ARC

Service Oriented Architecture (SOA) in the Real World

Table of Contents

Chapter 1: Service Oriented Architecture (SOA) 7
- Reader ROI 7
- Acknowledgements 7
- Introduction to SOA 8
  - The SOA Elephant 8
  - A simple definition for SOA 9
  - SOA Myths and Facts 11
  - The Evolution of SOA 12
  - Why should I care about SOA? 14
- Understanding Services 16
  - The Tenets of Service Design 18
  - Tenet 1: Boundaries are Explicit 18
  - Tenet 2: Services Are Autonomous 20
  - Tenet 3: Services share schema and contract, not class 21
  - Tenet 4: Service compatibility Is based upon policy 23
- An Abstract SOA Reference Model 24
  - Expose 25
  - Compose 25
  - Consume 26
- Recurring Architectural Capabilities 27
  - Messaging and Services 27
  - Workflow and Process 28
  - Data 28
  - User Experience 28
  - Identity and Access 28
  - Management 29
  - Supporting the Common Architectural Capabilities 29
- Common Architectural Capabilities and the Abstract SOA Model 30
  - Expose 30
  - Compose 33
  - Consume 34
- Summary 36
- References: 38

Chapter 2: Messaging and Services 39
- Reader ROI 39

Acknowledgements 40
Understanding Services 41
A SOA Maturity Model (*another* one??) 41
A Service Taxonomy 45
A Services Lifecycle 55
Service Analysis 55
Service Development 56
Service Testing 56
Service Provisioning 56
Service Operation 56
Service Consumption 57
Service Change Management 57
Service Decommission 57
SOA Scenarios 58
Information Integration 58
Legacy Integration 58
Process Governance 58
Consistent Access 59
Resource Virtualization 59
Process Externalization 59
Other Scenarios 59
SOA and the End User 60
What are Composite Applications? 62
What does a Composite Application look like? 65
Expected Benefits of Composition, and How to Achieve Them 67
Conclusion 67
SOA Case Study: Commonwealth Bank of Australia 69
References: 71
Chapter 3: Workflow and Process 72
Reader ROI 72
Acknowledgements 73
Understanding Workflow 74
What is Workflow? 74
Workflow Terminology 74
Why Workflow? 75
A Workflow Model 76
Workflow Contracts 77

Problem-Resolution Collaboration 78
Scripted Operations 80
Rule and Policy 81
Workflow Platform Value 83
More Semantic Exploitation 85
Platform Characteristics 86
A Common Workflow Runtime 87
Attacking the Problems 88
A Workflow Manifesto 89
Agility 89
Abstraction 90
Workflow is *Everywhere* 90
Workflow is *Expressive* 95
Workflow is *Fluid* 96
Workflow is *Inclusive* 97
Workflow is *Transparent* 97
Understanding the Relationship between BizTalk Server and WF 98
Conclusion 100
SOA Case Study: Dollar Thrifty Automotive Group 101
References: 102
Chapter 4: Data 103
Reader ROI 103
Acknowledgements 103
Data Challenges Facing SOA 104
Overview 104
Data Integration Issues 104
Database Scalability 107
Master Data Management (MDM) 109
What is MDM? 110
Customer Data Integration (CDI) 111
Product Information Management (PIM) 111
Master Data Management (MDM) Hub Architecture 111
Hub Architecture Styles 112
Architectural Issues 116
Versions and Hierarchies 117
Population and Synchronization 122
Publishing Updates 128

Data Integrity and Reliability 130
Metadata 130
Stewardship and Governance 131
Data Profiling 132
Export 132
Reporting 132
Workflow and Business Rules 132
Tools 133
Conclusion 133
SOA Case Study: London Stock Exchange 134
References: 135
Chapter 5: User Interaction 136
Reader ROI 136
Acknowledgements 136
What is Architecture? 137
Introducing a Framework for UX 138
Interface 139
Interaction 146
Infrastructure 151
SOA Case Study: Zurich Airport 162
References: 163
Chapter 6: Identity and Access 164
Reader ROI 164
Acknowledgements 165
Identity and Access 166
Overview 166
Trusted Subsystem Design 168
Current Practices 169
Trusted Subsystem Design 175
Trusted subsystem process extensions 177
Trusted Subsystem Policies 178
Flowing an Original Caller's Identity Claims 179
Identity/credential mapping 182
Benefits of the Design 182
An Identity Metasystem 183
What is the Identity Metasystem? 184
Identities Function in Contexts 185

The Laws of Identity 186
Roles within the Identity Metasystem 186
Components of the Identity Metasystem 187
Benefits of the Identity Metasystem 189
An Architecture for the Identity Metasystem: WS-* Web Services 190
Implementing the Identity Metasystem 191
Conclusion 194
SOA Case Study: OTTO 195
References: 196

Chapter 1: Service Oriented Architecture (SOA)

"SOAs are like snowflakes – no two are alike."

- David Linthicum
Consultant

Reader ROI

Readers of this chapter will learn about some of the general concepts generally associated with Service Oriented Architectures (SOA). The chapter provides several analogies for understanding service oriented concepts and some high level recommendations for designing services. This chapter provides an abstract model for describing SOA and introduces a set of architectural capabilities that will be explored in subsequent chapters of this book.

Acknowledgements

The vast majority of this book consists of content from a wide variety of people. Some of this content is new while other content may have appeared in other formats. Each chapter in this book will begin with an "Acknowledgements" section to thank the authors of the content that comprises each chapter.

Many of the concepts in Chapter One are based upon earlier published efforts. We wish to thank the following individuals for their work in this area: Don Box (Four Tenets), John deVadoss (Recurring Architectural Capabilities), and Kris Horrocks (Expose/Compose/Consume).

Introduction to SOA

The SOA Elephant

SOA has become a well-known and somewhat divisive acronym. If one asks two people to define SOA one is likely to receive two very different, possibly conflicting, answers. Some describe SOA as an IT infrastructure for business enablement while others look to SOA for increasing the efficiency of IT. In many ways SOA is a bit like John Godfrey Saxe's poem about the blind men and the elephant. Six blind men from Indostan encounter an elephant – each of the men then describes the elephant a bit differently because they are influenced by their individual experiences:

- The man touching the trunk believes it to be a snake
- The man touching the tusk believes it to be a spear
- The man touching the ear believes it to be a fan
- The man touching the elephant's side believes it to be a wall
- The man touching the tail believes it to be a rope
- The man touching the legs believes they are trees.

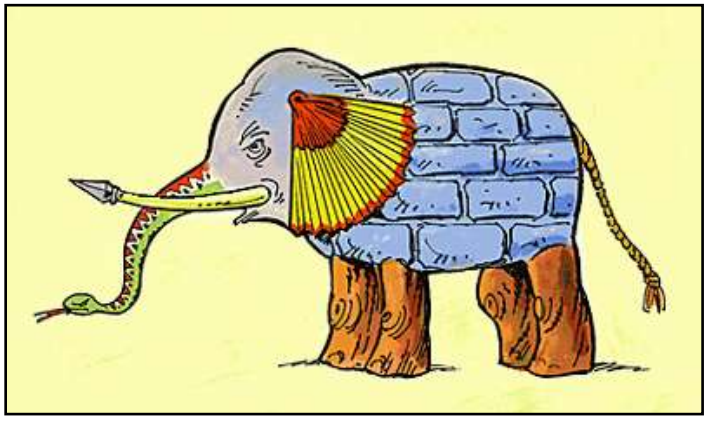


Figure 1: *Saxe's Elephant*

The blind men then engage in a series of debates about what they believe are facing them:

"…And so these men of Indostan
Disputed loud and long,
Each in his own opinion
Exceeding stiff and strong,

Though each was partly in the right,
And all were in the wrong!"

In many ways Mr. Saxe's poem has become a prophecy for SOA. Industry analysts, pundits, bloggers and reporters engage each other in an ongoing, never-ending debate about what is or isn't SOA. Like Mr. Saxe's blind men, people have correctly identified many of the capabilities of SOA but largely fail to communicate the concept as a whole. The challenge of defining SOA has become so important that various vendor consortia and standards organizations have launched initiatives to try and answer the question "What is SOA?"

A simple definition for SOA

For the purposes of this book, we will define SOA as:

A loosely-coupled architecture designed to meet the business needs of the organization.

At first glance this definition seems far too simplistic – where is SOAP, web services, WSDL, WS-* and other related standards? A SOA does not necessarily require the use of Web Services – Web Services are, for most organizations, the simplest approach for implementing a loosely coupled architecture. In the past, loosely coupled architectures have relied upon other technologies like CORBA and DCOM or document-based approaches like EDI for B2B integration. Many of these technologies are still in widespread use and are being augmented, replaced or extended with Web Services. Our definition works for us not because the focus here is not on the technology of SOA but upon meeting the needs of the organization. In simpler terms, one organization's SOA may look like nothing more than a bunch of Web Services (or other technologies) to another. There may be some common infrastructure capabilities such as logging and authentication, but for the most part a SOA for one organization will be quite different from the SOA used by another.

Many analysts and industry pundits have confused the concept of Service Oriented Architecture with Service Oriented Implementations. This has only added to the confusion associated with SOA and its related concepts. This can lead to disastrous results.

The Winchester Mystery House is an intriguing tourist attraction in the USA near San Jose, CA. The Winchester Mystery House was the home to the heiress of the Winchester fortune (amassed from the sales of Winchester rifles). According to the legend, the heiress went to see a fortune teller and learned she was cursed to be haunted by the spirits of everyone ever killed by a Winchester rifle. The only way to avoid the curse was to build a mansion – as long as she kept building the spirits would leave her alone. She promptly hired 147 builders (and 0 architects), all of whom began working on the mansion simultaneously. The builders worked on the mansion

until the heiress passed away, 38 years later. The result of their efforts is a classic example of implementation without architecture:

- The mansion contains 160 rooms, 40 bedrooms, 6 kitchens, 2 basements and 950 doors
- Of there 950 doors, 65 of them open to blank walls, 13 staircases were built and abandoned and 24 skylights were installed into various floors.
- No architectural blueprint for the mansion was ever created.

Figure 2: *The Winchester Mystery House*

Confusing architecture with implementation generates chaotic and unpredictable results – much like the Winchester Mystery House. Articles that try to explain SOA and jump into a tutorial for building Web Services are providing guidance for coding, not architecture. This is one of the many reasons that SOA is so misunderstood today – the rush to promote loosely coupled architectures focuses on the trees instead of the forest.

The architectural concepts associated with SOA are not new – many have evolved from ideas originally introduced by CORBA, DCOM, DCE and others. Unlike these previous initiatives, the key promise of SOA is to enable agile business processes via open, standards-based interoperability. While these standards are important we must remember that standards are not architecture and architectures are not implementations. At the end of the day it is the *implementation* of a well-designed architecture that will generate business benefits, not the architecture itself.

SOA is an architectural approach to creating systems built from autonomous services. With SOA, integration becomes forethought rather than afterthought - the end solution is likely to be composed of services developed in different programming languages, hosted on disparate platforms with a variety of security models and business processes. While this concept sounds incredibly complex it is not new – some may argue that SOA evolved out of the experiences associated with designing and developing distributed systems based on previously available

technologies. Many of the concepts associated with SOA such as services, discovery and late binding were associated with CORBA and DCOM. Similarly, many service design principles have much in common with earlier OOA/OOD techniques based upon encapsulation, abstraction and clearly defined interfaces.

Does the buzz around SOA and services mean that IT *wasn't* service-oriented in the past? No - IT (outsourced or not) exists solely to enable the business. Without IT businesses will have tremendous difficulty in both execution and competition. However, if IT cannot respond to business needs and opportunities fast enough then IT is perceived as a constrainer to the business instead of an enabler.

SOA promises to help IT respond to market conditions in a much timelier manner. SOA, however, is an architectural philosophy and is not necessarily an implementable concept. Many analysts and trade magazines have confused architecture with implementation – this leads one to believe that an implementation is, in fact, an architecture which can lead to disastrous results.

Organizations have different requirements and expectations for SOA because of vastly different business needs and objectives. This simple fact is one of the reasons that describing SOA can be such a challenge. SOA, like any initiative, must provide some level of value to the organization – otherwise there would be no use in ever considering it. The best way to ensure that SOA investments will provide a return to the organization is to align SOA with the organization's business drivers. Despite this obvious fact there is still a lot of confusion about SOA.

SOA Myths and Facts

There are several myths associated with SOA which are very important to understand before digging deeper into it. The table below describes some of the top myths surrounding SOA and the facts to help debunk them.

Myth	Fact
SOA is a technology	SOA is a design philosophy independent of any vendor, product, technology or industry trend. No vendor will ever offer a "complete" SOA "stack" because SOA needs vary from one organization to another. Purchasing your SOA infrastructure from a single vendor defeats the purpose of investing in SOA.
SOAs require Web Services	SOAs may be realized via Web services but Web services are not necessarily required to implement SOA

Myth	Fact
SOA is new and revolutionary	EDI, CORBA and DCOM were conceptual examples of SO
SOA ensures the alignment of IT and business	SOA is not a methodology
A SOA Reference Architecture reduces implementation risk	SOAs are like snowflakes – no two are the same. A SOA Reference Architecture may not necessarily provide the best solution for your organization
SOA requires a complete technology and business processes overhaul	SOA should be incremental and built upon your current investments
We need to build a SOA	SOA is a means, not an end

Focus on delivering a solution, not an SOA. SOA is a means to delivering your solution and should not be your end goal.

The Evolution of SOA

Service Orientation (SO) is the natural evolution of current development models. The, 80s saw object-oriented models; then came the component-based development model in the 90s; and now we have service orientation (SO). Service orientation retains the benefits of component-based development (self-description, encapsulation, dynamic discovery and loading), but there is a shift in paradigm from remotely invoking methods on objects, to one of passing messages between services. Schemas describe not only the structure of messages, but also behavioral contracts to define acceptable message exchange patterns and policies to define service semantics. This promotes interoperability, and thus provides adaptability benefits, as messages can be sent from one service to another without consideration of how the service handling those messages has been implemented.

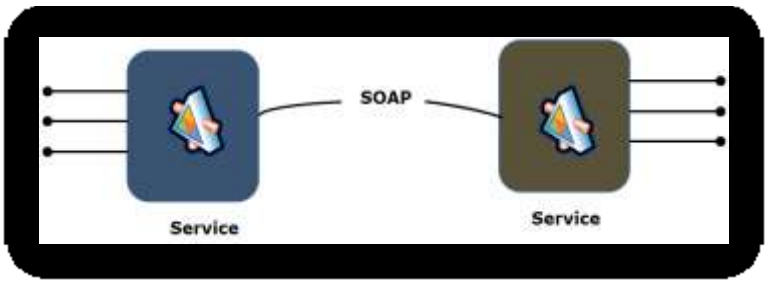


Figure 3: *Simple SOAP-based communications between Web Services*

Service orientation provides an evolutionary approach to building distributed software that facilitates loosely coupled integration and resilience to change. With the advent of the WS-* Web Services, architecture has made service-oriented software development feasible by virtue of mainstream development tools support and broad industry interoperability. Although most frequently implemented using industry standard Web services, service orientation is independent of technology and its architectural patterns and can be used to connect with legacy systems as well.

Unfortunately, the benefits offered by service orientation and SOA have been obscured by the hype and confusion that increasingly surround the terms. As awareness and excitement around SOA have swelled, the clear lines that once defined service orientation have been blurred. However SO does offer some specific benefits when utilized for the right purpose. There are three important observations about SO:

1. *It's evolutionary*: The principles of service-oriented development build on decades of experience in building real world distributed applications. SO incorporates concepts such as self-describing applications, explicit encapsulation, and dynamic loading of functionality at runtime – principles first introduced in the 1980s and 1990s through object-oriented and component-based development. What changes with SO is the metaphor with which developers achieve these benefits. Instead of using method invocation on an object reference, service orientation shifts the conversation to that of message passing – a proven metaphor for scalable distributed software integration.
2. *It's not a product or technology*: It is a set of architectural principles expressed independently of any product. Just as development concepts such as polymorphism and encapsulation are independent of technology, so is service orientation. And while Web services have in recent years facilitated the development of service-oriented applications, they are not required to do so.
3. *It's incremental:* Finally, service orientation can and should be an incremental process – one that can often be done in-house. Customers should not be required to dramatically re-engineer their businesses to attain the benefits of service orientation. Rather, they should be able to leverage existing IT assets in doing so. Service-oriented development can often be achieved using the skills and technologies customers already have today.

The fundamental building block of service-oriented architecture is a *service*. A *service* is a program that can be interacted with through well-defined message exchanges. Services must be designed for both availability and stability. Services are built to last while service configurations and aggregations are built for change. Agility is often promoted as one of the biggest benefits of SOA—an organization with business processes implemented on a loosely-coupled infrastructure is much more open to change than an organization constrained by underlying monolithic applications that require weeks to implement the smallest change. Loosely-coupled systems

result in loosely-coupled business processes, since the business processes are no longer constrained by the limitations of the underlying infrastructure. Services and their associated interfaces must remain stable, enabling them to be re-configured or re-aggregated to meet the ever-changing needs of business. Services remain stable by relying upon standards-based interfaces and well-defined messages— for example using SOAP and XML schemas for message definition. Services designed to perform simple, granular functions with limited knowledge of how messages are passed to or retrieved from it are much more likely to be reused within a larger SOA infrastructure.

Service Orientation does not necessarily require rewriting functionality from the ground up. Following the four tenets (see below) enables reuse of existing IT assets by wrapping them into modular services that can be plugged into any business process that you design. The goals for doing this should be:

- Connect into what is already there - Layer business process management, collaborative workflows, and reporting on top of existing IT assets.
- Extract more value from what is already there - Enable existing applications to be re-used in new ways.
- Extend and evolve what we already have - Create IT support for new cross-functional business processes that extend beyond the boundaries of what the existing applications were designed to do.

One of the key benefits of service orientation is loose coupling. No discussion of Web services seems complete without some reference to the advantages of looser coupling of endpoints (applications) facilitated by the use of Web service protocols. The principle is that of using a resource only through its published service and not by directly addressing the implementation behind it. This way, changes to the implementation by the service provider should not affect the service consumer. By maintaining a consistent interface, the service consumer could choose an alternative instance of the same service type (for example change service provider) without modifying their requesting application, apart from the address of the new instance. The service consumer and provider do not have to have the same technologies for the implementation, interface, or integration when Web services are used (though both are bound to use the same Web service protocols).

Why should I care about SOA?

Service Orientated Architecture is important to several stakeholders:

- To developers and solution architects, service orientation is a means for creating dynamic, collaborative applications. By supporting run-time selection of capability providers, service orientation allows applications to be sensitive to the content and

context of a specific business process, and to gracefully incorporate new capability providers over time.

- To the IT manager, service orientation is a means for effectively integrating the diverse systems typical of modern enterprise data centers. By providing a model for aggregating the information and business logic of multiple systems into a single interface, service orientation allows diverse and redundant systems to be addressed through a common, coherent set of interfaces.
- To the CIO, service orientation is a means for protecting existing IT investments without inhibiting the deployment of new capabilities. By encapsulating a business application behind capability-based interfaces, the service model allows controlled access to mission-critical applications, and creates the opportunity for continuous improvement of the implementation behind that interface. Service orientation protects investments from the swirl of change.
- To the business analyst, service orientation is a means of bringing information technology investments more in line with business strategy. By mapping employees, external capability providers and automation systems into a single model, the analyst can better understand the cost tradeoffs associated with investments in people, systems and sourcing.
- To Microsoft, service orientation is a crucial prerequisite to creating applications that leverage the network to link the actors and systems that drive business processes. This application model transcends any single device, crossing boundaries respectfully, and rejecting the restrictions of synchronicity. SOA-enabled solutions pull together a constellation of services and devices to more effectively meet your business challenges than the disconnected applications of the past.

The architectural concepts associated with SOA enable loose coupling. Loose coupling is the fundamental principle behind SOA, enabling us to summarize up the benefit of SOA in a single word: ***agility***.

As traditional application architectures evolved from the mainframe application through client server to multi-tier web applications, the applications have remained to a large extent tightly coupled. In other words, each of the subsystems that comprise the greater application is not only semantically aware of its surrounding subsystems, but is physically bound to those subsystems at compile time and run time. The ability to replace key pieces of functionality in reaction to a change in a business model, or of distributing an application as individual business capabilities, is simply not possible.

With a Service Oriented Architecture the application's functionality is exposed through a collection of services. These services are independent and encapsulate both the business logic and its associated data. The services are interconnected via messages with a schema defining their format; a contract defining their interchanges and a policy defining how they should be exchanged.

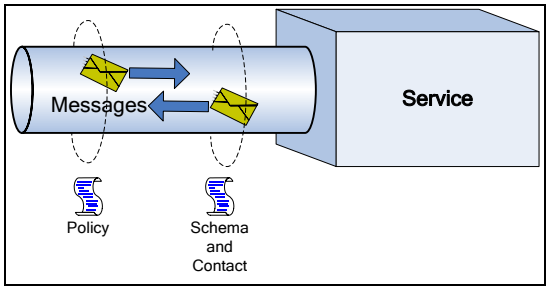


Figure 4: *Anatomy of a Web Service*

An application's services are designed to last with the expectation that we cannot control where and who consumes them. This is one of the key differences between SOA and traditional application architectures. Traditionally applications are designed to interact with humans, and the application vendor would provide the user-interface, the underlying business components and data stores. While good engineering disciplines separated the business components from the user interface, the only consumers of this business logic were the user interfaces delivered as part of the application. These UI and business components were traditionally deployed and versioned as a single entity. With service orientation, the business functionality exposed through the service can be used by any consumer outside of the control of the application. These consumers can be other services (incorporating the encapsulated business process) or alternatives to the user interfaces provided by the application itself. Therefore, contracts for these services, once published, must remain constant as we have no idea who is consuming them, nor when. In addition to offering services and exploiting its own services, an application itself should have the flexibility to adapt to new services offered after deployment. The availability and stability of these services therefore becomes a critical factor.

Understanding Services

The first step in any SOA undertaking is to clearly identify the critical business problems or challenges. The more precisely these can be defined the easier it will be to determine the scope and direction of each SOA project. By setting clear vision and direction from the top, it will be easier to obtain buy in on projects that are cross-functional in nature. Once the business drivers of the organization are defined, the service analysis process can begin. Service analysis is one of several steps that comprise a Service Lifecycle (Chapter Two provides more information about

the Service Lifecycle). The Service Lifecycle explains the necessary steps an organization must go through to define, develop, deploy and operate a service.

Services are commonly used to expose IT investments such as legacy platforms and Line of Business applications. Services can be assembled (or "composed") into business processes, and be made available for consumption by users, systems or other services. The process is an iterative one of creating ("exposing") new services, aggregating ("composing") these services into larger composite applications, and making the outputs available for consumption by the business user.

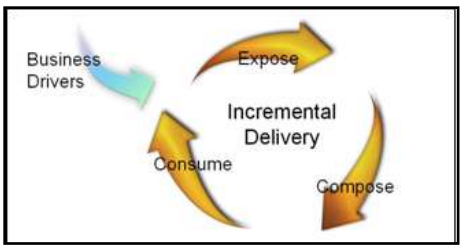


Figure 5: *An incremental, business-driven approach to SOA*

Fundamental to the service model is the separation between the interface and the implementation. The invoker of a service need only (and should only) understand the interface; the implementation can evolve over time without disturbing the clients of the service. Several key benefits of service orientation derive from this abstraction of the capability from how the capability is delivered. This means that, the same interface can be offered by many implementations, or conversely, that implementations can change without affecting the aggregate application. At its most abstract, service orientation views everything — from the mainframe application to the printer to the shipping dock clerk to the overnight delivery company — as a service provider. The service model is "fractal:" the newly formed process is a service itself, exposing a new, aggregated capability.

What is a service? In this book we will avoid using the term *Web* Services simply because all services are not necessarily *Web* Services. A service might also be manifested as an OS-specific process like a Unix daemon or a Windows Service. A service might also be an application that uses a well-defined contract that may or may not be based upon Web Services. Regardless of how the actual services are developed, they must be capable of participating within a loosely coupled architecture. There are four distinct principles (sometimes referred to as "Tenets") that can help ensure a loosely-coupled architecture. These Tenets of Service Design are defined below:

The Tenets of Service Design

The acronym SOA prompts an obvious question – what, exactly, is a service? Simply put, a service is a program that can be interacted with via well-defined message exchanges. Services must be designed for both availability and stability. Services are built to last while service configurations and aggregations are built for change. Agility is often promoted as one of the biggest benefits of SOA – an organization with business processes implemented on a loosely-coupled infrastructure is much more open to change than an organization constrained underlying monolithic applications that require weeks to implement the smallest change. Loosely-coupled systems result in loosely-coupled business processes, since the business processes are no longer constrained by the limitations of the underlying infrastructure. Services and their associated interfaces must remain stable, enabling them to be re-configured or re-aggregated to meet the ever-changing needs of business. Services remain stable by relying upon standards-based interfaces and well-defined messages – in other words, SOAP and XML schemas for message definition. Services designed to perform simple, granular functions with limited knowledge of how messages are passed to or retrieved from it are much more likely to be reused within a larger SOA infrastructure. As stated earlier, recalling basic OO design principles regarding encapsulation and interface design will serve us well as we design and build reusable Web Services. We can extend these OO principles into the world of Web Services by further understanding the frequently cited “four tenets” of Service Orientation:

Tenet 1: Boundaries are Explicit

Services interact through explicit message-passing over well-defined boundaries. Crossing service boundaries may be costly, depending upon geographic, trust or execution factors. A boundary represents the border between a service’s public interface and its internal, private implementation. A service’s boundary is published via WSDL and may include assertions dictating the expectations of a given service. Crossing boundaries is assumed to be an expensive task for several reasons, some of which are listed below:

- The physical location of the targeted service may be an unknown factor.
- Security and trust models are likely to change with each boundary crossing.
- Marshalling and casting of data between a service’s public and private representations may require reliance upon additional resources – some of which may be external to the service itself.

- While services are built to last, service configurations are built to change. This fact implies that a reliable service may suddenly experience performance degradations due to network reconfigurations or migration to another physical location.

- Service consumers are generally unaware of how private, internal processes have been implemented. The consumer of a given service has limited control over the performance of the service being consumed.

The Service Oriented Integration Pattern tells us that "service invocations are subject to network latency, network failure, and distributed system failures, but a local implementation is not. A significant amount of error detection and correction logic must be written to anticipate the impacts of using remote object interfaces." While we should assume that crossing boundaries is an expensive process, we must also exercise caution in the deployment of local methods designed to minimize such boundary crossings. A system that implements monolithic local methods and objects may gain performance but duplicate functionality of a previously defined service (this technique was referred to as "cut and paste" in OOP and shares the same risks regarding versioning of the service).

There are several principles to keep in mind regarding the first Tenet of SO:

- Know your boundaries. Services provide a contract to define the public interfaces it provides. All interaction with the service occurs through the public interface. The interface consists of public processes and public data representations. The public process is the entry point into the service while the public data representation represents the messages used by the process. If we use WSDL to represent a simple contract, the **< message>** represents the public data while the **<portType>** represents the public process(es).

- Services should be easy to consume. When designing a service, developers should make it easy for other developers to consume it. The service's interface (contract) should also be designed to enable evolving the service without breaking contracts with existing consumers. (This topic will be addressed in greater detail in future papers in this Series.)

- Avoid RPC interfaces. Explicit message passing should be favored over an RPC-like model. This approach insulates the consumer from the internals of the service implementation, freeing service developers to evolve their services while minimizing the impact on service consumers (encapsulation via public messages instead of publicly available methods).

- Keep service surface area small. The more public interfaces that a service exposes the more difficult it becomes to consume and maintain it. Provide few well-defined public interfaces to your service. These interfaces should be relatively simple, designed to

accept a well-defined input message and respond with an equally well-defined output message. Once these interfaces have been designed they should remain static. These interfaces provide the "constant" design requirement that services must support, serving as the public face to the service's private, internal implementation.

- Internal (private) implementation details should not be leaked outside of a service boundary. Leaking implementation details into the service boundary will most likely result in a tighter coupling between the service and the service's consumers. Service consumers should not be privy to the internals of a service's implementation because it constrains options for versioning or upgrading the service.

Tenet 2: Services Are Autonomous

Services are entities that are independently deployed, versioned, and managed. Developers should avoid making assumptions regarding the space between service boundaries since this space is much more likely to change than the boundaries themselves. For example, service boundaries should be static to minimize the impact of versioning to the consumer. While boundaries of a service are fairly stable, the service's deployment options regarding policy, physical location or network topology is likely to change.

Services are dynamically addressable via URIs, enabling their underlying locations and deployment topologies to change or evolve over time with little impact upon the service itself (this is also true of a service's communication channels). While these changes may have little impact upon the service, they can have a devastating impact upon applications consuming the service. What if a service you were using today moved to a network in New Zealand tomorrow? The change in response time may have unplanned or unexpected impacts upon the service's consumers. Service designers should adopt a pessimistic view of how their services will be consumed – services will fail and their associated behaviors (service levels) are subject to change. Appropriate levels of exception handling and compensation logic must be associated with any service invocation. Additionally, service consumers may need to modify their policies to declare minimum response times from services to be consumed. For example, consumers of a service may require varying levels of service regarding security, performance, transactions, and many other factors. A configurable policy enables a single service to support multiple SLAs regarding service invocation (additional policies may focus on versioning, localization and other issues). Communicating performance expectations at the service level preserves autonomy since services need not be familiar with the internal implementations of one another.

Service consumers are not the only ones who should adopt pessimistic views of performance – service providers should be just as pessimistic when anticipating how their services are to be consumed. Service consumers should be expected to fail, sometimes without notifying the

service itself. Service providers also cannot trust consumers to “do the right thing”. For example, consumers may attempt to communicate using malformed/malicious messages or attempt to violate other policies necessary for successful service interaction. Service internals must attempt to compensate for such inappropriate usage, regardless of user intent.

While services are designed to be autonomous, no service is an island. A SOA-based solution is fractal, consisting of a number of services configured for a specific solution. Thinking autonomously, one soon realizes there is no presiding authority within a service-oriented environment - the concept of an orchestration “conductor” is a faulty one (further implying that the concept of “roll-backs” across services is faulty– but this is a topic best left for another paper). The keys to realizing autonomous services are isolation and decoupling. Services are designed and deployed independently of one another and may only communicate using contract-driven messages and policies.

As with other service design principles, we can learn from our past experiences with OO design. Peter Herzum and Oliver Sims’ work on Business Component Factories provides some interesting insights on the nature of autonomous components. While most of their work is best suited for large-grained, component-based solutions, the basic design principles are still applicable for service design,

Given these considerations, here are some simple design principles to help ensure compliance with the second principle of SO:

- Services should be deployed and versioned independent of the system in which they are deployed and consumed
- Contracts should be designed with the assumption that once published, they cannot be modified. This approach forces developers to build flexibility into their schema designs.
- Isolate services from failure by adopting a pessimistic outlook. From a consumer perspective, plan for unreliable levels of service availability and performance. From a provider perspective, expect misuse of your service (deliberate or otherwise),expect your service consumers to fail – perhaps without notifying your service.

Tenet 3: Services share schema and contract, not class

As stated earlier, service interaction should be based solely upon a service’s policies, schema, and contract-based behaviors. A service's contract is generally defined using WSDL, while contracts for aggregations of services can be defined using BPEL (which, in turn, uses WSDL for each service aggregated).

Most developers define classes to represent the various entities within a given problem space (e.g. Customer, Order and Product). Classes combine behavior and data (messages) into a single programming-language or platform-specific construct. Services break this model apart to maximize flexibility and interoperability. Services communicating using XML schema-based messages are agnostic to both programming languages and platforms, ensuring broader levels of interoperability. Schema defines the structure and content of the messages, while the service's contract defines the behavior of the service itself.

In summary, a service's contract consists of the following elements:

- Message interchange formats defined using XML Schema
- Message Exchange Patterns (MEPs) defined using WSDL
- Capabilities and requirements defined using WS-Policy
- BPEL may be used as a business-process level contract for aggregating multiple services.

Service consumers will rely upon a service's contract to invoke and interact with a service. Given this reliance, a service's contract must remain stable over time. Contracts should be designed as explicitly as possible while taking advantage of the extensible nature of XML schema (xsd:any) and the SOAP processing model (optional headers).

The biggest challenge of the Third Tenet is its permanence. Once a service contract has been published it becomes extremely difficult to modify it while minimizing the impact upon existing service consumers. The line between internal and external data representations is critical to the successful deployment and reuse of a given service. Public data (data passed between services) should be based upon organizational or vertical standards, ensuring broad acceptance across disparate services. Private data (data within a service) is encapsulated within a service. In some ways services are like smaller representations of an organization conducting e-business transactions. Just as an organization must map an external Purchase Order to its internal PO format, a service must also map a contractually agreed-upon data representation into its internal format. Once again our experiences with OO data encapsulation can be reused to illustrate a similar concept – a service's internal data representation can only be manipulated through the service's contract.

Given these considerations, here are some simple design principles to help ensure compliance with the third principle of SO:

- Ensure a service's contract remains stable to minimize impact upon service consumers. The contract in this sense refers to the public data representation (data), message exchange pattern (WSDL) and configurable capabilities and service levels (policy).
- Contracts should be designed to be as explicit as possible to minimize misinterpretation. Additionally, contracts should be designed to accommodate future versioning of the service via the extensibility of both the XML syntax and the SOAP processing model.
- Avoid blurring the line between public and private data representations. A service's internal data format should be hidden from consumers while its public data schema should be immutable (preferably based upon an organizational, defacto or industry standard).
- Version services when changes to the service's contract are unavoidable. This approach minimizes breakage of existing consumer implementations.

Tenet 4: Service compatibility Is based upon policy

While this is often considered the least understood design tenet, it is perhaps one of the most powerful in terms of implementing flexible web services. It is not possible to communicate some requirements for service interaction in WSDL alone. Policy expressions can be used to separate structural compatibility (what is communicated) from semantic compatibility (how or to whom a message is communicated).

Operational requirements for service providers can be manifested in the form of machine-readable policy expressions. Policy expressions provide a configurable set of interoperable semantics governing the behavior and expectations of a given service. The WS-Policy specification defines a machine-readable policy framework capable of expressing service-level policies, enabling them to be discovered or enforced at execution time. For example, a government security service may require a policy enforcing a specific service level (e.g. Passport photos meeting established criteria must be cross-checked against a terrorist identification system). The policy information associated with this service could be used with a number of other scenarios or services related to conducting a background check. WS-Policy can be used to enforce these requirements without requiring a single line of additional code. This scenario illustrates how a policy framework provides additional information about a service's requirements while also providing a declarative programming model for service definition and execution.

A policy assertion identifies a behavior that is a requirement (or capability) of a policy subject. (In the scenario above the assertion is the background check against the terrorist identification system.) Assertions provide domain-specific semantics and will eventually be defined within

separate, domain-specific specifications for a variety of vertical industries (establishing the WS-Policy “framework” concept).

While policy-driven services are still evolving, developers should ensure their policy assertions are as explicit as possible regarding service expectations and service semantic compatibilities.

The four tenets are primarily designed to assist you in designing and developing your services.

An Abstract SOA Reference Model

While a well planned and executed SOA undertaking can help organizations realize greater responsiveness in a changing marketplace, not all service-oriented efforts have been successful. SOA projects may experience limited success when they are driven from the bottom up by developers unfamiliar with the strategic needs of the organization. Building SOA for the sake of SOA without reference to the business context is a project without organizing principles and guidance. The result is a chaotic implementation that has no business relevance. On the other hand, taking a top-down mega-approach to SOA requires such enormous time investments that by the time the project is complete, the solution no longer maps to business needs. (This of course is one of the problems SOA is supposed to solve!)

By contrast, Microsoft advocates a “middle out” approach which combines both top-down and bottom-up methodologies. In this approach, SOA efforts are driven by strategic vision and business need, and are met through incremental, iterative SOA projects that are designed to deliver on business goals one business need at a time. Microsoft has been using this technique to assist customers with their SOA initiatives since the .NET framework was first released in 1999.

The concept of SOA can be viewed from several possible perspectives. While no single perspective or set of perspectives represents a definitive view of a SOA, from a holistic view these perspectives assist in understanding the underlying architectural requirements. Microsoft believes that there are three abstract capability layers exposed within a SOA:

An illustration of these categories and their relationships to one another appears below:

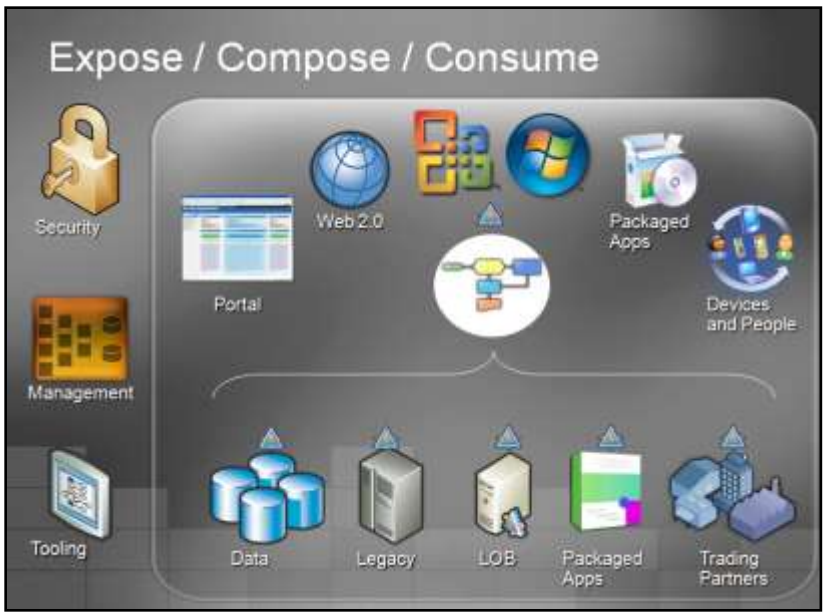


Figure 6: *An Abstract Reference Model for SOA*

Expose

Expose focuses on how existing IT investments are exposed as a set of broad, standards-based services, enabling these investments to be available to a broader set of consumers. Existing investments are likely to be based upon a set of heterogeneous platforms and vendors. If these applications are unable to natively support Web services a set of application or protocol-specific set of adapters may be required. Service creation can be fine grained (a single service that maps on to a single business process), or coarse grained (multiple services come together to perform a related set of business functions). Expose is also concerned with how the services are implemented. The functionality of underlying IT resources can be made available natively if they already speak Web services, or can be made available as Web services though use of an adapter. A *Service Implementation Architecture* describes how services are developed, deployed and managed.

Compose

Once services are created, they can be combined into more complex services, applications or cross-functional business processes. Because services are exist independently of one another they can be combined (or "composed") and reused with maximum flexibility. As business processes evolve, business rules and practices can be adjusted without constraint from the limitations of the underlying applications. Services compositions enable new cross-functional processes to emerge, allowing the enterprise to adopt new business processes, tune processes for greater efficiency, or improve service levels for customers and partners. A *Service Integration*

Architecture describes a set of capabilities for composing services and other components into larger constructs such as business processes. Composing services requires some sort of workflow or orchestration mechanism. Microsoft provides these capabilities via BizTalk Server 2006 (BTS) or Windows Workflow Foundation (WF). While BTS and WF may appear to serve similar needs, they are actually quite different. WF and BTS are *complementary* technologies designed to serve two very different needs:

- BTS is a ***licensed product*** designed to implement workflow ("orchestrations") ***across*** disparate applications and platforms.
- WF is a ***developer framework*** designed to expose workflow capabilities ***within*** your application. There are no fees or licensing restrictions associated with using or deploying WF.

We will examine workflow, orchestration and the use of BizTalk/WF in Chapter Three (Workflow and Business Processes).

Consume

When a new application or business process has been created that functionality must be made available for access (consumption) by IT systems, other services or by end-users. Consumption focuses on delivering new applications that enable increased productivity and enhanced insight into business performance. Users may consume "composed" services through a broad number of outlets including web portals, rich clients, Office business applications (OBA), and mobile devices. "Composed" services can be used to rapidly roll out applications that result in new business capabilities or improved productivity. These application roll-outs can be used to measure the return on investment (ROI) in an SOA. A *Service Oriented Application Architecture* describes how "composed services" are made available for consumption through as business processes, new services or new end-user applications. This concept is sometimes referred to as *Composite Applications* since it implies service consumption by end-user applications. Microsoft's Office Business Applications (OBAs) support the Composite Application notion of transactional systems while expanding the scope of user interaction via the familiar Office environment.

We will examine consumption in greater detail in Chapter Five (User Experience).

While the architectures described within Expose / Compose / Consume may be interdependent, they are designed to be loosely coupled. This enables services to be managed, versioned and configured independently of how they are exposed,

Recurring Architectural Capabilities

As we saw earlier, the SOA architectural model is fractal. This means that a service can be used to *Expose* IT assets (such as a Line of Business system), be *Composed* into workflows or Business Processes (each of which may also be exposed as a service) and be *Consumed* by end users, systems or other services. SOA is a fractal, not layered model. While the Abstract SOA Reference Model provides a holistic view of several important SOA concepts, the Expose / Compose / Consume portions of the model should not be interpreted as layers (despite their apparent appearance in the model). Designing a SOA as a set of well-defined tiers (or layers) will constrain the value and flexibility of your services, resulting in dependencies across unrelated components. This is why the Expose / Compose / Consume portions of the model can be thought of as independent architectural initiatives referred to as a *Service Implementation Architecture* (Expose), a *Service Integration Architecture* (Compose) and an *Application Architecture* (Consume). While each of these architectures are designed to be independent of one another, they share a set of five common capabilities

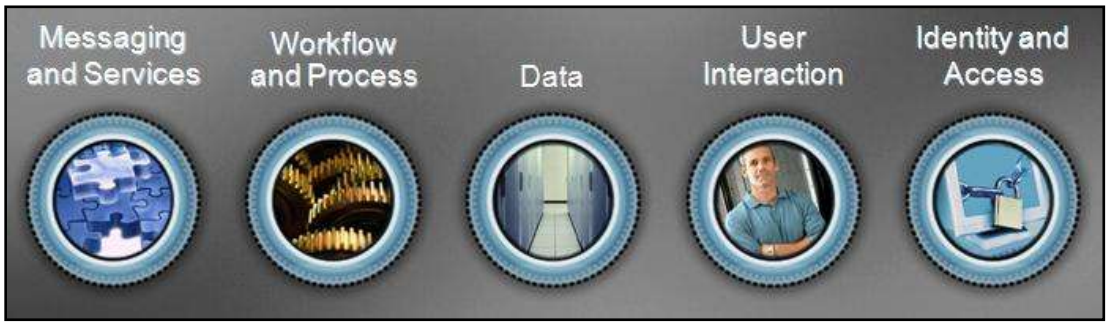


Figure 7: *Recurring Architectural Capabilities*

Messaging and Services

Messaging and Services focuses on how messaging is accomplished between senders and receivers. There are a broad array of options and patterns available – from pub/sub and asynchronous to message and service interaction patterns. Services provide an evolutionary approach to building distributed software that facilitates loosely coupled integration and resilience to change. The advent of the WS-* Web Services architecture has made service-oriented software development feasible by virtue of mainstream development tools support and broad industry interoperability. While most frequently implemented using industry standard Web services, service orientation is independent of technology and architectural patterns and can be used to connect with legacy systems as well. Messaging and Services are not a new approach to software design – many of the notions behind these concepts have been around for years. A

service is generally implemented as a coarse-grained, discoverable software entity that exists as a single instance and interacts with applications and other services through a loosely coupled (often asynchronous), message-based communication model. Messages tend to be based upon an agreed-upon set of semantics (such as an industry-standard Purchase Order) and serialized using an interoperable, extensible syntax (usually XML, although alternate approaches like JSON, RNC and ASN1 are sometimes used).

Workflow and Process

Workflow and Process are pervasive across multiple layers of an integration architecture – from formal orchestration of processes to flexible ad hoc systems and collaborative workflow across teams. Since business processes are dynamic and evolve with the organization, the workflow that models these processes must be equally adaptable. In addition, effective workflow involves not only process modeling, but also monitoring and analytics in order to respond to exceptions and optimize the workflow system over time.

Data

The lynchpin to success in many integration architectures is the ability to provide ***Data*** management. The need to deliver a shared view across disparate, often duplicate sources of data is more important than ever, as businesses strive to achieve a 360-degree view of organizational information. Entity aggregation, master data management, and the ability to make data useful via analytics and mining are crucial elements of integration architecture.

User Experience

Successful integration architectures depend upon both service delivery and the ability to consume services in a rich and meaningful way. Service consumption needs to be contextual, mapping to the natural workflow of employees, customers, and partners. To that end, an integrated ***User Experience*** spanning smart clients, rich clients, lightweight Web applications, and mobile devices enables service consumption by the broadest possible audience.

Identity and Access

To support integrated user experiences, customers require the ability to manage the identity lifecycle – providing integrated Single Sign-On (SSO), access management, directory services, and federated trust across heterogeneous systems. Today, many solutions are built using fragmented technologies for authentication and authorization. In the new application model, access decisions and entitlements need to be made at multiple layers and tiers, in which a federated ***Identity and Access*** across trust boundaries becomes a key requirement.

Management

During the lifetime of a service, the service most probably changes in different perspectives as listed below. As a result, one service will probably have to be available in several versions.

- Difference in interface (e.g. extended interface, but same business object)
- Difference in semantics with same interface (business objects changed)
- Difference in QoS, e.g. slower but cheaper or high-available but more expensive

Service management encompasses many capabilities, some of which are listed below:

- A comprehensive solution for change and configuration management, enabling organizations to provide relevant software and service updates to users quickly and cost-effectively.
- Reducing the complexity associated with managing the IT infrastructure environment with a focus on lowering the cost of operations.
- Centralized backup services capturing changed files to disk. Centralized backup should enable rapid, reliable recovery from disk while providing end-user recovery without IT intervention.
- Pre-deployment capacity planning coupled with best-practice guidance and hardware-specific knowledge to help information technology professional make low-risk architectural decisions.
- Data warehouse and reporting to help IT better support corporate decision making, improve the quality of service provided, and better administer resources through improved reporting capabilities and management data integration from a broad variety of resources.

Supporting the Common Architectural Capabilities

The five architectural capabilities discussed above are supported by the Microsoft SOA Platform. The remainder of this book discusses the common architectural capabilities in greater detail, starting with Messaging and Services in Chapter Two.

Figure 8: *SOA Capabilities on the Microsoft Platform*

Common Architectural Capabilities and the Abstract SOA Model

We can also think of these five common architectural capabilities as set of perspectives for viewing and understanding the Abstract SOA Model. The five architectural capabilities serves as a set of lenses to help us view and better understand the challenges associated with *Exposing* existing IT investments as services, *Composing* the services into business processes and *Consuming* these processes across the organization.

Expose

Service Enablement

Expose focuses on how we design and expose our services. We will most likely start out by enabling our IT investments to be exposed as web services.
As our organization matures we will start adding new services, most likely as proxies for other resources within the organization.
One of the hardest parts of service implementation is deciding where to begin. The are a variety of choices here and there is no single recommendation that will work for everyone. Motion is a methodology that provides some guidance for identifying business capabilities that could be exposes as services.

What are some best practices that one should follow when exposing IT investments as services?

- Think big– but start small
    - Show value at every step along the way – not build it and they will come
- Middle-out – not Top-down or bottom-up

    - Be pragmatic
- Vertical-slice – not build it and they will come
    - Risk-mitigation
- Demonstrate value in rapid iterations – not waterfall
    - New approaches to development
- Successful customers 'snowball'

The recurring architectural capabilities provide us with a set of considerations when exposing IT investments as services. Let's take a quick look at some of the considerations associated with each capability for service exposure (this is by no means a complete list):

Messaging and Services

Determining what to expose and how - Avoid falling into the granularity trap – focus on meeting your business requirements
Service Operation Contracts
Message and Data Contracts
Configuration, behaviors and control
SLAs
Governance
Versioning

Workflow and Process

Coordinator services for distributed, long-running processes
Tracking services capable of logging specific events within a workflow
Compensation services

Data

Entity services
Entity Aggregation services: acts as a single point to access information that may exist in multiple systems. An Entity Aggregation service has the following responsibilities:

- Acts as a unified source of entities.
- Provides a holistic view of an entity.
- Provides a holistic view of the entity model—entities and their relationships with other entities
- Provides location transparency—consumers of the Entity Aggregation layer do not need to know who owns the information.

- Enforces business rules that determine the segments of entities retrieved in a given context.
- Determines the system of record for each data element constituting an entity.
- Enriches the combined data model across systems—*the whole being better than the sum of its parts*.
- Entity factoring

MDM focuses on exposing data across corporate or departmental boundaries. We'll discuss MDM in greater detail in Chapter Four.

User Experience

Specialized services for supporting user interfaces (caching resources, communications between UI and services, etc). Service wrappers provide coarse-grained interfaces for user app consumption, lightweight mash-ups, etc.

Identity and Access

Identity Management

Impersonation and Delegation services

Trusted Subsystem - A trusted subsystem model implies that services are trusted to perform specific tasks, such as processing customer orders.

Authentication (Kerberos, SSL)

Role-based access control (RBAC)

Create/revoke trust relationships

Services need to make authorization decisions, such as approving an order submission before performing the business transaction.

The service must know the identity of the end user submitting the order.

Need to flow the identity of the end user is an inherent property of the delegation model, it is not so for the trusted subsystem model and special efforts must be made to include this feature.

To support the notion of trust as defined by the model, the services must at least be able to:

1. Authenticate / verify identity of upstream / downstream services
2. Decide if the service is a trusted subsystem for specific functions (including propagating identity claims)
3. Protect the integrity of the data being communicated between trusted subsystem and services.

Besides application data, application plumbing data, such as the identity claims of the original user, must also be protected so that no man-in-the-middle can modify the identity information that is in transit.

Compose

Service Composition

Compose focuses on how we can combine or aggregate granular services into more complex processes. We will most likely start by using services that expose our existing IT investments. Service composition results in a new service instance that the rest of the organization can make use of. The composition provides capabilities such as correlated asynchronous service invocation, long running processes and other capabilities for orchestrating autonomous services.

The recurring architectural capabilities provide us with a set of considerations when composing granular services into complex processes. Let's take a quick look at some of the considerations associated with each capability for service composition (this is by no means a complete list):

Messaging and Services

Service interaction patterns
Exposing orchestrations as services
Asynchronous service invocation patterns

Workflow and Process

Transactions
High frequency of change
Business Rules
Service Orchestration
Service Interaction Patterns (SIPs)
Process Externalization
Long Running Processes
Auditing and analytics

Data

Tracking the state of a given workflow instance
Data transformation (ETL)
Reliable message processing and storage
Replication

Synchronization
Metadata repository and Management
Instance reconciliation
Schema reconciliation
Document Replication
Syndication/Aggregation

User Experience
Composite applications (OBAs)
Human workflows (MOSS)
Orchestrations initiate human workflows via SharePoint adapter
Pageflows

Identity and Access
Impersonation and Delegation
Provisioning
Identity Repository synchronization
Approval workflows

Consume

User Experience

Consume focuses on how services and orchestrated processes (which may be exposed as services) are consumed by other services, applications and end-users. Any resource capable of interacting with services can be referred to as a "consumer". Consumers may appear across the organization in several possible forms:

- Lightweight, browser-based applications
- Rich internet applications (RIA) are browser-based applications that can address and cache local and remote resources
- Configurable, portal-based user experiences
- Applications that are installed on the local machine (such as a custom Windows application)
- Corporate business applications with solution-specific extensions (such as Microsoft Office with context-aware activity panes)
- Applications designed for mobile devices
- Services may act as consumers of other services

Recall that the SOA model is fractal – services may be consumed by other services and service compositions may be exposed as new services. In the last couple of years a "new breed" of consumers has emerged, enabling consumers to be aggregated and consumed by additional consumers. This "new breed" of consumers is usually referred to as a "mashup". A mashup is a set of services, websites or applications that combine content from multiple resources into a new integrated user experience. Content used by mashups is typically sourced from a third party (such as a service or website) via a public interface or API. Alternative methods of sourcing content for mashups include newsfeeds and JavaScript constructs (JSON).

The recurring architectural capabilities provide us with a set of considerations for User Experience. Let's take a quick look at some of the considerations associated with each capability for User Experience (this is by no means a complete list):

Messaging and Services
Forms-based service consumption
Web parts
Service Registry – check in / check out / search
AJAX, REST

Workflow and Process
Human workflows (MOSS)
Event brokering (CAB)
Page flows

Data
Entities (OBA Business Data Catalog)
Single view of the customer problem
JSON

User Experience
Composite applications (OBAs)
Personalization, user profiles
Portals
Business Intelligence
Reporting
Content aggregation
Declarative UX

Identity and Access

Single Sign-On (password synchronization)

User identification

Role-based access (RBAC)

Directory Services

Password management

Privacy (firewalls, encryption)

Compliance

Summary

In this chapter we provided some useful analogies for understanding the fractal nature of SOA. Services are the fundamental building blocks of SOA, although services do not necessarily need to be ***web*** services. Ideally these services should follow the four service design tenets which describe a set of best practices for service scopes, dependencies, communications and policy-based configuration. While these tenets focus upon service design, it is important to realize that services alone are not necessarily solution architecture – Microsoft uses an abstract reference model to describe the various aspects of SOA. The abstract SOA reference model provides three fundamental concepts to help most organizations understand the role that services can play within their solution architectures:

- ***Expose*** focuses on how existing IT investments are exposed as a set of broad, standards-based services, enabling these investments to be available to a broader set of consumers. A *Service Implementation Architecture* describes how services are developed, deployed and managed.
- ***Compose*** focuses on combining services into applications or cross-functional business processes. A *Service Integration Architecture* describes a set of capabilities for composing services and other components into larger constructs such as business processes.
- ***Consume*** focuses on delivering new applications that enable increased productivity and enhanced insight into business performance. A *Service Oriented Application Architecture* describes how “composed services” are made available for consumption through as business processes, new services or new end-user applications.

Each aspect of the Expose / Compose / Consume abstract reference model encompasses a set of five recurring architectural capabilities: Messaging and Services, Workflow and Processes, Data, User Experience and Identity and Access. The five architectural capabilities serves as a

set of views to better understand the challenges associated with *Exposing* existing IT investments as services, *Composing* the services into business processes and *Consuming* these processes across the organization.

In the Chapter Two we will enter into a more detailed analysis of both Service Orientation and the **Messaging and Services** architectural capability.

References:

1. "Enabling "Real World" SOA through the Microsoft Platform", A Microsoft White Paper, December 2006. Available at http://www.microsoft.com/biztalk/solutions/soa/whitepaper.mspx
2. Service Oriented Integration Pattern: http://msdn2.microsoft.com/en-us/library/ms978594.aspx
3. "Business Component Factory", Peter Herzum and Oliver Sims, Wiley, 1999

Chapter 2: Messaging and Services

"SOA is not something you buy,
it's something you do."

– Jason Bloomberg
Analyst

Reader ROI

Readers of this chapter will build upon the concepts introduced in Chapter One, specifically focusing on the Messaging and Services architectural capability.

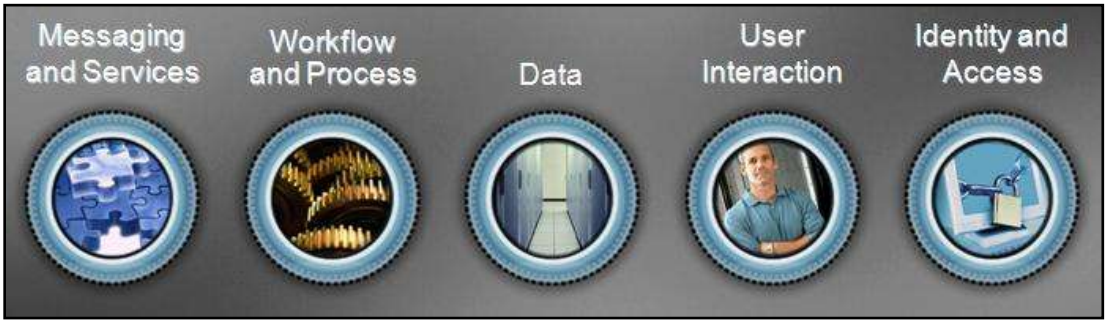


Figure 1: *Recurring Architectural Capabilities*

The Messaging and Services architectural capability focuses on the concept of service orientation and how different types of services are used to implement SOA. This chapter also touches upon the role of the user – specifically how a user will interact with services within a SOA.

Topics discussed in this chapter include:

- A maturity model for SOA
- A services lifecycle
- Sample SOA scenarios
- The role of the user within a SOA

The concepts covered in this chapter are not necessarily new. SOA Maturity Models and Service Lifecycles have been published by a broad variety of vendors, consultants and industry analysts. Like SOA itself, there is no single Maturity Model or Service Lifecycle that everyone

agrees upon. Readers should review several of these efforts and draw from them the aspects that best fit your organizational needs.

Acknowledgements

This chapter consists of work from the following individuals: Mark Baciak (Service Lifecycle), Atanu Bannerjee (OBA), Shy Cohen (Service Taxonomy), William Oellermann (Enterprise SOA Maturity Model), and Brenton Webster (SOA Case Study).

Understanding Services

A SOA Maturity Model (*another* one??)

There is an abundance of SOA Maturity Models available from vendors, consulting firms, analysts and book authors. Most of these Maturity Models are either based upon or inspired by the Software Engineering Institute's (now retired) Capability Maturity Model (CMM). A recent search on the terms "SOA Maturity Model" returned almost 10,000 relevant hits (including articles on what to look for in a Maturity Model). Given the intense interest and variety of maturity models available why introduce another? Unlike other Maturity Models, the one discussed here doesn't attempt to simply apply service orientation to the CMM. ESOMM (Enterprise Service Orientation Maturity Model) borrows CMM's notion of capability-driven maturity models and applies these principles to service-orientation paradigms, essentially building a road map from scratch. Unlike CMMI, the ESOMM Maturity Model doesn't focus on processes because the focus is on IT capabilities, not organizational readiness or adoption. While there are some conceptual similarities with CMM, ESOMM is a decidedly different application of the maturity model concept. The layers, perspectives, and capabilities defined in the ESOMM are designed as a road map to support services—not any specific service with any specific use, implementation, or application, but any service, or more specifically, any *set* of services.

Developing an enterprise-wide SOA strategy is not a trivial undertaking and should not be treated as a short-term or one-time effort. SOA is an attempt to enable a higher level of agility for the entire organization to facilitate an expedient response to the needs of the business and customers. There are many aspects of SOA that will be much more critical in the near- instead of long-term, so it is important to align your group's efforts accordingly. To do so successfully, the development of a prioritized roadmap should be a high priority to plan for success.
An ESOMM can provide a capability-based, technology-agnostic model to help identify an organization's current level of maturity, the short- and long-term objectives, and the opportunity areas for improvement.

Figure 2 provides an overview of ESOMM. ESOMM defines 4 horizontal maturity layers, 3 vertical perspectives, and 27 capabilities necessary for supporting SOA.

	Implementation	Consumption	Administration
Extensible	Service collaboration Service orchestration	Service SDKs External policy	Business analytics Automated policy management
Supportable	Versioning Executable policy Schema bank	Explicit SLAs Service portal Execution visibility	Auditing Monitoring Provisioning model
Repeatable	Common schema Service blocks	Self provisioning Service discoverability	Deployment management Enterprise policies
Usable	Design patterns Development processes	Testability Explicit contracts	Security model Basic monitoring

Figure 2: *An Enterprise Service Orientation Maturity Model (ESOMM)*

Most organizations will not naturally completely implement all of one layer's capabilities before moving up the model, but jumping too far ahead can be risky. This becomes very apparent as you recognize that some poor decisions concerning key building block capabilities can severely impact your ability to mature at higher layers.

Capabilities are categorized into one of three perspectives: Implementation, Consumption, and Administration. Implementation capabilities target the development and deployment of Web services from the provider's perspective. Consumption capabilities are those that cater to the consumers of your services, making them easier to implement and therefore more widely and successfully adopted. Administration capabilities include those that facilitate the operational and governance aspects of Web services across the organization. While an organization can choose to focus on a single perspective, the overall maturity, and hence value, of your SOA depends on the appropriate level of attention to all three.

ESOMM is a tool that can be used to:

- simplify the complexity of a massively complex and distributed solution.
- Identify and discuss an organization's adoption of service orientation.
- understand the natural evolution of service adoption (such as skipping certain capabilities at lower levels comes with certain risk).
- provide a cohesive and comprehensive service orientation plan for customers based on their objectives.

- align an organization's activities and priorities with a distinct level of value providing specific benefits through specific capabilities,

In a SOA assessment, each capability is assessed independently as a level of strength, adequacy, or weakness. A level of *strength* demonstrates that the organization is well positioned to safely grow their use of Web services without fearing pitfalls in this area down the road. A level of *adequacy* signals that an organization has sufficient capabilities to meet today's needs, but is vulnerable to a growth in Web services that could cause problems in that area. A *weakness* represents a deficiency that is a problem for today's use of Web services and could be a serious detriment in the future. Any capabilities that are obviously not part of the organization's objectives and appear to have minimal impact in the near term for the organization are classified as *not applicable*. These areas may quickly become weaknesses if business or technical objectives were to change or accelerate.

As in the CMM, individual capability levels drive an overall layer grade ranging from 1 to 5. A grade is assessed based on the average of the capability maturities within the layer. A rating of 5 represents a mastery of that level with little or no room for improvement. A 4 represents a solid foundation within an area that can be successfully built upon. A 3 is assigned when good efforts are established in that layer, but extending efforts to the next layer carries some risks. 2 means that only initial efforts have been made in the area and much more work is needed to extend to the next layer. A grade of 1 represents no effort to address the capabilities at that layer.

Now that we've had a chance to briefly review the components of ESOMM, you are hopefully already thinking about how it can be applied to your organization. To be clear, applying this model should not be looked at as a one-time activity or short-term process. Instead, the model is best leveraged as a working plan that can be modified over time as the usage of services and your experience grows.

Unfortunately, the downside of using the term *maturity* with a model is that people will immediately want to know what layer their organization is at to get a sense of their status or identity. As it happens, there is no appropriate way to answer the question, "what layer is my organization?" Instead of giving an overall grade based on one of the layers, we take the approach of giving each layer its own level of maturity, ranging from one to five, based on half-point increments.

ESOMM is intended to be leveraged as a road map, more so than as an SOA “readiness” or assessment tool. While it is important to know where you are, getting an exact bearing is less important than identifying the capabilities you need to address to continue advancing the value of service enablement in your organization. As long as you are willing to ask yourself some hard questions in an objective manner across all the relevant groups, you should be able to get a good understanding for your current challenges. If you apply the strategy and objectives of your organization, you should be able to identify which capabilities you will need to address in the near, short, and long term.

ESOMM is one of many possible maturity models that can be used to assess the capabilities of the enterprise in adopting and implementing SOA. It is terribly difficult to have a concise, constructive conversation about a service-enabled enterprise without some common agreement on capabilities and maturity – ESOMM is one of many maturity models that attempt to address this issue. Unlike other maturity models, however, ESOMM can also be leveraged as a prioritized roadmap to help enterprises identify the capabilities necessary for a successful implementation. The goal of ESOMM and other maturity models is to empower your organization with the tools and information needed for a successful SOA implementation.

In a way, SOA does render the infrastructure more complex because new capabilities will need to be developed that did not exist before, such as registry and repository. In some sense, it can be compared to the construction of a highway network—broad, safe roads are more expensive to build, and the users need to upgrade their means of transportation to make the most of that infrastructure, but the cost per trip (in time and safety) is driven down. Until the network reaches a critical mass, drivers still need to be prepared to go “off road” to reach their destination.
Many enterprise applications were not developed with SOA in mind, and are either incapable of leveraging an SOA infrastructure, or will need to be upgraded to take advantage of it. However, with more/new applications being created consistently, there is a great opportunity to drive down new interoperability costs as the various technology vendors enable their products for SOA.
The biggest challenge for SOA is convincing the application owners to invest more today to achieve those promised long term savings.

While maturity models like ESOMM can help clarify the capabilities necessary for SOA, the types of services your organization will need usually remains unanswered. A simple service taxonomy can assist you in better understanding the breadth of services that will typically exist within a SOA.

A Service Taxonomy

As we examine service types we notice two main types of services: those who are infrastructural in nature and provide common facilities that would not be considered part of the application, and those who are part of the application and provide the application's building blocks.

Software applications utilize a variety of common facilities ranging from the low-level services offered by the Operating System such as the memory management and I/O handling, to the high-level runtime-environment-specific facilities such as the C Runtime Library (RTL), the Java Platform, or the .NET Framework. Solutions built using a SOA make use of common facilities as well, such as a service-authoring framework (for example, Windows Communication Foundation) and a set of Services that are part of the supporting distributed computing infrastructure. We will name this set of services **Bus Services**.

Bus Services further divide into **Communication Services** which provide message transfer facilities such as message-routing and publish-subscribe mechanisms, and **Utility Services** which provide capabilities unrelated to message transfer such as service-discovery and federated security.

The efficiency of software applications development is further increased through reuse of coarse grained, high-level building blocks. The RAD programming environments that sprang up in the Component Oriented era (such as Delphi or Visual Basic) provided the ability to quickly and easily compose the functionality and capabilities provided by existing building blocks with application specific code to create new applications. Examples of such components range from the more generic GUI constructs and database access abstractions, to more specific facilities such as charting or event-logging. Composite applications in a SOA also use building blocks of this nature in their composition model. We will name these building blocks **Application Services**.

Application services further divide into **Entity Services** which expose and allow the manipulation of business entities, **Capability Services** and **Activity Services** which implement the functional building blocks of the application (sometimes referred to as components or modules), and **Process Services** which compose and orchestrate Capability and Activity Services to implement business processes.

The following diagram shows a possible composition of services in the abovementioned service categories.

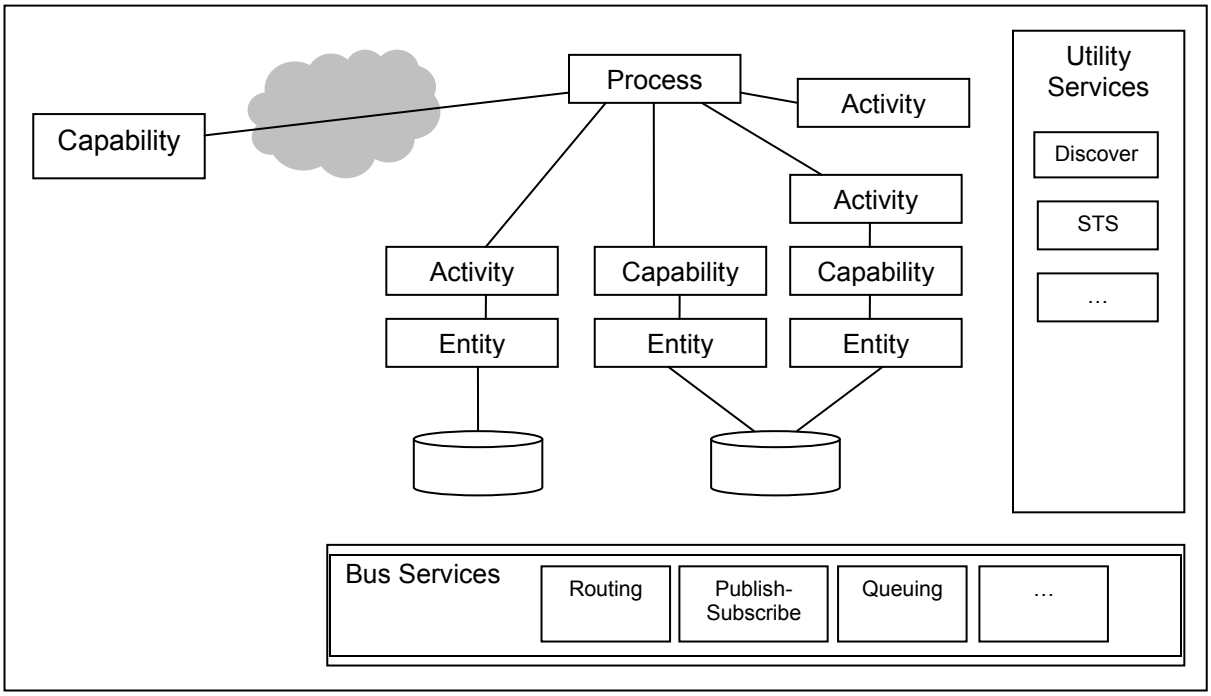


Figure 3: *Bus Services*

Bus Services are common facilities that do not add any explicit business value, but rather are a required infrastructure for the implementation of any business process in a SOA. Bus Services are typically purchased or centrally built components that serve multiple applications and are thus typically centrally managed.

Communication Services

Communication Services transport messages into, out of, and within the system without being concerned with the content of the messages. For example, a Bridge may move messages back and forth across a network barrier (i.e. bridging two otherwise-disconnected networks) or across a protocol barrier (e.g. moving queued messages between WebSphere MQ and MSMQ). Examples of Communication Services include relays, publish-subscribe systems, routers, queues, and gateways.

Communication Services to not hold any application state, but in many cases they are configured to work in concert with the applications that use them. A particular application may need to instruct or configure a Communication Service on how to move the messages flowing inside that application such that inter-component communication is made possible in a loosely coupled architecture. For example, a content-based router may require the application to provide routing

instructions such that the router will know where to forward messages to. Another example may be a publish-subscribe server which will deliver messages to registered subscribers based on a filter that can be applied to the message's content. This filter will be set by the application. In both cases the Communication Service does not process the content of the message but rather (optionally) uses parts of it as instructed by the application in advance for determining where it should go.

In addition to application-specific requirements, restrictions imposed by security, regulatory, or other sources of constraints may dictate that in order to use the facilities offered by a particular Communication Service users will need to possess certain permissions. These permissions can be set at the application scope (i.e. allowing an application to use the service regardless of the specific user who is using the application), at the user scope (i.e. allowing a specific user to use the service regardless of the application that the user is using), or at both scopes (i.e. allowing the specific user to access the service while running a specific application). For example, a publish-subscribe service may be configured to restrict access to specific topics by only allowing specific users to subscribe to them.

Other application-level facilities that may be offered by Communication Services pertain to monitoring, diagnostics, and business activity monitoring (BAM). Communication Services may provide statistical information about the application such as an analysis of message traffic patterns (e.g. how many messages are flowing through a bridge per second), error rate reports (e.g. how many SOAP faults are being sent through a router per day), or business-level performance indicators (e.g. how many purchase orders are coming in through a partner's gateway). Although they may be specific to a particular application, these capabilities are not different than the configuration settings used to control message flow. This information is typically provided by a generic feature of the Communication Service, which oftentimes needs to be configured by the application. The statistical information being provided typically needs to be consumed by a specific part of the application that knows what to do with it (e.g. raise a security alert at the data center, or update a BAM-related chart on the CFO's computer screen).

Utility Services

Utility Services provide generic, application-agnostic services that deal with aspects other than transporting application messages. Like Communication Services, the functionality they offer is part of the base infrastructure of a SOA and is unrelated to any application-specific logic or business process. For example, a Discovery service may be used by components in a loosely coupled composite-application to discover other components of the application based on some specified criteria (e.g. a service being deployed into a pre-production environment may look for another service which implements a certain interface that the first component needs and that is also deployed in the pre-production environment). Examples of Utility Services include security

and identity services (e.g. an Identity Federation Service or a Security Token Service), discovery services (e.g. a UDDI server), and message transformation services.

As in the case of Communication Services, Utility Services may too be instructed or configured by a particular application on how to perform an operation on their behalf. For example, a Message Transformation service may transform messages from one message schema to another message schema based on a transformation mapping that is provided by the application using the Message Transformation service.

Although Utility Services do not hold any application state, the state of a Utility Service may be affected by system state changes. For example, a new user being added to the application may require an update to the credential settings in the Security Token Service. Unlike in the case of Communication Services, client services directly interact with the Utility Services who process and (if needed) respond to the messages that the clients send to them.

Users of Utility Services may require a permission to be configured for them in order to use the service, be it at the application, user, or the application-user scope. For example, a Discovery service may only serve domain-authenticated users (i.e. users who have valid credentials issues by a Windows domain controller).

Like Communication Services, Utility Services may provide application-level facilities for monitoring, diagnostics, BAM, etc. These may include statistical information about usage patterns (e.g. how many users from another organization authenticated using a federated identity), business-impacting error rates (e.g. how many message format transformations of purchase orders failed due to badly formatted incoming messages), etc. As with Communication Services, these facilities are typically generic features of the Utility Service and they need to be configured and consumed by the particular solution in which they are utilized.

Application Services

Application Services are services which take part in the implementation of a business process. They provide an explicit business value, and exist on a spectrum which starts with generic services that are used in any composite-application in the organization on one end, ends with specialized services that are part of a single composite-application on the other end, and has services that may be used by two or more applications in between.

Entity Services

Entity Services unlock and surface the business entities in the system. They can be thought of as the data-centric components ("nouns") of the business process: employee, customer, sales-order, etc. Examples of Entity Services include services like a Customers Service that manages the

customers' information, an Orders Service that tracks and manages the orders that customers placed, etc.

Entity Services abstract data stores (e.g. SQL Server, Active Directory, etc.) and expose the information stored in one or more data stores in the system through a service interface. Therefore, it is fair to say that Entity Services manage the persistent state of the system. In some cases, the information being managed transcends a specific system and is used in several or even all the systems in the organization.

It is very common for Entity Services to support a CRUD interface at the entity level, and add additional domain-specific operations needed to address the problem-domain and support the application's features and use-cases. An example for a domain-specific operation is a Customers service that exposes a method called FindCustomerByLocation which can locate a customer ID given the customer's address.

The information that Entity Services manage typically exists for a time span that is longer than that of a single business process. The information that Entity Services expose is *typically* structured, as opposed to the relational or hierarchical data stores which are being fronted by the service. For example, a service may aggregate the information stored in several database tables or even several separate databases and project that information as a single customer entity.

In some cases, typically for convenience reasons, Entity Service implementers choose to expose the underlying data as DataSets rather than strongly-schematized XML data. Even though DataSets are not entities in the strict sense, those services are still considered Entity Services for classification purposes.

Users of Entity Services may require a permission to be configured for them in order to use the service, be it at the application, user, or the application-user scope. These permissions may apply restrictions on data access and/or changes at the "row" (entity) or "column" (entity element) level. An example for "column" level restriction would be an HR application might have access to both the social security and home address elements of the employee entity while a check-printing service may only have access to the home address element. An example for "row" level restriction would be an expense report application which lets managers see and approve expense reports for employees that report to them, but not for employees who do not report to them.

Error compensation in Entity Services is mostly limited to seeking alternative data sources, if at all. For example, if an Entity Service fails to access a local database it may try to reach out to a remote copy of the database to obtain the information needed. To support system-state consistency, Entity Services typically support tightly-coupled distributed atomic transactions. Services that support distributed atomic transactions participate in transactions that are flowed to them by callers and subject any state changes in the underlying data store to the outcome of these distributed atomic transactions. To allow for a lower degree of state-change coupling, Entity

Services may provide support for the more loosely-coupled reservation pattern, either in addition to or instead of supporting distributed atomic transactions.

Entity Services are often built in-house as a wrapper over an existing database. These services are typically implemented by writing code to map database records to entities and exposing them on a service interface, or by using a software factory to generate that mapping code and service interface. The Web Services Software Factory from Microsoft's Patterns & Practices group is an example of such a software factory. In some cases, the database (e.g. SQL Server) or data-centric application (e.g. SAP) will natively provide facilities that enable access to the data through a service interface, eliminating the need to generate and maintain a separate Entity Service.

Entity Services are often used in more than one composite-application and thus they are typically centrally managed.

Capability Services

Capability Services implement the business-level capabilities of the organization, and represent the action-centric building blocks (or "atomic verbs") which make up the organization's business processes. A few examples of Capability Services include third-party interfacing services such as a Credit Card Processing service that can be used for communication with an external payment gateway in any composite-application where payments are made by credit card, a value-add building block like a Rating Service that can process and calculate user ratings for anything that can be rated in any application that utilizes ratings (e.g. usefulness of a help page, a book, a vendor, etc.), or a communication service like an Email Gateway Service that can be used in any composite-application that requires the sending of emails to customers or employees. Capability Services can be further divided by the type of service that they provide (e.g. third-party interfacing, value-add building block, or communication service), but this further distinction is out of scope for this discussion.

Capability Services expose a service interface specific to the capability they represent. In some cases, an existing (legacy) or newly acquired business capability may not comply with the organization's way of exposing capabilities as services, or even may not expose a service interface at all. In these cases the capability is typically wrapped with a thin service layer that exposes the capability's API as a service interface that adheres to the organization's way of exposing capabilities. For example, some credit card processing service companies present an HTML-based API that requires the user to fill a web-based form. A capability like that would be wrapped by an in-house-created-and-managed-façade-service that will provide easy programmatic access to the capability. The façade service is opaque, and masks the actual nature of the capability that's behind it to the point where the underlying capability can be replaced without changing the service interface used to access it. Therefore, the façade service is

considered to be the Capability Service, and the underlying capability becomes merely an implementation detail of the façade service.

Capability Services do not typically directly manage application state; to make state changes in the application they utilize Entity Services. If a Capability Service does manage state, that state is typically transient and lasts for a duration of time that is shorter than the time needed to complete the business process that this Capability Service partakes in. For example, a Capability Service that provides package shipping price quotes might record the fact that requests for quotes were sent to the shipping providers out until the responses come back, thereafter erasing that record. In addition, a Capability Service that is implemented as a workflow will manage the durable, transient execution state for all the currently running instances of that workflow. While most of the capabilities are “stateless”, there are obviously capabilities such as event logging that naturally manage and encapsulate state.

Users of Capability Services may require a permission to be configured for them in order to use the service, be it at the application, user, or the application-user scope. Access to a Capability Service is typically granted at the application level. Per-user permissions are typically managed by the Process Services that make use of the Capability Services to simplify access management and prevent mid-process access failures.

Error compensation in Capability Services is limited to the scope of meeting the capability’s Service Level Expectation (SLE) and Service Level Agreements (SLA). For example, the Email Gateway Service may silently queue up an email notification for deferred delivery if there’s a problem with the mail service, and send it at a later time, when email connectivity is restored. A Shipping Service which usually compares the rates and delivery times of 4 vendors (e.g. FedEx, UPS, DHL, and a local in-town currier service) may compensate for a vendor’s unavailability by ignoring the failure and continuing with the comparison of the rates that it was able to secure as long as it received at least 2 quotes. These examples come to illustrate that failures may result in lower performance. This degradation can be expressed in terms of latency (as in the case of the Customer Emailing Service), the quality of the service (e.g. the Shipping Service would only be comparing the best of 2 quotes instead of 4), and many other aspects, and therefore needs to be described in the SLE and SLA for the service.

Capability Services may support distributed atomic transactions and/or the reservation pattern. Most of the Capability Services do not manage resources whose state needs to be managed using atomic transactions, but a Capability Service may flow an atomic transaction that it is included in to the Entity Services that it uses. Capability Services are also used to implement a reservation pattern over Entity Services that do not support that pattern, and to a much lesser extent over other Capability Services that do not support that pattern.

Capability Services can be developed and managed in-house, purchased from a third party and managed in-house, or “leased” from an external vendor and consumed as SaaS that is externally developed, maintained, and managed.

When developed in-house, Capability Services may be implemented using imperative code or a declarative workflow. If implemented as a workflow, a Capability Service is typically modeled as a short-running (atomic, non-episodic) business-activity. Long running business-activities, where things may fail or require compensation typically fall into the Process Service category.

A Capability Service is almost always used by multiple composite-applications, and is thus typically centrally managed.

Activity Services

Activity Services implement the business-level capabilities or some other action-centric business logic elements (“building blocks”) that are unique to a particular application. The main difference between Activity Services and Capability Services is the scope in which they are used. While Capability Services are an organizational resource, Activity Services are used in a much smaller scope, such as a single composite-application or a single solution (comprising of several applications). Over the course of time and with enough reuse across the organization, an Activity Service may evolve into a Capability Service.

Activity Services are typically created to facilitate the decomposition of a complicated process or to enable reuse of a particular unit-of-functionality in several places in a particular Process Service or even across different Process Services in the application. The forces driving for the creation of Activity Services can stem from a variety of sources, such as organizational forces, security requirements, regulatory requirements, etc. An example of an Activity Service create in a decomposition scenario is a Vacation Eligibility Confirmation Service that due to security requirements separates a particular part of a vacation authorization application’s behavior such that that part could run behind the safety of the HR department’s firewall and access the HR department’s protected databases to validate vacation eligibility. An example of an Activity Service used for sharing functionality would be a Blacklist Service that provides information on a customer’s blacklist status such that this information can be used by several Process Services within a solution.

Like Capability Services, Activity Services expose a service interface specific to the capability they represent. It is possible for an Activity Services to wrap an existing unit of functionality, especially in transition cases where an existing system with existing implemented functionality is being updated to or included in a SOA-based solution.

Like Capability Services, Activity Services do not typically directly manage application state, and if they do manage state that state is transient and exists for a period of time that is shorter than the

lifespan of the business process that the service partakes in. However, due to their slightly larger granularity and the cases where Activity Services are used to wrap an existing system, it is more likely than an Activity Services will manage and encapsulate application state.

Users of Activity Services may require a permission to be configured for them in order to use the service, be it at the application, user, or the application-user scope. Like in the case of Capability Services, access to an Activity Service is typically granted at the application level and managed for each user by the Process Services that are using the Activity Service.

Activity Services have the same characteristics for error compensation and transaction use as Capability Services.

Activity Services are typically developed and managed in-house, and may be implemented as imperative code or a declarative workflow. Like in the case of a Capability Service, if implemented as a workflow an Activity Service is typically modeled as a short-running business-activity.

Activity Services are typically used by a single application or solution and are therefore typically managed individually (for example, at a departmental level). If an Activity Service evolves into a Capability Service, the management of the service is typically transitions to a central management facility.

Process Services

Process Services tie together the data-centric and action-centric building blocks to implement the business processes of the organization. They compose the functionality offered by Activity Services, Capability Services, and Entity Services and tie them together with business logic that lives inside the Process Service to create the blueprint that defines the operation of the business. An example of a Process Service is a Purchase Order Processing service that receives a purchase order, verifies it, checks the Customer Blacklist Service to make sure that the customer is OK to work with, checks the customer's credit with the Credit Verification Service, adds the order to the order-list managed by the Orders (Entity) Service, reserves the goods from the Inventory (Entity) Service, secures the payment via the Payment Processing Service, confirms the reservation made with the Inventory (Entity) Service, schedules the shipment with the Shipping Service, notifies the customer of the successful completion of the order and the ETA of the goods via the Email Gateway Service, and finally marks the order as completed in the order-list.

Process Services may be composed into the workflows of other Process Services but will not be re-categorized as Capability or Activity Services due to their long-running nature.

Since Process Services implement the business processes of the organization, they are often fronted with a user interface that initiates, controls, and monitors the process. The service interface that these services expose is typically geared towards consumption by an end user

application, and provides the right level of granularity required to satisfy the use cases that the user facing front-end implements. Monitoring the business process will at times require a separate monitoring interface that exposes BAM information. For example, the Order Processing Service may report the number of pending, in-process, and completed orders, and some statistical information about them (median time spent processing and order, average order size, etc.).

Process Services typically manage the application state related to a particular process for the duration of that process. For example, the Purchase Order Processing service will manage the state of the order until it completes. In addition, a Process Service will maintain and track the current step in the business process. For example, a Process Service implemented as a workflow will hold the execution state for all the currently running workflow instances.

Users of Process Services may require a permission to be configured for them in order to use the service, be it at the application, user, or the application-user scope. Access to a Process Service is typically granted at the user level.

Process Services very rarely support participating in a distributed atomic transaction since they provide support for long-running business activities (a.k.a. long-running transactions) where error compensation happens at the business logic level and compensation may involve human workflows. Process Services may utilize distributed atomic transactions when calling into the services they use.

Process Services are typically developed and manages in-house since they capture the value-add essence of the organization, the “secret sauce” that defines the way in which the organization does its business. Process Services are designed to enable process agility (i.e. to be easily updatable) and the process that they implement is typically episodic in nature (i.e. the execution comprises of short bursts of activity spaced by long waits for external activities to complete). Therefore, Process Services are best implemented as declarative workflows implemented using an integration server (such as BizTalk Server) or a workflow framework (such as Windows Workflow Foundation).

Process Services are typically used by a single application and can therefore be managed individually (for example, at a departmental level). In some cases a reusable business process may become a commodity that can be offered or consumed as SaaS

When designing business software, we should remind ourselves that the objective is delivering agile systems in support of the business; not service orientation (SO). Rather, SO is the approach by which we an enable business and technology agility, and is not an end in itself. This must particularly be borne in mind with references to Web services. Achieving the agility that so often accompanies Web services is not just a consequence of adopting Web service protocols in the deployment of systems, but also of following good design principles. In this article, we consider

several principles of good service architecture and design from the perspective of their impact on agility and adaptability.

A Services Lifecycle

Now that we have examined the types of services that may exist within a SOA, a more holistic look at services is needed. A Services Lifecycle can be used to understand the activities, processes and resources necessary for designing, building, deploying and ultimately retiring the services that comprise a SOA.

A Service comes to life conceptually as a result of the rationalization of a business process and decomposition and mapping of that business process into the existing IT assets as wells the new IT assets to fill the gaps. The new IT assets once identified will be budgeted and planned for SDLC activities that result in deployable services (assuming that our goal is to create reusable IT assets). Following are various important activities that happen (not necessarily in this strict order) during the life time of a service from the service provider perspective:

Figure 4: *A Services Lifecycle*

Service Analysis

Service Analysis is the rationalization of business and technical capabilities with the express notion of enabling them via services. Other aspects such as SLAs, localization / globalization, and basic service contracts will be established for future use in the life cycle.

Service Development

Rationalization of contracts (XML Schemas) and designing new contracts will be one of the primary activities in this phase. Object libraries supporting the service implementation will be acquired or designed. Security policies, trust boundaries, authentication/authorization, data privacy, instrumentation, WSDL, etc. will be the outcome of this phase. Distributing WSDL or service consumer proxies will be strategized during this phase.
Services will be developed using the selected IDE, Web services stack and the language of choice.

Service Testing

Services will be unit, smoke, functional and load tested to ensure that all the service consumer scenarios and SLA ranges are met.

Service Provisioning

Service metadata as identified in the “Service Consumption” will be deployed into the directory. This will be associated with a deployment record into a repository that models deployment environment. Supported SLA policies will be an important metadata for successful operation of a service. Service gets a production endpoint in an appropriately designed production infrastructure. Support teams will be trained and appropriate processes for support among various roles (business versus IT) will be established. Access to service consoles and reports will be authorized to these roles.

Service Operation

This is the most important activity as the ROI will be realized through the operation of the services in production. The management infrastructure will do the following:

- Service Virtualization
- Service Metering (client usage metering and resource metering)
- Dynamic discovery of service endpoints
- Uptime and performance management
- Enforce security policies (authentication, authorization, data privacy, etc.)
- Enforce SLAs based on the provisioning relationship
- Generate business as well as technology alerts for a streamlined operation of the service
- Provide administrative interfaces for various roles
- Generate logs and audit trails for non-repudiation
- Dynamical provisioning (additional instances of the service as necessary)
- Monitor transactions and generate commit/rollback statistics

- Integrate well with the systems management tools
- Service, contract and metadata versioning
- Enforce service decommissioning policies
- Monetization hooks
- Reporting

Service Consumption

This activity is equally applicable to service consumers and providers as providers may consume services as well. During this activity, services will be discovered to understand the following:

- Service security policies
- Supported SLA policies
- Service semantics (from the lifecycle collateral attached to the service definition)
- Service dependencies
- Service provisioning (will be requested by the consumer)
- Pre and post-conditions for service invocation
- Service development schematics (proxies, samples, etc.)
- Service descriptor artifacts
- Service impact analysis
- Other documentation (machine readable as well as for human consumption)

During this activity, service consumers will be authorized to discover the service and its metadata. SLAs will be pruned to meet the desired level of availability based on the negotiated contract.

Service Change Management

Service like any IT application asset will go through several iterations during its lifetime. Service contracts will change, service security as well as SLA policies will change, the implementation will change, and the technology platform may change. Some of the above changes may be breaking changes. So, the management infrastructure has to be resilient for all the mutations by providing necessary deployment support across all the above changing dimensions.

Service Decommission

As a result of a change in the business strategy or as a result of better alternatives or as a result of waning consumer interest, a service may be decided for decommissioning. Management infrastructure should be able to enforce retirement policies by gracefully servicing the consumers until the last request.

SOA Scenarios

There are a number of business and technical scenarios for which SOA delivers a clear benefit. This section lists several of the most commonly used scenarios for SOA (this is not a comprehensive list).

Information Integration

The Information Integration scenario is sometimes referred to as “the single view of the customer problem”. The complete description of a customer might be spread across a dozen business applications and databases. This information is rarely completely in sync, and aggregating this information for optimal customer (or partner or employee) interaction is poorly supported. Information integration services are an effective means for both presenting your application portfolio with a unified view of these key entities, and for ensuring the consistency of the information across all of your back-end systems. Information integration projects can run from the tactical to the broadly strategic; incrementally re-engineering information access and management across the enterprise. This scenario is frequently associated with the following industry acronyms (each of which are sometimes used interchangeably):

- MDM: Master Data Management is an approach for providing and maintaining a consistent view of the organization’s core business entities (not just customers).
- EII: Enterprise Information Integration is broader than MDM, using data abstraction to address the challenges typically associated with data heterogeneity and context.
- CDI: Customer Data Integration is the combination of the technology, processes and services needed to create and maintain a complete view of the customer across multiple channels, business lines and enterprises. CDI is typically associated with CRM systems.

Chapter Four discusses the Information Integration scenario in greater detail.

Legacy Integration

The Legacy Integration scenario focuses on the tactical use of services to preserve existing investments in business applications, while extending the functionality of the capabilities upon which they deliver. For example, a service might add support to comply with new regulations in front of an existing ERP package. Applications would be engineered to exchange messages with the service, which would extract the compliance-relevant data and then communicate the request to the ERP package.

Process Governance

Process Governance is a far broader than either Information or Legacy Integration. In a Process Governance scenario, "header" elements are used to communicate key business metadata; from the turnaround time on customer requests to the identity of the approvers for specific business

decisions. This metadata is captured by a *utility service* (as discussed previously), for real-time and/or aggregated analysis. "Service native" processes would include this information in SOAP headers, while non-native applications would need to be re-engineered to transmit the metadata as a message to the governance server.

Consistent Access

Consistent Access is a more technical and subtly different scenario than any of the scenarios previously discussed. This scenario enables a services layer to ensure consistent enforcement of a variety of operational requirements when a diverse set of applications needs to connect to a critical back-end resource. By mandating that all access be routed through a service facade, an organization might enforce consistent access authorization, cost distribution and load management.

Resource Virtualization

A Resource Virtualization scenario can be utilized to help enforce loose coupling between resources and consumers, effectively insulating consumers from the implementation details of the targeted resources. Typical examples of Resource Virtualization may include:

- Context-sensitive and content-sensitive routing of requests, such as sending a real-estate inquiry to the agent in the specified geography who specializes in farm properties.
- Routing of requests to partitioned information stores (without requiring the requestor to understand partitioning schemes).
- Load balancing requests across available resources; from customer service representatives to streaming video feeds.

Process Externalization

Process Externalization scenarios utilize Web services to help securely negotiate common processes such as payroll processing, employee expense reimbursement, and logistical support. Cell phone service providers and Internet portals frequently use Web services to aggregate content while customer-facing organizations may use services to build composite offers (such as travel packages that include airfare and rental cars). The key to successful process externalization on today's technology stack is to manage your own expectations; compromise your requirements to the limits of the existing technologies so that you don't spend your profits or savings on building infrastructure services that you will replace in a few years' time.

Other Scenarios

There are far too many SOA scenarios to document them all. The scenarios discussed above represent some of the more common scenarios in which we have seen SOA succeed. Another common scenario is human interaction. How can services within a SOA interact with end users?

SOA and the End User

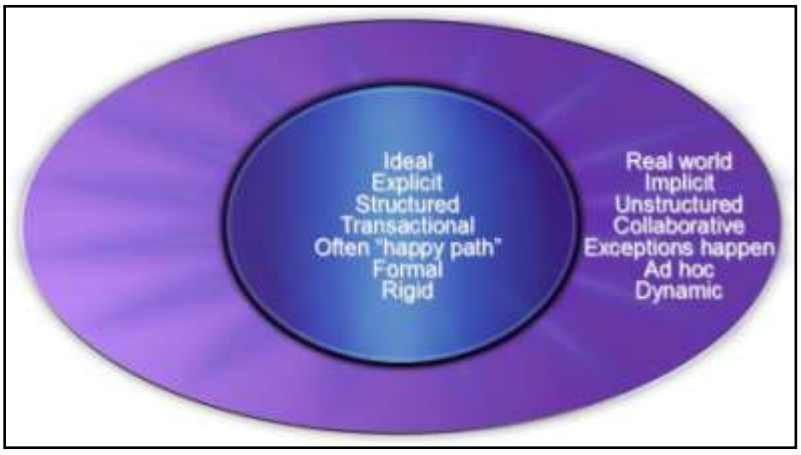


Figure 5: *Comparing Systems and Users*

Enterprise Application Integration (EAI) typically deals with system-to-system integration, ignoring computer-human interactions. System-to-system interactions tend to be very structured in terms of both process and data – for example, an Order to Cash process will use well-defined processes (Process PO) and business documents (Purchase Order). System-to-system interactions rarely mirror the real world. Processes tend to follow what is traditionally termed the "happy path," since exceptions are poorly or rarely handled effectively. How people work in the real world is much different – processes are ad-hoc and may change frequently within a given time period. The data we work with is equally unstructured since it may take the form of Office documents, videos, sound files, and other formats. The intersection of these two worlds represents the intersection of system-to-system interactions and human workflows (we will discuss this topic in greater detail in Chapter Three). Applications that effectively support these human workflows can be difficult to design and develop. The vast majority of end users today use Microsoft Office for email and information work. Microsoft Office represents the results of many years of research and investments geared towards effectively supporting human workflows (in terms of both data and processes). Microsoft Office 2007 has evolved to become a first-class integration platform, providing a familiar user experience for service consumption, enabling:

- Models for many business concepts including business entities, business events and event-driven business rules, task assignment and fulfillment, modeled workflows, and many others.
- An application lifecycle model with customization and versioning, self-contained zero-touch deployment, and management through the whole lifecycle.

- Tools support for creating models, composing applications and managing through the lifecycle, based upon Visual Studio Tools for Office (VSTO) and other platform tools.

Microsoft Office 2007 supports simple event handling support for common line of business (LOB) events, including workflow synchronization events. Since most requirements for integration of LOB content with Office revolve around surfacing business entities such as Account, Invoice, Quote and Product, Office Business Entities are exposed in a way that creates unique value for the knowledge worker. Entity relationship models are hardly new. The design of business applications has traditionally been based on business entities along with the business rules and logic associated with the data. Office Business Entities (OBEs) have a few special aspects:

- OBEs can be turned into Office native content, maintaining the link to the LOB sources while being subject to all the rules and features of the Office application to which the content is native. For instance a list of parts in a product entity can be inserted as a table in Word, while maintaining the ability to refresh and drill into the data in the LOB application source. This is a kind of self-describing smart tag in that it does not have to be recognized, it comes with its special behavior already built into the content.

- OBEs enable business entities to be treated as first-class citizens of the Office world. OBEs will be used offline, attached to e-mails, created, correlated, shared and edited in collaborative environments like SharePoint and Groove with user control over the commitment of their data to LOB data sources. Most of the knowledge work in a business happens in applications like Excel, Word, and Outlook, before, after and aside from creating the data of record—for instance a quote to be created or updated must be worked on by many people before being committed to the system of record. Business work is like a 3-dimensional creative and collaborative world from which the transactional applications capture a 2-dimensional projection of committed data. With the Office-friendly behavior of OBEs, the full lifecycle of business data usage can maintain coherence without resorting to awkward and error-prone cut-and-paste transitions between the document world and the business application world.

- OBEs are coupled with reusable UI experiences that can be used in rapid application development (RAD) to quickly produce context-driven business solutions. UI parts can be associated with OBE views, which can be used in a drag-and-drop design experiences to surface LOB data within Office. Relationships between UI Parts can be navigated dynamically using links, creating a web-like experience around business entities. The client runtime also provides a declarative programming model that allows user experience to be driven by standard Office context events (such as item open) with the experience tailored by parameters such as role and locale.

- The content of OBEs can be *bound* to the content of Office entities, without becoming a part of it. This is easiest to observe in Outlook items, where, for instance, a contact item in Outlook can be bound to a customer contact in a CRM system. Both the Outlook entity and the CRM entity exist independently, each with its own identity and behavior, but some of their properties (such as address) are conceptually shared by being bound to each other and synchronized automatically. Thus a hybrid Outlook/CRM entity is created conceptually with correlation and data synchronization rather than data sharing. This becomes visible in Outlook as extended data and user experience for such hybrid contacts; surfacing some CRM contact data and behavior as extensions of the Outlook contact UX. Hybrid entities create a deep but non-invasive association. The use of hybrid Office/LOB entities is most interesting for Outlook items today because Outlook items possess a firm identity which is needed for correlation with OBEs. As document sharing occurs in more controlled SharePoint/Groove environments as part of processes like document assembly, based for instance on Word templates such as “contract” or “RFP”, more Office entities will gain stable identities and become available for two-way correlation with OBEs.

LOB entities in many cases are fragmented into data silos and the data in these silos is often of questionable quality. OBEs can mask these problems while creating a rich user experience linked deeply to the real work context of the knowledge worker.

What are Composite Applications?

A composite application is a collection of software assets that have been assembled to provide a business capability. These assets are artifacts that can be deployed independently, enable composition, and leverage specific platform capabilities.

Figure 6: *High-level representation of a composite application*

In the past, an enterprise's software assets were usually a set of independent business applications that were monolithic and poorly-integrated with each other. However, to get the business benefits of composition, an enterprise must treat its software assets in a more granular manner, and different tiers of architecture will require different kinds of assets such as presentation assets, application assets, and data assets. For example, a Web service might be an application asset, an OLAP cube might be a data asset, and a particular data-entry screen might be a presentation asset.

An inventory of software assets by itself does not enable composite applications. This requires a platform with capabilities for composition—that is, a platform that provides the ability to deploy assets separately from each other, and in combination with each other. In other words, these assets must be *components*, and the platform must provide *containers*.

Containers provided by the platform need to be of different types, which map to the different tiers in the architecture. Enterprise architectures are usually decomposed into three tiers: presentation, application (or business logic), and data. So the platform needs to provide containers for these. However the 3-tier architecture assumes structured business processes and data, where all requirements are made known during the process of designing and building the system. By their very nature, composite applications presume that composition of solutions can occur after assets have been built and deployed – and so need to explicitly account for people-to-people interactions between information workers that are essential to get any business process complete. Usually these interactions are not captured by structured processes, or traditional business applications, and therefore it is critical to add a fourth tier - the productivity tier – to account for these human interactions. This is shown in Figure 7.

Figure 7: *The four tiers of a composite application*

Traditional discussions around the architecture of business applications tend to focus on the application tier as being the connection between people and data. Typically, however, the application tier contains structured business logic; and this holds for discussions around Service Oriented Architectures (SOAs), Enterprise Service Buses (ESBs), Service Component Architectures (SCAs), or most other architectural perspectives in the industry today – including first-generation discussions around composite applications. However, building a composite application requires a mindset that not only is the productivity tier a critical element of the stack, but also contains the most business value.

To expand on the comparison between composite applications and SOA, both of them target flexibility and modularization. However, SOA provides flexibility at just one tier: the structured business logic in the middle tier. Composite applications target flexibility at all four tiers. That said, a composite application is a great way to surface information out of an SOA, and having line-of-business (LOB) applications exposed as services makes it easier to build support for cross-functional processes into a composite application.

Therefore to design a composite application, a solutions architect must:

- **Choose a composition stack** –Pick one or more containers from each tier, and a set of components types that are deployable into those containers.
- **Choose components** – Define the repository of assets that must be built from this set of component types, based on business needs.

- **Specify the composite application** – Define the ways in which those assets will be connected, to provide a particular cross-functional process. The platform should enable these connections to be loosely-coupled.

Then after deployment, users will have the opportunity to personalize both assets and connections, as the composition stack should enable this through loose coupling and extensibility mechanisms.

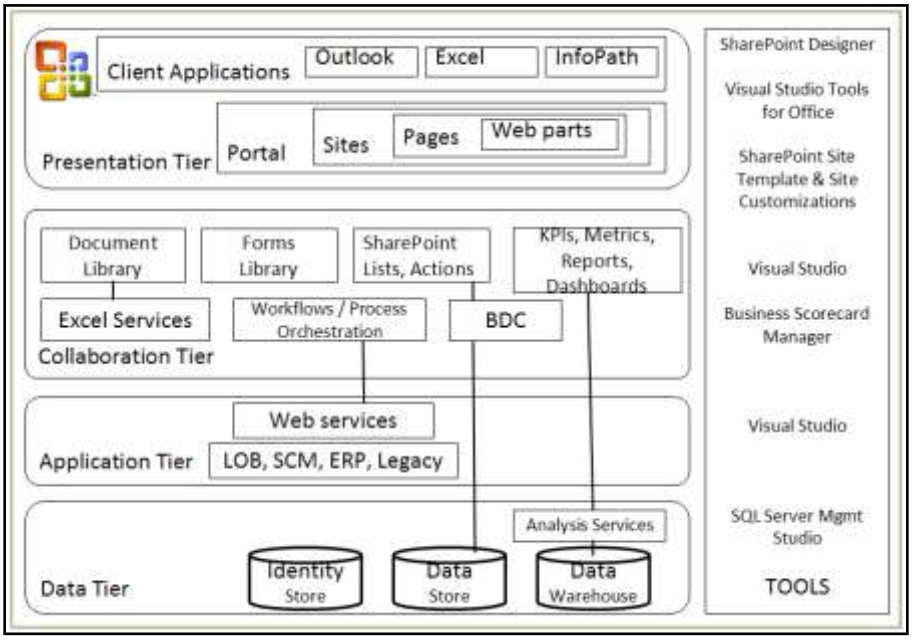


Figure 8: *A Composite Application Architecture*

What does a Composite Application look like?

A figurative representation of a composite application is shown in Figure 8, which shows a very abstract representation of an enterprise solution, deconstructed along the lines of Figure 7.

At the top are information workers, who access business information and documents through portals that are role specific views into the enterprise. They create specific documents during the course of business activities, and these activities are part of larger business processes. These processes coordinate the activities of people and systems. The activities of systems are controlled through process specific business rules that invoke back end LOB applications and resources through service interfaces. The activities of people plug into the process through events that are raised when documents specific to the process are created, or modified. Then business rules are applied to the content of those documents, to extract information, transform it, and transfer it to the next stage of the process.

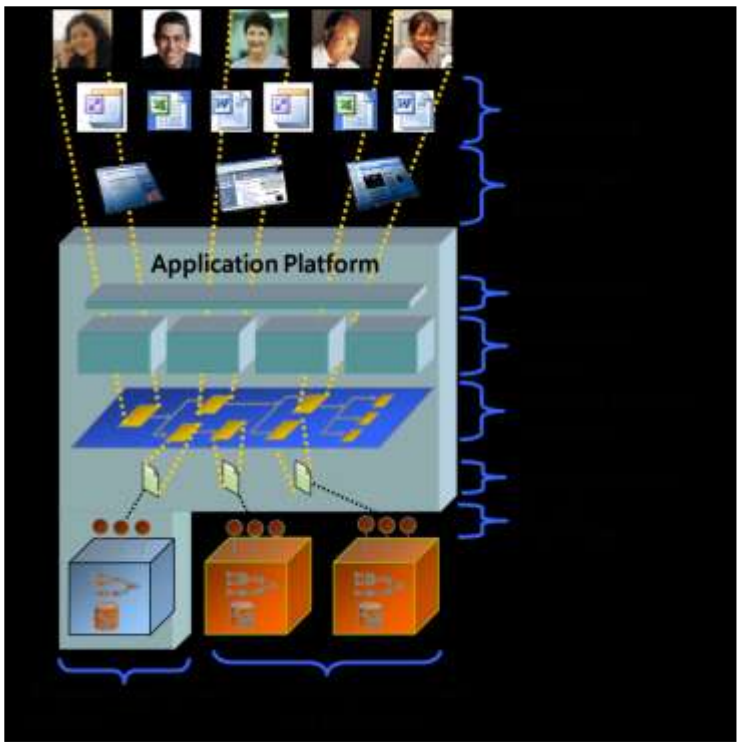


Figure 9: *Deconstructing an enterprise application*

Today most line-of-business applications (LOB) are a collection of resources, hard-coded business processes, and inflexible user interfaces. However based on the previous section, it is clear that enterprise solutions need to be broken down into a collection of granular assets that can be assembled into composite applications. A high-level approach for doing this to any business process is listed below:

1. Decompose the solution for a business process into software assets corresponding to the elements shown in Table 1 below.
2. Package all assets corresponding to a given business process into a “process pack” for redistribution and deployment. This would contain metadata and software components, and solution templates that combine them. The process pack would also contain service interface definitions that would enable connections to other IT systems. These connections would be enabled by implementing the service interfaces, for example. to connect to LOB applications and data. The goal is to be able to easily layer a standardized business process onto any heterogeneous IT landscape.
3. Deploy the process pack onto a platform that provides containers for the types of assets that the solution has been decomposed into. The platform should provide capabilities for rapid customization, personalization, reconfiguration, and assembly of assets.
4. Connect the assets within the process pack, to existing LOB systems, and other enterprise resources by implementing the services interfaces. These connections could be made using Web services technologies, other kinds of custom adapters, or potentially even Internet protocols like RSS.

• Documents • Workflows • Business activities • Business rules • Schemas • Interfaces to connect to back end systems (Web service APIs)	• UI Screens • Data connections • Authorizations • Reports • Metrics

Table 1: *List of application assets for composition*

Expected Benefits of Composition, and How to Achieve Them

Deployment of enterprise applications should be tied to business benefits in the Triple-A sense (agility, adaptability, alignment). These benefits need to be demonstrated from two perspectives:

- **The Solution Provider Perspective (or Development Perspective) –** This is the perspective of the organization that builds an enterprise application. This might be an Independent Software Vendor (ISV), or a Systems Integrator (SI), or even an in-house IT department. The solution provider perspective is concerned primarily with benefits gained in activities relating to designing, implementing, and deploying enterprise applications.
- **The Solution Consumer Perspective (or User Perspective) –** This is the perspective of the organization that uses an enterprise application. Typically this is the business unit that commissioned the enterprise application. The solution consumer perspective is concerned primarily with benefits gained by the business after the solution has gone into production

The benefits of composition that can be reasonably expected in each of these two perspectives are listed here, along with some high-level best practices to achieve these expected benefits.

Conclusion

In this chapter we examined the concept of SOA from the perspective of services: architectural and organizational maturity, service types, service lifecycles, scenarios and the role of users and composite applications within a SOA initiative.

Service oriented architecture (SOA) is a design approach to organizing existing IT assets such that the heterogeneous array of distributed, complex systems and applications can be transformed into a network of integrated, simplified and highly flexible resources. A well-executed SOA project aligns IT resources more directly with business goals, helping organizations to build stronger connections with customers and suppliers, providing more accurate and more readily available business intelligence with which to make better decisions, and helping businesses

streamline business processes and information sharing for improved employee productivity. The net result is an increase in organizational agility.

In an SOA, the concept of applications will still exist, especially when one considers an enterprise's IT investments. However, the concept of one vendor supplying a complete "SOA solution" with a monolithic set of products is being replaced with a "best-of-breed" approach, enabling customers to adopt a capabilities-based approach to implementing their architectural requirements.

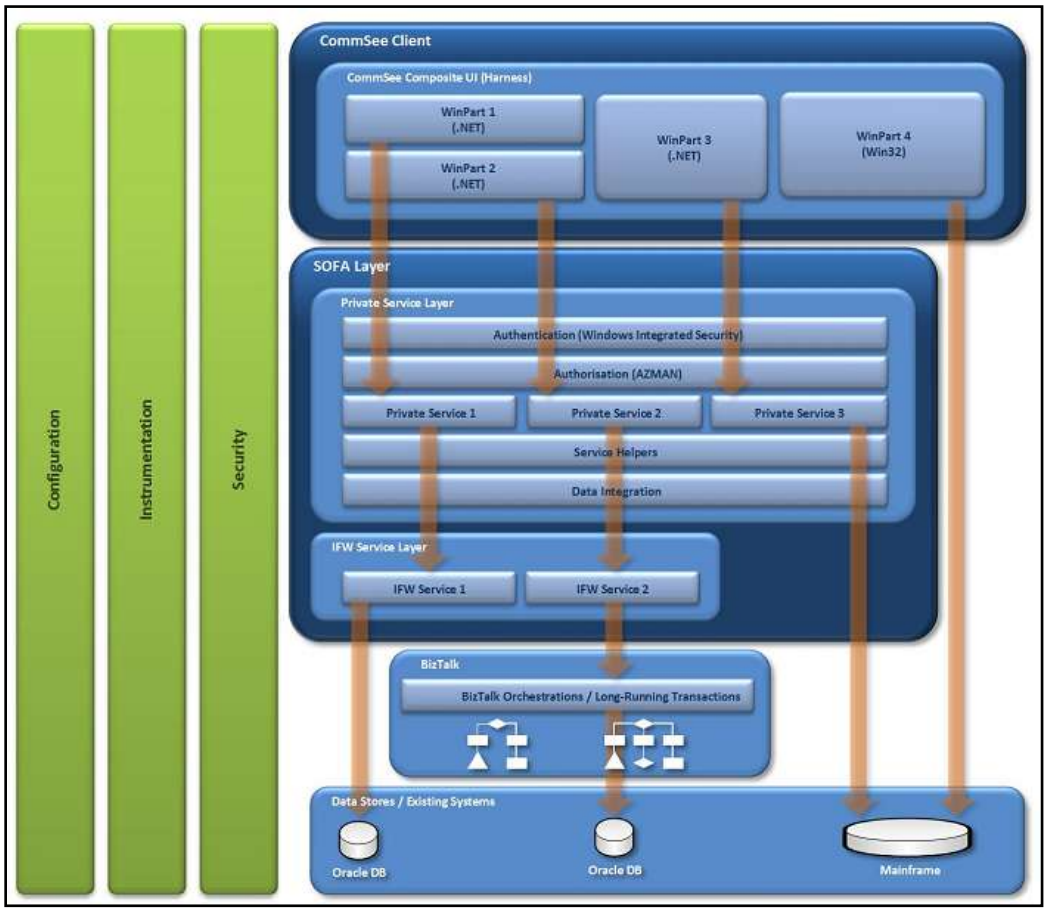
Organizations should remain focused on solving their business problems and avoid being distracted by integration trends and buzzwords. SOA should be a means for making the business more agile, not the end goal. Designing and implementing SOA should be an incremental process with rapid deployments and ROI realization. SOA should not be a top-down, multi-year "boil-the-ocean" effort – these types of projects rarely succeed because they are unable to keep up with the shifting needs of the organization.

Users will also undergo a transformation in the way they work with applications. Depending on the type of application, a user could either be exposed to specific tasks of a process, e.g. working in the context of a document workflow in Microsoft Office SharePoint Server, or, an application might encapsulate a business process internally and let a user start the process, but not interact with it during its execution.

Chapter Three provides a more detailed discussion of the **Workflow and Process** recurring architectural capability.

SOA Case Study: Commonwealth Bank of Australia

This case study describes how the Commonwealth Bank of Australia designed, developed, and implemented its *CommSee* application – a relationship banking solution, custom-built by the Commonwealth Bank of Australia using Microsoft® .NET technologies. This solution was developed as a Microsoft® Windows® Forms–based Smart Client that consumes standards-based .NET Web services. The Web services are responsible for orchestrating data from a variety of back-end data sources including mainframes, databases, and various backend legacy systems. At the time of this writing, CommSee has been successfully deployed to 30,000 users at more than 1,700 sites across Australia. After considering the business needs that CommSee was designed to address, this case study will examine the solution architecture, technical project details, and best practices employed in this major software development effort.



CBA Architectural Overview

The entire Case Study is available online at http://msdn2.microsoft.com/en-us/library/bb190159.aspx. .

See other SOA case studies at
http://www.microsoft.com/casestudies/search.aspx?Keywords=SOA.

References:

1. "Enabling the Service-Oriented Enterprise", Architecture Journal, April 2006. Available at http://msdn2.microsoft.com/en-us/library/bb245664.aspx
2. "Ontology and Taxonomy of Services in a Service-Oriented Architecture", Architecture Journal, April 2007. Available at http://msdn2.microsoft.com/en-us/library/bb491121.aspx
3. "Office Business Applications: Building Composite Applications Using the Microsoft Platform", December 2006. Available at http://msdn2.microsoft.com/en-us/library/bb220800.aspx
4. "Service Oriented Infrastructure", Mark Baciak. Available at http://msdn2.microsoft.com/en-us/architecture/aa973773.aspx
5. "Service Orientation and Its Role in Your Connected Systems Strategy". Available at http://msdn.microsoft.com/architecture/solutions_architecture/service_orientation/default.aspx?pull=/library/en-us/dnbda/html/srorientwp.asp

Chapter 3: Workflow and Process

"The art of progress is to preserve order amid change and to preserve change amid order."

- Alfred North Whitehead
Mathematician, philosopher
(1861-1947)

Reader ROI

Readers of this chapter will build upon the concepts introduced in previous chapters, specifically focusing on the Workflow and Process architectural capability.

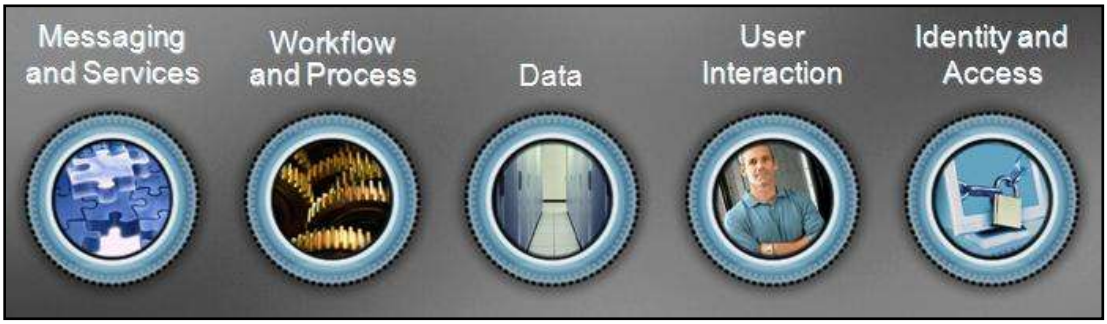


Figure 1: *Recurring Architectural Capabilities*

The Workflow and Process architectural capability focuses on the concept of *workflow* and how workflow can be used to orchestrate and aggregate granular services into larger constructs that we call *processes*.

Topics discussed in this chapter include:

- Explaining the concepts of workflow
- Clarifying the relationship of BPM, SOA, orchestration and choreography to workflow
- Describing the capabilities necessary for supporting workflow
- The benefits of adopting a model-based approach to workflow

Acknowledgements

This chapter consists of work from Dave Green (concepts, semantics, value and capabilities, Windows Workflow Foundation) and John Evdemon (workflow concepts, terms and manifesto).

Understanding Workflow

What is Workflow?

The concept of workflow is not new. Workflow technologies first emerged in the mid-1970s with simple office automation prototypes at Xerox Parc and the University of Pennsylvania's Wharton School of Business. Interest in workflow and office automation began to wane in the early 1990s until the book *Reengineering the Corporation* reignited interest in workflow and business processes. The reengineering trend of the 1990s gave us several books and methodologies for process analysis – unfortunately the technologies such as CASE and their ilk were immature and required significant manual intervention, exposing projects and executive stakeholders to significant levels of risk. Even worse, other "workflow products" were thinly disguised efforts to sell hardware such as scanners, printers, and other peripherals. Clearly the term "workflow" was being abused to take advantage of confusion in the market

So what is workflow? Workflow is fundamentally about the organization of work. It is a set of activities that coordinate people and / or software. Communicating this organization to humans and automated processes is the value-add that workflow provides to our solutions. Workflows are fractal. This means a workflow may consist of other workflows (each of which may consist of aggregated services). The workflow model encourages reuse and agility, leading to more flexible business processes.

Workflow Terminology

As stated above, workflow is about the organization of work. This is a very broad definition and may be interpreted in a number of ways. To avoid confusion we will identify several terms that are commonly associated with the concept of workflow:

- Business Process Management (BPM): BPM is a business philosophy that views processes as a set of competitive assets to be managed. The processes that business seeks to manage are largely composed upon and executed by a SOA infrastructure. SOA and BPM have several objectives in common:
    - Both are focused on making the organization more agile. SOA focuses on providing a loosely coupled infrastructure for service enablement while BPM focuses on effectively designing and managing loosely coupled business processes (which are aggregations of granular services).
    - Both seek to make the business proactive instead of reactive. A proactive business is able to spot market patterns, trends and competitive threats before they can negatively impact the organization. Simply identifying a threat or opportunity is not enough – the organization must also be able to proactively

modify its existing processes to take advantage of the opportunity (or avoid the threat). Achieving this level of agility requires loosely-coupled business processes atop by an SOA infrastructure capable of supporting rapid reconfigurations to meet the needs of the organization.

    - Both are focused on process-centric instead of functional capabilities. Processes can no longer be constrained by a limited set functional capabilities – an SOA infrastructure is a fabric of configurable, composable services that can be assembled/disassembled or reconfigured to meet the shifting business needs of the organization.
- Orchestration: Workflows and orchestrations are effectively the same thing since each is designed to compose and execute a series of tasks. Orchestrations require a "conductor" that is always in charge of the execution. The conductor is typically manifested as an integration server (such as BizTalk Server) which monitors execution, raises events, creates execution logs and performs various other duties to ensure the process executes as expected.
- Choreography: Choreography is a set of peer to peer relationships between individual participants. There is no "conductor" in choreography. Participants interact with one another based upon a set of agreed-upon principles or contracts. For example, a Supplier might enter into an agreement with a Retailer to ship a specific set of goods and services within a given timeframe. This means the Supplier and Retailer must agree upon business events, business documents, service level agreements (SLAs) and penalties if the SLAs are not met. The Supplier may in turn enter into another agreement with a Manufacturer to supply the raw materials that the Supplier needs to meet the Retailer's demands. A choreography is usually implemented by "splitting" it up into multiple orchestrations (one for each participant involved in the choreography).

Discussions about workflow frequently mention one of the terms explained above. Orchestration is the only term that is comparable to the concept of workflow.

Why Workflow?

The problems which a workflow approach has been applied often display three characteristics: The key business value delivered is *coordination*, for instance, in organizing multiple contributions to the preparation of a quote or driving a document review. Each instance of the business process concerned is of *long duration*, measured often in days, weeks, or months, rather than minutes. The business process has *human* participants, who usually contribute most of the work product.

However, only a small proportion of the business problems with these characteristics are solved using a workflow approach. Most commonly, the business process is not recorded as machine-

readable information at all. Rather, each of the humans participating in the business process interacts with business systems that are not aware of the semantics of the process as a whole, such as a customer information system, and with other human participants through content-neutral communications channels such as e-mail. Each human participant uses a mental model of their part in the overall business process to determine their behavior.

Three key benefits that a workflow model can bring are insight, monitoring, and optimization. A set of related workflow models can be used to gain *insight* into the flow of work through an organization. For *monitoring*, knowing which individuals are contributing work to which business process is very useful when trying to understand costs and workloads. For *optimization*, having a model of the work being undertaken, and being able to use the model to interpret behavior, together make it possible to reason about how to optimize the business process.

The concept of a model-driven approach is attractive it is not new – developers have been using UML and other modeling techniques for many years prior to the maturation of workflow development tools. Workflow provides either a sequential or state machine model that is used to develop, manage and run the workflow itself. Code-only approaches may take advantage of models to lesser success – for example we can parse C# code into an abstract CodeDOM model – but the usefulness of such a model is limited.

A Workflow Model

Given these compelling benefits, why haven't workflow models been used more widely? The most likely answer is that the cost of using them has been too high. These costs include *product* costs, that is, the direct cost of purchasing a workflow product; *integration* costs, where processes modeled as workflows need to be integrated as part of a larger business system; and *standardization* costs, where it is difficult for a large organization to standardize on a single workflow technology. Variations in workflow products also mean that skills and model portability are issues.

Let's look at the possibility of addressing these blocking issues by building applications on a workflow platform that is low cost, ubiquitous, uniform, and easily integrated in applications. To be clear, the idea is not to replace workflow products. Rather, the hypothesis is that it is useful to factor out support for some core workflow concepts into a platform on which both workflow products and other applications can be built (see Figure 1, below).

A workflow is a *model*, which means it is a machine-readable description of business behavior that is not code. The meaning and benefits of this concept in the context of the value of a workflow platform will be discussed later.

A workflow model describes an organization of *work units*. For instance, suppose that a document review process specifies that Joe writes the document and then Fred reviews it. Here, the work units are first writing and second reviewing the document, and the organization is that one task must follow the other. This concept is not a radical idea. Code that makes successive calls to two subroutines is a valid example of the concept. The interest lies rather in the forms that this organization takes.

To test the workflow platform hypothesis, we will consider a range of real-world applications and explore the characteristics that a workflow platform should have if it is to prove useful.

A *document review* process takes as an input parameter a set of [reviewer, role] pairs that describe which humans are involved in the workflow in which roles. Possible values for the role are required, optional, final approver, and owner. The review process then proceeds until all reviewers have performed their assigned roles and notifies the owner of the outcome.

Here, the work items are the document reviews organized by the review process. There are three interesting characteristics to call out, namely, multiple points of interaction, human and automated activity, and the need to handle dynamic change.

Workflow Contracts

The workflow has multiple points of interaction, or contracts. First, there is a contract with a reviewer. This contract involves asking a reviewer to review a document, accepting the verdict and any review comments, and also telling a reviewer that his or her input is no longer required (if the review is canceled, or perhaps if enough reviewers have voted yes). The contract might also allow a reviewer to delegate a review. Then there is a second contract with the final approver, which is a specialization of the reviewer contract. Third, there is a contract with the owner of the review that allows the owner to cancel the review and be notified of the outcome of the review. Finally, there is a contract with the initiator of the review process, who instantiates the review and supplies the required parameters.

It is typical of workflows that they connect multiple parties through a variety of contracts (see Figure 2). The document review workflow is essentially a coordinator, initiated through one contract that is coordinating a variety of participants through one or more additional contracts.

The *document review* workflow drives human activity. However, it might also drive automated activities, such as storing versions of the document in a repository as the review progresses. From the point of view of the workflow, there is no essential difference. A workflow can be thought of as communicating, in general, with services through contracts. One special case of a service is another workflow. Another special case is a human. In many ways, a human is the original asynchronous service: one never knows when or if it is going to respond.

A characteristic of this type of workflow is that the participants will ask for changes to the workflow as it executes. For example, a reviewer might delegate a review task to a colleague or share the work involved in a review task with a subordinate.

There are two ways of addressing this requirement. One is to build an understanding of all the possible changes into the workflow. Then, a delegation request becomes just another function of the contract between the workflow and the reviewer. The other possibility is to see change as something separate from the workflow, where change is implemented as an external function that changes the workflow model. In this approach, the result of delegation is a new workflow model identical to one in which the review task was assigned to the delegate from the beginning.

Requesting an additional approval step would add a new approval task to the workflow model, which might well have contained no approval steps at all in its original form. The workflow no longer has to anticipate all possible modifications; at the most it will be concerned with restricting the areas of the model that are subject to change.

Both approaches are useful. Building understanding into a workflow is simple to model and understand. Generalizing operations is more complex to model, but more powerful and agile.

In an extreme but interesting case of the latter approach, the workflow begins execution with little or no content, and the required behavior is added dynamically by the participants in the workflow. Here, the available operations for modifying the workflow become a vocabulary that a user can use to construct the desired behavior as the workflow progresses.

Problem-Resolution Collaboration

To look at a specific example of a *problem-resolution collaboration* application, consider an inventory shortfall. An assembly line is making a gadget, and the computer indicated that there were enough widgets in stock for the purpose. However, when the stockroom manager went to fetch the widgets for delivery to the assembly line, a shortfall of 10 widgets was discovered.

Collaboration among the stockroom manager, the customer's account manager, the supplies department, and the production manager is required to resolve the problem. Each role in the collaboration may take characteristic actions. The supplies department could order more widgets, perhaps using a different supplier or paying an existing supplier more money for faster delivery. The account manager could go to the customer and request deferred delivery or split the delivery into two parts and bear the extra shipping cost. The production manager could divert assembled gadgets from an order for another customer. The stockroom manager could search his or her physical stock in an attempt to find the missing widgets. Any given action might be performed multiple times.

One obvious constraint is that the collaboration is not completed until the shortfall is resolved by some combination of the previous actions. There will often also be business constraints. For

instance, there might be a rule that says deferral of delivery to gold customers is never permitted. Also, the actions will affect each other. For instance, there might be a policy that states that total added cost from corrective action may not exceed 5 percent of original factory cost. Thus, placing an order for accelerated supplies at a higher price might prevent a shipment from being split.

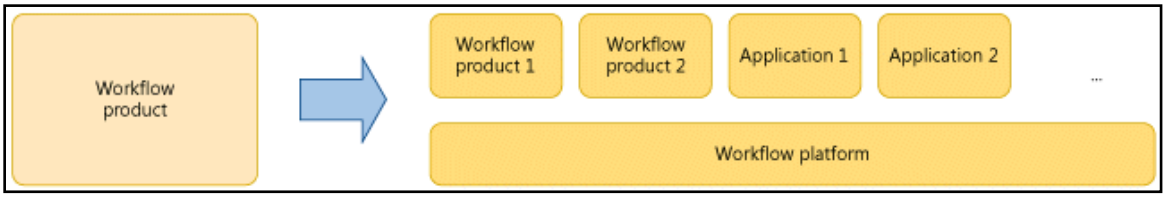


Figure 9: *A monolithic workflow stack*

In this case the work items are the actions that the various participants can take as they seek to resolve the inventory shortfall. The organization, however, is not the same as that required in document review. The participants are not dictated to; instead, they choose which actions to perform and when to perform them. However, these choices are constrained by the organization of the workflow, which has two aspects: 1) The actions are focused on achieving a goal; in this case, resolving the inventory shortfall. A bounded collaboration space is created when the problem resolution starts, and is not closed until the goal has been reached. 2) The participants are not free to perform arbitrary actions. Instead, the available actions are determined by the role the participant is performing and the state of the collaboration. The set of actions available is determined by policies related to the goal and global policies such as the restriction on shorting gold customers. The actions available vary as the collaboration progresses.

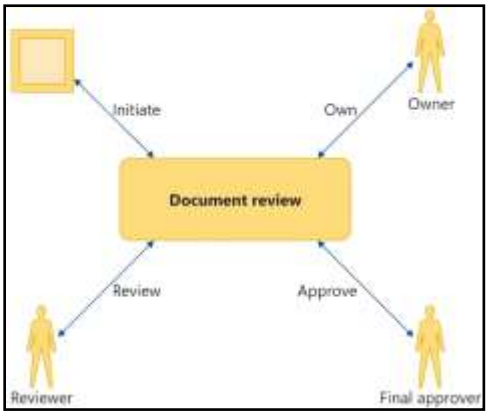


Figure 10: *A contract diagram for the document review application*

The experience of the participant is no longer that of performing assigned tasks. Instead, a participant queries for the actions currently available to him or her, performs none or one of these actions, and then repeats the cycle.

The main new requirement here, therefore, is for a form of organization of work items that is essentially data state and goal driven. There is also a requirement to support a query/act-style of contract with a workflow participant.

Scripted Operations

Scripted operations are simply a set of operations that are composed using a script. An example might be a desktop tool that allows a user to define and execute a series of common tasks, such as copying files and annotating them.

It would be unusual to consider using a typical workflow product for this purpose. However, it does fit the workflow platform pattern of a set of work units organized by a model. In this case the model is a sequence, perhaps with support for looping and conditional execution. Therefore, if a workflow platform were sufficiently low cost and ubiquitous, it would be possible to consider applying it to this sort of problem. Would doing so add any value?

One feature of scripted operations that is not addressed by their typical implementations today is the question of data flow. It is common for the data required by one operation to be the output of some previous operation, but this information is not typically modeled in the script. Thus, a user assembling tasks using a desktop tool might not be told when creating a script that the prerequisite data for a task hasn't been supplied, and would only discover the error when running the script. A workflow model that can describe these data dependencies would add clear value for script authors.

One approach is simply to include data flow constructs in the workflow model. It is highly arguable that the basic workflow model needs to include basic structural features such as sequences, conditions, and loops; but it is not clear that data flow is sufficiently universal to be represented by first-class elements of the model.

An alternative approach is to layer support for data flow on top of an extensible, basic workflow. A workflow model that can be enriched with abstractions appropriate to a variety of problem domains fits well with the notion of a workflow platform. This approach avoids both the complexity created by including in the base model a large variety of semantic constructs specialized for different problems and also the limitations imposed by limiting the workflow model to a fixed set of constructs.

Now let's look at a *guided user* application. One example is an interactive voice response (IVR) system, and another is a call center system guiding telephone operators through support or sales scripts. The essence of these applications is to guide users through the series of operations needed to achieve their goal. The organization of these operations is typically used to drive the presentation to the user, whether this is generated speech or a set of enabled and disabled command buttons on a form.

A characteristic of this type of application is that the workflow is the most frequently changed part of the application. Also, the business sponsors of the system are often heavily involved in specifying the changes, making it important to provide a way for IT staff and business personnel to communicate clearly and efficiently about the changes. A workflow model that expresses the core business purpose of the application, stripped of any irrelevant technical material, is an effective way to achieve this communication.

These applications also require flexibility within the workflow structure. In an IVR application the user will typically be heavily constrained, moving through a hierarchically structured set of menus. However, there will also be escape commands—for example, "return to root menu" or "skip out of current subtree."

A call center application will have more flexibility than an IVR application, changing the options offered to the user in response to the state of an order or in response to the input from a customer, such as skipping sales prompts if the customer starts to react negatively.

This sort of application requires support for a mix of organizations of work items, combining sequences, loops, and conditions with jumps from one state to another, and also the kind of data-driven behavior seen in problem-resolution collaboration.

Rule and Policy

As discussed previously, one way in which the workflow approach can deliver value is by isolating the focus of change in an application. Often, this focus is on the way in which the work items are structured, but in some applications the focus of change is on expressions tied to a relatively slow-changing structure.

An example of this focus is an insurance policy quotation system, where a set of frequently changing calculations is used to drive decision making in the quotation process. The requirement is for the workflow to model these expressions, which has two key benefits: First, the testing and deployment costs are much lower than those that would typically be incurred if the expressions were written as code, since the model provides a strong sandbox restricting the scope of possible changes. Second, the changes can be made by personnel who understand the business significance of the expressions but do not have the skills to understand the technical code in which expressions written as code would inevitably need to be embedded.

The Model-View-Controller (MVC) pattern often is used to wire a UI to an underlying object model (see Figure 3). The *model* represents the behavior of the system, independent of any particular UI representation. The *controller* is a part of the UI layer that is used to map the events generated by the UI into the method invocations required to drive the model. The UI itself is thus not polluted by any assumptions about the underlying model.

The workflows considered so far, viewed from this standpoint, all fall into the category of Models in the MVC sense. However, the controller can also be seen as a workflow. The work items it organizes are the methods provided by Model objects. The controller also interacts with the UI and the model through well-defined contracts. A model of this kind is often termed a *page flow*.

As with scripted operations, page flow would not today be implemented using a typical workflow product. There are two reasons to consider building a page flow using a workflow platform. First, a model readily can be represented visually, helping developers and analysts to express and communicate the required behavior. Second, if the page flow is frequently changing, then the abstraction of the page flow as a model improves agility.

There are two main requirements if this problem is to be addressed using a workflow platform. The workflow runtime must be lightweight, since a page flow may be running within a small application on a desktop, and the contracts supported must include the event-based contract characteristic of UIs, as well as the object-method contracts exposed by the Model.

Now let's look at a *test record/replay* application example. The intent of this final example is to test the limits of the applicability of the workflow platform hypothesis.

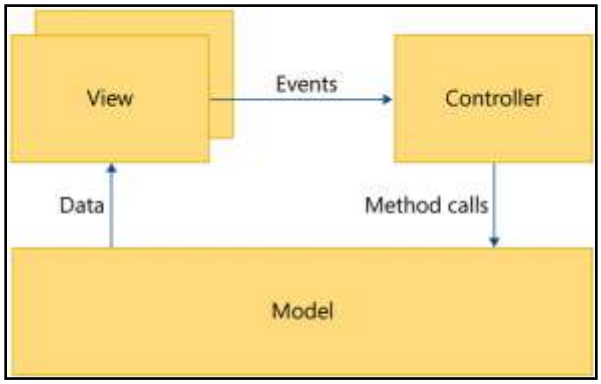


Figure 11: *An MVC application*

The application here is a tool for testing applications built as a set of services. The tool uses an interception mechanism to record all the interaction between services that occur during manual performance of a test case for the application. This recording can then be replayed. During replay, externally sourced messages are generated without manual intervention, and messages between the set of services that make up the application are checked for sequence and content against the original recording.

The workflow is the test case, organizing the work units that are the participating services. The workflow is both active, in that it simulates the behavior of externally sourced messages, and passive, in that it monitors the interactions between services.

A unique feature of this application is that the workflow is written, not by a developer or a user, but by a program, as part of the act of recording a test case. Workflow model creation must be fully programmable. There are also requirements for extensibility and dynamic update.

Extensibility is required because the structural semantics are rich. For instance, just because two messages arrived at a service one after the other in the recording, there is no necessary implication that this order needs to be preserved in a replay. If there is no causal dependency between the messages, then a replay that reverses the order of the messages is correct. So the semantics of sequence in the model used to record the test cases need to include a notion of causality, which is not likely to be a feature of the core workflow model of sequence.

Dynamic update is required because human interaction with the model will occur during replay. Discrepancies discovered during replay between recorded and observed behavior are popped up to a tester. If the discrepancy is because a message includes a timestamp that varies from run to run, then the tester would update the model to mark the field "don't care." If the discrepancy occurs in a regression test because the software has changed, then the tester might approve the change and update the test to expect the new behavior in all subsequent runs.

Workflow Platform Value

A workflow platform does not, by definition, have the full set of features offered by today's typical workflow products. Rather, the workflow platform considered here focuses on supporting the concept of a workflow as a model of the organization of work items. We have seen that this idea of an organization of work items is indeed applicable across a broad range of applications, but the fact that a workflow platform can be used doesn't mean that it should be used.

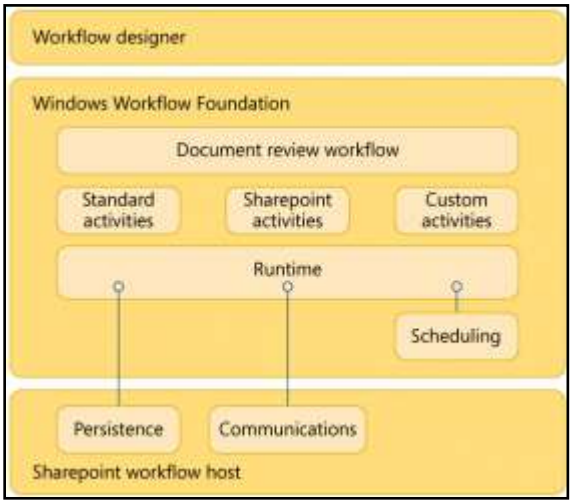


Figure 12: *The document review implementation schematic*

Two questions must be asked: What additional value is derived from the workflow platform approach? And, is this approach practical? This value of the workflow platform approach must

come from the expression of the organization of work as a model, which we'll discuss later. Let's summarize the characteristics that a practical and effective workflow platform must display.

To demonstrate how a model differs from code, this code snippet is a valid workflow by the definition used here, that is, an organization of work units.

```
public void HandleLoanRequest (string customerID, Application app)
{
   if (CheckCredit(customerId, app.Amount))
   {
     MakeOffer (customerId, app);
   }
}
```

And, in a sense, it is a model. It is possible to parse this code and build a CodeDOM tree that represents it.

However, the semantics of the resulting model are so general as to be opaque. It is possible to tell that the code contains function invocations, but it isn't too easy to distinguish a function that represents the invocation of a work item from a function that converts integers to strings. A workflow model explicitly distinguishes these ideas. Typically, a specialized model element is used to represent the invocation of a work item, and conversion functions cannot be expressed directly in the model at all. A workflow model, then, is one in which its graph is built from elements that are meaningful in the workflow domain. The semantic richness of such a model can be exploited in several ways.

Visualization. Visual representation of the model—typically in graphical form—is useful for the developer, during both development and maintenance, and also for workflow users who want to know why they have been assigned a given task or the IT operations worker who wants to understand what a misbehaving application should be doing.

Insight. The workflow model is amenable to programmatic access for a variety of purposes. An example is static analysis to determine dependencies and flow of work across a set of cooperating workflows or using the model to drive a simulation that predicts the workloads that will be generated by a new version of a process.

Expressiveness. The specialization of the workflow model to the workflow domain means that characteristic problems can be expressed more quickly and compactly. It is a domain-specific language (DSL), specialized to support characteristic problems. Consider a document review process where three positive votes out of five reviews mean that the document is good, and any outstanding reviews can be canceled. This process is quite difficult to code, but a workflow model can supply out-of-the-box constructions that address such problems.

More Semantic Exploitation

As we have seen in the scripted operations application discussion, extending the workflow model to further specialize the out-of-the-box model language is a very powerful technique for delivered additional value. An example is the creation of a language intended for end users, as in the document review conducted using an improvised definition of the review process that was discussed previously.

Execution. The specialization of the model makes it possible to add run-time support for common problems. A good example is long-running state. Of the applications discussed here, management of long-running state is required for the document review process, problem-resolution collaboration, and guided user applications. The workflow platform runtime can solve such difficult problems once, using simple expressive model elements to control a common capability and freeing up the developer to focus on the business problem.

Monitoring. The existence of a model makes it possible to produce an event stream with a meaningful semantic without any additional developer effort. Of the applications described here, this event stream is useful in the document review, problem-resolution collaboration, test record/replay, and guided user applications. The event stream can be used to monitor instances of workflows or build aggregate views of the state of a large number of workflow instances. The standardization of the event stream makes it much easier to build such aggregate views across workflows that were developed independently of each other.

Another powerful idea is the presentation of errors using a business semantic. Often, a technical failure such as the nondelivery of a message leads to escalation to a technical expert because the significance of the failure is unclear without specialist investigation. If the error can be mapped to a workflow model—so that it is clear that the error concerns a noncritical change notification, for instance—then escalation can be restricted to cases where it is necessary.

Composition. If an application is factored into work units, then these work units, with their well-understood interfaces, can be reused by other workflows. Workflows themselves also define work units that can also be used by other workflows.

Customization. Suppose that an ISV ships a workflow, which is customized by a VAR, and then again by a customer. Reapplying these customizations when the ISV ships a new base version is a challenging maintenance problem. The use of a shared, well-understood model for the workflow makes the consequent three-way merges much more tractable. Customization and composition together enable ecosystems where definitions of work and flow become shared or traded artifacts.

Manipulation. As we have seen in the discussions of the document review and test record/replay applications, there are often requirements to invent or modify workflows on the fly. This

modification cannot be done securely if changing code is required. Using a model makes possible dynamic manipulation that is both controllable and comprehensible.

These benefits make a compelling list, and it demonstrates clearly that the description of an organization of work items as a model has a lot to offer.

Platform Characteristics

There must be support for basic structural concepts like sequences, conditions, and loops. However, there also needs to be support for data-driven approaches to deal with the less-structured organizations that appear in applications like problem-resolution collaboration and guided user.

It is also important to allow new semantic elements to be added to create rich, specialized languages such as the data flow-aware composition in scripted operations. Adding new semantic elements might go so far as to require the redefinition of such fundamental ideas as sequence—for example, in the test record/replay application.

The workflow must also be able to communicate in a rich variety of ways. Workflows respond to UI events, drive different types of services (human, programmatic, other workflows), and support queries over the current state of their contracts—for instance, when determining the actions available to an actor in a problem-resolution collaboration application.

If the workflow platform is to be used in all the applications where it adds value, such as MVC, then it must be lightweight. Equally, it needs to address the scale and performance requirements implied by applications such as document review.

In addition, the workflow model itself must be fully programmable, which includes model creation—such as in the test record/replay application—and dynamic model update to support unanticipated change, as in both the document review and test record/replay applications.

Now let's look at the realization of these required characteristics in the Windows Workflow Foundation (WF).

WF implements the idea of workflow as an organization of work items, abstracted away from the related ideas with which it has been coupled in traditional workflow products. The abstractions fall under three main categories: design and visualization, hosting, and semantics.

Design and visualization. A workflow in WF is a tree of work items (called activities). This tree can be manipulated directly as an object model. A designer is provided, but its use is not mandated. It is possible to create new designers specialized to particular user communities or to particular organizations of work items. It is also possible to specialize the provided designer, which can be used not only within Visual Studio but from within an arbitrary hosting application.

Hosting. The WF runtime is sufficiently lightweight to be hosted in a client context such as a controller in a rich-client application shell. It is also performant enough to scale when embedded in a server host, such as the SharePoint Server delivered by Office 2007. The WF runtime's expectations of its host are abstracted as provider interfaces for services such as threading, transactions, persistence, and communications. Useful provider implementations are supplied out of the box, but they may be substituted as required.

Semantics. Different problems respond to different model semantics. WF supports three main styles of workflow out of the box: flow, state machine, and data driven. Flow is optimal for applications where the workflow is in control such as the scripted operations example. State machine is best when the workflow is driven by external events, as in the MVC or guided user applications. A data-driven approach is suited to applications where actions depend on state, as in problem-resolution collaboration.

These semantics can be extended by building custom activities to create a domain-specific vocabulary for use in any of these styles. However, since the structure of a workflow is itself expressed as a set of activities, the same approach can be used to define new styles, and entirely novel semantics, if required.

A Common Workflow Runtime

The focus of the WF runtime is to deliver facilities required by any workflow, and therefore avoid the need to re-implement them time and again in different applications, but without compromising the flexibility of the workflow abstraction. These common facilities fall into four main categories: activity scheduling, transactions and long-running state, exceptions and compensation, and communications. Let's look at each in more detail.

Activity scheduling. The WF runtime defines an activity protocol that all work items implement. This protocol defines the basic activity life cycle (initialized, executing, and closed) and the additional states needed to handle exceptions (faulted, canceling, and compensating). This definition enables the WF runtime to provide work scheduling for all workflows.

Transactions and long-running state. The WF runtime supports the execution of ACID transactions. These transactions are particularly useful for maintaining consistency across workflow state and external state such as application and message state. However, ACID transactions are not suitable for managing long-running state because of their resource and locking implications. The WF runtime implements a broader checkpoint-and-recovery mechanism to handle long-running state. From this point of view, ACID transactions become units of execution within a larger framework. The developer needs not do any work to get the benefit of WF's support for long-running state, as it is default behavior. However, if more detailed control is required, a set of simple model elements are supplied for the purpose.

Exceptions and compensation. The familiar idea of throw-try-catch exceptions is supported by the WF runtime and represented in the out-of-the-box workflow model. However, the WF runtime also supports a broader view of fault handling that includes the idea of compensation for successfully completed transactional units.

Communications. As we have seen, workflows need to communicate in a variety of ways, which is reflected in the WF, that supports communication through .NET method, event interfaces, and Web service interfaces. Support for Windows Communication Framework will also be made available in the future. Thus, WF does indeed realize the workflow-platform approach proposed here.

Figure 4 illustrates the high-level implementation schematic of the document review application and how all of the foregoing comes together. An implementation uses SharePoint as the workflow host. The WF runtime uses the default scheduling service provided out of the box with WF. However, the default persistence and communications services are replaced with implementations specialized for the SharePoint host. The persistence service stores long-running workflow state in the SharePoint database, and the communications service makes the rich-user interaction facilities of SharePoint available to the workflow. Both of these services are in fact delivered out of the box with Microsoft Office 2007.

Three sorts of activities are used to define the document review workflow itself. First, out-of-the-box WF activities are used to provide structural elements such as If-Else and While. Second, activities provided as part of Office 2007 are used to access the user communication services of SharePoint. Third, custom activities are used to implement organization-specific semantics for forwarding and delegation in a standard and reusable way. The WF designer is used as a means to define the workflow and also provide diagrammatic representations of the state of a document review workflow instance to the workflow owner.

Attacking the Problems

In summary, the workflow platform supports an abstraction of the ideas that have made workflow products an attractive attack on business problems. It does not replace today's workflow products, however. Rather, it factors them into platform and superstructure.

The workflow platform embodies two key ideas: a workflow is an *organization of work units*, and a workflow is a *model*, that is, a machine-readable description other than code. These ideas are valuable in a broad range of applications, both within and beyond the problem domain addressed by typical workflow products. Such a workflow platform is most useful if it is low cost and ubiquitous.

The principal benefits delivered arise from the expression of an organization of work items as a model, which has several advantages over a representation in code:

- *Transparency*. The business purposes of the system are clear, allowing users and IT staff to communicate effectively about the desired behavior and IT staff coming onto the project to get up to speed quickly.
- *Isolation of change*. The areas of the application most likely to change are expressed as workflow rather than code. By isolating the rapidly moving parts of the application, changes can be made more reliably.
- *Agility*. The bottom line of all these benefits is business agility. If business users can understand the system, developers can get up to speed quickly, and the risks associated with change are minimized. Then the system may be termed agile.

A broadly useful workflow platform must have these characteristics: define a core workflow model as a standard that is extensible and fully programmable at design time and runtime, be able to communicate in a rich variety of ways, be lightweight and embeddable, and be able to scale and perform well in high-volume environments. WF is a product that displays all of these characteristics. As a component of .NET 3.0 and a part of the Windows platform, WF is also low cost and ubiquitous.

A Workflow Manifesto

The "workflow manifesto" is a set of statements designed to help you consider the flexibility, value and benefits that workflow adds to your solutions architecture. The emergence of a flexible, extensible framework like WF enables the implementation of workflow-oriented solutions in ways that may have been impossible in the past. The "workflow manifesto" was designed to help clarify some of the opportunities for applying and using workflow in our solution architectures:

- Workflow is everywhere
- Workflow is expressive
- Workflow is fluid
- Workflow is inclusive
- Workflow is transparent

While each of these statements provides a significantly different perspective of workflow, they all share two specific benefits in common: *agility* and *abstraction*. Before reviewing the manifesto statements in greater detail let's better understand the reason these two benefits are shared in such a broad fashion.

Agility

Agility is a key concept for both IT and business – a flexible and highly extensible solutions architecture is much more likely to meet the rapidly changing needs of the organization. Because they are model based, *workflows are more suited for change than a pure-code approach.* For

example, workflows typically enable developers to quickly and easily add new steps to a given workflow instance at runtime. Performing a similar function in a code-only approach requires developers to understand how to use a reflection API and manipulate the low-level internals of a given set of classes. This approach exposes the solution to risk and significantly limits the pool of maintenance developers qualified to work on it. Workflows also provide support for business rules that can be modified and re-deployed at run-time, impacting the behavior of one or more workflows and instances that may utilize the associated business rules.

> *Note: While workflows are better suited for change than a pure code approach, workflows are not necessarily designed to replace all code. Workflows are nothing more than a set of specialized classes generated by the workflow modeling environment.*

Abstraction

Abstraction is another key concept for IT and business, but for slightly different reasons. From an IT perspective, abstraction can be used to hide complexity, potentially reducing both development time and long-term maintenance costs. From a business perspective, the maintenance costs are attractive while the workflow model makes it easier for non-technical personnel to understand the objective of a given workflow without having to review the code used to implement the solution. The scope of the workflow model can also be established by a business analyst prior to handing it off to a developer for implementation.

The concept of a model-driven approach is not new – developers have been using UML and other modeling techniques for many years prior to thSe maturation of workflow development tools. Workflow provides both sequential and state machine models that can be used to develop, manage and run the workflow itself. Developers usually prefer to work with the text (code) representation while a business analyst may well be overwhelmed by the detail of the model itself. Most workflow models also enable the model itself to be modified to meet the needs of the organization. For example, WF's presentation surface can be "skinned" to change the look and feel of the modeling surface prior to embedding within your application.

The Workflow Manifesto

The remainder of this chapter examines the Workflow Manifesto statements in greater detail. Several examples of each topic are also included.

Workflow is *Everywhere*

Workflow doesn't just live in the datacenter on integration servers. Any sequence of steps or activities can be represented as a workflow. These steps may be handled by automated processes in a system-to-system (S2S) workflow, a human–to-human (H2H) workflow, or a fractal model that touches upon both (discussed below). Workflow can be used to orchestrate services across applications or within a single application, brokering user events or exposing workflow

capabilities to an end-user. Workflow may also be used within dedicated devices. For example, WF-based solutions can easily be deployed to x86-based XP-embedded devices such as a point-of-sale (POS) terminal or a bank ATM.

There are several scenarios which may benefit from a workflow-oriented approach. The two scenarios we will examine are User Interface (UI) flows and creating and Web services.

UI Flows

A workflow model can serve as the controller within a Model-View-Controller (MVC) pattern. The MVC pattern provides a separation of objects into one of three categories — models for maintaining data, views for displaying all or a portion of the data, and controllers for handling events that may impact the model or views. There are three UI flow examples worth considering:

- WinForms: A workflow can act as the event broker for common user events such as button clicks, data selection, and others. The state of a given workflow instance can also be used to configure the user interface. For example, Figure 5 shows how a workflow state is used to enable or disable buttons on a WinForm based on user interaction with a given workflow instance. (The code for this example is available in the WF Samples folder after downloading and installing the Windows SDK for .NET 3.0.)

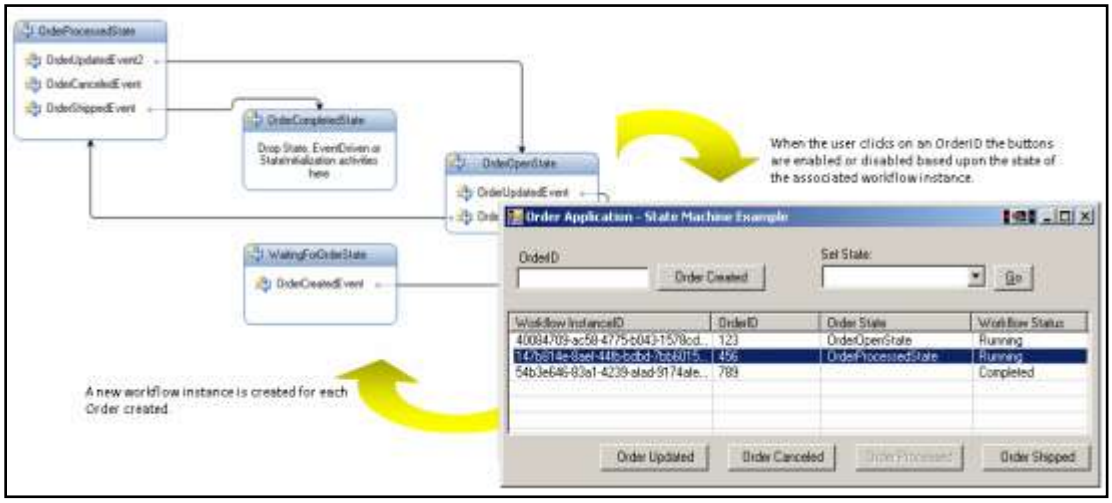


Figure 5: *Workflow instances drive the WinForms user interface*

- Composite UI Application Block (CAB): CAB is an application block that provides a programming model for building smart client applications based upon the Composite Pattern. The Composite Pattern is applied to UI by combining multiple components to create complex user interfaces.

The individual UI components can be independently developed, tested, and deployed (conceptually similar to SharePoint's Web Parts model). The original implementation of CAB was developed before the advent of WF and required a significant amount of code for event brokering and management between the components that make up the interface (see Figure 6 for an overview of CAB). CAB is being re-architected to include WF for event brokering and UI component management (see the area highlighted in Figure 3).

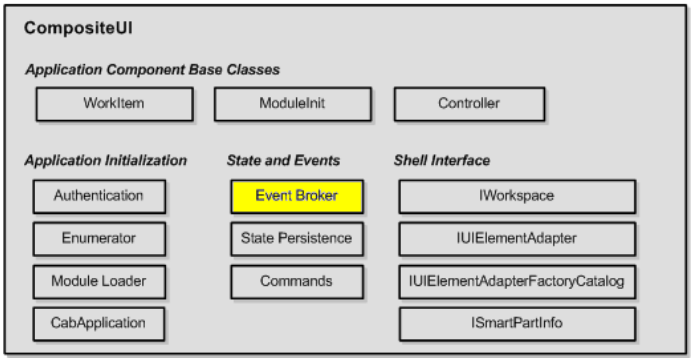


Figure 6: *The Composite UI Application Block*

- Pageflow – A page flow is a directory of Web application files that work together to implement a UI experience. A server-side workflow implementation is used to control the order in which pages are served to the user, possibly changing the user experience on the fly based upon events raised by the current page. Navigation logic remains within the workflow, effectively isolating navigation from presentation. While there is no "official" way to implement Pageflow using WF, the WF product team has made an example available at http://wf.netfx3.com/files/folders/sample_applications/entry10996.aspx. The example illustrates how one can implement a generic navigation framework with WF that is capable of supporting UI technologies like ASP.NET and WPF.

Web services

There are two approaches in which we can consider the relationship of workflows and Web services. A workflow can be used to orchestrate multiple Web services, potentially repurposing the return values from one Web service as input values to another. Using workflows to invoke Web services simplifies the process of working with asynchronous services since the invocation and correlation requirements can be handled by the workflow itself. We can also use workflows

to build Web services, publishing the workflow as a Web service. The model is fractal since we can use workflows to both create and consume (or orchestrate) Web services. Since most services were developed and designed to operate autonomously, bringing them together into a new service composition (workflow) requires that several challenges must be met. Figure 7 illustrates some of the most common challenges facing service orchestration.

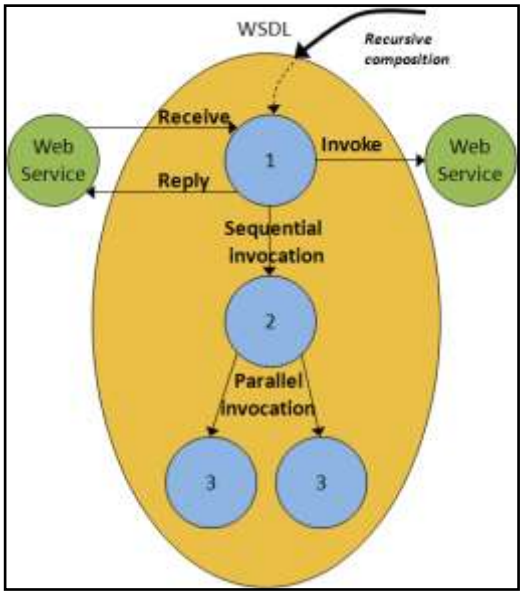


Figure 7: *Capabilities for Service Composition*

Most integration servers (such as BizTalk Server) support the capabilities in Figure 7, relieving developers from having to write, test and maintain "plumbing code" that provides little value to the organization. Figure 8 illustrates how a workflow can be both exposed as a service and orchestrate services.

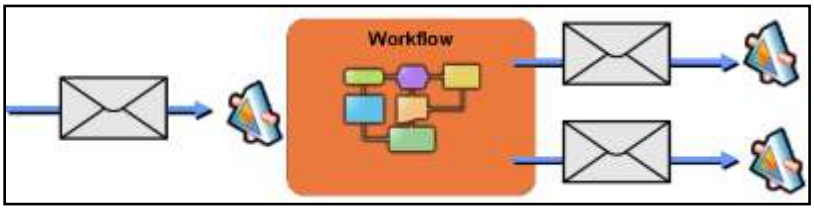


Figure 8: *Workflow and Web services*

In .NET 3.0, WF includes three out of the box (OOB) activities for consuming or publishing simple ASMX Web services:

- The InvokeWebService activity enables a workflow to consume a simple ASMX Web service. Adding an InvokeWebService activity to your workflow triggers the Add Web Reference wizard, enabling you to enter

the URI and proxy name of the service to be invoked. The MethodName dropdown property enables you to select which method on a given endpoint should be invoked. The InvokeWebService activity also includes optional parameter properties to support One-Way or Request-Response Message Exchange Patterns (MEPs).

- A WebServiceReceive activity is typically added as the first element of a workflow to be published as a Web service. WebServiceReceive binds the activation of a workflow instance to a Web service invocation. The interface definition is automatically generated based upon the value of the WebServiceReceive activity's InterfaceName property. The InterfaceName property is a simple interface that defines the method names and signatures to be exposed by the Web service. The WebServiceReceive activity also includes optional parameter properties to support One-Way or Request-Response Message Exchange Patterns (MEPs). A workflow with a WebServiceReceive activity is published as an ASMX Web service by simply right-clicking on the project containing the workflow and selecting "Publish Workflows as Services…"
- A WebServiceResponse activity may be added to a workflow to be published as a Web service with a Request-Response MEP (there is no WebServiceResponse in a One-Way MEP). WebServiceResponse includes a ReceiveActitivityID property which is used to associate the response message with a previously defined WebServiceReceive activity.

The NET 3.0 WF out-of-box activities do not provide support for Windows Communication Foundation (WCF). WCF and WF are more closely aligned In NET 3.5, enabling you to more easily expose workflows as WCF services and orchestrate WCF services.

Using WF to consume or create Web services provides a far richer set of options than a "pure-code" approach. WF enables developers to take advantage of various features such as service orchestration, asynchronous invocations, persisting/tracking staIn this section we illustrated how we can use workflows within user interfaces and how we can use workflows to create and consume Web services. Additional scenarios could include implementing a business logic layer within your applications, supporting long-running processes and exposing a set of configurable business rules that may be modified at runtime. Any sequence of steps or events can be represented as a workflow.

Workflow is *Expressive*

Workflows can do virtually anything that a "pure-code" approach can do, at a higher level of abstraction. This doesn't necessarily mean you should replace all of your code with workflows. What it does mean is that you can use a workflow-oriented approach to raise the level of abstraction in your solutions, potentially raising your level of productivity and reducing long term maintenance and ownership costs. Let's examine these assertions in greater detail.

The model is the workflow

Since the model is a direct representation of the actual workflow there is no loss of fidelity in the implementation of the model. This is a major change from older model-based technologies such as CASE and some UML-based development tools. These tools suffered from the fact that their modeling environments were far more expressive than their code generation capabilities. This meant that the models were, in fact, pictures because they were unable to generate the code necessary for implementing the solution. Moving between a workflow model and the underlying implementation code is a lossless process – the model and the code are different views of the same workflow.

The model also makes communicating and maintaining the workflow much easier than a "pure-code" approach. The model becomes part of the documentation of the workflow and can be shared with (or created by) a business analyst to illustrate the objective of the workflow. The model can also be used by maintenance programmers to understand and maintain the workflow without having to read through large sections of opaque, possibly undocumented code.

Which Workflow Model to use when?

Workflow helps us realize the fundamental difference between a picture and a model. A workflow typically includes some sort of model used to both develop and execute the workflow. With WF we have two models available – a sequential model and a state machine. Sequential models flow from top to bottom in a simple sequential fashion while a state machine model is used to trap various events that may arise during the execution of the workflow. If we review the state machine model in Figure 1 we see there are several states that a Purchase Order moves through while being processed (e.g. OrderOpened, OrderProcessed, OrderCompleted, and so on.). Each state may include one or more activities that fire based upon the context of the currently executing instance.

WF supports two models for designing your workflows – Sequential and State Machine. The model you select should be determined by your solution requirements.

- Sequential models are most effective when the workflow is in control, dictating both the activities and the order in which they are to be performed. Sequential models are most likely to be used in system-to-system (S2S) workflows. For example, an Order Validation process consists of a well-defined series of steps and operates on a highly structured set

of data (e.g. a Purchase Order). An Order Validation process may be best defined using a sequential workflow model.

- State machine models are most effective when the workflow is reacting to events that are raised outside of the workflow itself. The workflow supplies a list of events that may occur while the work is being performed. The order in which these events initially occur is not controlled by the workflow – the workflow only needs to know what can be done next based upon the current state. State machine models are most likely to be used in human-to-human (H2H) workflows. For example, in a document routing and review process, a reviewer may optionally decide to make revisions to the document or route the document back to the original authors for a complete rewrite. Modeling these types of actions in a sequential model can be quite challenging.

There is a third approach that some people call "data driven." A "data driven" approach uses a set of business rules to react to data being processed by the workflow (business rules are created within WF using the Policy activity). A "data driven" approach can be used with both sequential and state machine workflow models. Your solution may choose to use a combination of these workflow models. For example, a given state within a state machine may encapsulate a sequential workflow. The final choice is up to you and should ultimately be influenced by your solution requirements.

Workflow is *Fluid*

Because they are model-based, workflows can be modified, interrogated or otherwise manipulated at design time or run time. That is, both users and analysts participate in both the design and updates to the workflow. Items within the workflow can change frequently, such as the order of operations; method calls; and services, data, or applications to be integrated. For example in a mortgage solution, rates can change based on the market, the loan type, the lendee's history, or several other reasons. Changes may need to flow from design to deployment in a matter of minutes. This requires a different approach than classical compiled and distributed applications. Some workflow systems use a runtime repository of business applications and rules that require updating on a frequent basis. This approach enables workflow systems to rapidly respond to market and organizational conditions. Workflow flexibility can be maximized by judicious use of business rules and orchestrations.

Business Rules

One approach to fluidity is using business rules. Business rules govern conduct of business processes. Business processes tend to stay the same over time but the business rules used by these processes are highly volatile**.** Separating rules from application code enables users to invoke or change rules at run time, resulting in greater workflow flexibility. Business rules are best used for:

- Discrete, point-in-time evaluations and calculations.

- Large no of permutations to encode in a control flow.
- Fact-based inferencing where control flow cannot be predefined.

Orchestrations

Orchestrations are similar to business rules, but more formal. You would use them in place of business rules where formal workflows require:

- Long-running semantics
- Transactions/ Compensations
- Messaging

Orchestrations are also used when there is a recognized control-flow that needs to be rigorously managed for performance or scale and visibility and tracking of the state of the orchestration is critical.

Business rules and orchestrations both enable workflows to be modified faster and better deal with changes in the organization. This flexibility lowers the costs typically associated with development, maintenance and ownership of a given process.

Workflow is *Inclusive*

Automation in a system-to-system (S2S) environment is fairly straightforward – processes tend to be fixed with a finite number of variations. The data processed within a S2S environment tends to be structured, representing well-known business messages such as a Purchase Order. Users, however, work in an ad-hoc manner and may use unstructured or semi-structured data. For example, a user may receive an email at home, convert it to a document, enter related data into a remote tracking system, and communicate these events by leaving someone a voice mail message. Computer systems are much more suited for structured, long-running asynchronous problems with limited flexibility. Workflow models can be used to describe with both human-to-human (H2H) and system-to-system (S2S) interactions. We can also use these models to describe system-to-human (S2H) and human-to-system (H2S) interactions. Translating such models into an executable representation can be challenging and usually results in a task-notification (S2H) and task-completion-response (H2S) interaction pattern.

Workflow is *Transparent*

Humans can be brought into process design right from the start by the abstraction and model-driven design workflow offers. User interfaces like SharePoint Designer let an experienced end user build or modify workflows quickly and easily. A workflow model makes the workflow easier to understand without having to review and maintain opaque blocks of code. Tools such as WFPad (http://www.schmidt6.com/blogfiles/WF/WFPad.exe) can be used to display the workflow model in one windows and the workflow code in another. This enables experienced users and developers to view or maintain a workflow using either a model or the code. As stated we stated earlier, the workflow *is* the model. The workflow model offers:

- state management across service interactions
- service aggregation and orchestration
- transparent implementation of services
- visibility into the state of the service
- configurable runtime behavior of: tracking, persistence, transactions, threading, and so on.

Workflow transparency is important because state persistence is a critical requirement for long running processes. Processes may last for days, weeks, months or even years. Processes may need to be started, held, and restarted by humans or runtime services – sometimes with different timeframes with no loss of fidelity.

Understanding the Relationship between BizTalk Server and WF

In this chapter we made numerous references to Windows Workflow Foundation (WF). When WF was first announced many people incorrectly assumed WF was a replacement for BizTalk Server (BTS). WF and BTS are *complementary* technologies designed to serve two very different needs:

- BTS is a *licensed product* designed to implement workflow ("orchestrations") *across* disparate applications.
- WF is a *developer framework* designed to expose workflow capabilities *within* your application. There are no fees or licensing restrictions associated with using or deploying WF.

Use BizTalk Server if…

- You need to implement system-to-system (S2S) workflows across disparate applications or platforms.
- You need an enterprise-strength Business Process Management (BPM) suite that enables complex transformations, support for popular application/wire-level protocols and integration with line-of-business (LOB) systems like SAP, PeopleSoft, Oracle, and JD Edwards.
- You need to interface with a human-to-human (H2H) workflow.
- You need to take advantage of Business Activity Monitoring (BAM).
- You need to map authentication information between Windows and non-Windows systems (Single Sign-On (SSO))
- You need to set up and manage trading partners in a B2B scenario.

- You need a complete set of tools for managing the infrastructure and scalability of your solutions architecture.

Use Windows Workflow Foundation if…

- You need to expose workflow capabilities to end users through your application
- Your application only needs a message or event broker.
- You need to build a Human to Human (H2H) workflow capable of interfacing with a System to System (S2S) workflow. (SharePoint 2007 includes several predefined Human Workflows built with WF).
- You only need workflow or simple business rules and are not interested in the other features that BizTalk Server provides.

Conclusion

In this chapter we examined the concept of workflow and discussed other terms and concepts frequently associated with workflow. Workflow is the organization of work using activities that represent both people and software. Models represent this organization, providing a higher level of abstraction and flexibility to our solutions. This chapter also clarified the relationship of workflow-oriented capabilities on the Microsoft platform (Windows Workflow Foundation and BizTalk Server).

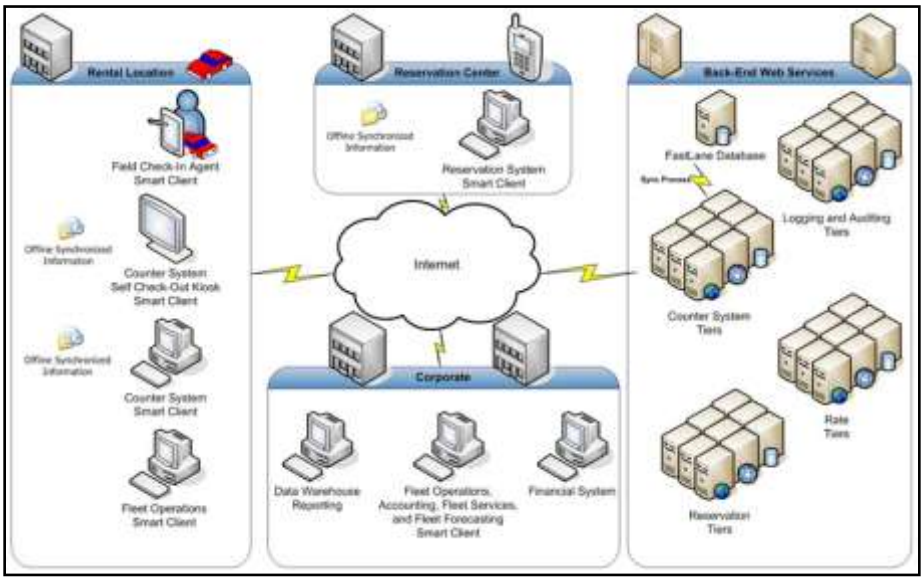
Chapter Four provides a more detailed discussion of the **Data** recurring architectural capability.

SOA Case Study: Dollar Thrifty Automotive Group

Dollar Thrifty Automotive Group provides car rentals through two brands: Dollar Rent A Car and Thrifty Car Rental. Approximately 8,300 Dollar Thrifty employees serve travelers from company headquarters in Tulsa, Oklahoma. With an ever-expanding network of North American and international corporate-owned and franchised locations in 70 countries, the Dollar or Thrifty brands operate in virtually all of the top U.S. airport markets.

As Dollar Thrifty grew and expanded, and competition in the rental car business accelerated, company decision makers became aware of issues with their mainframe-based car rental system. Dollar Thrifty wanted a car rental system that would enhance customer experience and help the company be more agile in the face of competition, now and in the future.

Developers at Dollar Thrifty took advantage of Windows Workflow Foundation (WF) to build validations in a smart client application. WF provides the programming model, engine, and tools for quickly building workflow-enabled applications. Because they used WF, developers didn't have to write the workflow or validation logic from scratch. Developers were able to create a flexible interface that lets agents work the way they want to work yet still satisfies all the business rules.



Dollar Thrifty Architectural Overview

The entire Case Study is available online at http://www.microsoft.com/casestudies/casestudy.aspx?casestudyid=1000003883.

See other SOA case studies at http://www.microsoft.com/casestudies/search.aspx?Keywords=SOA.

References:

1. "Building Applications on a Workflow Platform" by Dave Green, Architecture Journal, April 2006. Available at http://msdn2.microsoft.com/en-us/library/bb245670.aspx.

Chapter 4: Data

"Data is a precious thing and will last longer than the systems themselves."

- Tim Berners-Lee
inventor of the Web

Reader ROI

Readers of this chapter will build upon the concepts introduced in previous chapters, specifically focusing on the Data architectural capability.

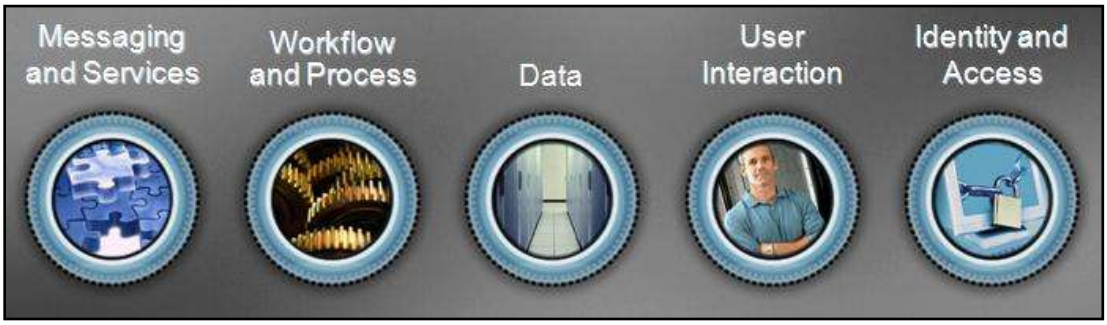


Figure 1: *Recurring Architectural Capabilities*

In this chapter, we will discuss the issues that should be considered when architecting the data portion of your SOA project. The following topics will be addressed:

- An overview of Data issues
- Scaling up and scaling out
- Replication and partitioning strategies
- Data integration and Master Data Management (MDM)

Acknowledgements

The concepts discussed in this chapter are entirely drawn from earlier efforts in this space. We wish to thank the following individuals for their work in this area: Roger Wolter (data issues, MDM overview, MDM Hub Architecture), Kirk Haselden (MDM).

Data Challenges Facing SOA

Overview

Any SOA project that ignores data issues is likely to fail. The conventional wisdom is that somewhere in the range of 20 to 40% of an SOA project should be set aside for data integration and the further into the project you are when you plan for data integration, the more expensive it becomes. In this section of the course, we will discuss the issues that should be considered when architecting the data portion of your SOA project.

One of the goals of SOA is to transform tightly coupled – single purpose systems into a set of loosely-coupled services that can be used and re-used across the enterprise. If an application is split into multiple services we could potentially use WS-AtomicTransaction (WS-AT) to simulate a distributed transaction. This approach tightly couples the services involved in the transaction - if any one of the services is unavailable the whole transaction will fail. One way to make service aggregation more robust is to design the service so it executes asynchronously within its own atomic transaction – this way if a service is unavailable the larger operation can still complete successfully. Data integrity requires a guarantee that if the service doesn't complete immediately, it will reliably complete when the service is available. The best way to ensure this is to use reliable, transactional messaging between services to ensure that once the original transaction commits, the messages required to complete all the other parts of the operation are also committed to a database so they won't be lost if something goes wrong (we briefly address reliable service execution later in this chapter).

The database could become a performance bottleneck during service reuse. When legacy services are used in new solution as your SOA infrastructure expands - the database where the service stores its data may not be able to handle the additional load of the new service calls. This implies that new SOA projects might require additional capacity for existing database systems.

Data Integration Issues

People often talk about using SOA to break-down data silos to provide a unified view of the data. Using services to break data silos can expose inconsistent and duplicate data. If the data is inconsistent or has duplicates, you must correct these issues prior to breaking down silos without first fixing the data will make the data unusable. If every data silo has a copy of the same customer with different data, breaking down the silos will expose multiple inconsistent copies of the customer data which is worse than the silo where even if the data was wrong there was at least only one copy. For example, a goal of a customer service may be to provide a single view of the customer - if the same customer appears in ten databases with a different address in each,

the single view idea is lost. *Using SOA to integrate data access without also integrating the data can lead to a confused view of inconsistent, duplicate data.*

Consolidating business-relevant data is a key requirement for SOA. Many enterprises rely upon a mix of custom and LOB systems across the organization. This has resulted in disparate, incompatible systems storing and maintaining inconsistent representations of business entities such as customers, purchase orders, invoices and many others. Semantics will also differ from one system to another. For example, the meaning of a particular data element on one system (e.g. Address represents a street address in a given city) may not be equivalent to the meaning of the same element on another (e.g. Address represents an endpoint for a Web Service). Consider two geographical databases – one that records distance in meters, the other in yards. Further, semantically equivalent data elements such as last name may vary in representation across systems, such as in capitalization and spelling. Some of the databases may also contain invalid data as a result of inconsistently enforced validation rules. Finally, data may be present in one or more systems that violate referential integrity. Together, these problems make it very difficult to maintain a consistent representation of key business entities across organization.

Semantic Dissonance

Three of the data challenges facing SOA include:

- Key business entities such as customer have inconsistent representations across multiple databases.
- Invalid data or referential integrity violations may be present within the various databases
- Semantic dissonance exists among the constituent data elements of key business entities.

Semantic dissonance describes a situation where data that initially appears to be the same may not necessarily mean the same thing. For example, one system might treat a monetary figure for annual revenues as if it includes sales tax, while another system treats the monetary figure as if it does not include any taxes. Likewise, one system might define the Western Region, in the context of revenue breakdown, to include the state of Colorado. Meanwhile, another system may define Colorado as part of the Central Region. These types of semantic dissonance can be difficult to detect and reconcile. Some rather elegant types of integration rely on the complete resolution of semantic dissonance, which might not be feasible in real-life situations. Other simpler forms of integration, such as Portal Integration, can accommodate ambiguity, but they do it at the expense of precision. In Portal Integration the business process that describes a sequence of tasks does not have to be represented in a precise process model, but instead

resides in the user's head. This enables the end user to compensate for semantic dissonance and make a case-by-case decision as needed.

Data Consistency

As stated earlier, designing services loosely-coupled services often requires re-thinking how database transactions work. For example, a legacy order entry system may add the order-header and order-lines to the database, update the customer record, update the inventory, create a ship order, and add a receivables record in the same database transaction. Enlisting services into a set of transactions breaks the loosely coupling model. If any one of the services is unavailable, the whole transaction will fail so aggregating services into a large distributed transaction can make the system very fragile.

The way to make this aggregation of service more robust is to ensure that each service executes asynchronously within its own transaction – this enables the operation to complete successfully, even if one or two of the services are unavailable. Data integrity requires a guarantee that if the service doesn't complete immediately, it will reliably complete when the service is available. The best way to ensure this is to use reliable, transactional messaging between services to ensure that once the original transaction commits, the messages required to complete all the other parts of the operation are also committed to a database so they won't be lost if something goes wrong.

When services are used in new applications as your SOA infrastructure expands, the database where the service stores its data may not be able to handle the additional load of the new service calls so an SOA project may require adding capacity to the database systems.

A SOA is frequently expected to break-down both application and data silos, providing a unified view of the data – this is sometimes referred to as the "single view of the customer" problem. For example, the goal of a customer service may be to provide a single view of the customer but if the same customer appears in 10 different databases with a different address in each, the goal of a single view of the customer becomes impossible. If data is inconsistent or has duplicates then the data must be corrected prior to making the data available to other services within the enterprise. If every data silo has a copy of the same customer with slightly different data, breaking down these silos exposes inconsistent representations of customer data – this issue far is worse than data silos since each used a single copy of the customer data. Using SOA to integrate data access without integrating the data leads to a confused view of inconsistent, duplicate data. Inconsistent data is a common cause of SOA failure.

Database Scalability

Scalability

Scalability is the ability of an application to efficiently use more resources in order to do more useful work. For example, an application that can service four users on a single-processor system may be able to service 15 users on a four-processor system. In this case, the application is scalable. If adding more processors doesn't increase the number of users serviced (if the application is single threaded, for example), the application isn't scalable. There are two kinds of scalability: scaling up and scaling out.

Scaling up means moving the data services onto bigger, more powerful servers (such as moving from four-processor servers to 16-processor or 32-processor servers. This is the most common way for databases to scale. When your database runs out of resources on your current hardware, you go out and buy a bigger box with more processors and more memory. Scaling up has the advantage of not requiring significant changes to the database. In general, you just install your database on a bigger box and keep running the way you always have, with more database power to handle a heavier load.

S*caling out* means expanding to multiple servers rather than a single, bigger server. Scaling out usually has some initial hardware cost advantages—eight four-processor servers generally cost less than one 32-processor server—but this advantage is often cancelled out when licensing and maintenance costs are included. In some cases, the redundancy offered by a scaling out solution is also useful from an availability perspective.

Replication and Partitioning

The two most common techniques for scaling out data are *replication* and *partitioning*. Replication requires making multiple copies of the data and distributing copies for broader access to the data. Replication typically uses one “write” (or “master”) copy of the data with multiple read-only copies for distribution. Partitioning splits the database into multiple databases based upon a specific partition column. The databases can be further segmented into organization-specific functional areas for even greater scale.

To illustrate the impact of partitioning let’s examine how an order database might be partitioned. One possibility might be to partition orders based on what was ordered such as book orders in one database and clothing orders in another. Aside from the obvious issues (e.g. what happens to an order that includes both books and clothes), this scheme wouldn’t work well if the majority of queries join orders to customers. The problem is that the join query would have to check all of the order databases for matching orders. An alternate approach for partitioning an order database would be to split the orders by order-number range. This might make sense if orders

are most often accessed by order number instead of by location. If there are a lot of joins with the customer table this scheme would require distributed joins. The only way to solve the join issue would be to partition the order database by customer number, so for a given customer the order database to use is always known. This is especially effective if the customer database is partitioned and the orders for each customer are in the same database as the customer. There will be other data that must be joined to the order data and if possible this data should be partitioned on the same scheme to avoid distributed joins. Some of this data may also be reference data – item descriptions for example – and this can be replicated to all the order databases to eliminate distributed joins to the inventory database.

Not all application data can be effectively partitioned and choosing the right partitioning scheme is essential for effective scaling out with partitioned data. The challenge is to align the partitioning scheme with how the data is accessed. If application data can be divided into multiple databases and the power of multiple servers outweighs the communications costs of assembling the results, the data can and should be partitioned.

Data Dependent Routing

Another scale out technique is known as Data Dependent Routing (DDR). DDR requires enough intelligence in the client application (typically in a middle-tier) to route database requests to the appropriate nodes. With DDR, there are no views across nodes — each federated server is independent of the others (with the exception of sharing the database schema). The middle tier contains mappings to how the data is partitioned and which node contains the data. While this approach is more complex it tends to result in far better performance than standard partitioning techniques.

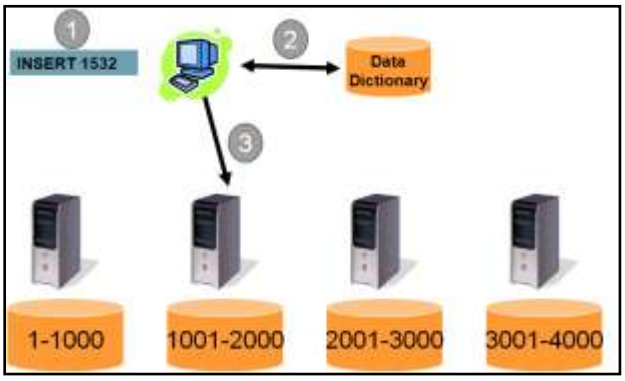


Figure 2: *In DDR the partition attribute value determines the location of the database*

Partitioning with Services

There are two basic architectural approaches when using services for partitioning:

1. Assume the client understands the partitioning and distribution so each copy of the service handles requests for a particular database
2. All services are equivalent and have the responsibility for determining which database to query to respond to each request.

The first option is appropriate when the client is a server-side application that can access the data dictionary to discover the service partitioning scheme. The second option requires more complex services but makes data distribution transparent to the clients. The second option is appropriate for smart client architectures where it's not practical for every client to track the partitioning. Service partitioning is illustrated below in Figure 3.

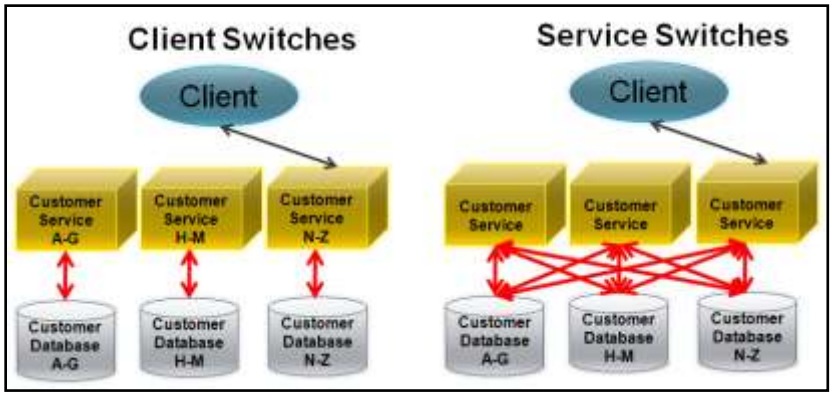


Figure 3: *Database routing from clients or services*

One of the expected of SOA is data integration. When services communicate using messages with well-defined formats and schema then exchanging data between systems is easy right? Unfortunately its not that easy when one considers data integration. In addition to having all your systems agree on what a customer message looks like, they are going to have to agree on customer identifiers for systems to exchange data in a meaningful way. For example, if I appear as a customer in three of your systems but one system uses my SSN as an ID, one uses my email address and another uses my phone number, chances are I am going to show up as three different customers so it will be nearly impossible for your company to understand me as your customer. As new systems are built and more customer data is acquired through mergers and acquisitions, this problem becomes more difficult to solve. To solve this issue we need a way to define and manage enterprise-wide representations of our business entities. This is the Master Data Management (MDM) problem.

Master Data Management (MDM)

Most software systems have lists of data that are shared and used by several of the applications that make up the system. For example, a typical ERP system will have a Customer Master, an Item Master, and an Account Master. This master data is often one of the key assets of a

company. It's not unusual for a company to be acquired primarily for access to its Customer Master data.

Master data is the critical nouns of a business and falls generally into four groupings: people, things, places, and concepts. Further categorizations within those groupings are called subject areas, domain areas, or entity types. For example, within people, there are customer, employee, and salesperson. Within things, there are product, part, store, and asset. Within concepts, there are things like contract, warrantee, and licenses. Finally, within places, there are office locations and geographic divisions. Some of these domain areas may be further divided. Customer may be further segmented, based on incentives and history. A company may have normal customers, as well as premiere and executive customers. Product may be further segmented by sector and industry. The requirements, life cycle, and CRUD cycle for a product in the Consumer Packaged Goods (CPG) sector is likely very different from those of the clothing industry. The granularity of domains is essentially determined by the magnitude of differences between the attributes of the entities within them.

What is MDM?

For purposes of this chapter, we will define Master Data Management (MDM) as “the technology, tools, and processes required to create and maintain consistent and accurate lists of master data”. There are a couple things worth noting in this definition. One is that MDM is not just a technological problem. In many cases, fundamental changes to business process will be required to maintain clean master data, and some of the most difficult MDM issues are more political than technical. The second thing to note is that MDM includes both creating and maintaining master data. Investing a lot of time, money, and effort in creating a clean, consistent set of master data is a wasted effort unless the solution includes tools and processes to keep the master data clean and consistent as it is updated and expanded.

While MDM is most effective when applied to all the master data in an organization, in many cases the risk and expense of an enterprise-wide effort are difficult to justify. It may be easier to start with a few key sources of Master Data and expand the effort, once success has been demonstrated and lessons have been learned. If you do start small, you should include an analysis of all the master data that you might eventually want to include, so you do not make design decisions or tool choices that will force you to start over when you try to incorporate a new data source. For example, if your initial Customer master implementation only includes the 10,000 customers your direct-sales force deals with, you don't want to make design decisions that will preclude adding your 10,000,000 Web customers later.

While Master Data management is fairly new, it includes two subsets of data management that have been around for several years: Customer Data Integration (CDI) and Product Information Management (PIM).

Customer Data Integration (CDI)

Customer Data Integration (CDI) is used to provide a single, consistent view of an organization's customers. This part of MDM is often the first to be implemented because customer data is a major pain point for many organizations. For example, a bank that has grown through a series of acquisitions may find it has several different sets of customer data for the same customer if the customer had accounts or loans with more than one of the acquired banks. In an acquisition scenario, customers may have already had accounts and loans with both banks. When they go into a one bank office they expect to be able to access all their accounts with both of the original banks. The SOA project must include a CDI implementation to provide this integration.

A variation of CDI is Party Management. This generalizes the Customer Data Integration techniques to all "parties" including customers, vendors, partners, employees, distributors, suppliers, etc.

Product Information Management (PIM)

Another subset of MDM is Product Information Management (PIM). PIM unifies information about the products that an enterprise handles. For example, an auto parts wholesaler that has grown by acquiring other wholesalers may have several part numbers for the same parts with inconsistent descriptions and prices. PIM will help detect and fix these inconsistencies to provide a single unified view of the parts in inventory. In our scenario, the "products" are the accounts, loans, and financial instruments offered by the two banks. Providing consistent information about the product offerings of the merged bank is the role of a PIM system.

Master Data Management (MDM) Hub Architecture

The MDM hub is a database with software to manage the master data that is stored in the database, keeping it synchronized with the transactional systems that use the master data. Figure 4 illustrates the architecture of a typical MDM hub.

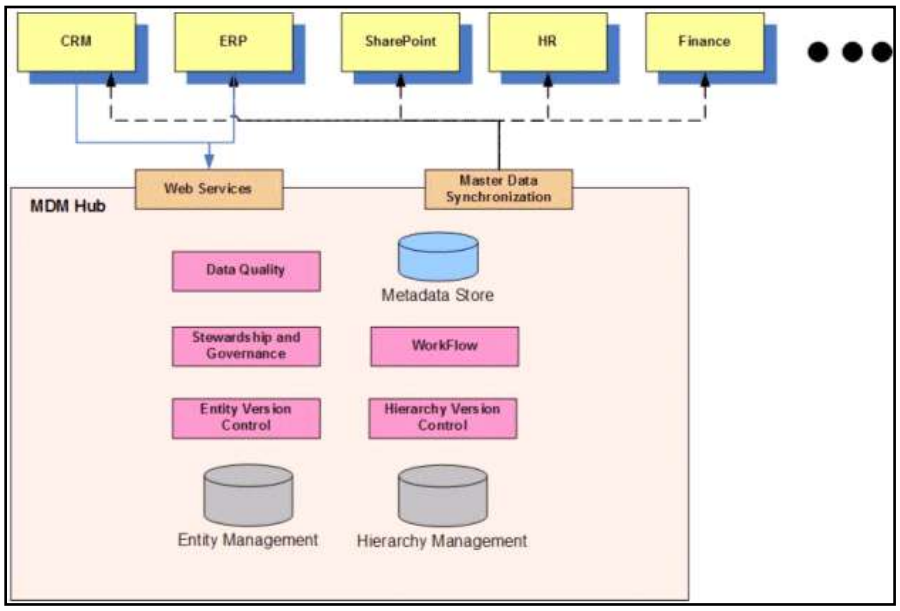


Figure 4: *MDM Hub Architecture*

The MDM hub contains the functions and tools required to keep the MDM entities and hierarchies consistent and accurate. In this architecture, the MDM data can be accessed through a Web services interface. The Master Data Synchronization function is responsible for keeping the data in the hub synchronized with the data in the transactional systems (depicted across the top in Figure 4). There are several alternative implementation styles used for MDM hubs. The next section describes three of the more commonly used styles.

Hub Architecture Styles

There are three basic styles of architecture used for MDM hubs: the registry, the repository, and the hybrid approach. The hybrid approach is really a continuum of approaches between the two extremes of registry and repository, so I'll spend more time on the two extremes.

Repository

In the repository approach, the complete collection of master data for an enterprise is stored in a single database. The repository data model must include all the attributes required by all the applications that use the master data. The applications that consume, create, or maintain master data are all modified to use the master data in the hub, instead of the master data previously maintained in the application database. For example, the Order Entry and CRM applications would be modified to use the same set of customer tables in the master-data hub, instead of their own data stores. The advantages of this approach are pretty obvious. There are no issues with keeping multiple versions of the same customer record in multiple applications synchronized, because all the applications use the same record. There is less chance of duplicate records

because there is only one set of data, so duplicates are relatively easy to detect. However, they obviously are not impossible, because things like alternate spelling, nicknames, multiple locations for the same company, typos, and so on are still possible, and the MDM hub must be designed to deal with them.

While the repository approach has significant advantages for maintaining a continuously consistent source of master data, there are major issues that must be considered when designing a repository-based MDM hub:

- The most obvious issue is that it's not always easy or even possible to change your existing applications to use the new master data. If you do not own the source for the application, you may not be able to modify it to use the new master-data hub. If the application's data model is pretty close to the MDM hub's data model, you may be able to use views and linked servers to make your application think it is talking to its own data, when in fact it is talking to the MDM hub.

I have also seen some systems that reduce the number of changes required in the applications by creating a stand-alone application that does some of the maintenance of the master data, so that not all of the application functionality needs to be ported to use the hub data. This approach is generally hard to implement in a way that users accept, however. Adding customers in a different application than the one used for updates is probably unacceptably complex. On the other hand, one of the more common reasons for implementing and MDM hub is to provide clean, consistent data for a SOA implementation. If you are rewriting and wrapping your applications as services, it might not be unreasonable to create new services to manage the master data.

- Another issue that must be resolved when implementing a repository-style MDM hub is coming up with a data model that includes all the necessary data, without it being so large that it's impossible to use. Because the hub database is used by all applications in the repository model, it has to include all the information required for all the applications. The simple answer to this is to make the hub database a superset of all the application data models. In this approach, a hub customer record would include all the attributes of the customer records of all the applications using the MDM hub. This is not practical, because it ignores many of the problems you need an MDM solution to solve. For example, if there are five formats for addresses, eight formats for telephone numbers, and six different customer IDs, making all of these columns in the customer MDM database would make the MDM hub almost unusable. Every query would have to decide which address, telephone number, and customer number to use. In many records, only one or two formats would be populated.

The obvious solution to this is to settle on an enterprise-wide standard for each of the data elements in the MDM hub and modify the applications to consume and produce the standard

formats. This is not only a lot of work for the IT department, but determining whose format should become the standard format is often a major political problem. All the application owners think that their data formats are the right ones—not necessarily because the formats are any better, but because the application owners do not want to make the changes required to use a different format. It's not unusual for meetings held to settle on a data model to take as much time as the actual implementation of the project. If there are data elements that are used by only one application, the data-modeling effort might decide to eliminate them, and this might require significant changes to the application.

- Another significant data-modeling issue is what to do with data elements that are not used by all applications. For example, a customer added by an order-entry application would likely have significantly fewer attributes than a customer added by the CRM application. Or a product added by marketing might have attributes that are very different from a product added by engineering. In some cases, it might make sense to assign default values to unpopulated attributes; and, in other cases, you might decide to modify the application to populate the extra attributes. In an SOA implementation, you may decide to populate all the attributes with the service program. In general, there will be cases in which it is not desirable or possible to populate all of the attributes from all the applications. A typical example is the Product Information Management (PIM) part of an MDM system, in which it may not make sense to maintain the same attributes for a product that is purchased for resale as for a product that is manufactured in-house.

Registry

The registry approach is the opposite of the repository approach, because none of the master-data records is stored in the MDM hub. The master data is maintained in the application databases, and the MDM hub contains lists of keys that can be used to find all the related records for a particular master-data item. For example, if there are records for a particular customer in the CRM, Order Entry, and Customer Service databases, the MDM hub would contain a mapping of the keys for these three records to a common key.

Because each application maintains its own data, the changes to application code to implement this model are usually minimal, and current application users generally do not need to be aware of the MDM system. The downside of this model is that every query against MDM data is a distributed query across all the entries for the desired data in all the application databases. If the query is going against a particular customer, this is probably not an unreasonable query. But if you want a list of all customers who have ordered a particular product in the last six months, you may need to do a distributed join across tables from 5 or even 10 databases. Doing this kind of large, distributed query efficiently is pretty difficult. This is the realm of Enterprise Information

Integration (EII). So, unless your requirements are relatively simple, you may want to look at EII-distributed query tools to implement query processing in a registry-model MDM hub.

There are basically two styles of repository databases used for MDM. The first has one row in a table for each master-data entity and columns for the keys of the application systems. This is the most straightforward to implement and the most efficient in operation, because all of the distributed queries for a given MDM record can start from the same database row. A NULL value for a particular key means that the corresponding database does not contain a record for the given MDM entity.

There are two significant issues with this scheme, however. First, adding an application to the MDM hub means adding columns to the key-matching table, which is not a big issue, but it may also mean changing queries to include the new source of information. The second, more significant issue is that this style assumes that a given database has only one record for a given MDM entity. While this would be ideal, it is rare to find this in a real application. One obvious solution to this is first to clean up the application databases, so there *is* only one record for each master-data item. This should be one of the goals of any MDM project, but it's not always possible to make the database cleanup a prerequisite for including an application in the MDM hub. If it is impractical to clean up the application database before integrating it into the MDM hub, the repository can be designed with one row for each mapping from the MDM entity to an application record. For example, if Ford has 20 records in the CRM database, the MDM hub would have 20 rows mapping the Ford MDM identity to each of the different CRM customer numbers. This style makes for much more complex queries and also raises issues, such as how to deal with 10 different addresses for the same customer. Nevertheless, it might be a necessary step in the evolution of your MDM solution. Knowing that there are 20 CRM records for Ford is a necessary first step in consolidating them into a single record.

Hybrid Model

As the name implies, the hybrid model includes features of both the repository and registry models. It recognizes that, in most cases, it is not practical (in the short term, at least) to modify all applications to use a single version of the master data, and also that making every MDM hub query a distributed query is very complex and probably will not provide acceptable performance. The hybrid model leaves the master-data records in the application databases and maintains keys in the MDM hub, as the registry model does. But it also replicates the most important attributes for each master entity in the MDM hub, so that a significant number of MDM queries can be satisfied directly from the hub database, and only queries that reference less-common attributes have to reference the application database.

While at first it seems that the hybrid model has the advantages of both of the other models, it is important to note that it has issues that neither of the other models has. Only the hybrid model includes replicated data (other than keys), so only the hybrid model must deal with update conflicts and replication-latency issues. The hybrid model also has the same data-model issues that the repository model has. Which attributes are stored in the hub, what they are called, and what format they are in can be very contentious issues when the hub integrates data from many disparate systems.

Architectural Issues

The following is a brief discussion of some of the architectural issues that must be considered in the design of an MDM hub database.

Data Model

In all three models, the design process must include a common data model for the hub database. In the repository model, the MDM data model becomes the hub-database data model. The model includes mapping from the application data models to the MDM data model, but these mappings are used only to create the hub database and define the application changes required to modify the application to use the hub database as the source of their master data.

The other two hub models also require an MDM data model and mappings from the current applications, but they are used differently. In the registry model, the data model is used to define queries and views, and the mapping is used to do the necessary transformations to map the application data to the MDM data model in each query. In the hybrid model, the common attributes are replicated to the hub database and the non-common attributes are transformed as part of queries, so both kinds of mapping are used. Almost by definition, there will be alternate mappings for some attributes, and rules must be defined for which mapping to use. For example, a customer address is generally stored in several databases so rules must be defined to control which address to use first and which alternate to use if the preferred address isn't available. (These business rules can get to be pretty complex if many databases are integrated in the MDM hub – we will cover business rules later.) The data models and business rules are documented in the MDM metadata and should be used as required to implement data-driven processing for populating, maintaining, and querying the MDM hub data.

MDM Hub Model

We have covered the three hub-database models, so let's discuss how to decide which model to use. The repository model is the most attractive, because it provides a real source of master data that is always current and consistent. The other choices involve data replication, so there is usually some latency between data updates and hub updates. Master data is generally fairly

static, so a little latency is not necessarily unacceptable. The non-repository approaches also maintain multiple copies of some data, so consistency (keeping the copies the same) is an issue these approaches must deal with.

The downside of the repository model is that it can be extremely expensive and take a long time to implement, because it requires changes to the applications that maintain and consume the master data. The repository model makes sense if: the number of applications involved in the MDM project is limited; you have enough control over the applications to make the required modifications; and the availability of authoritative and consistent master data provides enough business value to justify the time and cost required to build a repository-model MDM hub.

A registry-model MDM hub is appropriate when only a limited number of non-performance–critical queries involve access to a significant number of the application databases integrated with the MDM hub. Registry-model hubs are cheaper and quicker to implement and can be implemented one data source at a time, so they are good for incremental implementation and provide early return on investment (ROI). Registries are not good when queries routinely return attributes from many application databases or when there is enough duplication of data, so that determining which of several alternate sources of an attribute to return is a complex decision. In these cases, the pre-integrated and cleansed data provided by a hybrid-model MDM hub provide a more efficient and consistent source of master data.

It's important to note that the hybrid model is not a single model, but instead a whole continuum of options that start at the registry model and continue through to the repository model. For this reason, you may decide to start with a solution close to the registry model and gradually expand the number of attributes integrated into the MDM hub until you have an MDM repository implemented. Because MDM projects can be very expensive and time-consuming in a large enterprise with many applications, it's good to have a strategy that allows you to implement incrementally by both gradually increasing the number of attributes stored in the hub and incrementally adding applications to the hub. This allows you to show an early ROI from the MDM project, with a clear path to a long-term enterprise-wide solution.

Versions and Hierarchies

The previous section explained the options for implementing an MDM hub. This section drills into that a bit by discussing versions and hierarchies—two features that are keys to an MDM hub implementation. It covers why they are important and presents a few implementation options.

Link Tables

In the implementation options for both of these features, I refer to link tables frequently, so I thought I would explain what I mean when I say *link table*. (If you are already a link-table expert, feel free to skip to the next section.)

One of the fundamental concepts in relational databases is using a foreign key to define a relationship between related rows. This is done by storing they key of the related row in a column of the other row. For example, if I have a table of customers and another table of addresses, I can specify the shipping address for a customer by placing the primary key of the address table in a column named "shipping-address" in the customer table. When you want to find the shipping address for a customer, you use the value in the shipping-address column for that customer to look up the address. Many customers can use the same address by using the same key in their shipping-address column, but there's no good way to model a single customer with many shipping addresses. In reality, many customers can have the same address, and one customer can have many addresses. This is called a many-to-many relationship, and the easiest way to model this is with a link table. A link table looks something like Figure 5.

Figure 5: *A simple link table example*

Another useful property of link tables is that columns in the link table can be used to represent properties of the relationship. For example, a relationship between customers and addresses might represent a shipping address for a customer or a billing address for a customer. You could represent this by having two different link tables—one for shipping addresses and one for billing addresses—or by having a single link table to link customers and addresses with a link-type column that is used to differentiate between shipping-address links and billing-address links, as described in Figure 6.

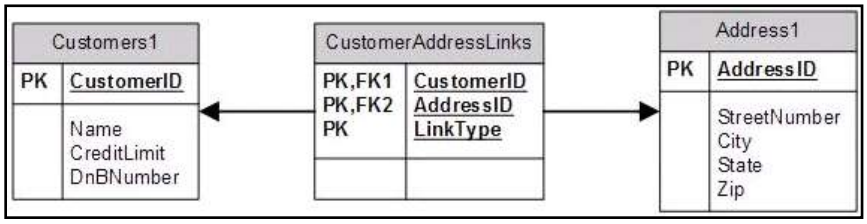


Figure 6: *A typed link table*

Notice that all the information about the relationship is included in the link table. Neither of the tables that are tied together has any information about the link. This means you can create a new

relationship between tables that are part of applications that can't be changed. For example, you can create a relationship between a customer record in the CRM application and a territory record in the Sales Force Automation application without changing either database.

Versioning

Data governance and regulatory compliance are much easier with a complete version history of all changes to the master data. It is often not enough to know what a customer's credit limit is today; you need to know what his credit limit was three months ago, when the customer was charged a high interest rate for exceeding his limit. While this is a simple example, there are many cases in which knowledge of past values for master-data attributes may be required. This leads to versioning as a key feature for master-data management systems. Versions are also required to support data stewardship and governance activities on master data. When master data is modified, business rules are applied to the modifications to determine if they meet the rules developed by the data-governance organization. Data stewards also use version information to monitor the results of the updates and, if necessary, restore the original values.

When most developers think of versioning, they picture source-code control systems that have full branching and merging capabilities. If your MDM hub needs this type of versioning, the versions are generally implemented with link tables that link rows in a version table with a particular version of the MDM record. A simplified diagram of the links might look something like Figure 7.

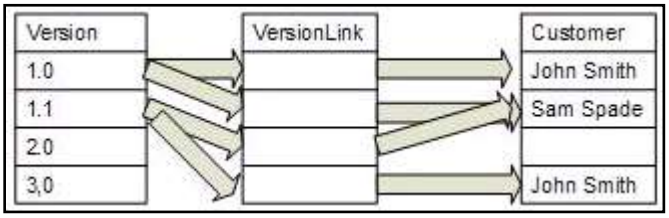


Figure 7: *Versions with a link table*

Notice that John Smith changed in version 1.1, so there are two different rows for John Smith; but Sam Spade did not change, so both versions point to the same row. In this schema, adding a new branch involves adding a row to the version table and creating rows in the VersionLink table for every customer. As customers are updated, a new row is inserted for each modified customer row and the link is changed to point to the new row. While this method offers a lot of flexibility, millions of customers and hundreds of branches produce huge link tables, so managing the volume of data can be an issue. Also, even fairly simple queries like "select all customers with a past-due invoice" involve multiple joins to obtain the right version of the customer records. In my opinion, most MDM systems do not require this level of versioning flexibility, and trading reduced flexibility for simplicity and performance is a good option.

One of the simplest versioning schemes is to add an "EffectiveDate" column to each master-data row. When a master-data item is modified, a new copy of the row is inserted with the "EffectiveDate" column set to the date and time that the change was made. (Okay, maybe it should be "EffectiveDateTime.") When you want to query the latest version of all customers, you look for the MAX(EffectiveDate). If you want to know what a customer record looked like on a particular date, you look for the row with the maximum EffectiveDate in which the EffectiveDate is less than the date you are looking for.

One of the downsides of maintaining a version history of all your master-data entities is that even simple queries have to deal with versions to retrieve the correct version of the data. One way to simplify this is to create a view that exposes the latest version of all objects, so that users who care only about the latest version can write simple queries and only users who need a particular version need to deal with the versioning complexity.

Another alternative solution that also may reduce the management overhead of the hub database is, instead of inserting a new row into the master-data table when a record is modified, to actually modify the master record in place and put the old version into a history table. This can make your master-data tables orders of magnitude smaller, in addition to making non-version queries simpler to write. Because the historical data is accessed less often than the latest version, it can be stored on slower, cheaper disks—reducing the overall cost of the system.

Another problem the history-table approach solves is what happens when the master-data schema changes. For example, when you add columns to the customer table, what value do you put into the new rows for old versions of the customer record that did not include the columns? Or, more importantly, if you drop a column, what happens to the information stored in older versions? With history tables, each schema version can be stored in a separate history table with the schema that was in use at the time the rows were created. This obviously makes queries against historical data more complex, because you will need to know which table contains the versions you want, but it provides a more accurate representation of history—another trade-off to consider.

The final option for representing versions is to use change records similar to the deltas maintained in a source-code control system. In this scheme, the current version is stored along with a log of the changes done to arrive at the current version. To obtain a past version, you start with the current version and undo the changes from the log until you arrive at the version you want. This is obviously much more complex than the previous options, but the total amount of data stored in this case is much less. You should not consider this model if you need to do a lot of queries against previous versions, because they can be very expensive. For example, obtaining a list of product prices for all products as of December 2 of two years ago would require rebuilding every customer from the change log.

Hierarchies

For purposes of this chapter, *hierarchy management* is defined as “the ability to define and store relationships between master-data records in the MDM hub”. Relationships are a critical part of the master data: Products are sold by salesmen, employees work for managers, companies have subsidiaries, sales territories contain customers, and products are made from parts. All these relationships make your master data more useful.

Many relationships exist in your current systems. For example, your HR system may track who works for whom or which organization pays your salary. Other relationships may be possible to define only because the MDM hub integrates the data from multiple systems. For example, linking a customer in the CRM system to a service contract in the customer-service system may be difficult to do if the systems are not aware of each other; but if both the customers and service contracts are stored in the MDM hub, a link table can be defined to track this relationship.

Some hierarchies are special-purpose or temporary. For example, if your development teams are organized in a matrix structure, expenses and salaries may be rolled-up to a management structure for budgeting and to a project structure for time and expense reporting.

MDM hierarchies should be named, discoverable, versioned, governed, and shared. For example, if I want to know how expenses for the XYZ project are rolled up or who reports to John Smith, I should be able to select the appropriate hierarchy from a list and know whether it is authoritative and when it took effect. This means that everyone who looks at project expenses will use the same structure, instead of everyone using whatever spreadsheet they happen to find. This also means that if an auditor wants to know who was working on the project on November 2, 2004, there is a single authoritative place to find the answer. CEOs love this stuff, because it tends to keep them out of jail.

To support relationships between entities without requiring changes to the entities, most hierarchies are implemented as link tables. If the data already contains relationships imported from the source systems, it generally makes sense to leave those relationships alone to maintain the fidelity between the MDM hub and the source system. But you may decide to convert them to hierarchies implemented as link tables to take advantage of the hierarchy-management features of the hub, as well as to provide a standard format for hierarchies.

Figure 8 shows a simplified view of what a hierarchy-management data model might look like.

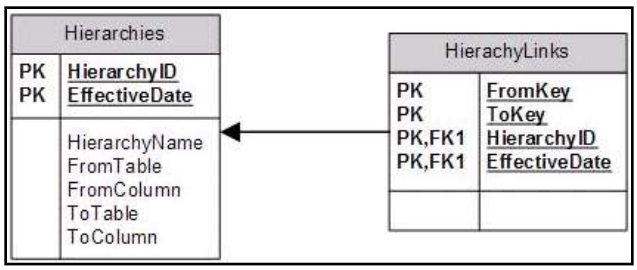


Figure 8: *Hierarchy link table*

In reality, there would be quite a bit more metadata about the hierarchy and probably more properties on the link-table rows. Whether you implement all hierarchies in the same table or create a table for each hierarchy will depend on how uniform and how big your hierarchies are. One hierarchy per table is the correct way to model it, from a relational-theory standpoint; but if you have hundreds of fairly small hierarchies, combining them may simplify database maintenance. There are a number of intermediate options, too. For example, you may group all the hierarchies that use the same pair of keys into a single table or group them by use—accounting in one table, HR in another, and CRM in a third.

Population and Synchronization

You should now have a good understanding of the architectural issues around deciding what your MDM hub database looks like and what kind of data is kept in it. In this section, we will discuss how to populate the hub with good, clean data and how to ensure that the data stays clean and consistent. This involves populating the hub database with data from the source systems initially and—with the exception of a pure repository-model hub—keeping the source systems synchronized with the hub database as the source systems make changes to the data.

Batch Loading: ETL

The initial population of an MDM hub is very similar to populating the dimension tables in a relational data warehouse. In many cases, the same Extract, Transform, and Load (ETL) tools used for data-warehouse loading can be used to populate the MDM hub. Many MDM implementations use either standard ETL tools or tools derived from ETL tools. A typical load process involves the following steps:

1. **Extract the data from the source system.** This should probably be done one subject area at a time, to make things easier. This is that part of the process that may require either buying or building an adapter that understands the data source. Again, the same adapters that are used to extract dimension data for data warehouses should work here, unless you are using a tool that is not compatible with standard adapters. This is basically a batch operation, so many tools will extract into a flat file, while others will extract directly into the ETL pipeline.

2. **Transform to the hub data model.** As part of the hub-design process, a data model was defined along with a mapping from each source to the common hub model. This step in the process makes the necessary changes to transform the master-data entity from the application data model to the MDM hub data model. This again is standard ETL stuff that might include changing column names, changing field sizes, changing formats of things like telephone numbers and addresses to match the standard formats for the MDM hub, combining columns into a single column, and parsing a single column value into multiple columns.

3. **Check for duplicates.** This process is the "secret sauce" of most MDM systems. It is both the hardest and most important part of populating the MDM hub. If you want a single view of your customer or product data, records describing the same business entity must be combined into a unique record for each unique entity; but if your MDM system is too aggressive in finding duplicates, entities might disappear when they are incorrectly determined to be already in the system. For example, your duplicate-detection algorithm might decide that George W. Bush and George H. W. Bush are the same person, so information about one of them might be lost. This is one of the reasons that both versions of the record should be stored in the version history, so this kind of error can be corrected if necessary.

   Some duplicate-checking algorithms are fairly simple and check for things like alternate spellings and missing words—for example, John Smith, Mr. John Smith, J. T. Smith, and so forth. While these are adequate for reasonably small databases, the potential for false matches is high. More-sophisticated algorithms might check for people at the same address or with the same telephone numbers. Other systems might use external data like telephone-directory data or Dun & Bradstreet listings to find matches. Many tools specialize in certain kinds of data—medical-patient data, consumer goods, or auto parts, for example. If there is a tool available for the kind of data you work with, these specialized tools can provide very accurate matching. Other tools are more generic and often allow you to specify your own matching rules to improve the matching for your specific data.

   Almost all of the matching tools provide a "degree of confidence" number for each match they detect, and your loading process should specify what confidence level is required for a match. For example, you may decide that a 95 percent confidence level is enough to automatically match an entity, confidence levels between 80 percent and 95 percent should be marked for manual processing, and levels below 85 percent are not considered matches. What values you choose will depend on the consequences of a false match or a missed match. If the result of a mistake is sending two marketing brochures when one

would have been adequate, the confidence level does not have to be high; but if a mistake results in someone getting arrested for tax evasion or treated for the wrong illness, it's good to be very sure.

4. **Load the MDM hub database.** If the new record is not already in the hub database, this is just a matter of inserting the data into the correct tables. But if it is a duplicate, the load process must check the business rules for this entity to decide what data to update with the incoming record. For example, if there is no shipping address in the current record and the incoming record includes a shipping address, the address is added. If there is already a shipping address and the incoming record also has one, there must be a rule specified to decide which one to keep or if both should be kept. If the business rules can't resolve the conflict, the incoming record should be put on a queue for manual processing. If the MDM hub is a registry or hybrid model, even if none of the data from the incoming record is used, the key of the record should be added to the database to record the connection from the hub record to the source record. This may be used by queries to find the source record or by the hub to publish hub updates to the source systems. See the next section for more on this.
5. **Update the source systems.** If loading a new record changes the hub database, the change may need to be propagated to one or more of the source systems. For example, if a new, authoritative shipping address is added to a customer record, other applications that stored information about that customer may want to use the new address. I say *may*, because there are cases where an application needs to continue with the old address and ignore the new address. I will cover this process in more detail in the synchronization discussion, but I just wanted to mention it here for completeness. As I said at the beginning of this section, if your MDM hub uses the repository model, it will replace the databases in the source systems, so this step is unnecessary.

The process of loading the data from a source application into the MDM hub can take a long time, if there is a lot of data and if a significant amount of manual processing is required to resolve data-quality issues. In many cases, it is wise to load a source application into the hub and then run for a few days or weeks to ensure everything is working correctly before loading the next application. The load process works best if the most authoritative and complete data sources are loaded first, so that subsequent loads make relatively few changes to the existing hub data. Primarily, however, it's best to record duplicates and synchronize the application data with the hub data. Loading the most critical databases first also leads to earlier time to value, which can be important in justifying the MDM investment.

Synchronization

Now that the MDM hub is populated with a single authoritative version of your master data, you need to develop a process to keep it clean and authoritative. This means implementing a method for changes to existing data and new master-data items to be transferred to the MDM hub, while maintaining the same level of data cleanliness that you achieved while loading the hub from the source applications.

One way of maintaining the MDM hub database is to keep any of the source applications from making changes to the master-data entities and thus force all additions and updates to the master data to be done to the hub database. This is the easiest technique to implement and manage, because only one database is updated and all updates can be closely monitored and controlled to ensure conformance to business rules. The primary difficulty with implementing this technique for maintaining the master data is that it requires that none of the source applications make updates to the master data. For example, nobody can add a customer to the CRM system and nobody can change a product definition in the ERP system. All changes must go through the new MDM system.

In many organizations, the retraining and operational changes required to make this work are unpalatable. On the other hand, if this MDM project is part of an SOA initiative, implementing new services to manage the master data can be incorporated into the overall SOA project. I will not spend a lot of time on how to build this service, because it is generally a pretty basic data-maintenance service. If you have access to the source systems, you might want to use a modified version of the best master-data maintenance procedures you currently have or, at least, use the business rules and validation logic from the source systems.

The one thing to remember here is that having a single master database does not mean you do not have to worry about duplicates. It's still possible for a user to create a new entity instead of modifying an existing one (and, in some systems, it is actually easer to create a new entry than to find and modify an existing one), so the MDM hub service must still check for duplicate entries.

If moving all master-data maintenance to the MDM hub is technically or organizationally impossible, you can consider a synchronization process that transfers changed master-data records from the source application that made the change to the MDM hub. The MDM hub then processes the change using much the same logic that was used originally to populate the hub. This introduces the possibility of conflicting updates and inserts from multiple systems, and it introduces some latency between the time a change is made and when it shows up in the MDM hub database; so the business must understand the limitations of this system.

In most systems, the rate of change to a given master-data entity is fairly low, so update conflicts should be pretty rare and thus reasonable to resolve either manually or with simple business

rules. This is especially true for data attributes that represent real-world entities. For example, the chances of two conflicting changes to a customer's telephone number or address happening the same day are pretty remote. To further reduce the chances of update conflicts, you might introduce the concept of a preferred source for data. For example, if it's not feasible to change the product-information–maintenance process to use a new service for maintaining product data, it may still be possible to limit the maintenance of any given product to a single system. This eliminates update conflicts, without requiring a total revamping of the product-maintenance process.

The most significant technical challenge in transferring master-data changes from the source applications to the MDM hub is detecting changes in the source system. If you have access to the source system, you may be able to add a little logic to send each master-data change to the MDM hub as it is made to the application database. Another option is to use database triggers to detect changes, if you have enough understanding of and control over the application database to do this. Replication might also be a good alternative, if the entities are simple enough that you can determine what the entity change was from the replicated data.

Unfortunately, you may find that none of these options works in your situation, so you might have to resort to periodically querying the application for changes or even parsing audit logs to find changes. After you have detected a change in the source system, it should be sent to the MDM hub as quickly as possible to reduce the update latency. I generally recommend reliable messaging for this task, to ensure that changes are not lost in network or system failures. Microsoft BizTalk and Microsoft SQL Server 2005 Service Broker are probably the best alternative for this on the Microsoft platform; but because the source applications can be running on a variety of platforms, other alternatives may be appropriate. On the other hand, if you are using the MDM Hub primarily for reporting and hierarchy management in a business-information (BI) environment, latency might not be a big issue; so loading changes into the hub database with batch-oriented MDM tools will provide adequate data freshness, with significantly less overhead and complexity.

Figure 9 shows a CRM application adding a customer to the MDM hub by calling the CreateEntity service.

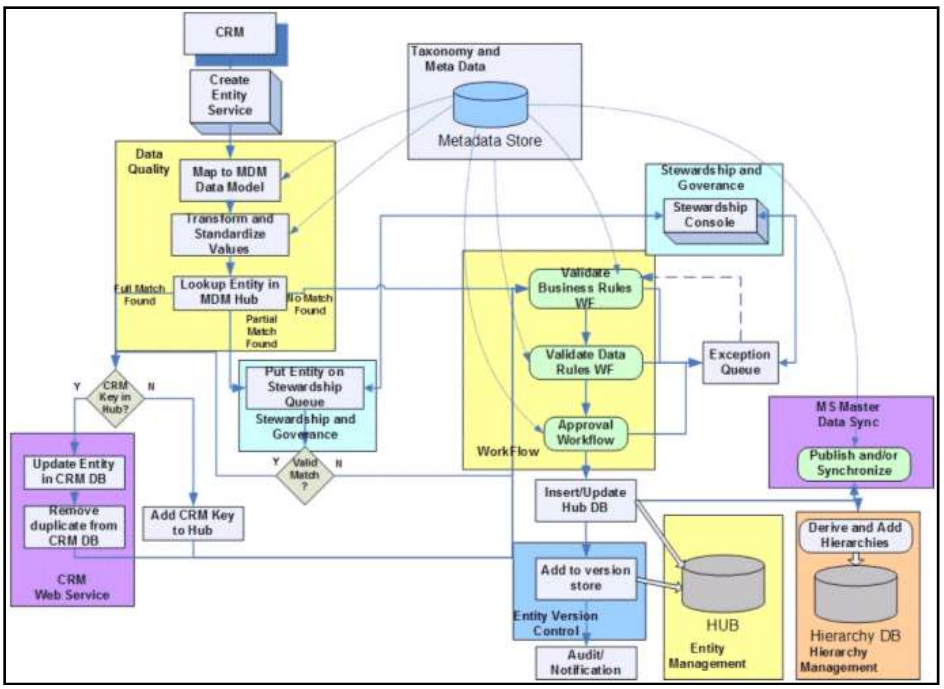


Figure 9: *CreateEntity service*

Adding a new customer to the MDM hub will typically require six steps:

1. The incoming data is mapped to the MDM data model using the same transformations used in the ETL process described earlier. This makes checking for duplication easier and puts the record into a common format that can be used throughout the rest of the process.
2. The hub looks up the entity in the hub database to see if it is already there. This is not a simple SQL query; it does all the fuzzy-matching logic that the duplicate-elimination process did when creating the hub database. For this reason, it's good to look for a tool that can look for duplicates in batch mode and also do the lookup one entity at a time. As I explained in the ETL section, there are three possible outcomes of the search: duplicate entry found, no entry found, and do not know. If the answer is do not know, the entity is put on a queue for the data steward to resolve (stewardship will be covered in a later section).
3. If a duplicate is found, another application has already added this entity, so this insert will be changed to an update. The entity in the hub is checked to see if there is already an entry from the CRM application for this entity. If there is, this entry is a duplicate in the CRM database; so the entity already in the CRM database is updated with the new data, and the entity that the CRM application is trying to add will be deleted to eliminate the duplication. On the other hand, if the entity in the MDM hub does not currently have a key

for the CRM application, the key for the incoming entity is added to the hub entity, and the incoming entity is passed on as an update to the approval workflow.

4. If no entry was found in the MDM hub for the incoming entity, it is passed to the approval workflow as an insert. At this point, the three streams converge again, and an automated workflow checks the data update or insert to verify that it meets all the business rules for the MDM hub, as defined by the data-governance activity of the enterprise. Some examples of business rules might be which fields require values, allowable value ranges, address verified with an external vendor, Dun & Bradstreet (D&B) number valid for this business, and prices in the correct currency. At some point, if there is not enough information to determine if a rule is satisfied, or a rule determines that manual approval is needed, the entity will be placed on the stewardship queue for manual approval.
5. If the entity passes the approval-workflow process, it is inserted or updated in the hub database as required. In the same transaction, the version information for this entity is updated with the previous values. If the entity contains information that can be used to derive any of the managed hierarchies for this record, the required entries are inserted in the hierarchy database. For example, the customer entity may be linked to a support-contract entity and a sales-territory entity based on **contract-id** and **territory-id** fields in the customer entity.
6. When the entity has been added to the hub database, the changes are published out to the other source systems. In some MDM systems, this means publishing every change to every system; but, in most cases, only a subset of the source systems are interested in changes to a given entity. For example, if your e-commerce system has hundreds of millions of customers, it probably does not make sense to push them all into the CRM system. A set of business rules can be used to determine which applications receive new master-data updates, based on entity type or source. Another alternative is to send updates to data only if the source application has a key in the MDM hub. In that way, an entity has to be added to the application before it is managed by the MDM system. The various methods of publishing updates back to the source applications are described in the next section.

Publishing Updates

The first architectural decision you must make about publishing updates is whether you need to do it. Some MDM systems are used to provide a single source of master data for enterprise reporting or performance management and do not require all the source applications to use the new master data. In many organizations, the political ramifications of an MDM system directly updating one of the key enterprise applications will prevent automatic propagation of updates to

some systems. On the other hand, creating a clean source of master data is a significant effort, so it seems like a waste of resources not to propagate the cleaned-up data to all the source applications.

If you determine that you need to publish master-data updates, the next decision is whether to push updates out to the source application or let the source applications pull the changes from the hub. Pull is generally easier to implement and manage, but push reduces the time between when the hub is updated and the updates are available in the source applications. Pull is also generally easier to implement between heterogeneous systems. If your MDM hub runs on SQL Server and one of the source systems is on a mainframe, it will probably be much easier to have the mainframe read a change file than to write an application to push changes into the mainframe application. This is the classic trade-off of capability against complexity, and the deciding factors are usually the requirement of up-to-date master data weighed against the difficulty of doing the integration.

The push option looks like replication on the surface; and, in some cases, replication may be the best way to push the changes. This works if the source application data model is pretty close to the MDM hub data model and there is a replication-connection available. If the two data models are significantly different, if replication is not available between the databases, or if directly updating the source application's database is not allowed, an integration server (such as BizTalk Server) is probably the best choice. If necessary, this can include complex data transformations and even an orchestration to do the update in multiple steps. Orchestration can also be used to publish updates selectively to only the applications that require them. For example, only CRM systems that contain a record for a customer would receive updates for that customer. If you are publishing from one SQL Server database to another SQL Server database, SQL Service Broker (SSB) is a good choice for a reliable asynchronous connection and transactional application of the required changes.

If the effort and complexity of implementing and maintaining a push solution are excessive, you may have to implement a pull solution. The simplest pull solution is to allow the application to query the MDM hub database (or preferably read-only views of the database) directly, to obtain the required data. If the amount of master data is pretty small, the application can periodically refresh its master data completely; but, in most cases, the application will want to refresh only what has changed. Time-stamp columns are the most common approach to this issue. Each application keeps track of the last time stamp it has on read-only retrieves data with time stamps greater than its remembered value. The downside of pulling data directly from the database is that, if it is done frequently by a large number of applications, it can cause significant performance degradation.

A pull alternative that makes it easy for applications to apply changes and reduces the load on the MDM hub database is to write changes into a journal or log. This can be either a database table or a flat file. If updates are sequentially numbered, an application can track which updates it has processed already. If the number of applications pulling data is relatively small, it might make sense to generate a separate journal for each application. This can be a lot of extra I/O, but it makes it easier to manage if each application can manage its own journal—by deleting the records they have processed, for example. On the other hand, you may want to maintain a journal of changes for auditing purposes, anyway, so this journal can do double duty. In a pull architecture, the application might pull updates itself or use an external tool that is either custom-written or implemented with an ETL tool to periodically read the changes and apply them to the source application. Some databases have Change Data Capture features that record all the changes to a specified set of tables in a file or table, so that a pull system can periodically read the captured changes instead of trying to determine what has changed.

It's also possible to do both push and pull, if your application requires it. For example, one of the destinations to which you push updates might be a service that writes a journal to support pull applications.

Data Integrity and Reliability

When designing an MDM infrastructure keep in mind that the complex processing that happens in the background to support master-data synchronization must be reliable. Losing updates will reduce the accuracy of the master data and can cause users to lose confidence in it. This means that, as you design the code that will handle data movement and manipulation, you must ensure that every action that changes or moves data is transactional and recoverable. Messaging should always be transactional to ensure messages don't get lost or duplicated; and all the asynchronous processing involved in the MDM workflows should be managed by transactional queues, so that it can be restarted from the previous state if the system shuts down in the middle of processing. The hub could use SQL Service Broker (SSB) to control its operations (although BizTalk Sever could provide the same reliability and make the workflow and business rules much easier to implement). There are also transactional messaging environments available in non-Microsoft environments. The point is not which tool you use, but instead making sure that, as you go through every step of the process design, you consider what happens if the system shuts down unexpectedly or a disk drive fails.

Metadata

Metadata is data about data. In MDM systems, as in any data-integration system, metadata is critical to success. As a minimum, for every column of every table in the MDM hub, there must be accurate data about where the value came from, what transformations were performed to get it

into the hub data-model format, what business rules were applied to it, and which applications receive updates to it. As with the master data itself, metadata is only useful if it is current and accurate.

For compliance reasons, you generally have to be able to prove that the metadata accurately describes the master data. The easiest way to furnish this proof is to show that the processes that handle the master data are derived directly from the metadata or vice versa. For example, the mappings and transformations done by the ETL tools might be driven directly from the metadata, or the ETL tools might populate the metadata repository whenever a transformation pipeline is created. Business rules should be taken directly from the metadata, whenever possible. This is especially important in MDM, because the business rules often determine which of several alternate values is used to populate a data element.

Reading business rules and transformations directly from the metadata repository whenever they are used can be a performance issue, so either the information is cached or the actual data-maintenance code is generated from the metadata. The MDM metadata may also include descriptions of processing done outside the MDM hub. For example, the transformations required to map the hub data model back to the source systems that subscribe to it will probably execute on the source system, but must be defined in MDM metadata.

Stewardship and Governance

Stewardship and governance are different things that often get confused with each other. There is, however, a close relationship between them.

Data governance is the process of defining the rules that data has to follow, and *data stewardship* makes sure that the data follows those rules. Poor-quality master data can affect many core applications of a business, so governance and stewardship are important functions in an MDM process.

The governance rules should include who can read, create, update, and delete data; what validity checks are required for data; which application is the preferred source for data items; how long data is retained; what privacy policies are enforced; how confidential data is protected; and what disaster-recovery provisions are required, to name a few. The data-governance function should include leaders of both the IT and business groups of a company.

The role of a data steward is to ensure that the master data is clean, consistent, and accurate. There may be cases in which data quality cannot be determined with automated rules and workflows- in this case so manual intervention will be required. The people doing the manual intervention are the data stewards. The data steward for a collection of master data should be the person who understands the data the best (not necessarily someone from IT).

The technical aspects of data stewardship involve a set of tools that help a steward find, analyze, and fix data-quality issues. These tools are generally integrated into a "stewardship console" that incorporates the data-profiling, data-analysis, and data-modification tools into a single user interface (UI). If your data stewards are business people, the stewardship console should be simple and highly automated. In organizations with complex governance rules and approval processes, workflow can be a useful part of the stewardship console. Basically, master-data updates that cannot be approved automatically are placed on a queue for the appropriate steward to resolve. An automation-supported human workflow can handle the routing and approval processes for these changes.

Data Profiling

Data-profiling tools can scan data for violations of business rules, missing values, incorrect values, duplicate records, and other data-quality issues. Profiling the data in your source systems is a good place to start an MDM project, so you can find out how much trouble you are in. Profiling can help you choose the authoritative source for data and design the ETL logic required to clean up and load the data into the MDM hub. Profiling should also be done periodically after the MDM system is in place, to find data-quality issues that the MDM system is not fixing.

Export

As soon as you have a clean, accurate source for master data, you will need to be able to export it to other systems that need it. For example, you may need to export your product master data periodically to be used in a data pool or a marketing campaign. (Most databases include tools for exporting data in a given format or XML schema.)

Reporting

Reporting includes reports on the master data itself—customer lists, product information, organization charts, and so on – including reports on the health of the MDM hub itself. Things like the number of rules violations detected, the number of manual interventions required, and the average latency of master-data updates will help the IT organization discover issues early enough to prevent major problems. A solid reporting system that produces "canned" reports and allows user to design their own reports is an important part of the MDM hub.

Workflow and Business Rules

A business rules engine is critical to the success of an MDM hub. In many cases, the rules are established by relatively unsophisticated data stewards, so a simple wizard or UI for developing rules may be required. The rules will often involve retrieving and manipulating database data, so a rules engine that has good database integration would also be useful.

Tools

An MDM project will also need data-modeling tools for recording the data models for the source applications and the MDM hub. If you have a repository, the modeling tool should integrate with the repository. ETL tools for loading the hub data, data-quality tools, profiling tools, and application-integration tools are also required for loading and synchronization. If your hub uses a registry or hybrid model, a distributed query tool may be needed for queries against the master entities when some parts of the data are stored in the source systems. A tool for defining and maintaining hierarchies will be required for hierarchy management.

Conclusion

In this chapter we provided an overview of some of the common issues associated with data management and integration for SOA. We reviewed a number of scale-up, scale-out, replication and database partitioning strategies. We closed with a review of MDM and review of how an MDM hub can assist in your data management strategies.

MDM is a fairly straightforward application of technologies that are probably already available in most large organizations. If your organization already has a data-administration function or at least some reasonably competent data modelers and data architects, you probably have the skills you need to succeed. Most large organizations have massive amounts of master data, so getting it all under control will take a while. Start with something that has limited scope, but also something that has a lot of value to the organization. Early success is important not only to get management to continue the project, but also for the project team to gain the satisfaction and confidence that deploying something that has significant impact brings. As soon as you have one part of the problem solved, learn from the experience, and repeat the process as many times as necessary to complete the MDM process.

Chapter Five provides a detailed discussion of the **User Interaction** architectural capability.

SOA Case Study: London Stock Exchange

The London Stock Exchange plc (the “Exchange”) is Europe’s largest exchange and operates the world’s biggest market for listing and trading international securities. It wanted to invest in a new market information and data access services system to deliver better and richer real-time price and value-added information to the market. The Exchange has an enviable record for reliability – its trading systems are at least as resilient as any other system used by major equities exchanges around the world. To deliver new real-time price and value-added information to market systems and consumers, the Exchange decided to enter a partnership with a best-of-breed technology provider and a global management and technology consulting firm.

The London Stock Exchange needed a scalable, reliable, high-performance, stock exchange ticker plant to replace an outmoded system. About 40 per cent of its revenue comes from selling real-time market data. The Exchange’s market information and data access system solution draws upon a newly created corporate data warehouse. This is a central data repository that gives customers access to a wide range of current and historical data going back thirty years. The system also consumes raw trading data and publishes it in real-time.

The Exchange wanted not only a scalable solution but also a high performance application with messages delivered in order, with consistency and with sub-second timing to its many customers. To ensure present and future scalability, the system uses parallel scaling over multiple CPUs per server, and multiple servers with Microsoft Clustering between servers.

The entire Case Study is available online at
http://www.microsoft.com/windowsserver/facts/casestudies/lse.mspx .

See other SOA case studies at
http://www.microsoft.com/casestudies/search.aspx?Keywords=SOA.

References:

1. "Portal Integration Pattern" available at http://msdn2.microsoft.com/en-us/library/ms978585.aspx
2. "Scaling Out SQL Server with Data Dependent Routing" by Man Xiong and Brian Goldstein. Available at http://www.microsoft.com/technet/prodtechnol/sql/2005/scddrtng.mspx
3. "The What, Why, and How of Master Data Management" by Roger Wolter and Kirk Haselden. Available at http://msdn2.microsoft.com/en-us/architecture/bb190163.aspx
4. "Master Data Management (MDM) Hub Architecture" by Roger Wolter. Available at http://msdn2.microsoft.com/en-us/library/bb410798.aspx#mdmhubarch_topic5

Chapter 5: User Interaction

"Design is inevitable.
The alternative to good design
is bad design, not no design at all"

- Douglas Martin
Author

Reader ROI

Readers of this chapter will build upon the concepts introduced in previous chapters, specifically focusing on the User Interaction architectural capability.

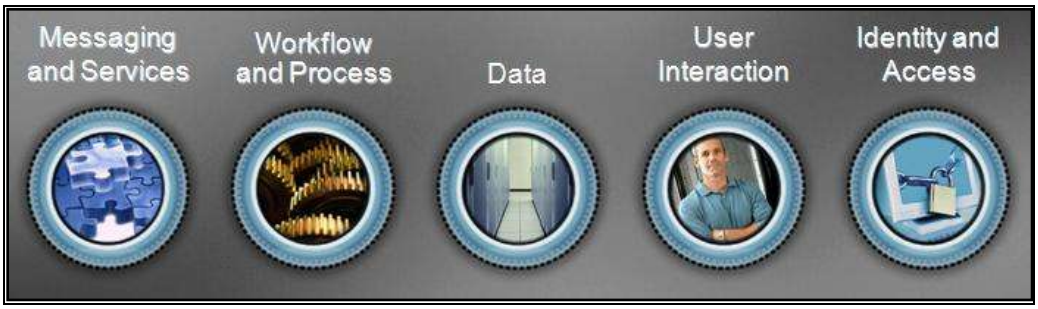


Figure 1: *Recurring Architectural Capabilities*

The User Interaction architectural capability focuses on the improving the user experience (UX) associated with applications and solutions.

Topics discussed in this chapter include:

- The impact of bad user experiences (UX)
- Prioritizing UX design
- Cognitive Friction
- A Framework for UX

Acknowledgements

This chapter is based upon presentations and keynotes given by Simon Guest.

What is Architecture?

Someone once said “architecture is the balance between art and engineering”. If we think about building architecture, such as the Acropolis, you can see the blend of art and engineering there. It’s beautiful, but a lot of engineering principles and concepts have gone into it.

Does the same apply to *software* architecture? Is there a balance between the art and engineering here as well? The engineering is obvious, most architects you talk to today all have different engineering concerns whether SOA, ESB, or a database. A lot of architecture today is focused on engineering and infrastructure. So the question isn’t “What is Architecture”, the real question is “Where is the art in software architecture?” Is there an art and does the analogy ring true?

Some people would say that the art is in the beauty of the design, or the beauty of this particular Visio diagram. This is incorrect - art should be about the user experience (UX). It should be the interface we give to users and the applications we give to them. This brings up the core problem: UX often comes last in most projects.

There are dedicated architects or teams that will work for 80-90% of the project on solution infrastructure, making sure the Web services are working well and the database is consistently available. Towards the end of the project schedule there is a mad scramble to move the solution into production when someone suddenly realizes the user interface still needs to be developed. This results in another scramble knock together a simple user experience (UX) in a week or so. The UX is the user facing component of the solution – why shouldn’t more time be spent on it?

We have all seen good examples of bad UX. Bad UX takes multiple forms - green on black, black on green, buttons that should be links, links that should be buttons, dialogs that do not make any sense, and confusing messages in error boxes. Figure 2 is a good example of a bad UX.

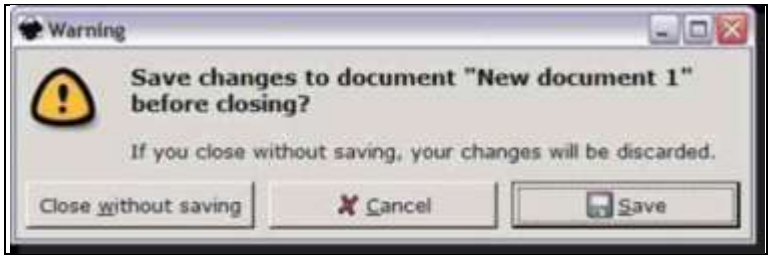


Figure 2: *A good example of bad UX*

Which button should one click? This type of confusing dialog frequently appears in both off-the-shelf and custom solutions. Error messages are problematic because reliability is key factor in UX - having problems pop up at the wrong time can detrimentally impact the user’s opinion of the solution.

Author and consultant Alan Cooper has coined the term “Cognitive Friction” to explain this problem. By providing a bad user experience we are introducing cognitive friction to the user’s work day. We are making it harder for users to think about how they should be using our applications - when they shouldn’t have to think about using them in the first place.

Why is UX always a low priority on most projects? Most organizations know their environment, their users and the hardware and software deployed across the organization. Many organizations also have a dedicated development staff. These are developers whose full time jobs are creating applications that deliver value to the organization. Most organizations also have stable business capabilities (e.g. the products being manufactured this week will most likely continue to be manufactured next week, next month, even six months down the line). These are typically not organizations that are manufacturing products one week and morphing into a services organization the next. Applications tend to mirror fairly standard, fairly stable business capabilities.

Some of these organizations react to bad UX by hiring designers. Unfortunately the designers are usually brought in late in the project, long after major UI decisions have been made. Designers should be added at the *beginning* of a project to maximize the impact they can have on the UX.

In some organizations bad UX causes users to become apologetic. Users feel the need to apologize for not understanding how to use the applications that IT has delivered to them. If a user doesn’t understand how to use an application, the most likely problem is the UX, not the user.

Introducing a Framework for UX

We need to fix this problem - we need a new perspective. We need a new way of thinking not only for UX, but for UX when talking to this architecture audience. The goal of this chapter is to introduce a framework you can use when facing these problems. The UX Framework is illustrated in Figure 3 below.

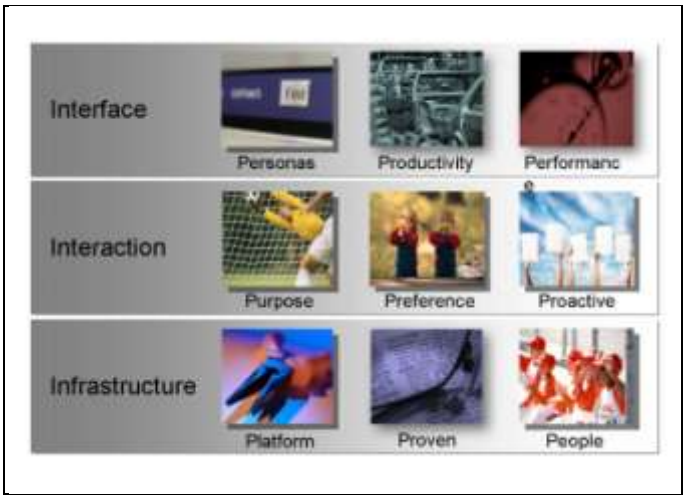


Figure 3: *A framework for thinking about UX*

One way to build a framework for UX is by asking users questions about their problems and concerns. Last year we conducted an internal study by asking users these types of questions – we received some fairly surprising answers. Here is a couple of the responses we received:

> *"Is this application designed for me or an engineer? I don't understand how to use this."*
> *"The performance is horrible. This application is totally unusable."*

Let's take a detailed look at the layers in the UX framework.

Figure 4: *The three aspects of the Interface layer*

Interface

Many of the types of questions we were asking could be categorized as "interface". The Interface is the divide between what people see on the screen and what users actually do. There are three aspects to consider within the Interface category: Personas, Productivity and Performance. We examine these aspects in greater detail below:

Personas

When an IT group is thinks about UX they will stumble across the concept of a single, generic user (e.g. *"The user won't do this, what the user really wants is this."*). This means that everyone on the development team has their own idea about who this user is. The UX Framework recommends using personas instead of a generic user. A persona is a fictitious person that is based upon as many real-world characteristics as possible. Here is a good example:

Sally is a “Road Warrior,” - here is a summary of her persona:

- 38, Married
- 2 Children
- 5 years of computer experience
- Mostly Windows/Office, experienced with PowerPoint
- Has between 10 and 20 clients – mix of small and medium businesses
- Engages in pre-sales
- Relies on laptop and mobile phone device

Once a summary is developed it’s easy to build up a picture of what your personas can and cannot do. By using personas in your project you tend to focus upon the things that really matter to your users. For example, if we are designing a call center application and we know Sally spends most of the day with customers – we also know she needs to be connected by a mobile phone or via Internet kiosks.

We can also develop other other personas – typically you will develop multiple personas for your solutions. Here are three additional examples:

Derek is a call center operator – here is a summary of his persona:

- spends 8 hours/day talking on the phone with mostly aggravated users
- needs to be super-productive - If he can access call records more quickly he can shave a few seconds off each call, servicing more customers.

Jim is an executive. – here is a summary of his persona:

- spends 8 hours a day thinking about customers, looking at the big picture
- thinks a lot about what should be done next, what kind of strategy the organization should take
- needs an abstract view, providing top level information as quickly as possible

Peter is a systems administrator – here is a summary of his persona:

- a member of the IT group
- going to have to support our application
- responsible for keeping everything up to date
- spends his entire day supporting users
- needs to do very complex things using the smallest amount of time (using scripts if possible)

Why not simply use real users? Unlike real users, personas let you abstract a little, changing the personas profiles to meet your application needs. It’s also important to realize that real users often do not really know what they want. While this may sound somewhat pompous it’s actually true – users are often not able to articulate their needs and do necessarily not have a good grasp of what goes into a good UX. Personas enable us to be abstract and think out of the box about UX. Personas let us move from conversations of “Most users will have access to a printer on that screen,” to “Derek will have access to a printer as he works in the call center, but Sally works on the road so may need another option for printing.” Personas make you think about different people differently. Personas enable us to move from conversations like “This would be quicker to develop as a Web app.” to “For Sally and Peter, a Web interface may make sense. But Derek has

a lot of keyboard shortcuts he's used to, so maybe we should think about something different for him?"

Persona Recommendations

1. Defining Personas should be the first step for any design experience.
2. Personas can help resolve arguments in the development team and help influence which bugs should be fixed or which features should be cut.

Productivity

When you deliver a new piece of software to a user you normally see a productivity curve similar to the one shown in Figure 6. Initially, productivity dips as the user adjusts to a new application. Eventually the user becomes more familiar with the new application and their productive rises as a result of using the application (preferably to a level higher than before the application was deployed).

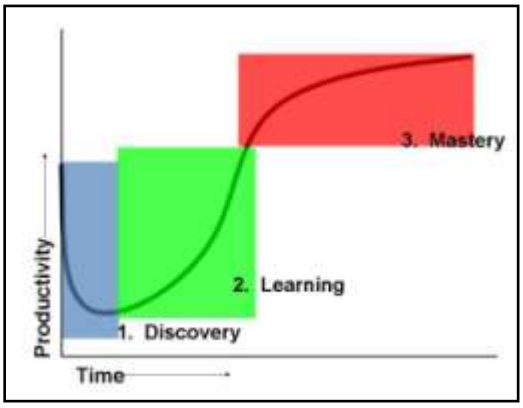


Figure 5: *Productivity versus time for the UX*

This productivity curve in Figure 5 is defined around three stages:

- Discovery – playing around with the features
- Learning – getting comfortable with the new application and integrating it with how you do your job
- Mastery – increased productivity from new application use

The productivity dip in Figure 6 can be reduced by making applications more familiar to our users, thereby reducing the level of Cognitive Friction. A well-designed UX can reduce the productivity dip that typically occurs when a new application is deployed (Figure 6).

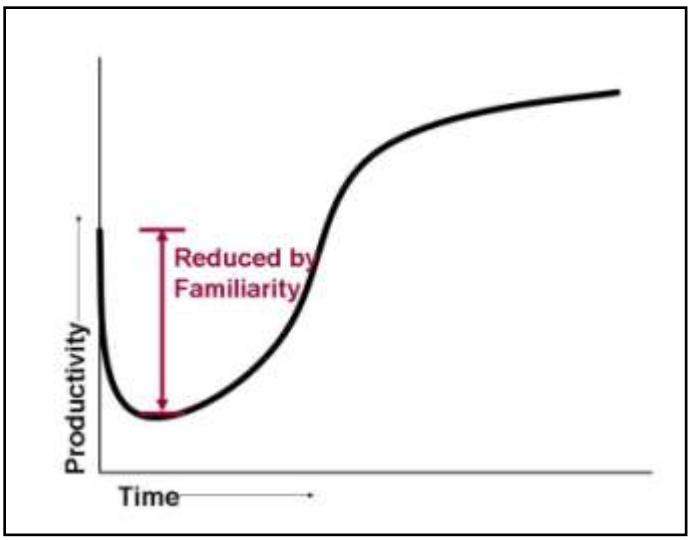


Figure 6: *Reducing the productivity dip with familiar elements*

We can see evidence of reduced productivity dip by comparing UX from unrelated websites. Many websites use tabs, search and shopping carts – this makes it easier for a user to interact with a site, even if they have never visited the site before. Listed below are a two such sites:

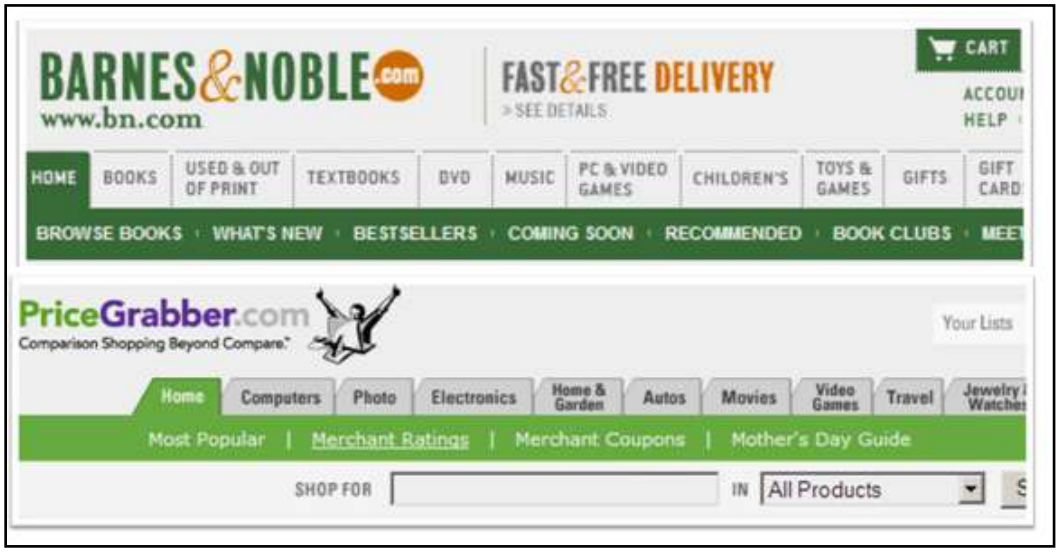


Figure 7: *Numerous sites use familiar elements*

You don't need to read a manual to use these sites because the concepts they use are familiar. As architects we often forget familiarity. We look for beautiful and obscure components and build interfaces that no one has ever seen before. We need to address this problem and figure out how to make things more familiar.

Figure 8 illustrates a typical scenario – an organization using Seibel CRM was unhappy with the default Siebel client.

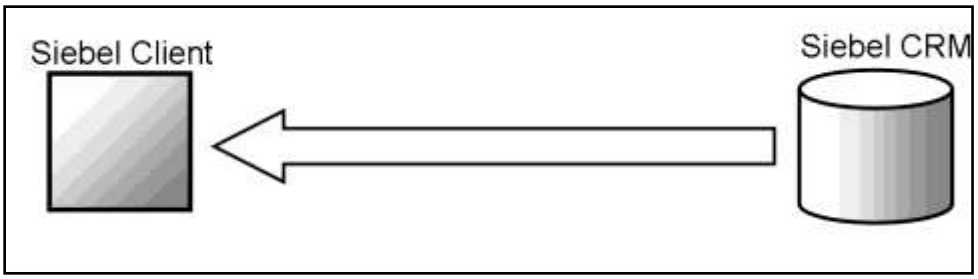


Figure 8: *A typical application*

The customer went through an internal project to move all of the CRM data to the Web using a Web services layer in front of the Siebel database. They then exposed their data using a custom ASP.net interface (Figure 9).

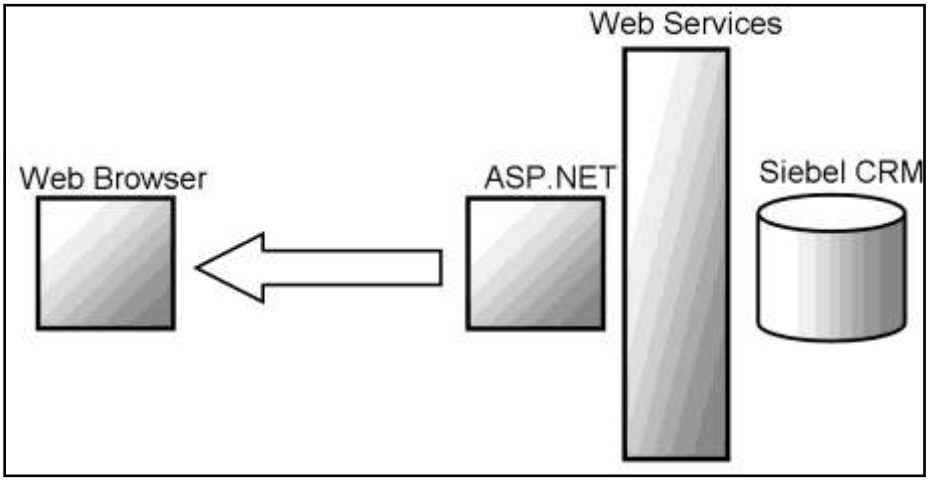


Figure 9: *A first attempt at a better UX*

Soon after the application was deployed the customer learned that, despite being the latest technology (SOA, Web), nobody was using it. The customer later learned that their traders were using an entirely *different* approach to working with the CRM application. The traders would take a call, and enter the information into Outlook instead of re-entering information into the new Web application.

The project was redesigned to feed Web services directly into Outlook. This initially presented some interesting technical challenges. The Siebel CRM data was used to create a set of objects within Outlook. These objects looked and acted like normal Outlook folders, contacts, and calendar items. The end result was a CRM interface that was better integrated into how the traders actually performed their work.

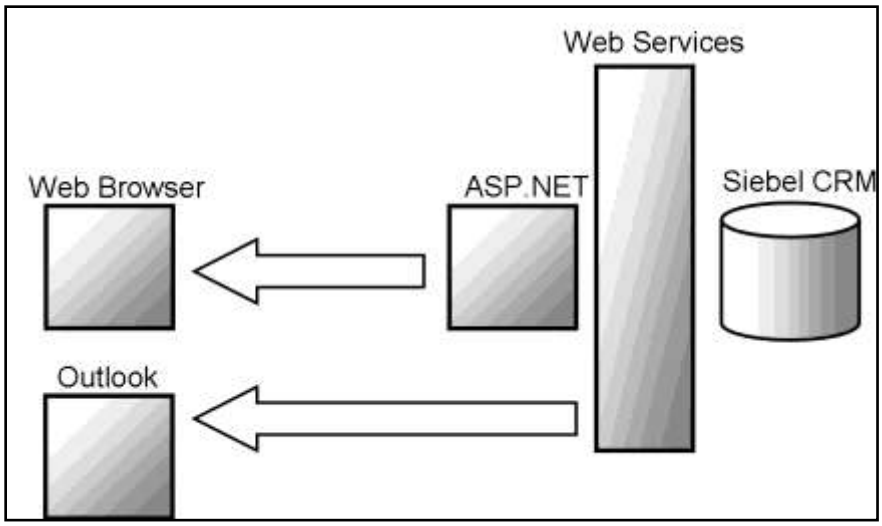


Figure 10: *Refactoring the application into a familiar interface*

The new approach avoided the productivity dip mentioned earlier because the UX was implemented using the familiar environment of Outlook. Training time went from 1.5 days for the Siebel client to 20 minutes for the new Outlook integrated UX. People already knew how to make contact, they knew how to drag-and-drop calendar items around, so they did not need training, they just needed to be given the application. All they needed were the clients and they were good to go.

If we go back to personas for a moment, it is worth noting that we can take the personas and map them to the productivity curves. Consider the Sally persona. Sally has been using computers for five years - she is going to experience an average drop in productivity for this application because of her relatively low level of computer experience. The Derek persona represents a much more experienced computer user. Derek is much more computer proficient and will have a much smaller dip in productivity. The Jim persona may experience a deeper dip in productivity depending up his familiarity with Outlook. Jim may actually give up trying to use the new application - he is a very busy guy with little time to learn new applications. Perhaps we can better help Jim by taking a closer look at his profile and deliver a different, more streamlined interface (such as a wizard, portal or similar UX that would be more valuable to him).

Productivity Recommendations

1. Often the most productive user experiences are not the most elegant or best looking. Mainframe green screens were not very pretty but the keyboard shortcuts used by such screens enabled users to be far more productive than a sleek, mouse-driven GUI.
2. Define the personas that make sense for your application and map productivity curves to your personas to help determine any issues or UX problems you might experience.

3. Look at existing experiences to see if there is anything you can extend. If people are already using Outlook, or Office or another application, is there anything you can tie into there to make them more productive rather than just starting from scratch?

Performance

The third and final aspect of the Interface layer is performance. Performance is interesting because a lot of performance analysis tends to focus on engineering (e.g. "we need this service and this client to have a latency of *x* milliseconds" or "there is no way we could survive having more than 1.2 seconds between these updates."). While these types of service level agreements (SLAs) may be true for industries such as financial, or manufacturing, performance is often about tolerance, not latency. Tolerance in this sense means focusing on what the users will or will not tolerate. There are two types of user tolerances in most projects:

- *Domain-specific tolerance* is when people know that something should happen quickly (or quicker) and does not. Domain experience enables users to know how the tools they use should work – if they do not work in an expected manner they will begin asking why the application is so slow.
- *Non-domain specific tolerance* is when a user does not really know what is going on. Tolerance for non-specific domains is greater, causing users to eventually start asking how long the application will take to respond.

When looking at performance and applications, be sure to look at the numbers and see if there are service levels that must be met. It's also worth reviewing the application components from a tolerance perspective. How tolerable is this user or persona going to be for this particular function?

We can sometimes address tolerance issues on the front end using technologies such as AJAX (asynchronous Javascript and XML). AJAX is a set of Web techniques that enables more interactions and granularity in the browser. Without AJAX web pages are rendered directly in the browser – updates to the UX required expensive post-backs to the server. With AJAX, the page gets rendered *to* the browser, but behavior gets handled in the background. We can give people more feedback and show them what is going on in a much better way. It's interesting to note here that Microsoft released the Microsoft Remote Scripting Toolkit in 1999, effectively supporting the same technologies, concepts and architectural approaches used by AJAX today..

Performance Recommendations

1. Avoid getting drawn into conversations about "millisecond" response times. While "sub-millisecond" response times may well be a valid SLA, UX perspectives are more about

user tolerance instead of application latency. There may be areas of the UX that can be modified to enable greater user tolerance.

2. The principles behind AJAX stretch well before 2004. The acronym is fairly new but the technology behind it is not.

Figure 11: *The three aspects of the Interaction layer*

Interaction

Interaction is about not what the user interacts with, but how they interact. The Interaction layer of the UX framework focuses on the following types of questions:

"Why doesn't this application help me get my work done?"
"If it wasn't for this app, I would do this differently." (forcing people to work a certain way)
"Okay, something's gone wrong, whom do I call?"

These types of problems fit into a middle tier called Interaction. The Interaction layer focuses on three aspects Purpose, Preference and Proactivity. We examine these aspects in greater detail below:

Purpose

Most applications and architects get confused by differences between *tasks* and *goals*. I need to create a new document, type some text, format some text, and send it to my editor. Is this a task or a goal? It's actually both. Each of those steps is a separate task, but my ultimate goal is to send something to my editor.

Office XP provided dynamic menus which only showed the most commonly used menu options, hiding all of the others. This was bad UX because it filtered out the ability to do particular tasks based upon how often these tasks might be performed. Office 2007 fixes this problem by applying context between each task. For example, if you launch Word and start typing some text the UI ribbon displays the text options for that particular function (bold, styles, and so forth). If you then insert a diagram, the ribbon automatically changes to display diagram formatting and editing options. Once you start typing again the ribbon switches back to displaying font formatting options.

This is a good example of how you can switch between contexts in the UI without hiding things from users. The UI ribbon can also be customized or reused by ISVs, enabling anyone to handle contexts and rules in customer applications.

Purpose Recommendations

1. Do not let customers confuse tasks and goals in applications.
2. Focus on the entire application - think about the context for each step. Check out the Office UI licensing and see if ribbons are a good fit for your applications.

Preference

Preference is looking to see if there are multiple ways to do the same thing; reaching the same goals using the right set of tasks. If a goal has been established there are probably multiple ways of getting there.

Microsoft's internal invoicing system is called MS Invoice. When you buy something from an external vendor you receive an invoice and an e-mail. To research an invoice, users must open a new instance of Internet Explorer and go into MS Market (Microsoft's vendor marketplace). Once in MS Market you must copy and paste the invoice number from MS Invoice into MS Market, then navigate through several screens to find the original order. The user must then Alt+Tab back over to MS Invoice, navigate through several additional screens, and approve the invoice for payment. The entire process typically takes 25 minutes or more.

A user's goal in this scenario was is to get more information about an invoice prior to approving or rejecting it. . The MS Invoice and MS Market applications force users to work in a specific manner. If these applications did not exist users would probably simply use the phone, Outlook, or send an instant message to the person that made the original order.

Instant messaging is becoming a favored means of communicating within the enterprise. It may be possible to being the IM UX into another context. For example, the Encarta bot is a "buddy" that you add to your IM list in Windows Messenger. Encarta bot is a scripted front end that interacts with the Encarta database. If you ask it some questions it will attempt to respond to you using the Encarta database. For example, if you ask "Who is the president?" it will ask me which country. If you reply with "USA" it will respond with "George Bush" (see below).

Figure 12: *Interacting with the Encarta bot*

What the application can then do is offer me more interaction, so it takes me to a Web page in the same context as the Web client and shows me a picture of George Bush.

We could utilize a similar UX to make invoice research and approvals easier for users. When a user checks e-mail an Invoice bot could notify the user when a new invoice exists and ask how to proceed. An example of this improved UX appears below:

Figure 13: *Using a specialized bot for a contextual UX*

This completely fits in with the context, completely fits in what we are doing. This approach moves the hard work of doing all of the look ups away from the user and into the back end (where

it belongs). This is an example of a UX that fits in with what we are already doing and what we want to do.

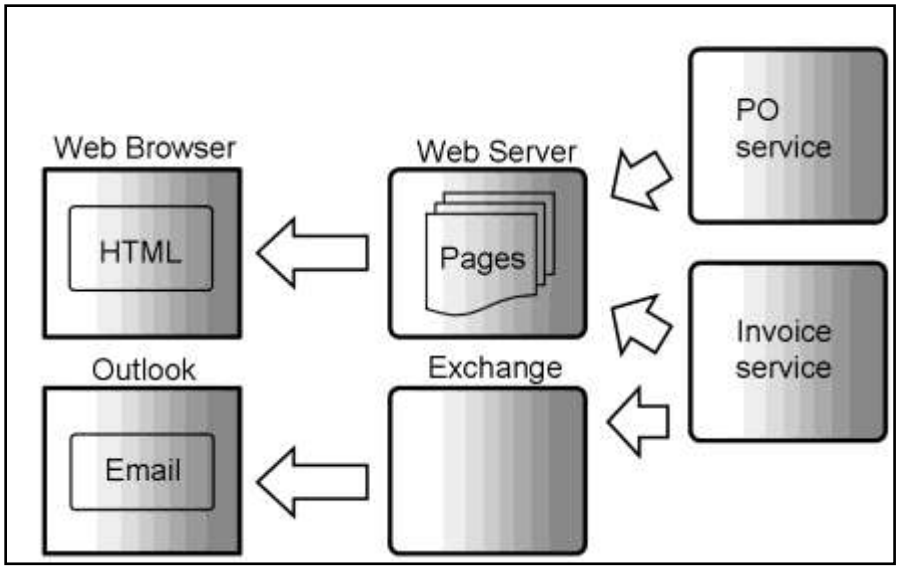


Figure 14: *The original MS Invoice architecture*

In the old system we had some services, a Web server which exposed the MS Invoice site and an Exchange server which sent out email reminders. If we move to a more natural IM solution, the architecture becomes much simpler.

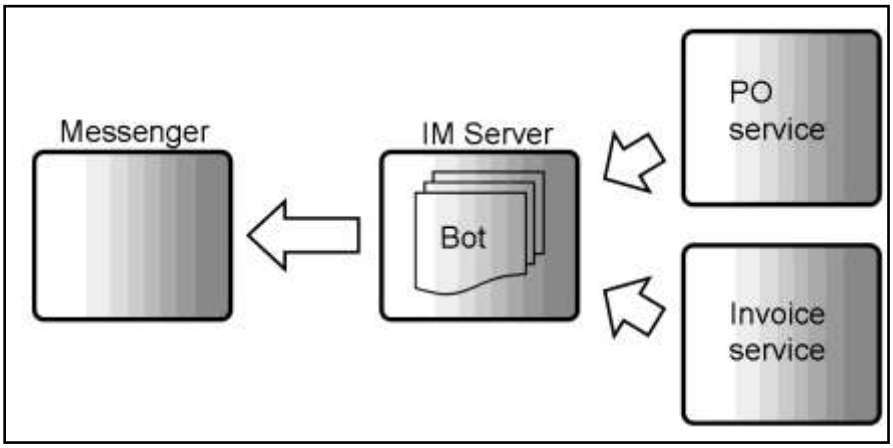


Figure 15: *The contextual bot architecture*

Preference Recommendations

1. There are many ways of reaching the same goals using a set of tasks. Instead of just diving into a set of tasks, step back and say “What is this persona trying to do?”
2. Building a set of personas will help you realize this efficiency. Thinking “This persona is not going to want to dial in from this location too often,” gets you thinking about different things.

3. Also look at applications that customers are using to see how you can integrate. So if everyone in the organization has MS Messenger installed, great! How can you use it? What can you do to kind of plug in there instead of creating something that is taking people out of their regular day?

Proactive

The third and final aspect of Interaction is Proactive. We have all been through this, we release a rev of our application and we feel really good about it. The application is a hit! Or at least we think it was a hit. We did not hear anything from our customers so we assume nothing is wrong. This scenario raises an interesting point because reactive feedback is very common. When we hear from customers its usually bad news about our applications (e.g. “this thing is broken” or “this thing wouldn’t install”). No news is not necessarily good news – in scenarios like this one must be proactive.

There are several ways you can be proactive and open up feedback channels to your users. The first approach is actually quite simple - a user rating scale for an application. Every screen in the application has a 1-5 rating (similar to how you might rate a book on Amazon.com). This approach enables users to provide immediate feedback (e.g. “I didn’t quite like this, it’s a 1,” to “Really loved it, it’s a 5.”). An average of the rating score across a several screens can be used to determine what should be improved in the next version of the application.

When Microsoft sent Vista to a number of families for testing the OS included an option called “send a smile.” This option displayed a red and green smileys that appeared in the system tray. When users found something they really liked, they clicked on a green smiley. When there was something they did not like they clicked on the red smiley. Regardless of which color they clicked on, a screen shot was created and automatically submitted back to Microsoft with some explanatory text. This is a great example of a non-intrusive UX for gathering feedback, and also coupled with the screen shot of the UX.

Figure 16: *A proactive approach to capturing feedback coupled with context*

Another way of getting proactive feedback is to have events and statuses that get logged backto a server for the support staff. This avoids having users call the Help Desk and cutting-and-pasting stack traces into an e-mail.

The third thing is thinking about effective status updates. What can you do in your application to let users know what has gone wrong? Things do go wrong. People will naturally accept that, but is there something you can do above and beyond, to maybe send them an e-mail, or maybe in another application actually give them some kind of status? You would not believe the kind of positive feedback you will get from users who say "I realize something has gone wrong, but now I have an expectation of when this thing is going to be up."

Proactive Recommendations

1. Think how users are going to provide proactive feedback. Think about those types of channels that you can open up.

Think about a strategy for when things go wrong. Thinks will go wrong. What do you need to circumvent that?

Wherever possible provide a status. Let people know what is good and bad and what their expectations should be.

Figure 17: *The three aspects of infrastructure.*

Infrastructure

The final set of concerns focuses on the infrastructure of the application. These are the types of user comments typically associated with infrastructure:

"To be honest, I couldn't even get the darn thing installed."
"It crashed the first time I used it. I don't think I'll try that again."
"This thing is sure ugly."

These types of problems fit into a foundational tier called Infrastructure. Infrastructure focuses on three aspects: Platform, Proven and People (developers and designers). We examine these aspects in greater detail below:

Platform

Infrastructure begins with the platform. Once you have the other areas covered, how should we create our application? There are a phenomenal number of choices today: Win32, Windows Forms, Flash, Java Swing, and Windows Presentation Foundation (WPF).

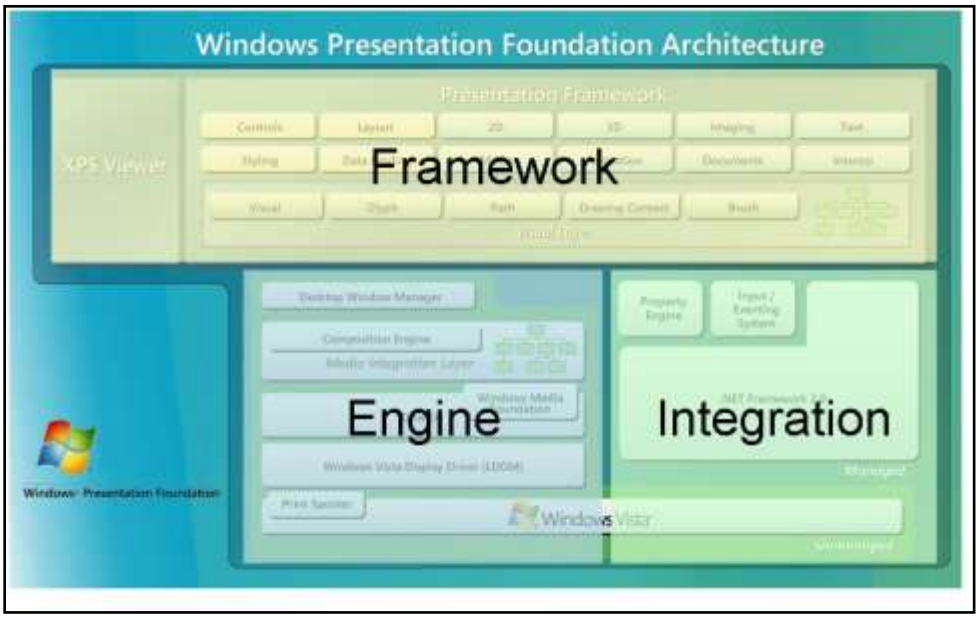


Figure 18: *The WPF platform*

WPF is interesting at the technical level, but there are three aspects to it that are worth reviewing:

- WPF adds a new engine – the evolution of GDI and GDI plus. The new engine is built upon 3D piping.
- WPF is a framework for both developers and designers.
- Integration: WPF is fully integrated into the platform

WPF supplies niceties that you can add to your current applications – nimbus, halo, drag and drop and more. WPF also enables applications that could not have been built before. For example, Zurich Airport mapped in WPF for traffic controllers so they can zoom in and get a view of the entire airport. This is a very rich UX that you could not have been easily built by application developers prior to the advent of WPF (see the Case Study in this Chapter for more information).

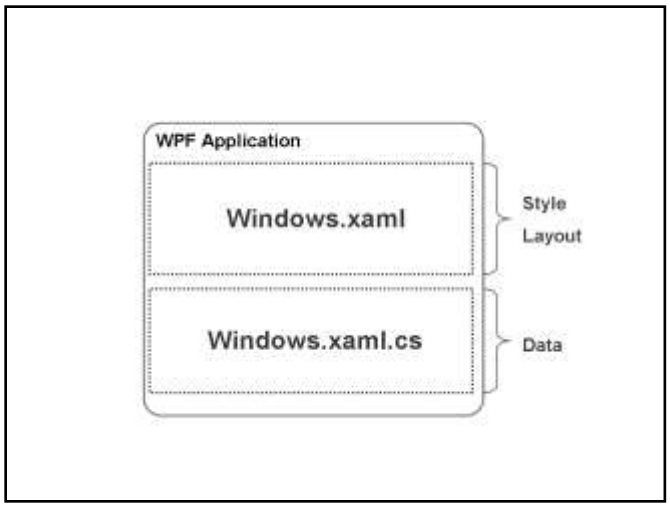


Figure 19: *WPF separation of style, layout, and data*

The separation of style, layout, and data is an important concept. WPF uses an markup language named XAML. XAML has a two way relationship with the underlying code – the XAML can be maintained as XML or with code alone. XAML can also be loaded directly into the browser using XBAP (XAML Browser APplication). XBAP is a new Windows technology used for creating Rich Internet Applications.

There are two main ways of deploying a WPF application: You can create a client executable that runs standalone or an XBAP that runs within the context of a browser. Regardless of the technical implications of selecting one over the other, there are a number of design implications associated with each that should be considered.

Depending upon your application requirements, an XBAP may offer the best end user experience since it is the easiest way to deploy an app (although there are security tradeoffs when running in a browser-based sandboxed environment). Deployment, however, is only one aspect of an application. Moving from a WinForms application to a browser-based XBAP raises several design considerations. For example, the original client application may have relied upon "right click context menus". While this feature is fairly simple to simulate in an XBAP, most users may realize they can right click on a Web page and not get the standard browser right-click menu.

Platform Recommendations

1. WPF is a declarative, vector-based, resolution independent platform for building rich user experiences (browser-based via XBAPs or incorporated into a traditional Windows application).

2. When thinking about deployment, think about the underlying design elements and principles and make sure they go across as well. Just because you can easily deliver a different experience from a technical standpoint doesn't mean that you can rely on the same, underlying user experience principles (e.g. right-click menus in a browser-based application).

Proven

Infrastructure also contains a number of elements that relates to proof. Proof comes down to reliability and how do you make an application reliable. This is the type of question many of us get everyday. The question instead should be: How can we make this application more proven?

Ultimately, reliability is a bad proof. One of the biggest hurdles for building trust or making your application proven is time. You cannot deploy version 1, 2, or 3 of an application and say “Wow, this is so reliable,” because it takes days, weeks, or months to build that proof point. Once that trust is lost, once the application crashes, dies or does not work as expected, it’s very difficult to recover.

For example, one of our architects got a new laptop, with all of the latest BIOS, but no OS. He got a clean version of Vista, the installation went fine, but the machine blue-screened on first login. At this point he lost trust in that machine. There were no third-party drivers, nothing hanging out in the back. This was a vanilla machine. The analogy was that this might not happen again, but he equated it to finding out his wife was a spy. He would still love his wife, but he would never be sure if things in the future were related to spying, or related to their relationship.

There are three things to do when thinking about making stuff more proven, more reliable in the eyes of the users: reliable installation, exception handling, and testing.

So first, think about the installation. Users hate it when developers spend twelve months on the application process and only two hours on the design. It’s like taking a beautifully expensive mobile phone, and wrapping it in old news paper. The product may be great, but if the install is problematic, that is the first impression we need to get over. It’s the only feature all of the customers will always use, so if that install is bad, if it dumps a load of stuff over your desktop, or if it Unzips and then closes and you have to search to find it and run setup from there, it leaves a really bad impression “right-out-of-the-box” so to speak.

Although it's not as bad as it used to be, many people discount the importance of installation as part of the overall experience. The installation is the first impression of a piece of software. It should be elegant, flawless, simple, and should just work - and if it is not going to work the feedback should be immediate

Here is a top ten list of things that to consider when creating your own installable software:

1. **Pre-requisites for the installation software.** Bad installation packages have many pre-requisites (certain versions of controls, components, frameworks and so on) - this can be bad - especially if you are forced to quit the installer to install these pre-requisites first!
2. **Requiring a restart.** Sometimes you need a restart of the machine, especially if you are changing part of the kernel that just cannot be stopped. There have been many installers that require a reboot "just to be safe." Do not make your installer require a reboot of the machine just because it's a simple checkbox that you can enable - if you need dependant components to restart after your install, then put in the time and work out how to do this without a complete restart of the machine.
3. **Annoying animations and/or menus.** Few people will be impressed by how visually exciting the installation menu is. Keep it simple, clean, and elegant.
4. **Too many options.** This is the killer mistake for most installers - option after option after option. Good installers do discovery, make some basic assumptions and keep the questions to a bare minimum. Do the majority of your users really care whether where your application lives anyway? At least customize the instalationl for the persona doing the install (such as basic or advanced user).
5. **Un-installation.** We have all seen rogue files, registry settings, ini files, icons, menus, and so forth get left behind by a bad installer. Make sure that your installer cleans up after itself if people choose to remove your software.
6. **Working out permissions.** If the installer requires admin permission (especially important with the UAC changes in Vista) let the user know before you start installing anything. Exiting half way through an installation (e.g. a “2869 MSI message”) is not good installation karma.
7. **Applications that need quitting.** If an installer relies on an application being terminated before installation can continue, give the user notice of this and, if applicable, have the installer shut it down. Do not quit the installer just because the user has a browser open. The installer should also *not* tell the user to close the associated application.
8. **Accurate account of time remaining.** If you do provide a progress box in an installer, please do not reset it or break into multiple unknown chunks. What good is a progress bar if it doesn’t accurately illustrate the progress of the installation?
9. **Unpacking the installation.** If you provide a package that has to unzip some temporary files, please automatically run the "setup.exe" after you've completed the unzip - don't just unzip files and let the user figure out what needs to be run next. Make sure the installer cleans up these temporary files once the installation is complete.

10. **Do you really need an installer at all?** If you are creating something lightweight – an add-in, or extension – why not creating a piece of standalone software that figures out how to run and co-exist on the user's machine without an install routine?

Think about how exceptions are handled. Popping up a system.NET Web exception is pretty easy to do, but what is there in letting the value in the user seeing that? How difficult is it for us to trap that?

Related to exceptions is thinking about how you are going to retry things. If something does happen, is it worth just retrying it again? Maybe a network connection was lost. Does your installer have to be so fragile that it fails with exceptions at the first sign of an error? There is probably an 80/20 rule where if we tried something else, we could probably fix 80% of the problems quickly.

The third thing to think about is testing. Proof also comes down to testing. If you are involved with customers they may be thinking about how to test this application before releasing it to our customers. User testing tends to be the same today as it always was. We get a room full of users for an hour and maybe buy them some pizza, we give them some tests – they can be sophisticated or basic – we collect the feedback and see what happens, and then we make a modification. Great, we should definitely do that; it will make things more reliable.

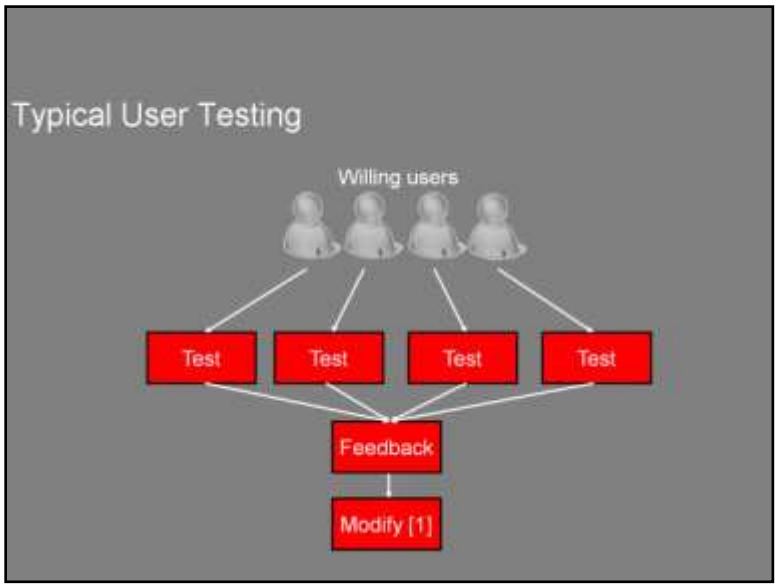


Figure 20: *Typical user testing*

This is really the best way of doing testing. Game developers use a test technique known as RITE: rapid, iterative testing and evaluation[1]. This is how RITE works: Take one user, put them

[1] *http://download.microsoft.com/download/5/c/c/5cc406a0-0f87-4b94-bf80-dbc707db4fe1/mgsut_MWTRF02.doc.doc*

through a test, collects the feedback, and then makes that modification. Then you take the modified version and give it to a second user. Repeat with a third, a fourth, and so on.

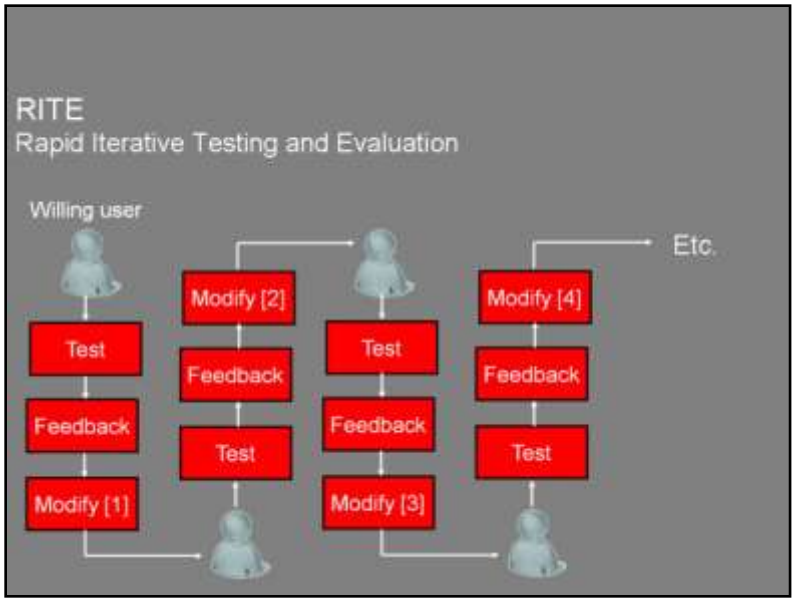


Figure 21: *RITE user testing*

What they found with this methodology is that they are using the same amount of users, and although the length between tests is longer because they need to make the modifications, that the amount of bugs they can fix is exponentially greater. The first person picks out all of the low hanging fruit, then next person is at a level higher, and so on.

Proven Recommendations

1. Trust takes time. Do not risk it by releasing a buggy version just to get things out of the door. If things go wrong, it's going to hurt you in the long term.
2. Treat install and uninstall as part of the application. In fact, it's more important, because if it doesn't install properly, you have no hope of people using it.
3. Look at the RITE methodology. If you involved in user testing it's a great and easy way of finding more bugs quicker.

People

The last aspect of the Infrastructure layer is People. The People aspect refers to everyone who designs and develops the application, essentially the software development lifecycle (SDLC). A typical SDLC looks something like Figure 22:

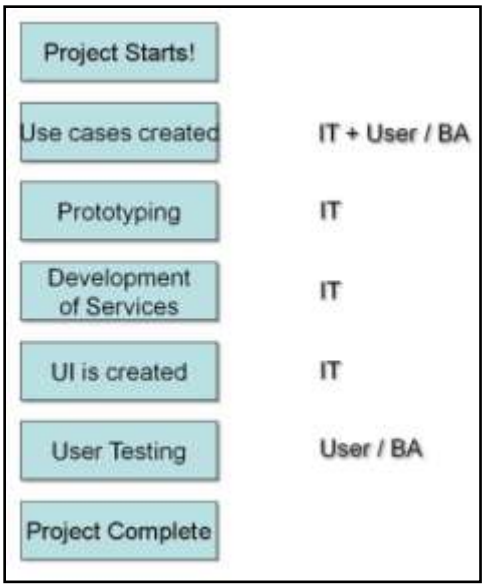


Figure 22: *The typical SDLC*

A project typically follows the pattern illustrated in Figure 22. There are two fundamental problems in this example: one is that IT owns most of the project. While IT should be involved, the owner/stakeholders are unclear. The other problem is how late in the schedule the UI is created. UI design and development is simply bolted onto the end of the project schedule. Many organizations push version one of their UI out the door quickly, get user feedback, and decide they have to fix it. A designer is brought in, but it's too late to add any value to the UX (as stated earlier). The designer's hands are effectively tied due to the state of the application. We need a new, more successful methodology that recognizes UX as a first-class citizen in the SDLC. A more successful UX scenario appears in Figure 23 below:

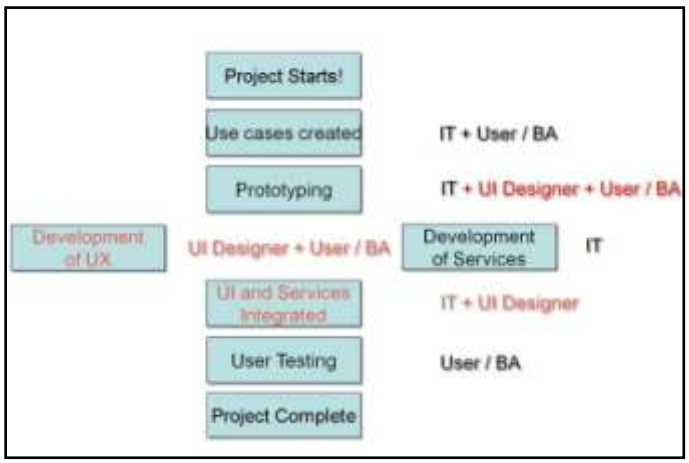


Figure 23: *A better UX development scenario*

In Figure 23 the project begins by developing the use cases, scenarios, and personas. At the prototyping stage, we split the project into two pieces - an IT prototype for the Web services, and a separate user prototype. The user prototype can utilize techniques such as paper prototyping, wireframes and speaking with end users to ensure the UX will meet their needs.

IT focuses on developing the service and the infrastructure side while the designer works with users, business analysts and IT to design the UX. In an agile methodology there would be points where we merge these initiatives back together for review and user testing. Eventually the project comes back together for completion.

Are all designers the same? Designers may have different skills, but are their functions different? When it comes to design there are two different personas. A graphic designer comes up with the correct shade of blue or image for the interface. An Interaction Designer is the developer who thinks about how the user will interact with the application. Microsoft Expression Blend can be used by Designers to design various components of the UX. The real challenge is sharing designer recommendations with developers.

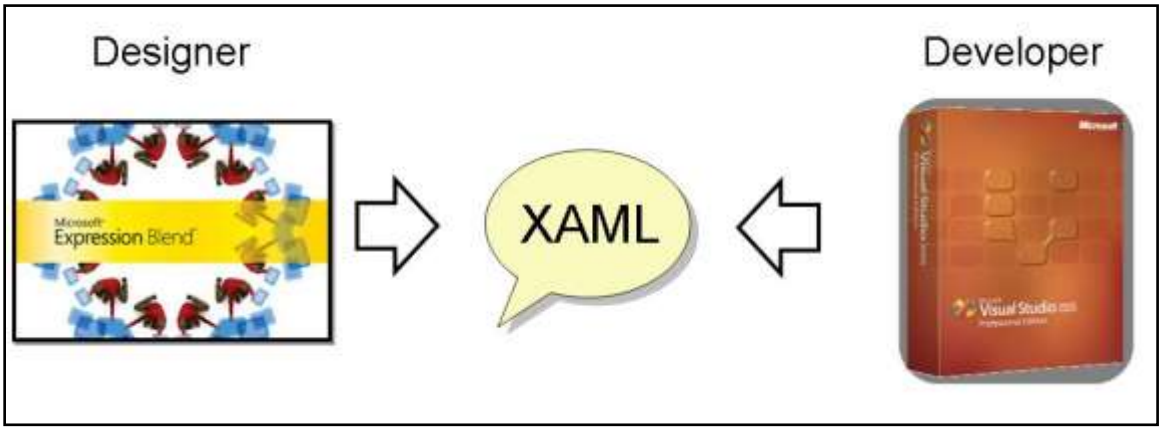


Figure 24: *Interaction between Designers and Developers*

Expression stores designs in XAML which can be shared with UX developers in Visual Studio, ensuring no loss of fidelity between the design and the implementation. From an architect perspective we need to think about the following design considerations up front:

- Events and the functions in Blend are going to require code-level understanding. A Designer might be able to use Blend to mark up a page, but what if the design needs to be more interactive? Are Designers expected to know how to write C# or VB code?
- There are going to be arguments between Developers and Designers about who owns the XAML artifacts. If both sides have to work with the same piece of XAML who trumps whom when a change needs to be made?

- This type of design interaction will lead to interface stubs for communication. Will patterns or guidance emerge helping Developers and Designers to communicate better?

People Recommendations

1. Look at the SDLC and customer projects. Normally, a great SDLC really enables great UX.
2. Understand what types of designers the SDLC model is going to require and support.

Support interactions between developers and designers.

Conclusion

This chapter proposed a UX Framework for architects. The three main points to remember are:

1. If you are a practicing architect use the Framework to better position UX for the customers you are involved with.
2. If you are an aspiring architect, use the Framework to put UX into the larger context. As you are think about the architecture for your solution think about the impact that UX will have upon your users.
3. Use the Framework to promote UX as a first class citizen in the domain of software architecture.

Chapter Six provides a detailed look at the Identity and Access architectural capability,

SOA Case Study: Zurich Airport

In just a single day, more than 60,000 travelers will pass through the Zurich Airport. Annually, approximately 20 million tourists, diplomats, and residents will travel through this busy hub in Switzerland. It takes an infrastructure of 24,000 employees and 180 businesses to support the Zurich Airport. Overseeing it all is Unique, a private company contracted by the Swiss Federation to manage the entire complex.

To ensure smooth, overall airport operation, Unique teamed with Microsoft to build an integrated business monitoring system. The goal was to provide a high-level visual overview of everything going on at the airport at any time, on any day. WPF was used to develop a fully interactive map capable of providing at-a-glance views of key operational information in real-time, including passenger forecasts, plane location, flight data, environmental conditions, baggage loads, management summaries, and other live reports. Native support for advanced animation within WPF enabled developers to write a few lines of code to illustrate the movements of aircraft on the airport runways. Users can zoom in on the plane, move the mouse pointer over it, and see a pop-up balloon containing information about the flight.

Red areas represent plane and passenger movement at Zurich Airport

The entire Case Study is available online at http://www.microsoft.com/casestudies/casestudy.aspx?casestudyid=200365.

See other SOA case studies at http://www.microsoft.com/casestudies/search.aspx?Keywords=SOA.

References:

1. "User Experience for Enterprise Applications", presentation by Simon Guest, 2006. More information on UX is available at http://msdn2.microsoft.com/en-us/architecture/aa699447.aspx

Chapter 6: Identity and Access

"Your identity is your most valuable possession – protect it.
If anything goes wrong, use your powers."

- Elasti-Girl,
from the film "The Incredibles"

Reader ROI

Readers of this chapter will build upon the concepts introduced in previous chapters, specifically focusing on the Identity and Access architectural capability.

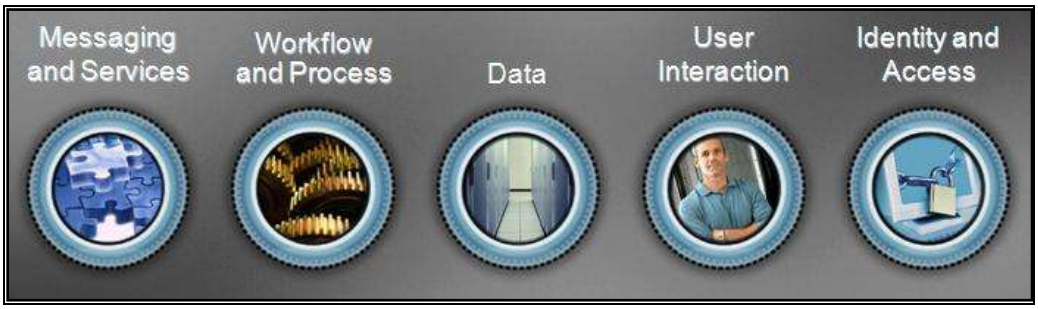


Figure 1: *Recurring Architectural Capabilities*

In this chapter readers will learn about Microsoft's identity and access strategy. We will discuss the laws of identity – a set of guiding principles that are learned from operating real world identity systems. We will also describe an identity metasystem that can lead to safer and more trustworthy Internet computing.

Topics discussed in this Chapter include:

- Terms and concepts associated with Identity and Access
- The Laws of Identity
- Identity federation scenarios
- Trusted Subsystems
- An Identity Metasystem

Acknowledgements

The concepts discussed in this chapter are entirely drawn from earlier efforts in this space. We wish to thank the following individuals for their work in this area: Kim Cameron (The Laws of Identity) and Fred Chong (terms, concepts and identity federation scenarios, and trusted subsystem design).

Identity and Access

Overview

Identity and Access focuses on the management of identities with support for the following aspects:

- Management of multiple roles (identities): An employee may serve different roles within an organization – each of these roles need to be served appropriately based upon context, process and other factors.
- Propagation of identities: Propagating the various identities used by both simple and complex business processes can be a challenge. Identities must be passed between each service involved in the business process in a seamless manner. The identities used by a given process must also be granted appropriate access rights to the resources necessary for the process to complete successfully.
- Cross-Boundary Integration: Business processes can span departments, companies and countries. Disparate cross-department, cross-company collaboration, regardless of resource or employee locations.
- Applying and managing business-rules to identities: Next to the creation and the management of identities, rules are needed that define how these identities should be handled (e.g. caching/time limitation and other issues).
- Delegation of management functions to business units: Identities convey much more information than a simple “Username / Password“. The responsibility of identity content, rules and management is evolving into a business function instead of an administrative function for IT. Business units must be able to manage and maintain their identities in an autonomous manner.

Most large organizations have multiple security systems in place. SOA projects need to leverage the organization’s existing IT investments, meaning the solution will need to interact with a broad range of security systems. Without comprehensive security integration strategy in place the organization will need to determine security, identity and access requirements on a case by case basis. This leads to increased development and deployment times as well as application-specific security issues. If applications fail to address these issues then the burden for interacting with various security systems falls to the user, resulting in overly complex user experiences, lower productivity and increased risk. Security Integration is generally concerned with five central issues:

- Who are the users?

- How do they prove it?
- What can they access?
- What level of privacy is required?
- How and when are access privileges granted, extended ore revoked?

The solution to these issues is to adopt a framework that effectively hides the security details of from new solutions. Once this framework has been established the organization's legacy applications must be adapted to make use of it. Policies for new applications should be adopted to ensure that new solutions will also use the security integration framework.

There are several additional benefits the organization can derive from a standardized security integration framework:

- Workers will no longer be forced to manage multiple sets of logon requests and demands, avoiding lost productivity and potential security vulnerabilities.
- New employees have to wait too long for their accounts to be created.
- Corporate security and privacy policies are inconsistently or ineffectively enforced.

As stated earlier, security integration and identity management are critical topics that must be addressed by any SOA solution. Organizations that adopt a consistent security integration and identity management strategy are:

- **More efficient:** Organizations can better connect and collaborate with their customers, partners and suppliers.
- **More productive:** Users can be more productive since delays accessing systems or resources are minimized. Management of user identities is also more productive associated with providing access to needed systems are reduced (e.g. like described, multiple password requests are handled by the applications themselves).
- **More efficient from an Operational Costs perspective:** A consistent security integration and identity management strategy enables common tasks like password resets to become user-driven self service options, greatly reducing overall help desk call volumes.
- **More Secure:** Consistent security integration and identity management strategies enable corporate and governmental policies and compliance mandates to be implemented at an enterprise level instead of on a department by department basis.

Now that we've got some familiarity of the benefits an organization can derive from a consistent identity management strategy, let's take a closer look at some of the principles for identity

management in a large distributed system. Increasingly organizations are deploying or using solutions that use the Internet. Some of these solutions use the Internet to communicate with trading partners while SaaS-based solutions may be delivered over the Internet.

Trusted Subsystem Design

While decomposing applications into multiple layers facilitates certain design and deployment activities, it also introduces challenges into the development of different facets of application solutions. Of those design issues, recurring ones concern controlling and accounting for access to business data and computing resources exposed by each application tier.

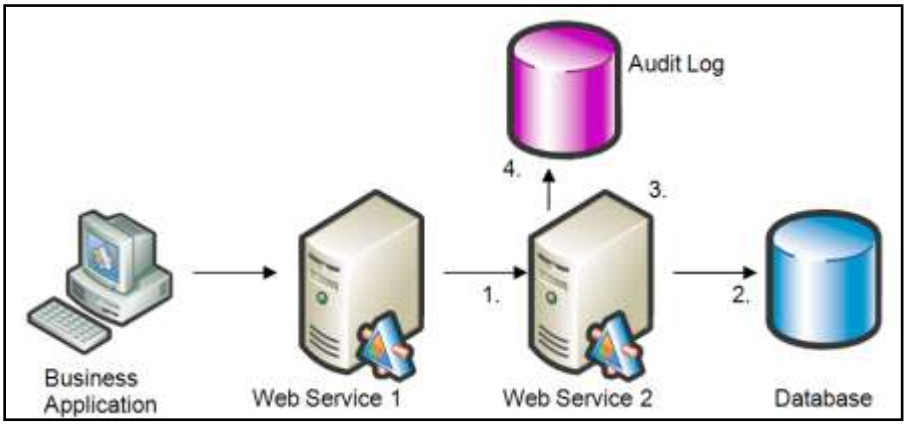


Figure 2: *Access control and auditing requirements for multi-tiered applications*

In Figure 2 we see a typical distributed system. Systems like this typically raise the following types of concerns:

1. Who can invoke interfaces at Web Service 2?
2. Who can connect to the business database?
3. How should the business data be filtered with respect to privacy and business regulatory policy requirements?
4. How do we audit and keep records of data access?

We will use the scenario illustrated by Figure 2 to explain these design considerations. Figure 2 shows a generic multi-tiered application that is a good representation of the current generation of line-of-business (LOB) applications. In this scenario, a business application initiates an application request through a method in Web Service 1. Web Service 1 is a façade service that in turn calls another method in Web Service 2 to process the user request. The method in Web Service 2 implements internal business logic. Part of the business logic processing requires business data that must be fetched from a backend database. Before responding to the call from Web Service 1, the business logic implementation also writes the business event into an audit log

for accounting purposes. The following access control and accounting requirements may arise from this scenario:

How should we control access to computing resources that provide business logic computation services (such as Web Service 2) and the database that provides data management services? Web Service 2 implements important business logic that is critical for enforcing business policies and processes. Enterprises may want to restrict the ability to execute such mission critical code to a subset of identities. Similarly, the database is where the crown jewels of the business reside. The rights to connect to the database should be handed out judiciously.

Who should be allowed to access the data? Data privacy requirements or business policies may limit the business information that can be viewed and manipulated by the business application user. For example, insurance agencies may restrict viewing of customer data to the assigned insurance agent. In this case, the business logic will need to make authorization decisions based on the identity of the application user (e.g. the insurance agent) and the customer data retrieved from the database. This enforcement is sometimes known as data entitlement.

Who has already accessed the data? Regulatory requirements may mandate businesses to keep an audit trail of all data related activities, with regard to who accessed what data and when. In our scenario, the Web service will need to know the identity of the business application users to keep accurate records.

Current Practices

Currently, there are two prevalent models for fulfilling the needs described above:

- Impersonation and Delegation
- Trusted Subsystem

We will describe the two models conceptually and mention how today's implementations of the models are used to solve the access control and auditing issues mentioned above. We will then detail some limitations with the current approaches and then apply a set of trusted subsystem design principles to derive an alternative solution.

Impersonation and Delegation

Delegation is the process of allowing another party to act on behalf of an identity. This process bestows upon a party the rights and privileges of another party to perform a set of tasks. Impersonation can be viewed as the most relaxed form of delegation such that one identity is assigned the complete set of permissions of the impersonated identity. There are two common ways of implementing delegation:

1. An identity creates a signed statement with a set of permissions that is assigned to the stated party for a specific duration of time, so that the stated party can present the signed statement and act on the identity's behalf.
2. A trusted third party issues signed statements to stated parties enabling those stated parties to act on another identity's behalf for a certain amount of time.

The first model is easier to implement when the subject delegating the rights is aware of the parties that are performing tasks on its behalf. However, in the current multi-tiered distributed computing environment, this is not the common case. More often than not, an end user is not aware of the services that are interacting to complete the user's requests. In these cases, a more suitable form of delegation is used; a third party authorizes on-behalf-of requests. The Kerberos security protocol is a popular implementation of such a scheme. The Kerberos delegation model does not assume that the end user knows about all the cooperating services beforehand. With Kerberos, the Kerberos Distribution Center (KDC) issues "forwardable" tickets that allow services to act on each user's behalf.

Using the Windows impersonation and Kerberos delegation approach, a service wanting to propagate the caller identity will first make an application request to the Windows operating system to impersonate the user locally. Note that impersonation is considered a highly restricted function and the application process identity must possess the Trusted Computing Base (TCB) privilege to impersonate a user. This right can only be assigned by a system administrator on the local machine. When the application process impersonates a user, everything it does locally is authorized and audited using the impersonated user's identity. This means that an application process may also acquire user privileges that the application process identity would not normally have when not impersonating a user.

Kerberos delegation enables an application process to acquire a Kerberos service ticket on behalf of the impersonated user, and the impersonated user's identity is carried forward in the acquired service ticket. The application process can then invoke the remote service using the acquired Kerberos service ticket. The downstream service being invoked will process the request according to the identity in the Kerberos service ticket. Thus, through the delegation process, the (original) user's identity is propagated to downstream services.

In our previous scenario, using Windows impersonation and Kerberos delegation, it is possible to flow the original user's identity downstream, so that the business logic Web service can perform more granular authorization, such as data-level entitlement, as well as record the impersonated user's identity in the audit trail.

Other than enabling multi-tier applications to make application authorization decisions, using Windows impersonation and Kerberos delegation also enhances the security of the application environment through the following features:

With Windows Kerberos constrained delegation, administrators can further configure delegation policies so that services can be restricted to act on behalf of users for only a given set of services.

The Kerberos protocol allows mutual authentication, which means that services can authenticate each other, thus reducing the risks of interacting with rogue services.

Impersonation and Kerberos delegation are readily available capabilities that are built into the Windows operating systems. This means enterprises that have an Active Directory directory services infrastructure can reuse the same security infrastructure to enable application security.

Despite these benefits, there are also limitations with using the impersonation and delegation model:

Impersonating users at the OS process level can elevate the process privilege, which may have unintended consequences as a result of elevation-of-privilege security attacks.

Kerberos delegation requires the application process to contact the KDC to obtain a service ticket on behalf of the user before accessing and invoking downstream services. For applications that are accessing downstream resources on behalf of a large number of users, this security negotiation with the KDC introduces an extra communication latency which may not be acceptable. The obvious solution to cache such security contexts across multiple requests violates stateless recommendations for high performance services.

Some applications optimize their performance by sharing a pool of pre-created communication channels for connecting and communicating with downstream services. For example, many business logic tier services have a pool of database connections that the services reuse to communicate with the downstream database. To meet security requirements, each of these database connections requires authentication. When impersonation/delegation is used to propagate the security context of the original caller, an authenticated database connection is created for each impersonated user and the connection pooling technique cannot be used to optimize application performance.

Access to services such as the business logic Web service must be secured using the impersonated user's identity. In other words, it is necessary to grant permissions to end users to call mission-critical code. In many cases, this is not an acceptable assumption, from the following perspectives:

- It violates the particular view of defense-in-depth which gradually reduces the number of identities that can access higher value assets. The fortress analogy is quite appropriate

here. Layers of defensive walls protect the center core of command and control. Fewer trusted officials of higher ranks are allowed access as we move toward the inner core of the fortress. To abide by this principle means that the ability to invoke inner business logic should be restricted to a small set of application identities. When end users can also invoke these services, the security of these services is also dependent on (and at the mercy of) the efficiency of the end user de-provisioning process.

- Giving end users the ability to invoke business logic services also increases the complexity of managing access control for the service. Instead of only having to manage the access rights of a few application identities, we now have to deal with end user identities instead.

Trusted subsystems

The limitations of the impersonation and delegation model provide the motivation for considering the trusted subsystem model.

By definition, the trusted subsystem model implies that application services are trusted to perform a specific set of application tasks, such as processing customer orders. Frequently, downstream services need to make application authorization decisions, such as approving an order submission before performing the business transaction. To do so, the service must know the identity of the end user submitting the order. While the ability to flow the identity of the end user is an inherent property of the delegation model, it is not so for the trusted subsystem model and special efforts must be made to include this feature.

The notion of “trusted” does not come for free. To support the notion of trust as defined by the model the services must at least be able to:

Authenticate and verify the identity of the upstream or downstream service they are communicating with.

Decide if the identified service is a trusted subsystem for a specific set of application functions, including propagating identity claims.

Protect the integrity of the data being communicated between trusted subsystem and downstream services. Besides application data, application plumbing data, such as the identity claims of the original user, must also be protected so that no man-in-the-middle can modify the identity information that is in transit.

However, many current implementations of alleged “trusted subsystems” do not meet these minimum safeguard requirements. We will substantiate this claim using some observations on how current trusted subsystems are propagating the original user’s context.

A common practice today is to propagate the identity of the caller as a custom HTTP or SOAP header. The downstream service looks in the custom HTTP and SOAP header when it needs the original caller's identity. Since the identity propagation often occurs within the organization's network, many application administrators naively assume that it is safe to propagate the data unprotected. In actual fact, many, if not most, application security breaches occur within the organization's internal networks.

Adding transport layer protection, such as using SSL\TLS with client and server authentication, can reduce the security attack surface quite significantly. In Web services environments where Web services endpoints may span multiple transport hops, adding Web service layer security can reduce the chances of man-in-the-middle attacks. This is done by requiring the services to mutually authenticate each other's integrity and protect the custom SOAP header.

We can also potentially build on existing X509 public key infrastructure to help a downstream service decide if an upstream service is a trusted subsystem. For instance, a custom certificate usage OID can be defined for a trusted subsystem so that it is issued certificates that contain the trusted subsystem usage OID. The issued certificate may also contain client and server authentication OID, so that the same certificate can also be used for HTTPS or Web services authentication using the X509Certificate token profile.

Up to this point, the proposed solutions are low hanging fruits that existing trusted subsystem solutions can incorporate to add additional security protection.

Trusted Subsystem Design Goals

In the next section, we will describe an infrastructure design that allows us to meet the trusted subsystem requirements previously stated. In addition, this trusted subsystem design also satisfies an additional set of security attributes for flowing identity claims. The design goals for this system are summarized below.

1. Enable downstream services to authorize service invocations based on application identities instead of end user identities. This requirement helps simplify access management of the service interface.
2. Enable a downstream service to positively identify an upstream caller and determine if the upstream caller is also a subsystem trusted for specific application functions. The design must not assume that all application services are trusted subsystems. Instead, it must use security policy mechanisms to specify and determine the kind of application functions that a service can perform.
3. Enable an upstream service to protect its communication with downstream services so that the data can be guarded against tampering and other attacks.

4. Enable an upstream service to propagate the identity of the original caller to downstream services. The identity context of the original caller is frequently used by application services to authorize business tasks and audit an individual's actions. The design should make this identity data available to the application.
5. Unlike the impersonation and delegation model, the design must not operate under the premise of "acting on behalf" of the original user, therefore changing the identity of the application process or thread. The solution should allow trusted subsystems to function like airline service representatives performing the check-in procedure at ticketing counters for identified travelers. The service representatives are not impersonating or acting on behalf of the travelers. Instead, they are the persons that are authorized to perform check-in procedures for travelers at check-in counters. The trusted subsystem design must not require an application to assume another identity to perform its application function.
6. In certain cases, a downstream service receiving a propagated identity from a trusted subsystem may also need additional evidence to accept the identity and proceed with the application requests. The process requirement here is akin to the U.S. green card application process. In many companies, corporate attorneys assist with initiating and preparing the green card application for foreign employees. After the initial preparation process is completed, the application document prepared by the corporate legal department is sent to the federal immigration office along with other official documents, such as the applicant's passport. The passport is the additional piece of information that the U.S. federal immigration office will use to prove the identity of the applicant. In the same way, downstream services may require additional evidence from a third party other than the trusted subsystem or user to support the identity claims received from trusted subsystems. For this requirement, our design takes into consideration the following specific evidences that may be required:

- It must be possible for a downstream service to verify that the owner of the propagated identity claim has been recently authenticated by a trusted third party. This reduces the window of time available for a compromised trusted service to propagate false identity claims.
- To further reduce the possibilities of false identity propagations, the service must be able to verify from material facts (for example, the original caller's signature) that a specific call originated from the identity previously identified through the set of propagated claims. This requirement builds on the previous one.

7. Decouple the credentials that are used for authentication from the ones that are used to make trusted subsystem claims. This requirement facilitates security policy management.

For example, authentication credentials such as X509 certificates usually have a longer lifespan, whereas a service's claim with respect to being a trusted subsystem may have a shorter lifespan that is dependent on the service location in the application environment.

Trusted Subsystem Design

There are two important requirements that must be met in the process of identifying a trusted subsystem:

1. The services that are interacting with each other must know that they are communicating with the intended party.
2. In situations where not all application services are considered as trusted subsystems, there must be a way for a service to further differentiate services that are trusted to propagate identities from those that are not trusted.

In the previous section, we have already described a few ways where services can mutually authenticate each other. It is certainly possible to leverage existing technologies to accomplish mutual authentication, such as using HTTPS with client and server side X509 certificates or the Kerberos protocol.

In addition to these solutions, it is also possible to use emerging Web service-based technology, such as a security token service that provides third-party brokered authentication to a group of Web service-based applications.

Beyond the authentication and SSO services that are provided by such brokered authentication services, we show here how it is possible to extend the infrastructure so that services can also ride on the authentication mechanism to authorize services as trusted subsystems.

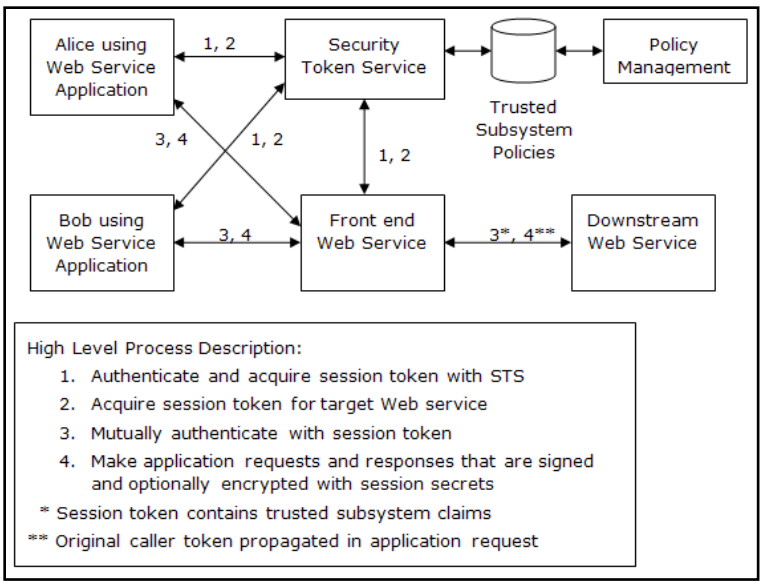


Figure 3: *A trusted subsystem design*

Figure 3 shows a trusted subsystem design that is built on a security token service and provides authentication and SSO service. The underlying security protocols supporting such an infrastructure can be composed by using existing Web services security specifications such as WS-Security, WS-Trust and WS-SecureConversation. This design demonstrates a multi-tiered application scenario where two users, Alice and Bob, are using two instances of Web service applications to access a front-end Web service. The front-end Web service communicates with a downstream Web service to complete the application request.

The next section provides background information detailing how a typical SSO infrastructure for Web services can be expected to function. The information is intended to provide the basic framework which we will extend to implement the trusted subsystem design.

Basic Web service SSO framework

Referring to the scenario shown in Figure 3, Alice and Bob authenticate with the security token service to acquire a session token from the STS as a proof of authentication. Subsequently, Alice and Bob can use their respective session tokens with the STS to request service access tokens to the front-end Web service without having to re-authenticate with their primary credentials (for example, password or certificates). The service access token is used to negotiate session keys and prove knowledge of secrets to establish mutual authentication.

The same SSO infrastructure used by Alice and Bob to authenticate and achieve SSO with the front-end Web service is also used by the Web services to mutually authenticate each other. For example, before the front-end Web service can communicate with the downstream Web service, it is required to authenticate with the STS to acquire an STS session token and a service access

token to the downstream Web service. Both Web services then mutually authenticate and set up a secure session with each other. The secure session handshaking protocol can be set up by leveraging Web services protocol building blocks such as WS-SecureConversation. This secure session negotiation process only has to be done once during the lifespan of the service access token. When the service access token is no longer valid, it may be renewed and the secure session must be renegotiated. Such an infrastructure proves to be very useful for large categories of distributed applications and services that are deployed in enterprises today. The mutually-authenticated security sessions improve security while minimizing performance overhead, since the initial security negotiation can be amortized over multiple application requests and responses transacted through the Web services.

It is also important to note that the concept of security sessions is orthogonal to the application communication paradigm. At first glance, it may appear that such SSO infrastructure is only useful for distributed applications that communicate synchronously. This is not true. The same infrastructure can also be used to secure distributed applications that service asynchronous long-running transactions.

Trusted subsystem process extensions

Returning to the topic of a trusted subsystem, we now propose an extension to the plain-vanilla SSO infrastructure by introducing the notion of trusted subsystem policies. The exact trusted subsystem policy model and language may take many forms, and it is not the goal of this document to specify or limit the mechanisms that can be used with this infrastructure level design.

In Step 2 of the high level process shown in Figure 3, the STS checks a trusted subsystem policy store to find out if the front-end Web service requesting a service access token to the downstream Web service is a trusted subsystem for the downstream Web service. If it is, the STS includes the trusted subsystem claim in the access token issued to the front-end Web service. The trusted subsystem claim and the rest of the data in the token are signed by the STS so that the downstream Web service can verify the party (STS) authorizing the claim.

When the downstream Web service processes the access token sent by the front-end Web service in Step 3, it verifies the authenticity of the trusted subsystem claim and notes the fact that the front-end Web service is a trusted subsystem. Depending on the specific nature of the claim, the downstream Web service may accept propagated identities in subsequent application requests from the front-end Web service. Note that the downstream Web service may still enforce requirements for additional evidence supporting the propagated identities. We will discuss the enforcement of this requirement later.

Trusted Subsystem Policies

In addition to the issuance, processing, validation, and enforcement of trusted subsystem claims, there is also a management component for manipulating the trusted subsystem policies. Trusted subsystem policies may take many forms. Existing authorization models such as role-based authorization may also be used to specify trusted subsystem policies. The example shown Figure 4 modeled different services (workflow services and data adapter services) as application resources. Each of the application resource services may permit the upstream service to perform certain actions according to the role that the accessing service is assigned to. For instance, the trusted subsystem role is defined for the workflow and data adapter services so that trusted services that belong to these roles are permitted to propagate identities to the application resource services. We can then assign individual services to their respective trusted subsystem roles defined for these application resource services.

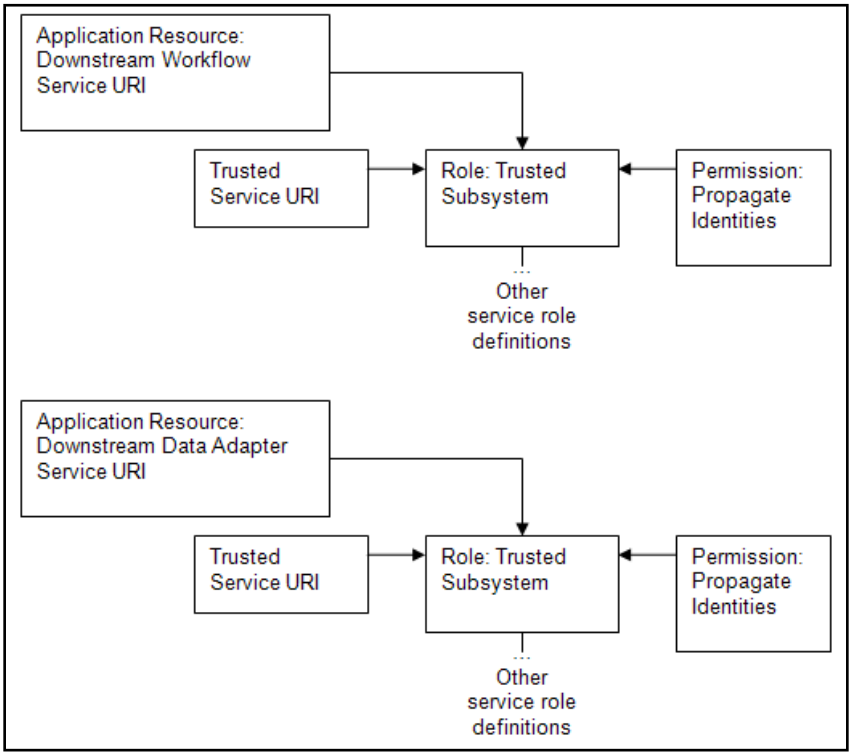


Figure 4: *An example of a trusted subsystem policy using role-based representation*

When the STS receives a service access token request for a particular downstream service, it checks whether the requesting service belongs to the trusted subsystem role defined for the downstream service. Alternatively, the STS can also check if the requesting service is allowed to propagate identities to the downstream service. If the result of the policy check asserts that the requesting service is a trusted subsystem, the STS inserts the trusted subsystem claim in the issued access token. It is important for the issued token to have scoping elements defined such

that the trusted subsystem claim applies only to the particular downstream service that the policy check has approved. There are many ways that the trusted subsystem claim may be transported inside the service access token. For example, if the service access token is a SAML 1.1 token, the trusted subsystem claim could be a SAML Authorization Decision Assertion inside the SAML token:

```
<saml:AuthorizationDecisionStatement Decision="Permit"
Resource="http://downstreamservice.multitierwebservice.com">
 <saml:Actions Namespace="http://..">
  <saml:Action>PropagateIdentities</saml:Action>
 </saml:Actions>
 <saml:Subject>
  <saml:NameIdentifier
Name="http://frontendservice.multitierwebservice.com" />
 </saml:Subject>
</saml:AuthorizationDecisionStatement>
```

In the example provided above, the upstream service identified by the URI http://frontendservice.multitierwebservice.com is a trusted subsystem permitted to propagate identities to the downstream service identified by the URI http://downstreamservice.multitierwebservice.com.

Flowing an Original Caller's Identity Claims

In the previous section, we illustrated how trusted subsystems may be permitted to propagate the original caller's identity claims to downstream services. At the beginning of the paper, we also mentioned the requirement that downstream services may require additional evidences to verify and support the genuineness of the flowed claims. The additional evidences are claims generated by parties other than the immediate upstream trusted subsystem, since different parties may be trusted to assert a limited set of claims. Such capabilities enable the downstream services to assemble additional evidence to securely process the request. For instance, for a downstream service to process a high value financial transaction, such as a $500,000 transfer, there is a need to know that the request in fact has been initiated by the subject identified in the flowed claim. We do not want compromised trusted subsystems to be able to generate such requests under the guise of any user.

In this section, we will show how trusted subsystems can support these requirements by describing the categories of propagated identity tokens.

Categories of flowed tokens

The following types of tokens may be propagated by the trusted subsystem:

1. Trusted subsystem generated identity tokens

2. Third party generated identity tokens
3. User self-signed tokens

The first type of propagated identity tokens is generated and issued by the trusted subsystem flowing the identity claims. This token can be as simple as a WS-security username token without the password element, or as complicated as a custom identity token that contains non-standard claims inserted by the trusted subsystem. In all cases, the trusted subsystem signs the identity token before propagating the generated token to downstream services.

For the next few examples, we use the `<OriginalCaller>` XML element to contain the propagated identities.

In this specific example, the UsernameToken is used to propagate the context of the original caller.

```
<OriginalCaller>
 <UsernameToken>
  <Username>...</Username>
 </UsernameToken>
</OriginalCaller>
A more complicated example of a custom token generated by a
trusted subsystem may look like:
<OriginalCaller>
 <CustomUserContextToken
Issuer="http://trustedsubsystem1.multitierwebservices.com">
  <Username>...</Username>
  <Roles>...</Roles>
 </CustomUserContextToken>
</OriginalCaller>
```

In addition to the name of the original caller, the issued custom token also contains application roles that the user belongs to. Such information may help downstream services make access decisions.

The second type of propagated token comes from a third party that is trusted to perform a set of specialized functions, such as a security token service that provides authentication and single sign-on services to Web services.

The following example shows a trusted subsystem propagating a signed SAML token issued by an STS that provides single sign-on services:

```
<OriginalCaller>
 <saml:Assertion AssertionId="SecurityToken-745d68f1-9c63-4216-
9fd5-9ebd96fab986" MajorVersion="1" MinorVersion="1"
Issuer="http://sso-sts.multitierwebservices.com" ...>
 ...
  <ds:signature>...</ds:signature>
 </saml:Assertion>
</OriginalCaller>
```

The third category of propagated tokens is generated and signed by the user that originated the call. The signature in the token enables trusted subsystems and downstream services to verify the asserted claims in the token.

Here is an example of a user self-signed token that can be used to correlate an application request with the original caller:

```
<OriginalCaller>
 <CallerEvidenceToken>
  <Subject>
    <X509CertficateToken>...</X509CertificateToken>
  </Subject>
  <Timestamp>...</Timestamp>
  <MessageID>uuid: 5b19a5f5-426c-4cfa-b953-
d5dc200e9508</MessageID>
  <Signature>...</Signature>
 </CallerEvidenceToken>
</OriginalCaller>
```

The original caller has attached an X509 certificate so that the subject in the X509 certificate can be identified as the original caller. Within the signed evidence, a timestamp element denotes the time when the evidence is generated and a message identifier is used to identify the chain of application requests initiated by the original caller. The timestamp and message identifier are then signed using the private key associated with the X509 certificate.

It is also possible to identify the subject in the evidence via some other means of tokens. For instance, we can also use a username token (without password) with a signature generated using a derived key from the password and some other random data.

Table 1 summarizes the conditions that may lead one to use one type of token over the other:

Token Type	**Usage Conditions**
Trusted Subsystem Generated Token	When downstream services trust the trusted subsystem to assert the original caller's identity, without requiring additional evidence from other parties.
Third Party Generated Token	When the downstream services trust the trusted subsystem to assert claims regarding the original caller in conjunction with third party evidence that satisfies an additional set of security requirements. For instance, downstream services may want to know that the originating user was recently authenticated by a trusted third party. In this case, the propagated token could be a SAML token issued by an STS providing SSO service.

Token Type	Usage Conditions
User Self-Signed Token	When the trusted subsystem is authorized to perform a set of application functions and when there must be evidence from the original caller that the caller initiated the request.

Table 1: *Token type and usage conditions*

Identity/credential mapping

With identity/credential mapping, a service takes a given identity, and with or without the help of a third party (such as a single sign-on service), maps the given identity into another related set of credentials. The set of mapped credentials may or may not contain authentication data, such as the logon password. The services then use this set of new credentials to gain access to downstream resources. We consider this to be a special function of the trusted subsystem role, where the goal is to transform an identity to another related identity (could be a change in type or namespace) for the purpose of gaining access to downstream resources that only recognize the transformed identity. It is important to note that the types of propagated identity tokens discussed above do not exclude token types that result from the process of identity or credential mapping.

Benefits of the Design

In view of the application requirements that we have discussed, the new approach described in this document brings about the following benefits:

It considers current application optimization techniques that focus on resource sharing, such as connection pooling, so that an application designer can minimize the tradeoff costs between security and performance. In our model, the trusted subsystem maintains a security session with downstream services. A security context is established for the trusted subsystem that allows the downstream services to exactly authorize and determine the scope of the trust. Subsequently, identity is flowed using one of the three token types mentioned. There is no need for services to establish per-user security context as required by the impersonation/delegation model.

It cleanly separates out the security concepts and goals of “acting-on-behalf-of” versus “authorized-to-perform.” Our trusted subsystem solution outlines the requirements and steps that are necessary to design “authorized-to-perform” systems, which is the model needed to secure a large number of the multi-tier applications currently deployed. Applications requiring delegation should continue to use existing protocols such as Kerberos.

Our design addresses the frequent need of downstream systems to know the identity of the original caller by implementing secure propagation of the original caller's identity claims.

Trusted subsystem behavior is managed and driven through policies, allowing for more flexibility in configuring and securing the tiers between applications. The same application service can be configured to be a trusted subsystem when communicating with a CRM service and not a trusted subsystem when communicating with a billing service. Such flexibility encourages reuse of loosely-coupled service components.

The credentials to prove trusted subsystem claims are decoupled from those that are used to authenticate and identify the services. Authentication credentials are typically longer-lived while application service roles may be different depending on how the service is being used within the larger distributed application context. Our design allows more flexibility for changing service capabilities without introducing additional management overhead and the need for revoking and reissuing authentication credentials.

Design principles and infrastructure implementations to facilitate trusted subsystem communications and identity claims propagation are some of the most needed but also most under-investigated areas in application security. The design we discussed is fully in tune with current service-oriented computing paradigm, where explicit service boundaries, loose–coupling, and policy-driven behavior are key design attributes. With our design approach, services do not make any a priori assumptions about whether a service is a trusted service, services are not tightly bound to a specific set of trusted services or trust settings, and the scope of service trust can be adjusted through configurable run time policies.

There are several important security requirements that are fundamental to trusted subsystem designs, which are:

- The ability for services to mutually and positively identify each other.
- The ability for services to determine if whether to trust the identity claims propagated from another service.
- The ability for a service to protect the integrity of application data and identity claims transacted through trusted subsystems.

An Identity Metasystem

For users and businesses alike, the Internet continues to be increasingly valuable. More people are using the web for everyday tasks, from shopping, banking, and paying bills to consuming media and entertainment. E-commerce is growing, with businesses delivering more services and content across the Internet, communicating and collaborating online, and inventing new ways to connect with each other.

But as the value of what people do online has increased, the Internet itself has become more complex, criminalized, and dangerous. Online identity theft, fraud, and privacy concerns are on the rise, stemming from increasingly sophisticated practices such as “phishing”. The multiplicity of accounts and passwords users must keep track of and the variety of methods of authenticating to sites result not only in user frustration, known as “password fatigue”, but also insecure practices such as reusing the same account names and passwords at many sites.

The root of these problems is that the Internet was designed without a system of digital identity in mind. In efforts to address this deficiency, numerous digital identity systems have been introduced, each with its own strengths and weaknesses. But no one single system meets the needs of every digital identity scenario. And even if it were possible to create one system that did, the reality is that many different identity systems are in use today, with still more being invented. As a result, the current state of digital identity on the Internet is an inconsistent patchwork of ad hoc solutions that burdens people with different user experiences at every web site, renders the system as a whole fragile, and constrains the fuller realization of the promise of e-commerce.

What is the Identity Metasystem?

Given that universal adoption of a single digital identity system or technology is unlikely ever to occur, a successful and widely employed identity solution for the Internet requires a different approach — one with the capability to connect existing and future identity systems into an **identity metasystem**. This metasystem, or system of systems, would leverage the strengths of its constituent identity systems, provide interoperability between them, and enable creation of a consistent and straightforward user interface to them all. The resulting improvements in cyberspace would benefit everyone, making the Internet a safer place with the potential to boost e-commerce, combat phishing, and solve other digital identity challenges.

In the offline world, people carry multiple forms of identification in their wallets, such as driver’s licenses or other government-issued identity cards, credit cards, and affinity cards such as frequent flyer cards. People control which card to use and how much information to reveal in any given situation.

Similarly, the identity metasystem makes it easier for users to stay safe and in control when accessing resources on the Internet. It lets users select from among a portfolio of their digital identities and use them at Internet services of their choice where they are accepted. The metasystem enables identities provided by one identity system technology to be used within systems based on different technologies, provided an intermediary exists that understands both technologies and is willing and trusted to do the needed translations.

It’s important to note that the identity metasystem does not compete with or replace the identity systems it connects. Rather, it plays a role analogous to that of the Internet Protocol (IP) in the

realm of networking. In the 1970s and early 1980s, before the invention of IP, distributed applications were forced to have direct knowledge of the network link, be it Ethernet, Token Ring, ArcNet, X.25, or Frame Relay. But IP changed the landscape by offering a technology-independent metasystem that insulated applications from the intricacies of individual network technologies, providing seamless interconnectivity and a platform for including not-yet-invented networks (such as 802.11 wireless) into the network metasystem.

In the same way, the goals of the identity metasystem are to connect individual identity systems, allowing seamless interoperation between them, to provide applications with a technology-independent representation of identities, and to provide a better, more consistent user experience with all of them. Far from competing with or replacing the identity systems it connects, the metasystem *relies* on the individual systems to do its work!

Identities Function in Contexts

The identities held by a person in the offline world can range from the significant, such as birth certificates, passports, and drivers' licenses, to the trivial, such as business cards or frequent coffee buyer's cards. People use their different forms of identification in different contexts where they are accepted.

Identities can be in or out of context. Identities used out of context generally do not bring the desired result. For example, trying to use a coffee card to cross a border is clearly out of context. On the other hand, using a bank card at an ATM, a government-issued ID at a border, a coffee card at a coffee stand, and a Passport Network (formerly .NET Passport) account at MSN Hotmail are all clearly in context.

In some cases, the distinction is less clear. You could conceivably use a government-issued ID at your ATM instead of a bank-issued card, but if this resulted in the government having knowledge of each financial transaction, some people would be uncomfortable. You could use a Social Security Number as a student ID number, but that has significant privacy implications, even facilitating identity theft. And you can use Passport accounts at some non-Microsoft sites, but few sites chose to enable this; even where it was enabled, few users did so because they felt that Microsoft's participation in these interactions was out of context.

Studying the Passport experience and other digital identity initiatives throughout the industry led us to work with a wide range of industry experts to codify a set of principles that we believe are fundamental to a successful, broadly adopted, and enduring digital identity system on the Internet. We call these principles "The Laws of Identity".

The Laws of Identity

The “Laws of Identity” are intended to codify a set of fundamental principles to which any universally adopted, sustainable identity architecture must conform. The Laws were proposed, debated, and refined through a long-running, open, and continuing dialogue on the Internet. Taken together, the Laws define the architecture of the identity metasystem.

They are:

1. **User Control and Consent:** Identity systems must only reveal information identifying a user with the user's consent.
2. **Minimal Disclosure for a Constrained Use:** The identity system must disclose the least identifying information possible, as this is the most stable, long-term solution.
3. **Justifiable Parties:** Identity systems must be designed so the disclosure of identifying information is limited to parties having a necessary and justifiable place in a given identity relationship.
4. **Directed Identity:** A universal identity system must support both “omnidirectional” identifiers for use by public entities and “unidirectional” identifiers for use by private entities, thus facilitating discovery while preventing unnecessary release of correlation handles.
5. **Pluralism of Operators and Technologies:** A universal identity solution must utilize and enable the interoperation of multiple identity technologies run by multiple identity providers.
6. **Human Integration:** Identity systems must define the human user to be a component of the distributed system, integrated through unambiguous human-machine communication mechanisms offering protection against identity attacks.
7. **Consistent Experience Across Contexts:** The unifying identity metasystem must guarantee its users a simple, consistent experience while enabling separation of contexts through multiple operators and technologies.

To learn more about the Laws of Identity, visit www.identityblog.com.

Roles within the Identity Metasystem

Different parties participate in the metasystem in different ways. The three roles within the metasystem are:

Identity Providers, which issue digital identities. For example, credit card providers might issue identities enabling payment, businesses might issue identities to their customers, governments

might issue identities to citizens, and individuals might use self-issued identities in contexts like signing on to web sites.

Relying Parties, which require identities. For example, a web site or online service that utilizes identities offered by other parties.

Subjects, which are the individuals and other entities about whom claims are made. Examples of subjects include end users, companies, and organizations.

In many cases, the participants in the metasystem play more than one role, and often all three.

Components of the Identity Metasystem

To build an identity metasystem, five key components are needed:

1. A way to represent identities using claims.
2. A means for identity providers, relying parties, and subjects to negotiate.
3. An encapsulating protocol to obtain claims and requirements.
4. A means to bridge technology and organizational boundaries using claims transformation.
5. A consistent user experience across multiple contexts, technologies, and operators.

Claims-Based Identities

Digital identities consist of sets of claims made about the subject of the identity, where “claims” are pieces of information about the subject that the issuer asserts are valid. This parallels identities used in the real world. For example, the claims on a driver’s license might include the issuing state, the driver’s license number, name, address, sex, birth date, organ donor status, signature, and photograph, the types of vehicles the subject is eligible to drive, and restrictions on driving rights. The issuing state asserts that these claims are valid. The claims on a credit card might include the issuer’s identity, the subject’s name, the account number, the expiration date, the validation code, and a signature. The card issuer asserts that these claims are valid. The claims on a self-issued identity, where the identity provider and subject are one and the same entity, might include the subject’s name, address, telephone number, and e-mail address, or perhaps just the knowledge of a secret. For self-issued identities, the subject asserts that these claims are valid.

Negotiation

Negotiation enables participants in the metasystem to make agreements needed for them to connect with one another within the metasystem. Negotiation is used to determine mutually

acceptable technologies, claims, and requirements. For instance, if one party understands SAML and X.509 claims, and another understands Kerberos and X.509 claims, the parties would negotiate and decide to use X.509 claims with one another. Another type of negotiation determines whether the claims needed by a relying party can be supplied by a particular identity. Both kinds of negotiation are simple matching exercises; they compare what one party can provide with what the other one needs to determine whether there's a fit.

Encapsulating Protocol

The encapsulating protocol provides a technology-neutral way to exchange claims and requirements between subjects, identity providers, and relying parties. The participants determine the content and meaning of what is exchanged, not the metasystem. For example, the encapsulating protocol would allow an application to retrieve SAML-encoded claims without having to understand or implement the SAML protocol.

Claims Transformers

Claims transformers bridge organizational and technical boundaries by translating claims understood in one system into claims understood and trusted by another system, thereby insulating the mass of clients and servers from the intricacies of claim evaluation. Claims transformers may also transform or refine the semantics of claims. For example, a claim asserting "Is an employee" might be transformed into the new claim "OK to purchase book". The claim "Born on March 22, 1960" could be transformed into the claim "Age is over 21 years", which intentionally supplies less information. Claims transformers may also be used to change claim formats. For instance, claims made in formats such as X.509, Kerberos, SAML 1.0, SAML 2.0, SXIP, and others could be transformed into claims expressed using different technologies. Claims transformers provide the interoperability needed today, plus the flexibility required to incorporate new technologies.

Consistent User Experience

Many identity attacks succeed because the user was fooled by something presented on the screen, not because of insecure communication technologies. For example, phishing attacks occur not in the secured channel between web servers and browsers — a channel that might extend thousands of miles — but in the two or three feet between the browser and the human who uses it. The identity metasystem, therefore, seeks to empower users to make informed and reasonable identity decisions by enabling the development of a consistent, comprehensible, and integrated user interface for making those choices.

One key to securing the whole system is presenting an easy-to-learn, predictable user interface that looks and works the same no matter which underlying identity technologies are employed. Another key is making important information obvious — for instance, displaying the identity of the site you're authenticating to in a way that makes spoofing attempts apparent. The user must be informed which items of personal information relying parties are requesting, and for what purposes. This allows users to make informed choices about whether or not to disclose this information. Finally, the user interface provides a means for the user to actively consent to the disclosure, if they agree to the conditions.

Benefits of the Identity Metasystem

Microsoft recognizes that the identity metasystem will only gain widespread adoption if participants filling all roles in the metasystem stand to benefit from their participation. Fortunately, this is the case. Key benefits of the identity metasystem include:

Greater user control and flexibility. Users decide how much information they disclose, to whom, and under what circumstances, thereby enabling them to better protect their privacy. Strong two-way authentication of identity providers and relying parties helps address phishing and other fraud. Identities and accompanying personal information can be securely stored and managed in a variety of ways, including via the online identity provider service of the user's choice, or on the user's PC, or in other devices such as secure USB keychain storage devices, smartcards, PDAs, and mobile phones

Safer, more comprehensible user experience. The identity metasystem enables a predictable, uniform user experience across multiple identity systems. It extends to and integrates the human user, thereby helping to secure the machine-human channel.

Increases the reach of existing identity systems. The identity metasystem does not compete with or replace the identity systems it connects, but rather preserves and builds upon customers' investments in their existing identity solutions. It affords the opportunity to use existing identities, such as corporate-issued identities and identities issued by online businesses, in new contexts where they could not have been previously employed.

Fosters identity system innovation. The identity metasystem should make it easier for newly-developed identity technologies and systems to quickly gain widespread use and adoption. Claims transformers can allow new systems to participate even when most participants don't understand their native claims formats and protocols.

Enables adaptation in the face of attacks. New technologies are needed to stay ahead of criminals who attack existing identity technologies. The metasystem enables new identity technologies to be quickly deployed and utilized within the metasystem as they are needed.

Creates new market opportunities. The identity metasystem enables interoperable, independent implementations of all metasystem components, meaning that the market opportunities are only limited by innovators' imaginations. Some parties will choose to go into the identity provider business. Others will provide certification services for identities. Some will implement server software. Others will implement client software. Device manufacturers and mobile telephone players can host identities on their platforms. New business opportunities are created for identity brokers, where trusted intermediaries transform claims from one system to another. New business opportunities abound.

A benefit we will all share as the identity metasystem becomes widely deployed is a **safer, more trustworthy Internet**. The metasystem will supply the widely adopted identity solution that the Net so desperately needs.

Participants in the identity metasystem can include anyone or anything that uses, participates in, or relies upon identities in any way, including, but not limited to existing identity systems, corporate identities, government identities, Liberty federations, operating systems, mobile devices, online services, and smartcards. Again, the possibilities are only limited by innovators' imaginations.

An Architecture for the Identity Metasystem: WS-* Web Services

Microsoft has worked for the past several years with industry partners on a composable, end to end architecture for Web Services. The set of specifications that make up this architecture have been named the WS-* Web Services architecture by the industry (see http://msdn.microsoft.com/webservices/). This architecture supports the requirements of the identity metasystem.

The encapsulating protocol used for claims transformation is WS-Trust. Negotiations are conducted using WS-MetadataExchange and WS-SecurityPolicy. These protocols enable building a technology-neutral identity metasystem and form the "backplane" of the identity metasystem. Like other Web services protocols, they also allow new kinds of identities and technologies to be incorporated and utilized as they are developed and adopted by the industry.

To foster the interoperability necessary for broad adoption, the specifications for WS-* are published and are freely available, have been or will be submitted to open standards bodies, and allows implementations to be developed royalty-free.

Deployments of existing identity technologies can be leveraged in the metasystem by implementing support for the three WS-* protocols above. Examples of technologies that could be utilized via the metasystem include LDAP claims schemas, X.509, which is used in Smartcards;

Kerberos, which is used in Active Directory and some UNIX environments; and SAML, a standard used in inter-corporate federation scenarios.

Identity Metasystem Architectural Diagram

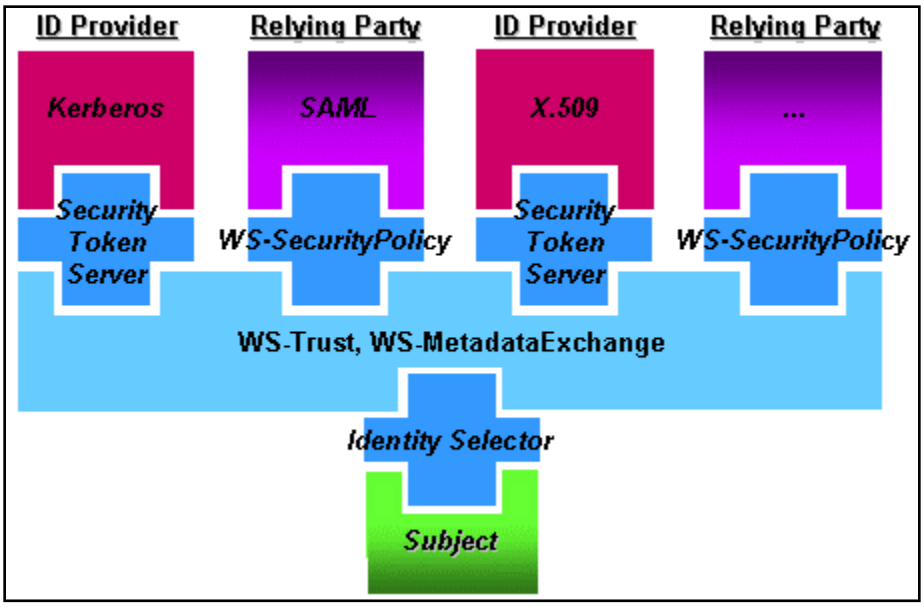


Figure 5: *An Identity Metasystem*

This figure depicts sample relationships between a subject, identity providers, and relying parties, showing some of the technologies used by the metasystem and by specific systems utilized through the metasystem.

- The *Security Token Server* implements the WS-Trust protocol and provides support for claims transformation.
- Relying parties provide statements of requirements, expressed in terms of the WS-SecurityPolicy specification, and made available through the WS-MetadataExchange protocol.
- The Identity Selector implements the consistent user experience. After being invoked by an application, it performs the negotiation between relying party and identity provider(s), displays the identities of “matched” identity providers and relying parties to the subject (e.g. the end user), obtains claims, and releases them to the application under the supervision of the subject.

Implementing the Identity Metasystem

.NET 3.0 includes the following software components for participation in the identity metasystem:

Identity selector: CardSpace is a .NET 3.0 component that provides the consistent user experience required by the identity metasystem. It is specifically hardened against tampering and spoofing to protect the end user's digital identities and maintain end-user control. Each digital identity managed by "InfoCard" is represented by a visual "Information Card" in the client user interface. The user selects identities represented by "InfoCards" to authenticate to participating services.

Simple self-issued identity provider: CardSpace also includes a simple identity provider that enables individual PC users to create and utilize self-issued identities, enabling password-free strong authentication to relying parties. A self-issued identity is one where the user vouches for the information they are providing, much like users do today when registering with a Web site. We are implementing the simple self-issued identity provider to help bootstrap the identity metasystem; we believe self-issued identities will continue to be accepted for certain classes of services. Identities hosted in the simple self-issued identity provider will not include or store sensitive personal information, such as Social Security numbers (or other national ID numbers if these are developed) or credit card numbers. Self-issued identities are not intended to provide the full range of features that a managed identity provider can offer - the market is wide open for companies to provide managed identity solutions to consumers.

Active Directory identity provider: This is a managed identity provider integrated with Active Directory. It includes a full set of policy controls to manage the use of Active Directory identities in the identity metasystem. Active Directory Federation Services, a new Active Directory feature shipping in Windows Server 2003 R2, is the first step to integrating identities in Active Directory with the identity metasystem.

WCF: Windows Communication Foundation web services provide developers a way to rapidly build and deploy distributed applications, including relying party services in the identity metasystem.

The identity metasystem preserves and builds upon customers' investments in their existing identity solutions, including Active Directory and other identity solutions. Microsoft's implementation will be fully interoperable via WS-* protocols with other identity selector implementations, with other relying party implementations, and with other identity provider implementations.

Non-Microsoft applications have the same ability to use CardSpace to manage their identities as Microsoft applications will. Others can build an entire end-to-end implementation of the metasystem without any Microsoft software, payments to Microsoft, or usage of any Microsoft online identity service.

What About Passport?

Microsoft's best-known identity effort is almost certainly the Passport Network (formerly .NET Passport). Microsoft has learned a great deal from building one of the largest Internet scale authentication services in the world, and applied these hard-won lessons in developing the Laws of Identity, the identity metasystem, and several of our products.

Passport was originally intended to solve two problems: to be an identity provider for the MSN and Microsoft properties, and to be an identity provider for the Internet. It succeeded at the first, with over 250 million active Passport accounts and over 1 billion authentications per day. As for the second original goal, it became clear to us through continued engagement with partners, consumers, and the industry that in many cases it didn't make sense for Microsoft to play a role in transactions between, for instance, a company and its customers.

Apart from its input to our thinking on the Laws of Identity, it is worth mentioning that operating the Passport service has helped Microsoft gain a deep understanding of the operational and technical challenges that large-scale identity providers face. These experiences have helped us ensure that our identity products meet the needs of large-scale deployments.

The identity metasystem is different from the original version of Passport in several fundamental ways. The metasystem stores no personal information, leaving it up to individual identity providers to decide how and where to store that information. The identity metasystem is not an online identity provider for the Internet; indeed, it provides a means for all identity providers to coexist with and compete with one another, with all having equal standing within the metasystem. Finally, while Microsoft charged services to use the original version of Passport, no one will be charged to participate in the identity metasystem.

The Passport system itself has evolved in response to these experiences as well. It no longer stores personal information other than username/password credentials. Passport is now an authentication system targeted at Microsoft sites and those of close partners – a role that is clearly in context and with which our users and partners are very comfortable. Passport and MSN plan to implement support for the identity metasystem as an online identity provider for MSN and its partners. Passport users will get improved security and ease of use, and MSN Online partners will be able to interoperate with Passport through the identity metasystem.

Conclusion

In this chapter we briefly discussed the benefits of a identity and access strategy. We also discussed a set of design principles for facilitating a trusted subsystem. The chapter closed with a detailed look at the capabilities of an identity metasystem. These capabilities are exposed today in the Microsoft .NET 3.0 Framework.

Many of the problems on the Internet today, from phishing attacks to inconsistent user experiences, stem from the patchwork nature of digital identity solutions that software makers have built in the absence of a unifying and architected system of digital identity. An identity metasystem, as defined by the Laws of Identity, would supply a unifying fabric of digital identity, utilizing existing and future identity systems, providing interoperability between them, and enabling the creation of a consistent and straightforward user interface to them all. Basing our efforts on the Laws of Identity, Microsoft is working with others in the industry to build the identity metasystem using published WS-* protocols that render Microsoft's implementations fully interoperable with those produced by others.

SOA Case Study: OTTO

Since its first mail-order catalog in 1950, OTTO has always sought retail innovation by introducing new distribution methods, customer channels, and shopping experiences. Today, OTTO is the world's number one mail-order company and the number two online retailer.

OTTO decided to build a virtual store for fashion apparel that would transcend traditional e-commerce limitations and foster closer relationships with customers. OTTO also wanted to introduce entirely new e-commerce features, such as community tools and drag-and-drop user controls.

OTTO built the virtual OTTO Store using the Microsoft .NET Framework 3.0. OTTO Store is a smart client that takes advantage of local hardware and software resources for a rich and responsive user experience, but it also integrates deeply with Web resources. OTTO Store uses Windows CardSpace for application authentication functions. Windows CardSpace provides users with an easier and safer shopping experience, saving them the hassle of typing in user names and passwords. OTTO is considering Windows CardSpace for use across its e-commerce operations, including future iterations of its leading retail Web site.

The entire Case Study is available online at http://www.microsoft.com/casestudies/casestudy.aspx?casestudyid=200504.

See other SOA case studies at http://www.microsoft.com/casestudies/search.aspx?Keywords=SOA.

References:

1. "How To: Use Impersonation and Delegation in ASP.NET 2.0", August 2005, J.D. Meier, Alex Mackman, Blaine Wastell, Prashant Bansode, Andy Wigley, Kishore Gopalan, Patterns and Practices (http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag2/html/PAGHT000023.asp)
2. "Authentication and Authorization", January 2004, Patterns and Practices (http://msdn.microsoft.com/library/default.asp?url=/library/en-us/secmod/html/secmod03.asp)
3. "Developing Identity-Aware ASP.NET Applications, Identity and Access Management Services, July 8th 2004. (http://www.microsoft.com/technet/security/topics/identitymanagement/idmanage/P3ASPD_1.mspx)