JOURNAL3

Dear Architect

While it is often difficult to reach agreement, one thing I think all architects can agree on is that our jobs are getting harder, not easier. We're facing an ever-increasing level of complexity with an ever-shrinking level of resources. Whether its new challenges such as increased regulatory compliance or old-fashion challenges like feature creep and slashed budgets, the architect's job continues to become more difficult as well as more important every day.

Abstraction is the architect's key tool for dealing with complexity. We've seen the evolution of architectural abstractions techniques such as models and patterns; however we have yet to realize much in the way of measurable value from them. Models and patterns are useful for communicating effectively with our peers, but so far they haven't helped drastically reduce the amount of resources it takes to build and operate a system. In order to continue to deal with increasing complexity, we need to get much more pragmatic about using abstraction to solve problems.

In this, the third issue of JOURNAL, we focus on raising the abstraction level. This focus ranges from general techniques for abstraction to the introduction of an industrialized manufacturing approach for software development. Along the way, we cover the use of patterns to describe business functions as well as identity and data management.

Regardless of our disagreements, the looming crisis of complexity threatens to severely impede the progress of software construction and the businesses that rely on that software. Going forward, JOURNAL will continue to deliver information from all aspects of the software architecture industry and to provide the guidance architects need.

Harry Pierson
architect,
D&PE Architecture Strategy,
Microsoft Corporation

Editorial

By Arvindra Sehmi

Dear Architect
Welcome to the summer issue of JOURNAL. These last few months have been really exciting for 'architecture' as a topic at Microsoft. The Microsoft® architecture center has become established as a leading portal for architectural content and a springboard for thousands of our customers and partners to dive into excellent guidance on architecting and developing solutions on the Windows® platform.

This issue has an abundance of architectural gems written by respected architects from Microsoft and valued partners and I'm confident that it has raised the level of content quality to a higher level.

We start with a paper by Don Awalt and Rick McUmber of RDA Corporation, also members of the Microsoft Architecture Advisory Board, who reveal many secrets of great architects. They tackle a very hard problem faced daily by Enterprise architects, namely the challenge of high complexity in systems development which is compounded by ever-changing needs of the business and pressure to adopt new technologies as they emerge. The key secret of great architects they reveal begins with a mastery of solution conceptualization and abstraction. The way in which the authors have dissected the problem and provided an exemplary walkthrough of the solution process is evidence itself of such mastery.

Jack Greenfield from Microsoft's Enterprise Frameworks and Tools division discusses in his article important new thinking on a critical business imperative troubling many organizations today – how to scale up software development? As currently practiced, software development is slow, expensive and error prone, and results in a multitude of well-known problems. Despite these shortcomings, the 'products' of software development obviously provide significant value to consumers, as shown by a long term trend of increasing demand. To address these shortcomings a case is made for a 'Software Factories' methodology to industrialize the development of software which is described in detail in a forthcoming book of the same name by Jack Greenfield and Keith Short, from John Wiley and Sons.

Feedback from customers to Microsoft on the challenges of implementing SOA systems has been very consistent; issues in managing identities, aggregating data, managing services, and integrating business processes have been cited over and over again as major road blocks to realizing more efficient and agile organizations. Frederick Chong from the Architecture Strategy team in Microsoft writes a paper on one of these challenges, namely Identity and Access Management. He provides a succinct and comprehensive overview of what I&AM means using a simple framework consisting of three key areas: identity life cycle management, access management, and directory services.

Microsoft's Philip Teale and Robert Jarvis of SA Ltd. follow with the second part of their paper discussing business patterns – which are essentially architectural templates for business solutions. In this paper they describe how to develop business patterns based on business functions, data, and business components and also show how these elements can be used to engineer software systems. A realistic but simplified example is used to show how to use standard techniques to develop descriptions of these elements required for a business pattern.

Next, Easwaran Nadhan and Jay-Louise Weldon, both from EDS, examine various data transfer strategies for the enterprise which enable timely access to the right information and data sharing such that business processes can be effective across the enterprise. They describe eight options and analyze those using criteria such as data latency requirements, transformation needs, data volume considerations, and constraints regarding the level of intrusion and effort that can be tolerated by an enterprise in order to realize the expected benefits.

The final paper is part two of Soumen Chatterjee's description of SOA messaging patterns. Traditionally messaging patterns have been applied to enterprise application integration solutions. Soumen uses these patterns to explain how SOA can be implemented. His insights are derived from the original work of Hohpe and Woolf's book on Enterprise Integration Patterns. However, Soumen shows us how the same messaging patterns described in the book can be applied equally effectively at the application architecture level, especially in SOA-based solutions, because they too are fundamentally message-oriented.

Please keep up to date on the web at the Microsoft® architecture center and specifically at the home for JOURNAL http://msdn.microsoft.com/architecture/journal where you'll be able to download the articles for your added convenience. And keep a look out for announcements of a new JOURNAL subscription service coming soon.

As always, if you're interested in writing for JOURNAL please send me a brief outline of your topic and your resume to asehmi@microsoft.com

Now put your feet up and get reading!

Arvindra Sehmi
Architect, D&PE, Microsoft EMEA

Secrets of Great Architects

By Don Awalt and Rick McUmber, RDA Corporation

Applying Levels of Abstraction to IT Solutions

Enterprise architects are challenged by the vast amount of complexity they face. It's one thing to develop an isolated, departmental application that automates a business task. However, it is quite another to design and assemble a worldwide network of IT labs filled with applications, servers and databases which support tens of thousands of IT users, all supporting various business activities. Compounding the complexity, the IT network must always be available, responsive and protect the corporation's precious information assets. In addition to all of this, the IT network must be flexible enough to support the ever-changing needs of the business and to adopt new technologies as they emerge.

Some architects clearly stand out and thrive in this complexity. We've been fortunate to work side by side with some truly great analysts and architects over our careers. Reflecting on these experiences, we've analyzed what makes an exceptional architect.

Without exception, all great architects have mastered the ability to conceptualize a solution at distinct levels of abstraction. By organizing the solution into discrete levels, architects are able to focus on a single aspect of the solution while ignoring for the moment all remaining complexities. Once they stabilize that part of the solution, they can then proceed to other aspects, continuously evolving and refining the layers into a cohesive model; one that can ultimately be implemented.

Most software developers know that they should decompose the solution into levels of abstraction. However, this is very difficult to apply in practice on actual projects. Upon encountering the first hurdle, it's easy to abandon the levels in the rush to start coding. Great architects work through the challenges and are disciplined to maintain the levels throughout the entire project lifecycle. They realize if they don't, they'll eventually drown in the complexity.

This article presents techniques for applying levels of abstraction to IT solutions. We first demonstrate the approach through a simple example, and then propose a structure of system artifacts based on formal levels of abstraction.

Levels of Abstraction: A Powerful Weapon for all Engineers

Other engineering disciplines, such as civil engineers, have been coping with complexity by leveraging levels of abstraction for centuries. Let's study how other, more mature, engineering disciplines apply levels of abstraction, starting with the electrical engineers who design computer systems which continually grow increasingly complex with each new generation.

Hardware Engineers

System designers model computer systems using levels of abstraction. Each level is well defined and provides a distinct perspective of the system. Many systems are designed at three primary levels – System, Subsystem and Component as shown in *Figure 1*.

Layering enables the engineers to integrate an enormous amount of complexity into a single, working computer system. It's impossible to comprehend a computer strictly at the level of its atomic parts. There are ~25,000,000 transistors on a single Intel Itanium® chip alone!

This approach of breaking complexity down into layers of abstraction is of course not unique to IT-related disciplines. A similar approach is used in countless other disciplines from aeronautical engineering to microbiology.

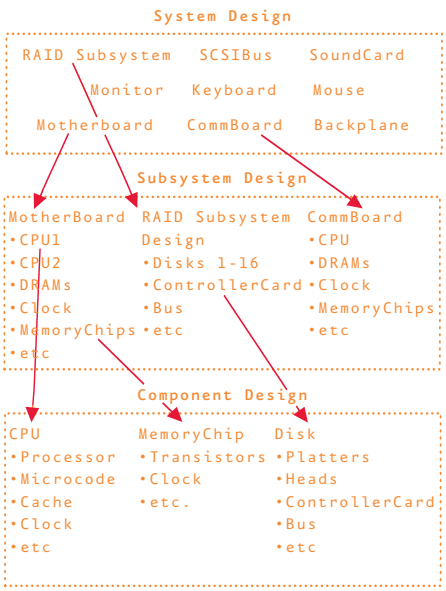
Core Principles When Applying Levels of Abstraction

All engineers follow this core set of principles when applying levels of abstraction. These principles also hold true when applying levels of abstraction to software.

The number and scope of the levels are well defined – In order for engineers to collaborate on a complex system, all team members must share the same understanding of the levels. So as designers make design decisions, they must file those decisions at the appropriate level of detail.

Multiple views within each level – The complexity within a single level

Figure 1. Levels of Abstraction for a Computer System



"By organizing the solution into discrete levels, architects are able to focus on a single aspect of the solution while ignoring for the moment all remaining complexities."

The three levels of abstraction are defined as follows:

Level Name	Level Scope
System	The computer engineer designs the system by integrating various subsystems, such as a backplane, circuit boards, a chassis, internal devices such as CD/DVD drives & disk drives and external devices such as a display monitor, keyboard and mouse. The engineer thinks of the system in terms of its subsystems, their interfaces and their interconnections. Each subsystem interface is documented as a formal specification to the subsystem designer.
Subsystem	The subsystem engineer designs the subsystem by integrating components. For example the motherboard designer architects the motherboard by integrating components such as memory chips, DMA chips and a CPU chip. Likewise, the display monitor engineer designs the monitor by integrating components such as a video card and CRT. The subsystem engineer thinks of the subsystem in terms of its components, their interfaces and their interconnections. Each component interface is documented as a formal specification to the component engineer.
Component	The component engineer designs the component by assembling and integrating subcomponents. For example, the memory chip designer architects the chip as a complex network of integrated circuits.

Level Name	Level Scope
Domain	– The company is the 'black box' central actor. – Model the business from the perspective of the business' external actors. – Model only the business interactions. Omit the communication mediums.
Business Processes	– Model the business process workflows that are realization of the domain level's business interactions. – The system serves as the 'black box' central actor. – Model the business processes from the perspective of the system's external actors. Include the communication mediums for completing the business transactions.
Logical	– Model the internal design of the system. – The major system components function as the main actors. – Model the system behavior from the perspective of inside the system 'black box'.
Physical	– Model the physical structure of the implementation.

can become too much to grasp all at once. In this case, engineers present the design within a single level through multiple views. Each view presents a particular aspect of the design, yet remains at the same level of abstraction. For example, the motherboard engineer creates a view for each layer of the board, modeling the design of the connection pathways for that layer.

Must maintain consistency among the levels – In order for the system to function as intended, each subsequent layer must be a proper refinement of its parent layer. If the computer system designer switches from an IDE bus to a SCSI bus, then the interface specifications for all devices must also switch to SCSI. If the levels are not synchronized, the system won't perform as envisioned at the top level.

Apply Levels of Abstraction to IT Systems

Now that we've examined how other disciplines apply levels of abstraction, let's apply the technique to IT solutions[1]. The following sections present techniques for applying levels of abstraction to model the requirements, design and implementation of a typical IT application. The techniques are presented through a simple, instructional example of an online order system for a hypothetical retailer. In our example, we not only include the architecture, but expand the scope to include the system requirements and the business context as defined by the retail industry.

[1] Many have successfully applied levels of abstraction to software. Ed Yourdon and Tom DeMarco proposed structured analysis and structured system design in 1979. Many branches of the US Government standardized on DoD's 2167A standard which requires systems to be composed of a hierarchy of hardware and software configuration items. The DBA community frequently applies levels of detail to model relational databases. In particular, the Bachman toolset and James Martin's Information Engineering Methodology (IEM) model databases logically, then physically. A Google search of 'software levels of abstraction' returns several results, however, most are from the academic community and seem to focus on formal computer languages.

Simple Framework: Four Levels of Abstraction

Our simple example defines the following four levels of abstraction for an IT solution:

Within each level, we present both the dynamic view and static view of the behavior for that particular level. Whereas the dynamic view models the messaging among the objects, the static view models the structure and relationships among the objects.

Domain Level of Abstraction

Applying the scoping rules above, the retailer serves as the black box central actor in the domain level. The customer serves as the external actor. The domain level is modeled from the perspective of the customer. Only the purchase interactions are modeled. The communication mediums used to complete the purchase are not included at this level but are introduced at the business process level.

Dynamic View

The dynamic view within the domain level models the interactions between the customer and the retailer. The following figure summarizes the domain context and contains a simple use case narration of the business interactions.

Static View

The static view of the domain level models the class structure and their relationships of the objects witnessed in the use case. In other words – 'What objects does the customer need to understand in order to accomplish the purchase transaction at this level of abstraction?' *Figure 3* presents the class

Figure 3. Domain level static view for purchasing items from a retailer

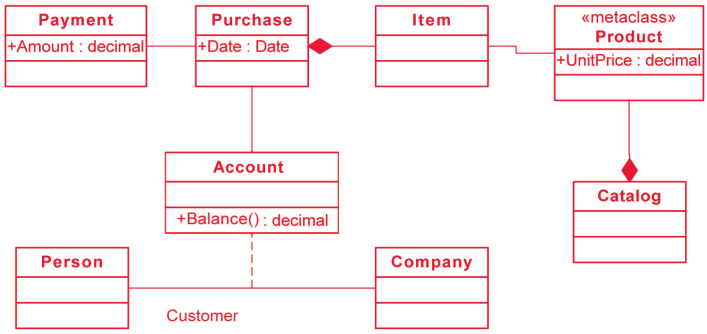


diagram for the static view of the domain level.

The customer is an instance of *Person*. The relationship between the customer and the retailer is embodied as an *Account*. All *Purchases* are associated with the customer's *Account*. The *Purchase* is associated with each of the *Items* being purchased. Each *Item* is of a specific *Product,* where *Product* follows the meta-class pattern. Instances of *Product* are in effect classes themselves. Modeling *Product* as a meta-class makes our model more flexible in that adding additional *Products* to the *Catalog* is purely a data driven process and doesn't impact the class model. Rounding out the classes, each *Payment* is associated with its *Purchase.*

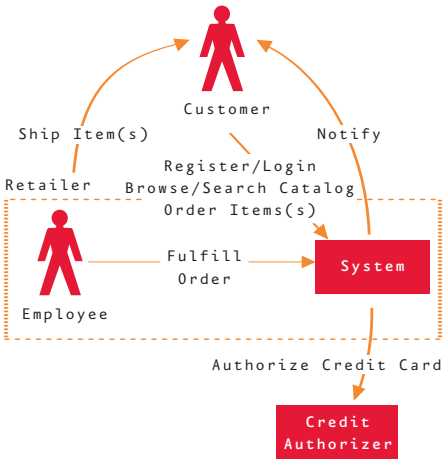
Figure 2. Domain level dynamic view for purchasing items from a retailer


```
Purchase items
1. Customer browses the retailer's
   catalog of items
2. Customer selects one or more items
3. Customer pays the retailer for
   the items
```

As you can see, the model at this level is representative for most any retailer – online or brick-and-mortar, large or small. This illustrates why the [Industry] Domain Model should indeed define the company as the black box central actor. Companies in the same industry tend to support the same set of business interactions with their external actors. Moreover, domain models exclude the specific business processes of the company as they can vary widely among companies in the same industry.

The domain level focuses strictly on business interactions viewed from the perspective of the external actor. We must be careful to not include implementation mechanisms for accomplishing the interactions. These details belong at the next level of abstraction. So in this case, we only model browse, select, purchase and pay. We do not model how these interactions are accomplished – via telephone, US Mail, email, web application, in person, check, credit card or cash.

Figure 4. Business process level dynamic view for purchasing items online from a retailer



Purchase Items Online

1. Customer visits the retailer's web site
2. Customer browses the retailer's online catalog
3. Alternatively, the customer performs a keyword search against the catalog
4. Customer selects item(s)
5. Customer logs into her online account
6. Customer creates an order for the items(s)
7. Customer pays for the items with her credit card
8. Employee checks the system for unfulfilled orders
9. Employee fulfills the order by pulling items from the inventory, packaging and shipping them to the order's shipping address
10. Employee updates the system with the shipping information which decrements the item inventory
11. System automatically sends an email notification to the customer indicating the shipping status

Variations

- Customer may login at any point prior to creating the order
- At step 5, customer can register a new user account if she does not yet have one
- At step 7, credit authorizer could reject credit card payment if authorization fails

Business Process of Abstraction

The next level of abstraction models the company's business processes for realizing the interactions that were captured at the domain level. The system level 'zooms inside' the company black box and identifies all employees and systems that collaborate in order to perform the business transaction. At this level, the system to be developed serves as the black box central actor.

Applying the scoping rules for the system level, the online order system serves as the black box central actor. The customer and the employee serve as external actors. The system level is modeled from the perspective of the customer and the employee. The customer performs the purchase online. Payment is made via credit card. The order is fulfilled by shipping the items to the customer's shipping address. Notification of shipment is sent via email.

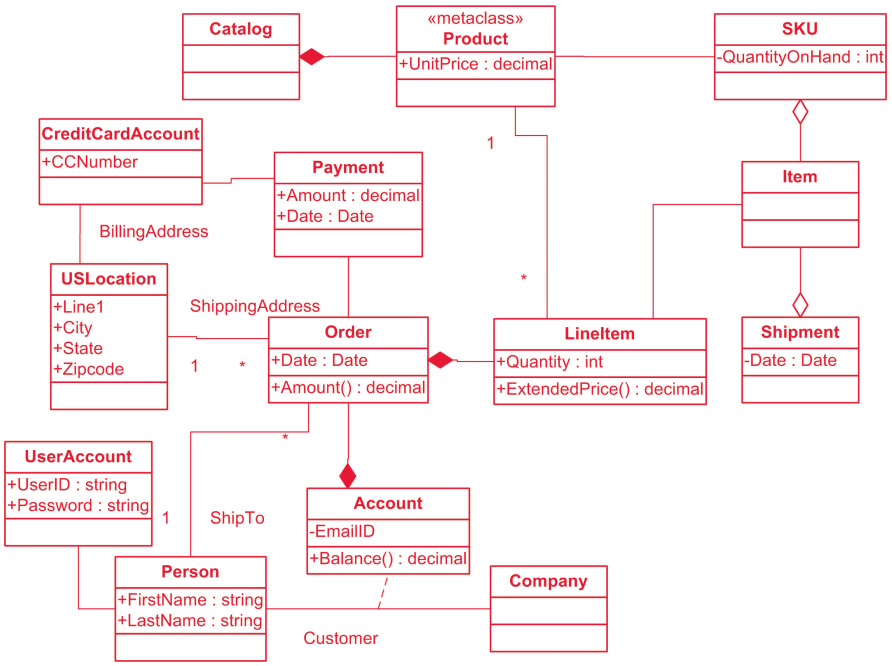
Dynamic View

The dynamic view replays the domain level purchase transaction, this time exposing the internal business processes of the retailer. *Figure 4* summarizes the business process context and contains a simple use case narration of the interactions among the system and its actors.

Static View

The static view at this level refines the class model to capture the objects witnessed in the business process level use cases. In other words – 'What objects do the customer and employee need to understand in order to create an order online and fulfill the order?' *Figure 5* presents the class diagram for the business process static view.

Figure 5. Business process level static view for purchasing items online from a retailer



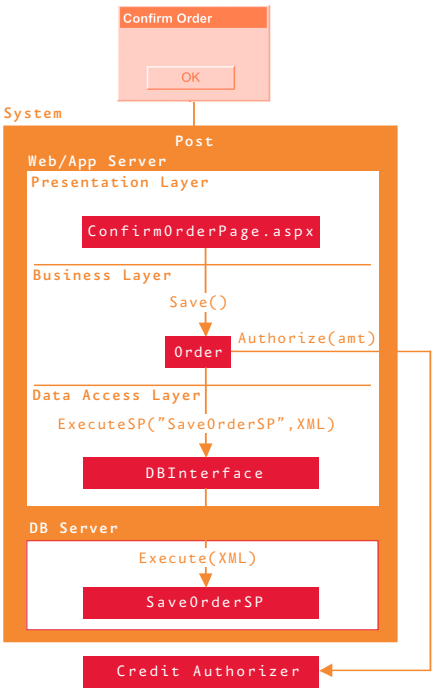
"In order for engineers to collaborate on a complex system, all team members must share the same understanding of the levels."

We refine the domain class model to capture the perspective at this level of abstraction. The *Person, Account* and *Company* abstractions remain the same as well as Catalog and Product. However, the abstract *Purchase* event from the domain model is replaced with an *Order*. *Orders* contain *LineItems* which are associated with the *Product* in the *Catalog*. Since this level models the company's internal business process, we need to capture the inventory onhand (an attribute of a stock keeping unit *(SKU)* which represents an inventory of items at a particular location). We also model the customer's *UserAccount* which provides access to the online system. *Payment* is accomplished by using a *CreditCardAccount*. *Location* represents a geopoint within the US and serves as the billing address as well as the *Order's* shipping address. *Shipment* contains *Items* included in the *Shipment*.

The system level of abstraction usually requires a great deal of creativity because it is at this level that we invent ways to streamline business processes. In doing so, it is common to realize a single domain level transaction using several different mediums at the business process level. For example, a purchase can be accomplished online, over the phone, by mailing or faxing an order form or in-person at the retail store. Each of these would need to be modeled at the business process level. Notice that even though the Credit Authorizer is an external actor to the retailer, it is introduced at this level because it is only needed to implement a business process that first appears at this level.

Lastly, notice that the system is technology independent. Our online purchase system could be implemented with any web technology. Selecting technologies inside the system black box is an architecture decision.

Figure 6. Logical level dynamic view for purchasing items online from a retailer



Confirm Order

1. Customer reviews the order summary and confirms by submitting the form
2. ASP.Net posts the ConfirmOrderPage who retrieves the current order and invokes Order.Save()
3. Order.Save() validates order business rules. If any are violated, Save returns the list of errors
4. Otherwise, Save() invokes the credit authorizer to authorize the payment amount
5. If successful, Save() serializes all parts of the Order and invokes the Data Access Layer to execute the SaveOrderSP
6. The Data Access Layer forwards the request to the stored procedure
7. SaveOrderSP saves the order in the database

Logical Level of Abstraction

The logical level zooms inside the system black box, exposing the high level design of the system. The architect selects the technology and defines the high level system structure. In our simple example, the system is comprised of an IIS/ASP.Net server that hosts the presentation, business and data access layers and a SQL Server database server that hosts the persistent data.

Dynamic View

The dynamic view at the logical level traces the message flows through the major components of the system. As an example, *Figure 6* traces the flow when submitting the ConfirmOrder web form.

Static View

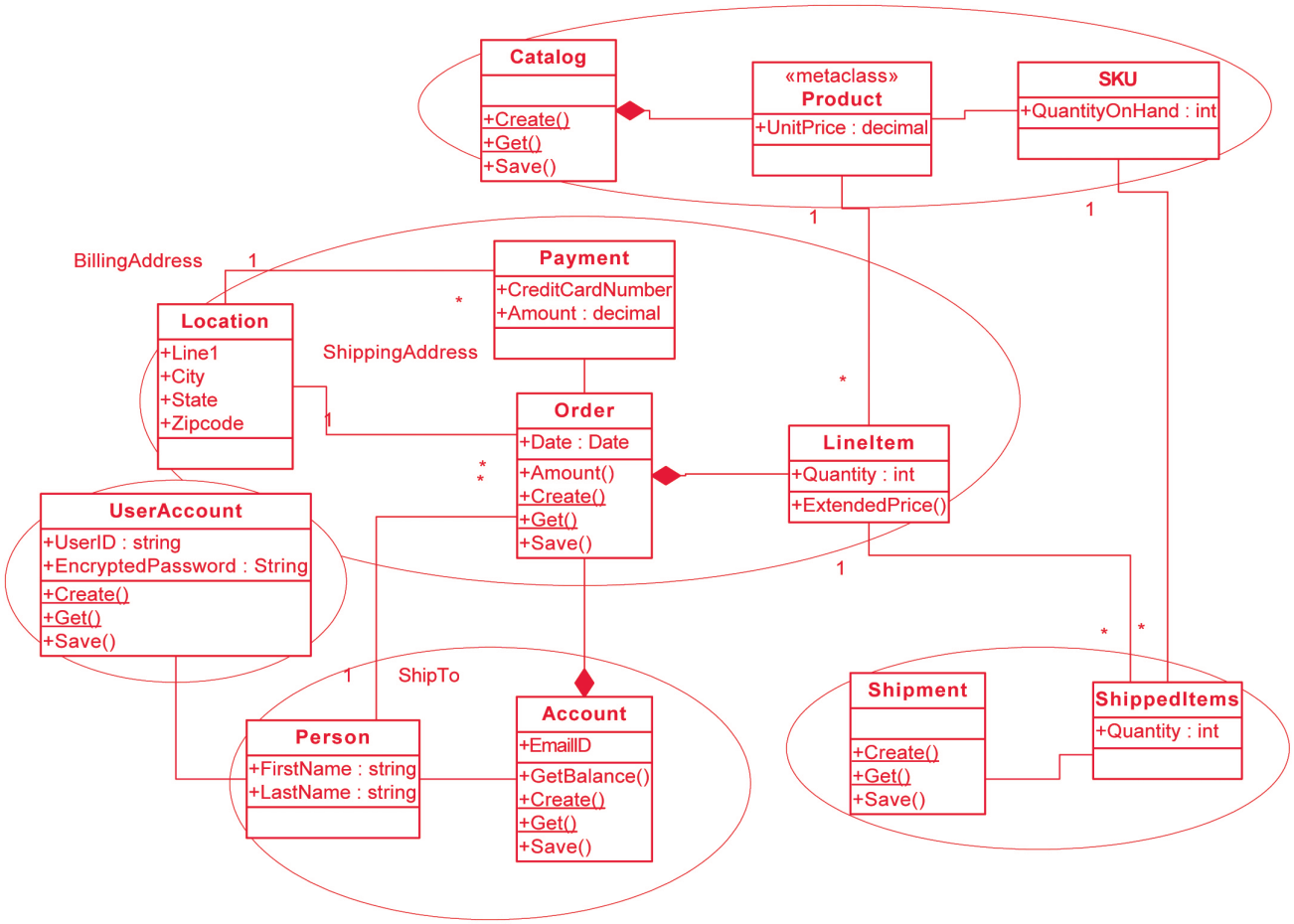
The static view at this level also switches our perspective to the system internals. Whereas the business process level modeled the real world abstractions that appear in the business processes, this level models the abstractions as they are to be represented inside the system. In an actual system, the architect would design the classes for each software layer (presentation, business and data access). To keep this article brief, *Figure 7* presents just the static design for the business layer to show how the system level abstractions are refined to the design.

The architect refines the system level classes to design the business layer interface.

All accounts and customers in the system are the retailer's, so it's not

"Domain models exclude the specific business processes of the company as they can vary widely among companies in the same industry."

Figure 7. Logical level static view for purchasing items online from a retailer



practical to create a single instance of Company and associate it with all accounts, so Company is omitted at this level. Rather than creating a separate instance for each *CreditCardAccount,* we simply store the credit card number and billing address with the *Payment.* Further, it is not practical for the system to create an instance for every *Item* sold, so *Item* is removed from the model and instead the model tracks the quantity of items ordered in the *LineItem* and the quantity of items shipped in the new *ShippedItems* class.

The architect also defines the granularity of the services that the business layer exposes. For this example, the business layer exports Create, Read, Update and Delete (CRUD) services for *Account, UserAccount, Order, Shipment* and *Catalog.* The ellipses indicate the CRUDing granularity.

Note that even though the classes at this level are not a proper superset of the business process classes, the architect arrives at this design by directly refining the business process classes, changing the perspective from outside the system to inside the system.

Physical Level of Abstraction

The physical level of abstraction captures the structure of the system implementation. The system is implemented as a network of nodes, each node configured with hardware and software. The three software layers in the logical view (presentation, business and data) are physically implemented as code and deployed onto these nodes. Persistent classes in the

logical view are physically stored in relational tables in a SQL Server database.

Dynamic View

The dynamic view traces the message flows through the nodes of the physical configuration. The ConfirmOrder HTTP post flows from the customer's browser through the internet through the retailer's firewall to the web server where Windows forwards it to IIS who passes it to ASP.Net who then dispatches ConfirmOrder.aspx. Fortunately, modern development tools insulate us from the majority of the physical network. Architects, however, need to understand the physical layer in order to avoid network bottlenecks and security exposures.

Static View

The static view (*Figure 7*) refines the persistent classes in the logical view to their physical representation. In our retail example, the business layer classes are stored in the following SQL Server tables.

Classes map to relational tables and attributes are implemented as columns. One-to-one relationships and one-to-many relationships are implemented using a foreign key. Optimistic concurrency is implemented by assigning a datetime field to each parent class being CRUDed.

When designing the logical level, the architect focuses mainly on implementing system functionality. Confident that the system functionality is covered, the architect can focus on optimizing the implementation at the physical level.

Evolve the Levels through Iterations

Having established this framework, the architect evolves the solution over several iterations. Each iteration incorporates additional functionality – invoices, back orders, order in person, order by phone, and so on. In each case, the architect updates the appropriate level of abstraction, and then refines the updates to the physical implementation level.

Revisit the Levels of Abstraction Core Principles

Let's test our example against the core level of abstraction principles.

The number and scope of the levels are well defined

We have four distinct levels – Company black box, System black box, Logical design inside the system and Physical implementation.

Multiple views within each level

In this simple example, we presented a dynamic view and static view at each level.

Must maintain consistency among the levels

If a change is made to the domain model, the impact of the changes must flow down to the lower levels. For example, if the retailer decides to offer maintenance contracts for its products, the analyst would add *MaintenanceContract* to the domain model and refine it to its physical representation. Synchronizing all levels is critical for maintaining large systems. As enhancement requests are submitted, the analyst performs an impact assessment to the appropriate level of detail. Some enhancements impact the domain level (and therefore all subsequent levels). Others merely impact the physical level.

Figure 8. Physical level static view for purchasing items online from a retailer

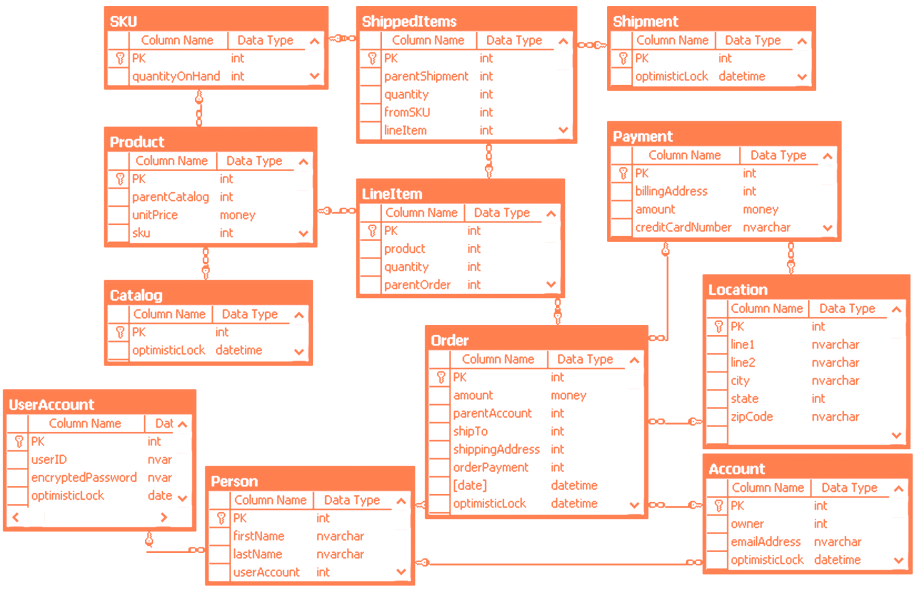


Figure 9. RUP configuration organizing project artifacts into well defined levels of abstraction

Level Name	Level Scope
Domain	The company is the 'black box' central actor. Model the business from the perspective of the business' external actor. Model only the business interactions. Do not include the communication mediums.
Project Vision	Project mission, project business objectives, project return on investment.
Business Process	Model the business process workflows that are realization of the domain level's business interactions. The system serves as the 'black box' central actor. Model the business processes from the perspective of the system's external actors. Include the communication mediums for completing the business transactions.
UI Specification	UI design and UI prototype of system functionality. Demonstrate how system users are able to realize the above business process workflows.
Detailed Requirements	Specify the lowest level details that represent the external interface of the system. For example, 'US zipcodes must be masked as xxxxx or xxxxx-xxxx'. Specify the proficiency requirements – performance, availability, security, internationalization, configuration, scalability and flexibility.
Architecture	Inside the system 'black box'.
Implementation	Database schemas, source code, reference data, configuration files.

Logical View	Concurrency View	Security View	Deployment View	Component View	Data View
Logical Design	Concurrency Design	Security Design	Network Configuration	Component Interfaces	Logical Data Model
	Concurrency Implementation	Security Implementation	Box Configuration	Component Implementations	Physical Data Model

Scaling Levels to Support Enterprise Solutions

Now that we've presented a simple example with four levels of abstraction, let's scale the approach to support solutions for IT enterprises. *Figure 9* presents a Rational Unified Process (RUP) configuration that organizes project artifacts in to well defined levels of abstraction.

The levels in the table are described below.

Domain – The domain level captures the business context for a project.

Project Vision – The project vision communicates the business impact the system will have on the business. It quantifies this impact in a return on investment analysis. The project vision represents the highest level of abstraction of the project.

Business Process – The system level models the business processes within the company. For extremely complex organizations, this level can be subdivided into sublevels – division, interdepartmental and intradepartmental.

UI Specification – The UI specification designs the user interface that realizes the business processes. It is comprised of a UI design document and a functional UI prototype.

Detailed Requirements – The detailed requirements specify the lowest level abstraction of the system requirements. It includes details such as data type formats and detailed business rules. It also contains the proficiency requirements such as performance, availability, security, internationalization, configuration, scalability and flexibility requirements.

"No longer do we overwhelm the business users with a single, monolithic functional specification."

Architecture – The system architecture is organized into six views –

– *Logical:* Defines the software layers and major abstractions that perform the system functionality.
– *Concurrency:* Captures the parallel aspects of the system, including transactions, server farms and resource contention.
– *Security:* Defines the approach for authentication, authorization, protecting secrets and journaling.
– *Deployment:* Defines the network topology and deployment configuration of the system.
– *Component:* Defines the system components, their interfaces and dependencies.
– *Data:* Defines the design structure of the persistent data.

Benefits

Organizing system artifacts into discrete levels of abstraction delivers several benefits:

– It separates the system requirements into three distinct levels of abstraction – Business Processes, UI Specification and Detailed Requirements. No longer do we overwhelm the business users with a single, monolithic functional specification. Instead, we communicate the system requirements in three refined levels of detail.
– Analysts and architects are able to harness the complexity into a single, integrated model of the system.
– Architects can focus on a single aspect of the system and integrate those decisions into the overall solution.
– The levels of abstraction form the structure of the system artifacts. For example, the software architecture document dedicates a subsection for each view.
– The levels of abstraction provide direct traceability from requirements to design to implementation. Traceability enables a team to perform an accurate impact assessment when evaluating change requests.
– After developing several systems using the same framework, patterns emerge at each level of abstraction. Organizations can catalog these patterns and other best practices within each level of abstraction. This catalog of best practices serves as the foundation of a process improvement program.

Summary

All engineering disciplines apply formal levels of abstraction in order to cope with complexity. Software is no exception. In order to realize the benefits of levels of abstraction, projects must

– Formally identify the layers, each with a well-defined scope
– Split complexity within a level into multiple views
– Maintain consistency among the levels

This article demonstrated how to apply levels of abstraction through a simple example, then scaled the approach to support enterprise IT solutions. It offered a RUP configuration framework that organizes system artifacts into well defined levels of abstraction.

Self-Assessment

Does your current project apply levels of abstraction?
Are the levels well-defined?
Are the levels well understood by the project team?
If the complexity becomes too great within a level, does the team split it into views?
Does the team maintain consistency among the levels?
Would your project benefit from levels of abstraction?

Great architects follow these principles instinctively. The rest of us must consciously apply levels of abstraction and exercise discipline to maintain the levels throughout the project lifecycle.

Resources

Cockburn, Alistair. *Writing Effective Use Cases.* New Jersey: Addison-Wesley, 2001

Kroll, Per and Kruchten, Philippe. *The Rational Unified Process Made Easy: A Practitioner's Guide to the RUP.* Boston MA: Pearson Education and Addison-Wesley, 2003

DeMarco, Tom and Plauger, P J *Structured Analysis and System Specification.* Prentice Hall PTR, 1979

Online copy of DoD standard 2167A can be found at http://www2.umassd.edu/SWPI/DOD/MIL-STD-2167A/DOD2167A.html

About the Authors

Don Awalt
CEO and founder of RDA Corporation AWALT@rdacorp.com
Don Awalt is the CEO and founder of RDA Corporation, founded in 1988 as a custom software engineering firm with offices in Washington DC, Baltimore, Atlanta, Philadelphia, and Chicago. The firm, a Microsoft Gold Certified Partner, is focused on the development of enterprise web and rich client systems using the .NET Framework. Don currently serves as the Microsoft Regional Director for Washington DC, and was formerly the founding Regional Director for Philadelphia. Don is a frequent speaker at industry events, including Tech Ed, Developer Days, MSDN events, and various SQL Server and Windows events. He has served as a contributing editor for SQL Server Magazine and PC Tech Journal Magazine, and has written for other publications as well. Don's particular areas of expertise include Web Services, SQL Server, the evolution of modern programming languages, and much of the architecture and process work seen in Microsoft's Prescriptive Architecture Group (PAG).

Rick McUmber
Director of Quality and Best Practices for RDA
McUmber@rdacorp.com
Rick McUmber is Director of Quality and Best Practices for RDA. For 11 years, he worked for IBM and Rational Software Corporation respectively, developing systems for the Department of Transportation, Department of Defense, NASA and Canada's Department of National Defense. Since 1994, he has worked with RDA developing business solutions for its customers.

The Case for Software Factories

By Jack Greenfield, Microsoft Corporation

This article briefly presents the motivation for Software Factories, a methodology developed at Microsoft. It describes forces that are driving a transition from craftsmanship to manufacturing. In a nutshell, a Software Factory is a development environment configured to support the rapid development of a specific type of application. Software Factories are really just a logical next step in the continuing evolution of software development methods and practices. However, they promise to change the character of the software industry by introducing patterns of industrialization.

Scaling Up Software Development

Software development, as currently practiced, is slow, expensive and error prone, often yielding products with large numbers of defects, causing serious problems of usability, reliability, performance, security and other qualities of service.

According to the Standish Group [Sta94], businesses in the United States spend around $250 billion on software development on approximately 200 projects each year. Only 16% of these projects finish on schedule and within budget. Another 31% are canceled, mainly due to quality problems, for losses of about $81 billion. Another 53% exceed their budgets by an average of 189%, for losses of about $59 billion. Projects reaching completion deliver an average of only 42% of the originally planned features.

These numbers confirm objectively what we already know by experience, which is that software development is labor intensive, consuming more human capital per dollar of value produced than we expect from a modern industry.

Of course, despite these shortcomings, the products of software development obviously provide significant value to consumers, as demonstrated by a long term trend of increasing demand. This does not mean that consumers are perfectly satisfied, either with the software we supply, or with the way we supply it. It merely means that they value software, so much so that they are willing to suffer large risks and losses in order to reap the benefits it provides. While this state of affairs is obviously not optimal, as demonstrated by the growing popularity of outsourcing, it does not seem to be forcing any significant changes in software development methods and practices industry wide.

Only modest gains in productivity have been made over the last decade, the most important perhaps being byte coded languages, patterns and agile methods. Apart from these advances, we still develop software the way we did ten years ago. Our methods and practices have not really changed much, and neither have the associated costs and risks.

This situation is about to change, however. Total global demand for software is projected to increase by an order of magnitude over the next decade, driven by new forces in the global economy like the emergence of China and the growing role of software in social infrastructure; by new application types like business integration and medical informatics; and by new platform technologies like web services, mobile devices and smart appliances.

Without comparable increases in capacity, it seems inevitable that total software development capacity is destined to fall far short of total demand by the end of the decade. Of course, if market forces have free play, this will not actually happen, since the enlightened self interest of software suppliers will provide the capacity required to satisfy the demand.

Facing the Changes Ahead, Again

What will change, then, to provide the additional capacity? It does not take much analysis to see that software development methods and practices will have to change dramatically.

Since the capacity of the industry depends on the size of the competent developer pool and the productivity of its members, increasing industry capacity requires either more developers using current methods and practices, or a comparable number of developers using different methods and practices.

While the culture of apprenticeship cultivated over the last ten years seems to have successfully increased the number of competent developers and average developer competency, apprenticeship is not likely to equip the industry to satisfy the expected level of demand for at least two reasons:

– We know from experience that there will never be more than a few extreme programmers. The best developers are up to a thousand times more productive than the worst, but the worst outnumber the best by a similar margin [Boe81].
– As noted by Brooks [Bro95], adding people to a project eventually yields diminishing marginal returns. The amount of capacity gained by recruiting and training developers will fall off asymptotically.

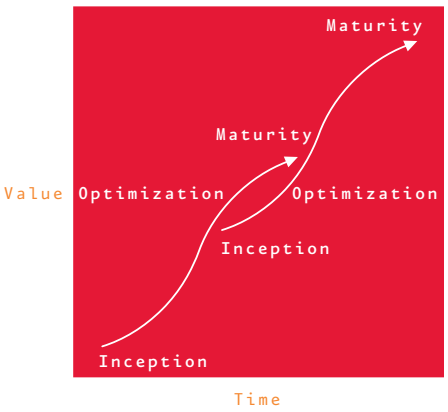
"A Software Factory is a development environment configured to support the rapid development of a specific type of application, and promises to change the character of the software industry by introducing patterns of industrialization."

The solution must therefore involve changing our methods and practices. We must find ways to make developers much more productive.

Innovation Curves and Paradigm Shifts

As an industry, we have collectively been here before. The history of software development is an assault against complexity and change, with gains countered by losses, as progress creates increasing demand. While great progress has been made in a mere half century, it has not been steady. Instead, it has followed the well known pattern of innovation curves, as illustrated in *Figure 1* [Chr97].

Figure 1. Innovation Curves



Typically, a discontinuous innovation establishes a foundation for a new generation of technologies. Progress on the new foundation is initially rapid, but then gradually slows down, as the foundation stabilizes and matures. Eventually, the foundation loses its ability to sustain innovation, and a plateau is reached. At that point, another discontinuous innovation establishes another foundation for another generation of new technologies, and the pattern repeats. Kuhn calls these foundations paradigms, and the transitions between them paradigm shifts [Kuh70]. Paradigm shifts occur at junctures where existing change is required to sustain forward momentum. We are now at such a juncture.

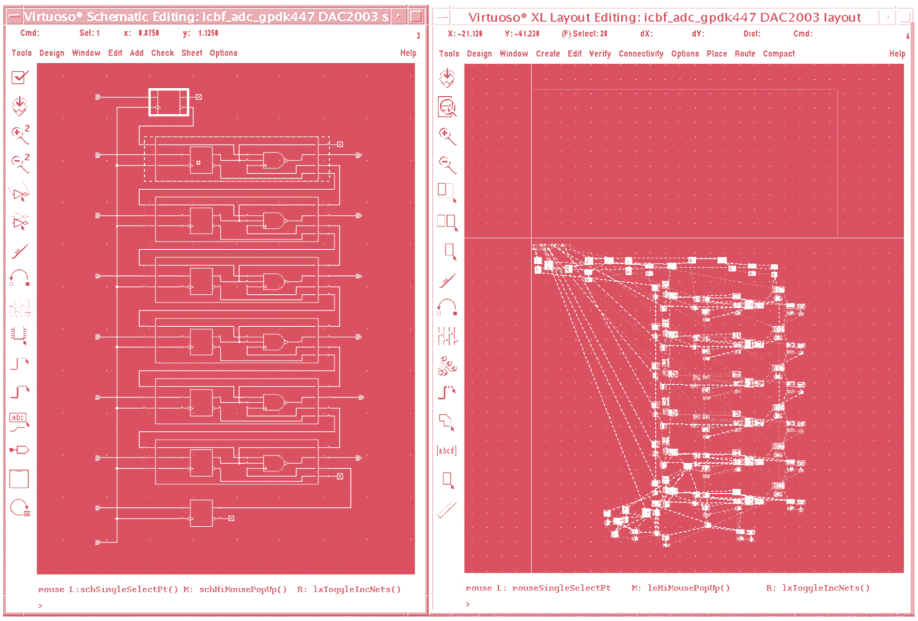
Raising the Level of Abstraction

Historically, paradigm shifts have raised the level of abstraction for developers, providing more powerful concepts for capturing and reusing knowledge in platforms and languages. On the platform side, for example, we have progressed from batch processing, through terminal/host, client/server, personal computing, multi-tier systems and enterprise application integration, to asynchronous, loosely coupled services. On the language side, we have progressed from numerical encoding, through assembly, structured and object oriented languages to byte coded languages and patterns, which can be seen as language based abstractions. Smith and Stotts summarize this progression eloquently [SS02]:

The history of programming is an exercise in hierarchical abstraction. In each generation, language designers produce constructs for lessons learned in the previous generation, and then architects use them to build more complex and powerful abstractions.

They also point out that new abstractions tend to appear first in platforms, and then migrate to languages. We are now at a point in this progression where language based abstractions have lagged behind platform based abstractions for a long time. Or, to put it differently, we are now at a point where tools have lagged behind platforms for a long time. Using the latest generation of platform technology, for example, we can now automate processes spanning multiple

Figure 2. ASIC Based Design Tools[1]



[1] This illustration featuring Virtuoso® Chip Editor and Virtuoso® XL Layout Editor has been reproduced with the permission of Cadence Design Systems, Inc © 2003 Cadence Design Systems, Inc. All rights reserved. Cadence and Virtuoso are the registered trademarks of Cadence Design Systems, Inc.

businesses located anywhere on the planet using services composed by orchestration, but we still hand-stitch every one of these applications, as if it is the first of its kind. We build large abstract concepts like insurance claims and security trades from small concrete concepts like loops, strings and integers. We carefully and laboriously arrange millions of tiny interrelated pieces of source code and resources to form massively complex structures. If the semiconductor industry used a similar approach, they would build the massively complex processors that power these applications by hand soldering transistors. Instead, they assemble predefined components called Application Specific Integrated Circuits (ASICs) using tools like the ones shown in *Figure 2,* and then generate the implementations.

Can't we automate software development in a similar way? Of course, we can, and in fact we already have. Database management systems, for example, automate data access using SQL, providing benefits like data integration and independence that make data driven applications easier to build and maintain. Similarly, widget frameworks and WYSIWYG editors make it easier to build and maintain graphical user interfaces, providing benefits like device independence and visual assembly. Looking closely at how this was done, we can see a recurring pattern.

– After developing a number of systems in a given problem domain, we identify a set of reusable abstractions for that domain, and then we document a set of patterns for using those abstractions.
– We then develop a run time, such as a framework or server, to codify the abstractions and patterns. This lets us build systems in the domain by instantiating, adapting, configuring and assembling components defined by the run time.
– We then define a language and build tools that support the language, such as editors, compilers and debuggers, to automate the assembly process. This helps us respond faster to changing requirements, since part of the implementation is generated, and can be easily changed.

This is the well known Language Framework pattern described by Roberts and Johnson [RJ96]. A framework can reduce the cost of developing an application by an order of magnitude, but using one can be difficult. A framework defines an archetypical product, such as an application or subsystem, which can be completed or specialized in varying ways to satisfy variations in requirements. Mapping the requirements of each product variant onto the framework is a non-trivial problem that generally requires the expertise of an architect or senior developer. Language based tools can automate this step by capturing variations in requirements using language expressions, and generating framework completion code.

Industrializing Software Development

Other industries increased their capacity by moving from craftsmanship, where whole products are created from scratch by individuals or small teams, to manufacturing, where a wide range of product variants is rapidly assembled from reusable components created by multiple suppliers, and where machines automate rote or menial tasks. They standardized processes, designs and packaging, using product lines to facilitate systematic reuse, and supply chains to distribute cost and risk. Some are now capable of mass customization, where product variants are produced rapidly and inexpensively on demand to satisfy the specific requirements of individual customers.

Can Software Be Industrialized?

Analogies between software and physical goods have been hotly debated. Can these patterns of industrialization be applied to the software industry? Aren't we somehow special, or different from other industries because of the nature of our product? Peter Wegner sums up the similarities and contradictions this way [Weg78]:

Software products are in some respects like tangible products of conventional engineering disciplines such as bridges, buildings and computers. But there are also certain important differences that give software development a unique flavor. Because software is logical not physical, its costs are concentrated in development rather than production, and since software does not wear out, its reliability depends on logical qualities like correctness and robustness, rather than physical ones like hardness and malleability.

Some of the discussion has involved an 'apples to oranges' comparison between the production of physical goods, on one hand, and the development of software, on the other. The key to clearing up the confusion is to understand the differences between production and development, and between economies of scale and scope.

"If the semiconductor industry used a similar approach, they would build the massively complex processors that power these applications by hand soldering transistors."

In order to provide return on investment, reusable components must be reused enough to more than recover the cost of their development, either directly through cost reductions, or indirectly, through risk reductions, time to market reductions, or quality improvements. Reusable components are financial assets from an investment perspective. Since the cost of making a component reusable is generally quite high, profitable levels of reuse are unlikely to be reached by chance. A systematic approach to reuse is therefore required. This generally involves identifying a domain in which multiple systems will be developed, identifying recurring problems in that domain, developing sets of integrated production assets that solve those problems, and then applying them as systems are developed in that domain.

Economies of Scale and Scope

Systematic reuse can yield economies of both scale and scope. These two effects are well known in other industries. While both reduce time and cost, and improve product quality, by producing multiple products collectively, rather than individually, they differ in the way they produce these benefits.

Economies of scale arise when multiple identical instances of a single design are produced collectively, rather than individually, as illustrated in *Figure 3*. They arise in the production of things like machine screws, when production assets like machine tools are used to produce multiple identical product instances. A design is created, along with initial instances, called prototypes, by a resource intensive process, called development, performed by engineers. Many additional

Figure 3. Economies of Scale

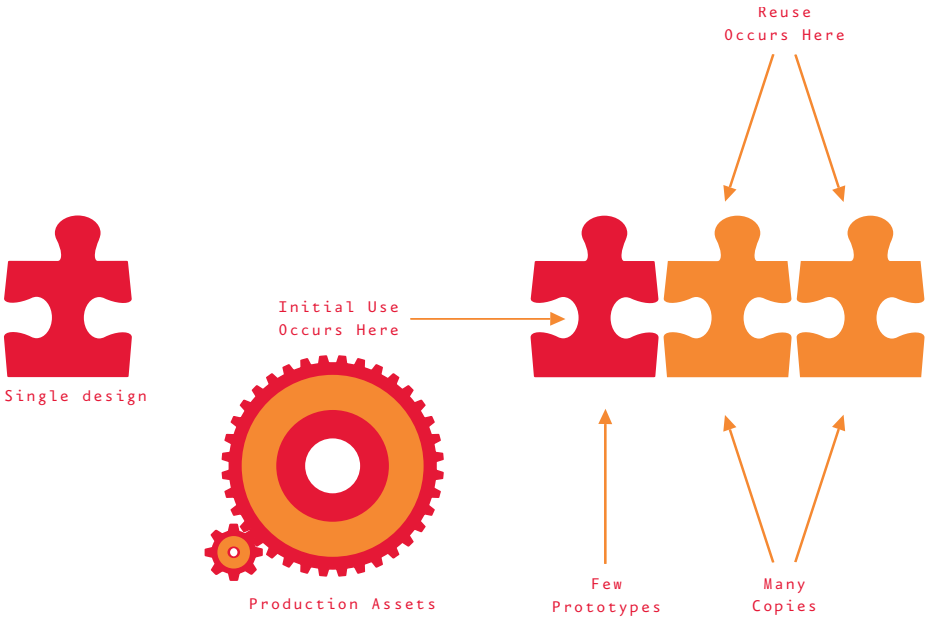
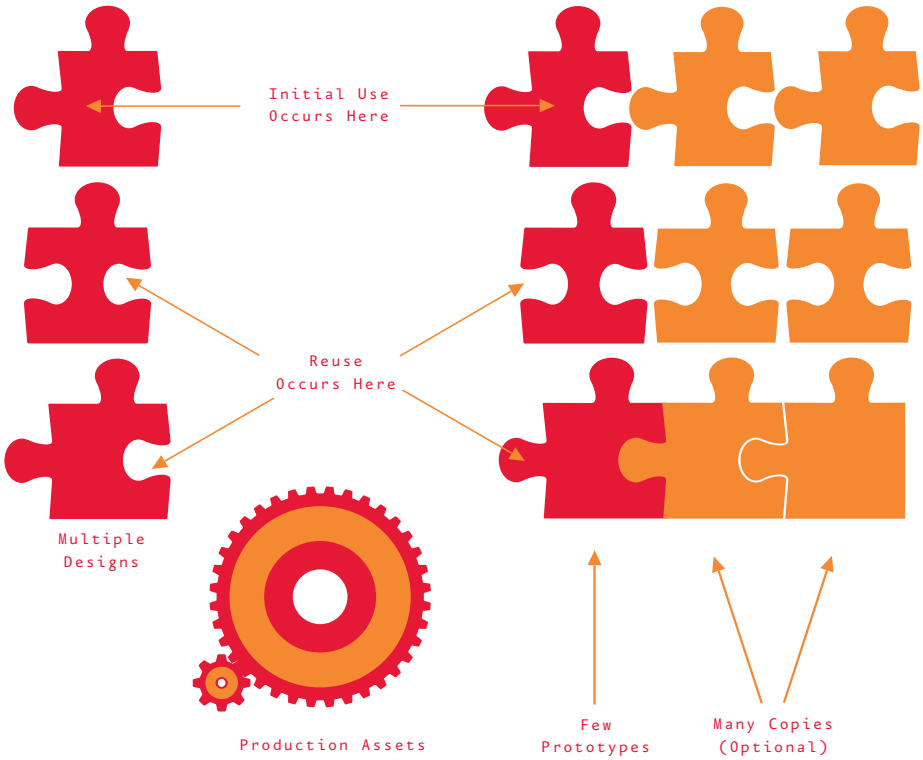


Figure 4. Economies of Scope



"Other industries increased their capacity by moving from craftsmanship... to manufacturing... where machines automate rote or menial tasks. [And] the key to meeting global demand is to stop wasting the time of skilled developers on rote and menial tasks."

instances, called copies, are then produced by another process, called production, performed by machines and/or low cost labor, in order to satisfy market demand.

Economies of scope arise when multiple similar but distinct designs and prototypes are produced collectively, rather than individually, as illustrated in *Figure 4.* In automobile manufacturing, for example, multiple similar but distinct automobile designs are often developed by composing existing designs for subcomponents, such as the chassis, body, interior and drive train, and variants or models are often created by varying features, such as engine and trim level, in existing designs. In other words, the same practices, processes, tools and materials are used to design and prototype multiple similar but distinct products. The same is true in commercial construction, where multiple bridges or skyscrapers rarely share a common design. However, an interesting twist in commercial construction is that usually only one or two instances are produced from every successful design, so economies of scale are rarely, if ever, realized. In automobile manufacturing, where many identical instances are usually produced from successful designs, economies of scope are complemented by economies of scale, as illustrated by the copies of each prototype shown in *Figure 4*.

Of course, there are important differences between software and either automobile manufacturing or commercial construction, but it resembles each of them at times.

– In markets like the consumer desktop, where copies of products like operating systems and productivity applications are mass produced, software exhibits economies of scale, like automobile manufacturing.
– In markets like the enterprise, where business applications developed for competitive advantage are seldom, if ever, mass produced, software exhibits only economies of scope, like commercial construction.

We can now see where apples have been compared with oranges. Production in physical industries has been naively compared with development in software. It makes no sense to look for economies of scale in development of any kind, whether of software or of physical goods. We can, however, expect the industrialization of software development to exploit economies of scope.

What Will Industrialization Look Like?

Assuming that industrialization can occur in the software industry, what will it look like? We cannot know with certainty until it happens, of course. We can, however, make educated guesses based on the way the software industry has evolved, and on what industrialization has looked like in other industries. Clearly, software development will never be reduced to a purely mechanical process tended by drones. On the contrary, the key to meeting global demand is to stop wasting the time of skilled developers on rote and menial tasks. We must find ways to make better use of precious resources than spending them on the manual construction of end products that will require maintenance or even replacement in only a few short months or years, when the next major platform release appears, or when changing market conditions make business requirements change, which ever comes first.

One way to do this is to give developers ways to encapsulate their knowledge as reusable assets that others can apply. Is this far fetched? Patterns already demonstrate limited but effective knowledge reuse. The next step is to move from documentation to automation, using languages, frameworks and tools to automate pattern application.

Semiconductor development offers a preview into what software development will look like when industrialization has occurred. This is not to say that software components will be as easy to assemble as ASICs any time soon; ASICs are the highly evolved products of two decades of innovation and standardization in packaging and interface technology. On the other hand, it might take less than 20 years. We have the advantage of dealing only with bits, while the semiconductor industry had the additional burden of engineering the physical materials used for component implementation. At the same time, the ephemeral nature of bits creates challenges like the protection of digital property rights, as seen in the film and music industries.

Conclusion

This article has described the inability of the software industry to meet projected demand using current methods and practices. A great many issues are discussed only briefly here, no doubt leaving the reader wanting evidence or more detailed discussion. Much more detailed discussion is provided in the book 'Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools', by Jack Greenfield and Keith Short, from John Wiley and Sons. More information can also be found at

http://msdn.microsoft.com/architecture/overview/ softwarefactories, and at

http://www.softwarefactories.com/

including articles that describe the chronic problems preventing a transition from craftsmanship to manufacturing, the critical innovations that will help the industry overcome those problems, and the Software Factories methodology, which integrates the critical innovations.

References

1. [Boe81] B Boehm. *Software Engineering Economics.* Prentice Hall PTR, 1981

2. [Bro95] F Brooks. *The Mythical Man-Month.* Addison-Wesley, 1995

3. [Chr97] C Christensen. *The Innovator's Dilemma,* Harvard Business School Press, 1997

4. [Kuh70] T Kuhn. *The Structure Of Scientific Revolutions.* The University Of Chicago Press, 1970

5. [RJ96] D Roberts and R. Johnson. Evolving *Frameworks: A Pattern Language for Developing Object-Oriented Frameworks. Proceedings of Pattern Languages of Programs*, Allerton Park, Illinois, September 1996

6. [SS02] J. Smith and D Stotts. *Elemental Design Patterns – A Link Between Architecture and Object Semantics.* Proceedings of OOPSLA 2002

7. [Sta94] The Standish Group. *The Chaos Report.* http://www.standishgroup.com/sample_research/PDFpages/chaos1994.pdf

8. [Weg78] P Wegner. *Research Directions In Software Technology*. Proceedings Of The 3rd International Conference On Software Engineering. 1978

Jack Greenfield
Architect, Microsoft Corporation
jackgr@microsoft.com

Jack Greenfield is an Architect for Enterprise Frameworks and Tools at Microsoft. He was previously Chief Architect, Practitioner Desktop Group, at Rational Software Corporation, and Founder and CTO of InLine Software Corporation. At NeXT, he developed the Enterprise Objects Framework, now called Apple Web Objects. A well known speaker and writer, he also contributed to UML, J2EE and related OMG and JSP specifications. He holds a B.S. in Physics from George Mason University.

Identity and Access Management

By Frederick Chong, Microsoft Corporation

Abstract
To date, many technical decision makers in large IT environments have heard about the principles and benefits of Service Oriented Architecture (SOA). Despite this fact, very few IT organizations are yet able to translate the theoretical underpinnings of SOA into practical IT actions.

Over the last year, a few individual solution architects on my team have attempted to distill the practical essence of SOA into the following areas: Identity and Access Management, Service Management, Entity Aggregation and Process Integration. These four key technical areas present significant technical challenges to overcome but yet provide the critical IT foundations to help businesses realize the benefits of SOA.

Note that it is our frequent interactions with architects in the enterprises that enable us to collate, synthesize and categorize the practical challenges of SOA into these areas. Our team organizes the Strategic Architect Forums that are held multiple times worldwide annually. In these events, we conduct small discussion groups to find out the pain points and technical guidance customers are looking for. The feedback from our customers has been very consistent: issues in managing identities, aggregating data, managing services, and integrating business processes have been cited over and over again as major road blocks to realizing more efficient and agile organizations. In addition, our team also conducts proof-of-concept projects with customers to drill deeper into the real world requirements and implementation issues. It is through these combinations of broad and deep engagements with customers that we on the Architecture Strategy team derived our conclusions on the four significant areas for IT to invest in.

The key focus of this paper is to provide an overview of the technical challenges in one of those areas, namely identity and access management; and secondarily, to help the reader gain an understanding of the commonly encountered issues in this broad subject.

Introduction
Identity and access management (I&AM) is a relatively new term that means different things to different people. Frequently, IT professionals have tended to pigeonhole its meaning into certain identity and security related problems that they are currently faced with. For example, I&AM has been perceived to be a synonym for single sign-on, password synchronization, meta-directory, web single sign-on, role-based entitlements, and similar ideas.

The primary goal of this paper is to provide the reader with a succinct and comprehensive overview of what I&AM means. In order to accomplish this purpose, we have structured the information in this paper to help answer the following questions:
- What is a digital identity?
- What does identity and access management mean?
- What are the key technology components of I&AM?
- How do the components of I&AM relate to one another?
- What are the key architecture challenges in IA&M?

Anatomy of a Digital Identity
Personal identifications in today's society can take many different forms. Some examples of these forms are driver licenses, travel passports, employee cardkeys, and club membership cards. These forms of identifications typically contain information that is somewhat unique to the holder, for example, names, address and photos, as well as information about the authorities that issued the cards, for example, an insignia of the local department of motor vehicles.

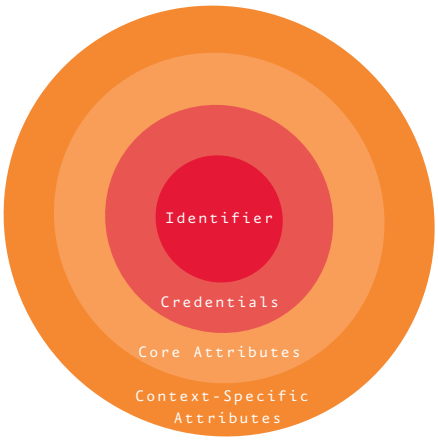
While the notion of identities in the physical world is fairly well understood, the same cannot be said about the definition of digital identities. To help lay the ground work for the rest of the discussions in this paper, this section describes one notion of a digital identity, as illustrated in *Figure 1*. Our definition of digital identity consists of the following parts:
- *Identifier:* A piece of information that uniquely identifies the subject of this identity within a given context[1]. Examples of identifiers are email addresses, X500 distinguished names and Globally Unique Identifiers (GUIDs).
- *Credentials:* Private or public data that could be used to prove authenticity of an identity claim. For example, Alice enters in a password

[1] Context defines the boundary which an identity is used. The boundary could be business or application related. For example, Alice may use a work identity with the identifier Alice@WallStreetAce.com to identify herself at her Wall Street employer (Wall Street Ace) as well as to execute a stock trade in NYSE. Her Wall Street employer and the NYSE would be two different business contexts where the same identifier is used.

to prove that she is who she says she is. This mechanism works because only the authentication system and Alice should know what the password for Alice is. A private key and the associated X509 public key certificate is another example of credentials.

– *Core Attributes:* Data that help describe the identity. Core attributes may be used across a number of business or application contexts. For example, addresses and phone numbers are common attributes that are used and referenced by different business applications.
– *Context-specific Attributes:* Data that help describe the identity, but which is only referenced and used within specific context where the identity is used. For example, within a company, the employee's preferred health plan information is a context specific attribute that is interesting to the company's health care provider but not necessarily so to the financial services provider.

Figure 1. Anatomy of a Digital Identity



What is Identity and Access Management?

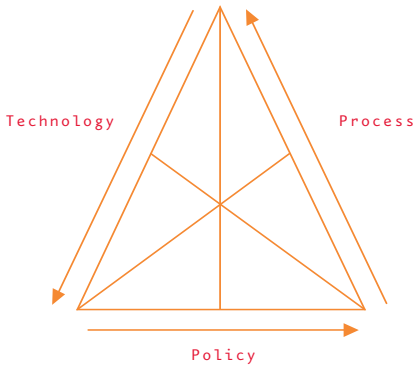
The Burton Group defines identity management as follows: *'Identity management is the set of business processes, and a supporting infrastructure for the creation, maintenance, and use of digital identities'*[2]

In this paper, we define identity and access management (I&AM) as follows: *'Identity and access management refers to the processes, technologies and policies for managing digital identities and controlling how identities can be used to access resources'*

We can make a few important observations from the above definitions:

– I&AM is about the end-to-end life cycle[3] management of digital identities. An enterprise class identity management solution should not be made up of isolated silos of security technologies, but rather, consists of well integrated fabric of technologies that address the spectrum of scenarios in each stage of the identity life cycle. We will talk more about these scenarios in a later section of this paper.
– I&AM is not just about technology, but rather, is comprised of three indispensable elements: policies, processes and technologies. Policies refer to the constraints and standards that needs to be followed in order to comply with regulations and business best practices; processes describe the sequences of steps that lead to the completion of business tasks or functions; technologies are the automated tools that help accomplish business goals more efficiently and accurately while meeting the constraints and guidelines specified in the policies.
– The relationships between elements of I&AM can be represented as the triangle illustrated in *Figure 2*. Of significant interest is the fact that there is a feedback loop that links all three elements together. The lengths of the edges represent the proportions of the elements relative to one another in a given I&AM system. Varying the proportion of one element will ultimately vary the proportion of one or more other elements in order to maintain the shape of a triangle with a sweet spot (shown as an intersection in the triangle).

Figure 2. Essential elements of an identity and access management system



– The triangle analogy is perfect for describing the relationships and interactions of policies, processes and technologies in a healthy I&AM system as well. Every organization is different and the right mix of technologies, policies and processes for one company may not necessarily be the right balance for a different company. Therefore, each organization needs to find its own balance represented by the uniqueness of its triangle.
– An organization's I&AM system does not remain static over time. New technologies will get introduced and adopted; new business models and constraints will change the corporate governance and processes to do things. As we mentioned before,

[2] From *Enterprise Identity Management: It's About the Business*, Jamie Lewis, The Burton Group Directory and Security Strategies Report, v1 July 2nd 2003.

[3] The term 'life cycle' when used in the context of digital identities is somewhat inappropriate since digital identities are typically not recycled. However, since the phrase 'identity life cycle management' is well entrenched in the IT community, we will stick with the 'life cycle' terminology.

when one of the elements change, it is time to find a new balance. It is consequently important to understand that I&AM is a journey, not a destination.

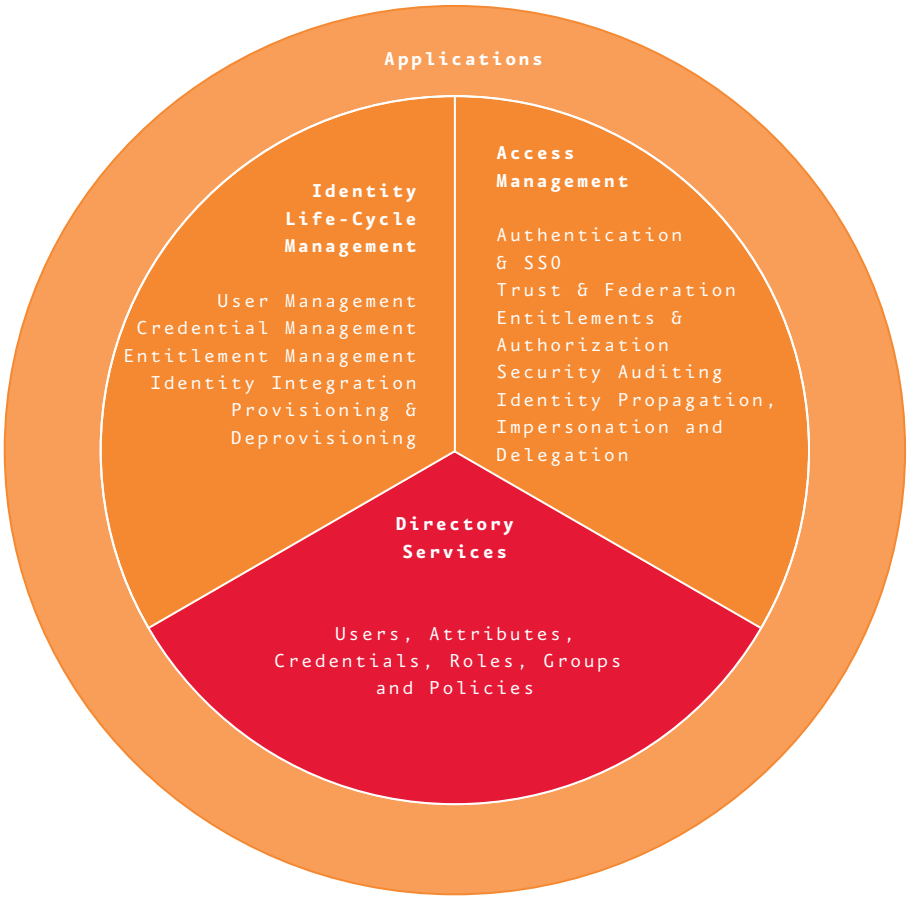
Identity and Access Management Framework

As implied in the previous sections, identity and access management is a very broad topic that covers both technology and non-technology areas. We will focus the rest of this paper around the technology aspects of identity and access management.

To further contain the technical scope of this topic that is still sufficiently broad, it is useful to abide by some structure for our discussions. We will use the framework shown in *Figure 3,* which illustrates several key logical components of I&AM to lead the discussions on this subject.

This particular framework highlighted three key 'buckets' of technology components:
– Identity life cycle management
– Access management
– Directory services

Figure 3. Logical Components of I&AM



The components in these technology buckets are used to meet a set of recurring requirements in identity management solutions. We will describe the roles that these components play in the next few sections.

Directory Services

As mentioned previously, a digital identity consists of a few logical types of data – the identifier, credentials and attributes. This data needs to be securely stored and organized. Directory services provide the infrastructure for meeting such needs. Entitlements and security policies often control the access and use of business applications and computing infrastructure within an organization. Entitlements are the rights and privileges associated with individuals or groups. Security policies refer to the standards and constraints under which IT computing resources operate. A password complexity policy is an example of a security policy. Another example is the trust configuration of a business application which may describe the trusted third party that the application relies upon to help authenticate and identify application users. Like digital identities, entitlements and security policies need to be stored, properly managed and discovered. In many cases, directory services provide a good foundation for satisfying these requirements.

Access Management

Access management refers to the process of controlling and granting access to satisfy resource requests. This process is usually completed through a sequence of authentication, authorization, and auditing actions. Authentication is the process by which

identity claims are proven. Authorization is the determination of whether an identity is allowed to perform an action or access a resource. Auditing is the accounting process for recording security events that have taken place. Together, authentication, authorization, and auditing are also commonly known as the gold standards of security. (The reasoning behind this stems from the periodic symbol for Gold, 'Au'; the prefix for all three processes.)

There are several technical issues that solutions architects may encounter when designing and integrating authentication, authorization, and auditing mechanisms into the application architecture:
– Single Sign-On
– Trust and Federation
– User Entitlements
– Auditing

We will describe these challenges and their solutions in more detail later on in this document.

Identity Life Cycle Management

The life cycle of a digital identity can be framed in similar stages to the life cycles of living things:
– Creation
– Utilization
– Termination

Every stage in an identity's life cycle has scenarios that are candidates for automated management. For example, during the creation of a digital identity, the identity data needs to be propagated and initialized in identity systems. In other scenarios an identity's entitlements might need to be magnified when the user represented by the identity receive a job promotion. Finally when the digital identity is no longer put to active use, its status might need to be changed or the identity might need to be deleted from the data store.

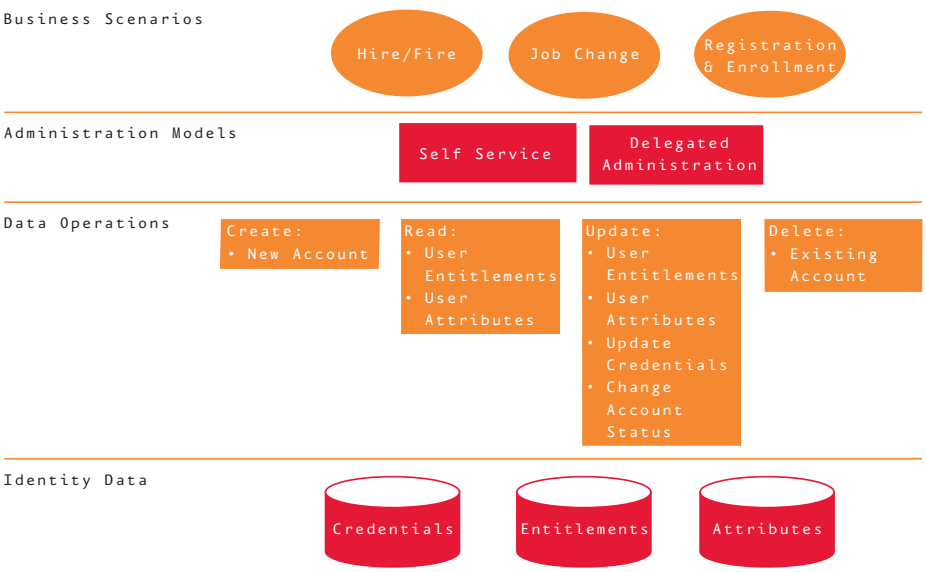
All events during the life cycle of a digital identity need to be securely, efficiently, and accurately managed, which is exactly what identity life cycle management is about.

The requirements for identity life cycle management can be discussed at several levels, as represented in *Figure 4*. The types of data that need to be managed are shown at the identity data level. Based on our previous definitions of digital identity, the relevant data includes credentials, such as passwords and certificates; and user attributes, such as names, address and phone numbers. In addition to credentials and attributes, there are also user entitlements data to manage. These entitlements are described in more detail later on, but for now, entitlements should be considered as the rights and privileges associated with identities.

Moving up a level in the illustration, the requirements listed reflect the kinds of operations that can be performed on identity data. Create, Read, Update, and Delete (CRUD) are data operation primitives coined by the database community. We reuse these primitives here as they provide a very convenient way for classifying the kinds of identity management operations. For example, we can classify changes to account status, entitlements, and credentials under the Update data primitive.

The next level in the illustration shows two identity life cycle administration models: self-service and delegated. In traditional IT organizations, computer administration tasks are performed by a centralized group of systems administrators. Over time, organizations have realized that there may be good economic and business reasons to

Figure 4. Levels of Identity Life Cycle Management Requirements



enable other kinds of administration models as well. For example, it is often more cost effective and efficient for individuals to be able to update some of their personal attributes, such as address and phone number, by themselves. The self-service administration model enables such individual empowerment. The middle ground between the self-service and centralized administration models is delegated administration. In the delegated model, the responsibilities of identity life cycle administration are shared between decentralized groups of administrators. The common criteria used to determine the scope of delegation are organization structure and administration roles. An example of delegated administration based on organization structure is the hierarchy of enterprise, unit and department level administrators in a large organization.

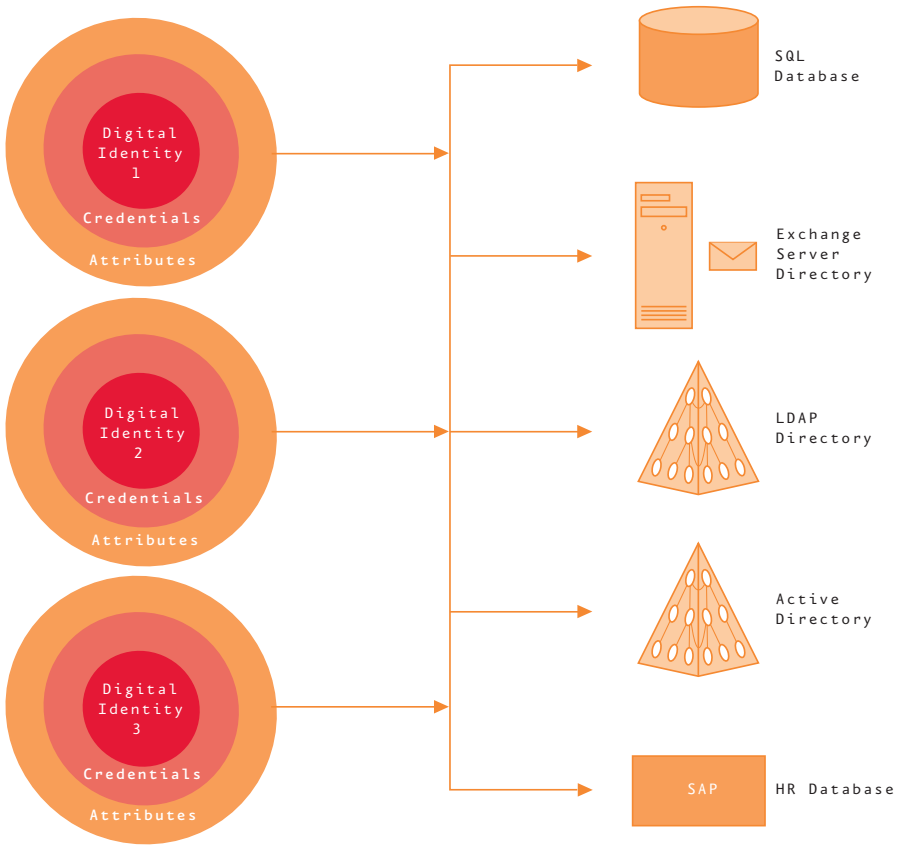
The above life cycle administration models can be used to support a variety of business scenarios, some of which are listed in *Figure 4.* For instance, new employees often require accounts to be created and provisioned for them. Conversely, when an employee is no longer employed, the existing account status might need to be changed. Job change scenarios can also have several impacts on the digital identities. For example, when Bob receives a promotion, his title might need to be changed and his entitlements might need to be extended.

Now that we have a better understanding of identity life cycle management requirements, we are ready to drill into the challenges involved in meeting those requirements. The illustration in *Figure 5* speaks to the fact that a typical user in an enterprise typically has to deal with multiple digital identities which might be stored and managed independently of one another. This current state of affairs is due to the ongoing evolution that every business organization goes through. Events such as mergers and acquisitions can introduce incompatible systems into the existing IT infrastructure; and evolving business requirements might have been met through third party applications that are not well integrated with existing ones.

One may presume that it would have been much easier for enterprises to deprecate the current systems and start over. However, this solution is seldom a viable option. We have to recognize that existing identity systems might be around for a long time, which leads us to find other solutions for resolving two key issues arising from managing data across disparate identity systems:

1. Duplication of information
   Identity information is often duplicated in multiple systems. For example, attributes such as addresses and phone numbers are often stored and managed in more than one system in an environment. When identity data is duplicated, it can easily get out of sync if updates are performed in one system but not the others.

Figure 5. Current state of affairs: multiple identity systems in the enterprise



2. Lack of integration
   The complete view of a given user's attributes, credentials and privileges are often distributed across multiple identity systems. For example, for a given employee, the human resource related information might be contained in an SAP HR system, the network access account in an Active Directory and the legacy application privileges stored in a mainframe. Many identity life cycle management scenarios require identity information to be pulled from and pushed into several different systems.

We refer to the above issues as the identity aggregation challenge, which we will describe in more detail in a later section.

Challenges in Identity and Access Management

Single Sign-On

A typical enterprise user has to login multiple times in order to gain access to the various business applications that they use in their jobs. From the user's point of view, multiple logins and the need to remember multiple passwords are some of the leading causes of bad application experiences. From the management point of view, forgotten password incidents most definitely increase management costs, and when combined with bad user password management habits (such as writing passwords down on yellow sticky notes,) can often lead to increased opportunities for security breaches. Because of the seemingly intractable problems that multiple identities present, the concept of single sign-on (SSO); the ability to login once and gain access to multiple systems, has become the 'Holy Grail' of identity management projects.

Single Sign-On Solutions

Broadly speaking, there are five classes of SSO solutions. No one type of solution is right for every application scenario. The best solution is very much dependent on factors such as where the applications requiring SSO are hosted, limitations placed by the infrastructure (e.g. firewall restrictions), and the ability to modify the applications. These are the five categories of SSO solutions:

1. Web SSO
2. Operating System Integrated Sign-On
3. Federated Sign-On
4. Identity and Credential Mapping
5. Password Synchronization

Web SSO solutions are designed to address web application sign-on requirements. In these solutions, unauthenticated browser users are redirected to login websites to enter in user identifications and credentials. Upon successful authentication, HTTP cookies are issued and used by web applications to validate authenticated user sessions. Microsoft Passport is an example of Web SSO solutions.

Operating system integrated sign-on refers to authentication modules and interfaces built into the operating system. The Windows security subsystem provides such capability through system modules such as Local Security Authority (LSA) and Security Specific Providers (SSP) SSPI refers to the programming interfaces into these SSP. Desktop applications that use the SSPI APIs for user authentication can then 'piggyback' on Windows desktop login to help achieve application SSO. GSSAPI on various UNIX implementations also provide the same application SSO functionality.

Federated sign-on requires the application authentication infrastructures to understand trust relationships and interoperate through standard protocols. Kerberos and the future Active Directory Federation Service are examples of federation technologies. Federated sign-on means that the authentication responsibility is delegated to a trusted party. Application users need not be prompted to sign-on again as long as the user has been authenticated by a federated (i.e. trusted) authentication infrastructure component.

Identity and credential mapping solutions typically use credential caches to keep track of the identities and credentials to use for accessing a corresponding lists of application sites. The cache may be updated manually or automatically when the credential (for example password) changes. Existing applications may or may not need to be modified to use identity mapping solutions. When the application cannot be modified, a software agent may be installed to monitor application login events. When the agent detects such events, it finds the user credential in the cache and automatically inputs the credential into the application login prompt.

The password synchronization technique is used to synchronize passwords at the application credential databases so that users and applications do not have to manage multiple passwords changes. Password synchronization as a silo-ed technology does not really provide single sign-on,

but results in some conveniences that applications can take advantage of. For example, with password synchronization, a middle tier application can assume that the password for an application user is the same at the various systems it need access to so that the application does not have to attempt looking up for different passwords to use when accessing resources at those systems.

Entitlement Management
Entitlement management refers to the set of technologies used to grant and revoke access rights and privileges to identities. It is closely associated with authorization, which is the actual process of enforcing the access rules, policies and restrictions that are associated with business functions and data.

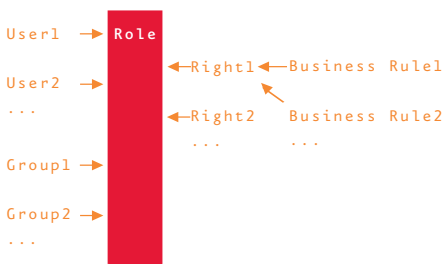
Today's enterprise applications frequently use a combination of role-based authorization and business rules-based policies to determine what a given identity can or cannot do.

Within a distributed n-tiered application, access decisions can be made at any layer in the application's architecture. For example, the presentation tier might only present UI choices that the user is authorized to make. At the service layer of the architecture, the service might check that the user meets the authorization condition for invoking the service. For example, only users in the manager role can invoke the 'Loan Approval' service. Behind the scenes at the business logic tier, there might need to be fine grain business policy decisions such as 'Is this request made during business hours'; at the data layer, the database stored procedure might filter returned data based on the relationship between the service invoker's identity and the requested data.

Given the usefulness, and often intersecting use, of both role and rule-based authorization schemes, it is not always clear to application architects how to model an entitlement management framework that integrates both schemes cleanly. Many enterprises have separate custom engines to do both.

Figure 6 Integrating role and rule based authorization illustrates a representation of how both schemes might be combined and integrated. In this representation, we can envision a role definition (which typically reflects a job responsibility) with two sets of properties. One set of properties contain the identities of people or systems that are in the given role. For example, Alice, Bob and Charlie may be assigned to the Manager role. The second set of properties contains the set of rights that a given role has. Rights can represent business functions or actions on computing resources. For example, *transfer fund* defines a business function and *read file* refers to an operation on a computing resource. Furthermore, we can assign a set of conditional statements (business rules) for each right. For example, the *transfer fund* right may have a conditional statement to allow the action if the current time is within business hour. Note that the conditional statement might base its decision on dynamic input data that can only be determined at application runtime.

Figure 6. Integrating role and rule based authorization



It is also important for most organizations to have a consolidated view of all the rights that a given identity possesses. To meet this requirement, entitlement management applications typically leverage a centralized policy store to help facilitate centralized management and reporting of users' rights.

Identity Aggregation
Enterprise IT systems evolve organically over the course of an organization's history. This is often due to reasons such as mergers and acquisitions, or preferences and changes of IT leaderships. The consequences of this are often manifested through hodge-podges of disconnected IT systems with undesirable architectural artifacts. Identity-related systems are no exceptions to such IT evolutions.

Frequently, the enterprise will have not just one identity systems, but several, each serving different business functions but storing duplicated and related data. Applications that need to integrate with those business functions are then forced to reconcile the differences and synchronize the duplications.

For example, a banking customer service application might need to obtain customer information from an IBM DB2 database, an Oracle

database and a homegrown CRM. In this case, the application's concept of 'Customer' is defined by three different systems. Attributes that describe customer such as customer name, address, and social security number might be stored and duplicated in multiple systems. On the other hand, non-duplicated data such as the financial products that the customer has purchased and the customer's bank balance might be kept in separate systems. The application will need to aggregate this data from different systems to get the necessary view of the customer.

Moving the underlying identity data into one huge giant identity system might seem like an obvious answer to this problem. However, there are many real world issues (for example, the risk of breaking legacy applications) that prevent such solution from being broadly adopted any time soon.

Identity aggregation therefore refers to the set of technologies that help applications aggregate identity information from different identity systems, while reducing the complexity of data reconciliation, synchronization and integration.

There are several technical challenges that identity aggregation technologies should help address:
– Maintaining relationships for data transformations
– Optimizing data CRUD operations
– Synchronizing data

The next few sub-sections provide an overview of these design issues.

(a) Maintaining Relationships for Data Transformations
An identity aggregation solution can provide several benefits to applications with disparate views of identity information that manipulate data in different systems. The first benefit involves providing applications with a consolidated view or an aggregated view of the data in the individual systems. In order to transform and represent data in different views, the identity aggregation solution needs to maintain meta-data describing the schema representing the consolidated view and its relationship with the data schema in the various identity systems.

Let's look at a specific example of an identity life cycle management application that is managing data across a few existing systems. The consolidated view of the management application is represented by a new schema which is made up of new and existing identity attributes.

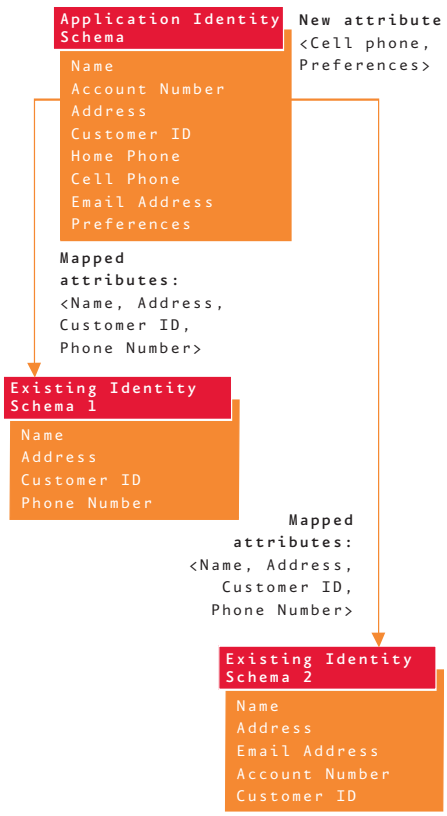
Figure 7 Reconciling identity schemas illustrates an example where a new identity schema is defined for the application. The new schema refers to attributes in two existing identity schemas and also defined new ones that are not currently specified (Account Number, Cell phone and Preferences).

In order for applications to query and update data in existing stores, we will need to maintain data relationships between the attributes defined in the new schema and the corresponding attributes in the existing schemas. For example, the following relationships will need to be maintained:

Reference – A reference refers to a piece of information that unambiguously identifies an instance of data as represented by a particular data schema. For example, the 'Customer ID' attribute allows a data instance as represented by the application scheme in *Figure 7* to find its corresponding data instance as represented by existing schema 1. Different schemas may use different references to identify their own data instances.

Ownership – A data attribute may be defined in more than one existing schema. Using the same example shown in *Figure 7* again, we can see that the 'Name' and 'Address' attributes are defined in both existing schemas. In the event of a data conflict, the application needs to know which

Figure 7. Reconciling identity schemas



version holds the authoritative copy to keep. In addition, there may be scenarios where an attribute can get its value from a prioritized list of owners. In those scenarios, when the value for an attribute is not present in the first authoritative source, the aggregation service should query the next system in the prioritized list of owners.

Attribute Mapping – Attributes defined in multiple data store may have the same semantic meaning, but have the same or different syntactic representations. When querying or updating a data attribute, the identity aggregation service needs to know the attributes that are semantically equivalent to one another. For example, although customer id is defined in all three schemas, it is named differently as CustID in identity schema 2. When the application performs an update for an identity's customer id number, the system must also know to update the CustID attribute for the data instance represented by schema 2.

(b) Optimizing Data CRUD Operations
As previously identified, CRUD is a database acronym that stands for Create, Read, Update and Delete. CRUD defines the basic primitive operations for manipulating data. The reason why CRUD is raised as a technical challenge is because the performance for completing an aggregation-related activity that involves CRUD operations across multiple data backend systems can vary significantly depending on the data relationships.

In the best case scenario, CRUD operations can be parallelized. This is mostly true in situations where the data instances can be resolved using the same reference and the data reference always resolves to a unique instance. For example, if the social security number is the only key used for querying and aggregating data across data stores, that particular query can be issued in parallel.

In the worst case scenario, the CRUD operations are serialized across data stores. Serialized operation is common in situations where the references used for resolving data instances have dependencies on other data instances. As a simple illustration, let's suppose we need to aggregate data from the HR and Benefits databases and the instance references are employee ID and social security ID respectively. If the only initial key we have for the query is the employee ID, and the social security ID can only be obtained from the HR database, then we will need to serialize the query in the following order:

1. Query the HR database using the employee ID as the key.
2. Extract the social security number from the above query results.
3. Query the benefits database.
4. Aggregate the query results.

Replication is a common technique used to address performance degradation due to CRUD operations across data stores. To address the performance issue, a portion of the backend data attributes may be replicated to a store maintained by the identity aggregation service. In addition, the local copy of the replicated data can be further de-normalized to help improve the CRUD performance.

(c) Synchronizing Data
Data synchronization is needed in situations when one or both of the following conditions are true:

- Duplicate identity attributes exist in multiple backend stores.
- Data is replicated to an intermediate identity aggregator store.

However, the use of data synchronization may also introduce other design issues:

- Data conflict resolution. In situations where the data can be updated from more than one source, it is often easy to introduce conflicting data. Some common practices to help mitigate the situations are as follows:
  - Assign data ownership priority so that in the event of conflict, we will use the assigned authority.
  - In a conflict, the last writer wins.
  - Synchronization triggers. A couple of common approaches are:
    - Scheduled updates.
    - Event notification based.

Trust and Federation
As mentioned in the single sign-on section, federation offers a form of single sign-on solution. However, federation is more than just single sign-on. Federation implies delegation of responsibilities honored through trust relationships between federated parties. Authentication is just one form of delegated responsibility. Authorization, profile management, pseudonym services, and billing are other forms of identity-related functions that may be delegated to trusted parties.

There are three technology elements that are crucial to the concept of federation:

- A federation protocol that enables parties to communicate.
- A flexible trust infrastructure that supports a variety of trust models.

– An extensible policy management framework that supports differing governance requirements.

Federation protocols are the 'languages' that are used by federating parties to communicate with each other. Since federation implies that a responsibility is delegated to and performed by a different party, the protocol must allow individuals to obtain 'capabilities' – essentially tamper-proof claims that a given identity has successfully completed an action or is entitled to a collection of privileges. For example, in the case of federated sign-on, an authenticated identity obtains a capability that proves that the individual has successfully authenticated with an approved authentication service.

In a complex business world, it is possible to have relatively complex trust schemes involving multiple business parties. Federation technology must be able to capture the essence of those real world trust relationships into simple to understand but powerful trust models that will help enable various business scenarios. Some common trust models, illustrated in *Figure 8*, are as follows:
– Hub-and-spoke
– Hierarchical
– Peer-to-peer Web of Trust

The hub-and-spoke model is the simplest to understand. In this model, there is a central broker that is directly trusted by the federating parties. The European Union is an example of this federation model where the EU countries directly trust the EU body to provide common economic guidelines and trade opportunities to federated parties.

In the hierarchical model, two parties have an indirect trust relationship if they both have a trust path in their respective branches in the hierarchical tree to a common root authority. The American political system demonstrates this trust model. This political system has federal, state, county and local city political bodies, each existing at various levels of the hierarchy.

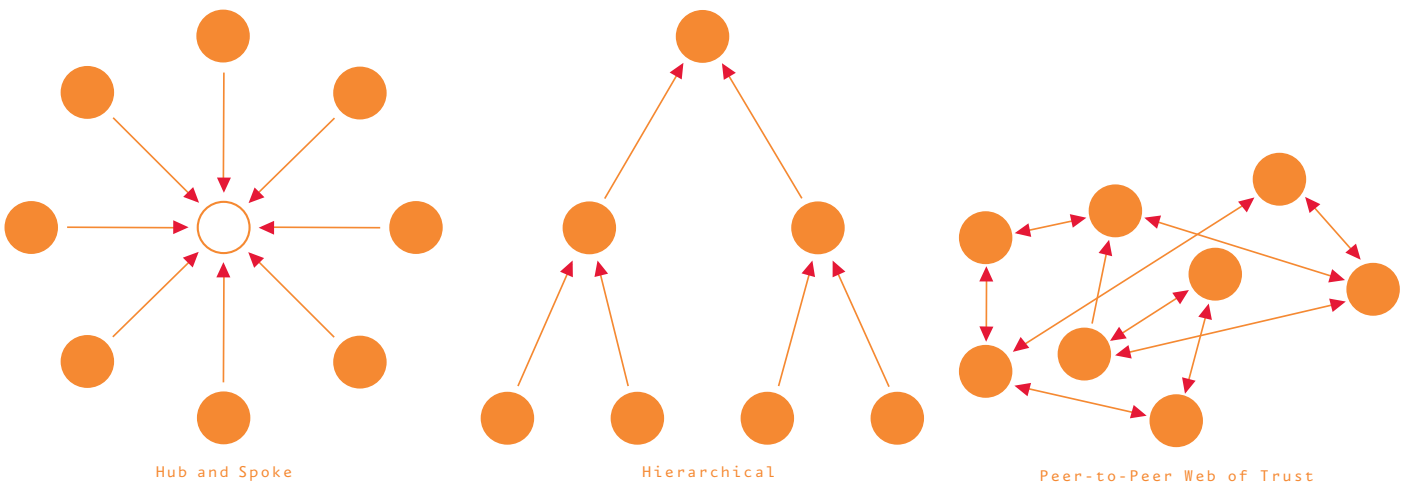
The peer-to-peer model represents a collection of ad-hoc direct trust relationships. Personal relationships between friends in the physical world are good examples of this trust model.

Note that it is also possible to extend and form new networks of federations that consist of different trust models, or hybrid models.

A basic policy management framework must allow policies to be created, deleted, modified and discovered. In order to promote federated systems that enable new business models and partnerships to be quickly integrated, the policy management framework must also be extensible to reflect the dynamicity of the environment it aims to support.

Some examples of application policies that are relevant in the federated world are:
– Trusted issuer of identity-related capabilities.
– The types of capabilities required to invoke an application's operation.

Figure 8. Common trust models



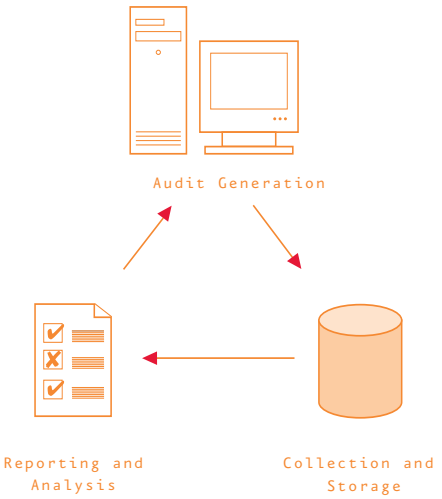
Federation implies delegation of responsibilities honored through trust relationships between federated parties."

– The kinds of identity information that the application expects in capabilities.
– The kinds of privileges that an identity must demonstrate in order to invoke a service.

Auditing

Auditing in the context of I&AM, is about keeping records of 'who did what, when' within the IT infrastructure. Federal regulations such as the Sarbanes-Oxley Act are key drivers of the identity-related auditing requirements.

Figure 9. IT audit process



(a) IT Audit Process

The IT audit process typically involves the following phases as illustrated in *Figure 9*:
– Audit generation
– Data collection and storage
– Analysis and feedback

Audit trails can be generated by different infrastructure and application components for different purposes. For example, firewalls and VPN servers can generate events to help detect external intrusions; middleware components can generate instrumentation data to help detect performance anomalies; and business applications can produce audit data to aid debugging or comply with regulatory audit requirements.

After the audit data has been produced, it needs to be collected and stored. There are two main models to consider here: distributed and centralized. In the distributed model, audit data typically remains in the system where the data is generated. With the centralized approach, data is sent to a central collection and data storage facility.

Once the data is collected, they may be processed and analyzed automatically or manually. The audit analysis is designed to lead to conclusions on what corrective actions, if any, are needed to improve the IT systems and processes.

(b) Audit Systems Design Considerations

Given the typical auditing process described in the previous sections, there are several considerations that are important to the design of auditing systems:
– Locality of audit generation and storage
– Separation of auditor's role
– Flow of audited events

Once an audit trail is generated, the audit data can be stored locally on the same system or transferred to another storage location. This consideration is important from a security perspective, as audit data can be easier to change and modify if it is stored on the same system that generates it. This point can be illustrated by a simple example: In the event that a system has been compromised, the attacker might be able to modify the audit trail on the local system to cover up the hacking attempt. Therefore, for higher assurance against such attacks, you might want the audit system to store the data separately on a remote machine.

Role separation is a common best practice to help minimize the occurrence of illegal activities resulting from actions that might circumvent accountability checks. For example, a corporate acquisition officer should not be able to approve purchasing requests. In the field of IT audit, it is common practice to separate the system administrator role from the auditor's role. Doing so prevents the system administrator who usually has 'god status' on computers to cover up audit trails of unauthorized activities.

In a distributed design model, where audit data is generated and stored in different systems, only allowing data to flow out of the audit generation systems can add another level of safeguard. This preventive measure reduces the chances of tampered audit data replacing actual trails.

In addition, identity auditing infrastructures are also expected to be:
– Efficient (typically means a message-based asynchronous communication channel).
– Available (typically means distributed, clustered and fault tolerant).
– Accurate (keep accurate records and traces).
– Non-repudiated (can be admitted into the courts of law as evidence, typically means the records need to be digitally signed).

Conclusions

Organizations are made up of people and systems, represented within the IT systems as digital identities. Everything that occurs in businesses is the consequence of actions initiated by and for those identities. Without identities (even anonymous identities) there would be no activities and businesses would be lifeless constructs.

At the highest level, SOA can be seen as a way to organize the business IT infrastructure that executes and manages business activities. Therefore, enterprises seeking to realize SOA must figure out how to resolve the identity and access management challenges they face today. We have provided an overview on a few keys technology areas:

– Achieving user and application single sign-on.
– Aggregating, transforming, synchronizing and provisioning identity data from various identity systems.
– Managing access to business functions and data using roles and rules.
– Federating with business partners.
– Auditing identity-related activities.

We also hope that the technical discussions on challenges have helped the readers gain some appreciation of the products and solutions in this space.

Our final conclusion is:

Identity and access management is a cornerstone to realizing SOA. Show me an enterprise that claims to be 'SOAccessful' and I'll show you an enterprise that has good handle on its identity and access management.

References

1. *SOA Challenges: Entity Aggregation*, Ramkumar Kothandaraman, .NET Architecture Center, May 2004 (URL: http://msdn.microsoft.com/architecture/default.aspx?pull=/library/en-us/dnbda/html/dngrfsoachallenges-entityaggregation.asp)
2. *Enterprise Identity Management: It's About the Business*, Jamie Lewis, The Burton Group Directory and Security Strategies Report, v1 July 2nd 2003
3. *Microsoft Identity and Access Management Series*, Microsoft Corporation, May 14th, 2004 (URL: http://www.microsoft.com/downloads/details.aspx?FamilyId=794571E9-0926-4C59-BFA9-B4BFE54D8DD8&displaylang=en)
4. *Enterprise Identity Management: Essential SOA Prerequisite Zapflash*, Jason Bloomberg, Zapflash, June 19th, 2003 (URL: http://www.zapthink.com/report.html?id=ZapFlash-06192003)

Frederick Chong
Solutions Architect,
Microsoft Corporation
fredch@microsoft.com

Frederick Chong is a solutions architect in the Microsoft Architecture Strategy Team where he delivers guidance on integrating applications with identity and access management technologies. He discovered his interest in security while prototyping an electronic-cash protocol in the network security group at the IBM T J Watson Research Center. Since then, he has implemented security features and licensing enforcement protocol in Microsoft product teams and collaborated with various enterprise customers to architect and implement a variety of security solutions including web single sign-on, SSL-VPN and web services security protocol.

Business Patterns for Software Engineering Use – Part 2

By Philip Teale, Microsoft Corporation and Robert Jarvis, SA Ltd

Introduction
This is the second article in a series of two. The purpose of these articles is to explore whether business patterns[1] can be defined in a structured way, and if so – whether these business patterns would be of value for those who build software systems that support the business. Our point of view is that this value does exist.

The first article was published in JOURNAL2 and it explored how to *define* business patterns that would be useful for software engineers. Article 1 sets the scene for this article and should be read before this one.

In the first article we used an Enterprise Architecture Framework called SAM[2] to analyse the opportunity and we concluded that such business patterns will describe:

– The *business functions* being supported.

– The *data* that is required to support the described functions.

– The *business components* that are the IT representations of the data and functions the business needs.

– Optionally, the *infrastructure* needed to support the functions, data and components. This is necessary in highly distributed enterprises or those made up of divisions or units with diverse technical or operational environments.

In addition, we defined the business patterns that describe the key relationships between these dimensions.

In this second article, we describe how to *develop* business patterns based on business functions, data, and business components. We also show how these can be used to engineer software systems.

Summary of Article 2
In this article we use a realistic but simplified example to show how to use standard techniques to develop descriptions of the business functions, data and business components required

Difference between patterns and systems
If you haven't had much experience with patterns, you might look at the examples in this article and see them as a system development rather than a pattern. This is because, for software engineering, a pattern is a stepping stone on the way to developing part of a system. The pattern gets you part of the way to the final objective by starting you off from a tried-and-trusted place. It looks familiar, but it is not a solution in itself – it is an abstraction of the solution that you have to embellish to make it the solution that suits your needs. The tricky bit in creating patterns is to get the level of abstraction right. Very little abstraction constrains the usefulness of the pattern, as it is then very specific. But a lot of abstraction also limits the usefulness as it provides very little practical value.

Think of a common situation today where you might have designed a service called 'message broker' to provide message routing and transformation between a variety of IT services. A highly abstracted pattern would be exactly what we just said – 'if you want to provide message routing and transformation between a variety of IT services then use a message broker'. How useful was that? Well, it starts you thinking the right way, so it is some use but does not provide a lot of practical help.

On the other hand, after you've built a message broker for handling the IT services required by a bank to handle a consolidated customer view, could you take that solution and call it a pattern? Yes, but how many would it be useful for? It is too specific and is a potentially-repeatable solution rather than a pattern.

Somewhere in between is the 'sweet spot', where the key ideas at all the levels involved in building the system can be abstracted and reapplied to solve many industry problems. These are the most powerful patterns. In this article we think the Business Component specification example we show is in that sweet spot for the *healthcare industry.* We think this is an important aspect of business patterns that should be clearly recognised – *some will not be of interest outside the industry (like this one) and some will (that represent common functions).* We speculate that the common ones that cross industries actually identify the areas mostly ripe for outsourcing – but that's a whole different discussion. [0]

[1] Our definition of a 'pattern' follows the classic definition created by Christopher Alexander in *A Timeless Way of Building,* Oxford University Press, 1979. When writing patterns we use a Coplien-style notation.

[2] For information on SAM – see *Enterprise Architecture – Understanding the Bigger Picture*, Bob Jarvis, a Best Practice Guideline published by the National Computing Centre, UK, May 2003 or http://www.systems-advisers.com

for a business pattern. We do not describe the infrastructure needed, and hence will not explore the relationships to infrastructure either. Our goal is to say 'it can be done, and you already know how' rather than to provide a detailed guide to every step.

First we show a way to define the business functions, data and components we need, and then second we use the PRM[3] to show a roadmap for the journey. The example we use is based on the Healthcare industry but the techniques are valid for any industry. The techniques shown are those that were actually used in a real project. It is important to stress that we are not saying that you must use these techniques. The techniques in themselves are not important – it's the results that matter. These should enable you to engineer either more refined patterns or real software systems from the deliverables.

Business Functions

To develop business patterns our first goal must be to discover, define and document relevant business functions. This has two main tasks:

1. We want to discover the atomic level function set that describes the problem space. These are known as 'primitive functions' in functional modelling. The same things are known as 'elementary processes' in Business Process Re-engineering.
2. We want to aggregate the atomic functions into larger and larger grained functional groups.

Note that by 'atomic' we mean that the function cannot be meaningfully divided any further – once started, the function has to be completed or aborted. This can be more fine-grained, and therefore voluminous, than is strictly necessary for business pattern definition. Therefore what we actually seek for a business pattern is business functions defined at to level of decomposition at which meaningful relationships can be formed with the other spheres, particularly the data sphere. In the case of business functions we find this in practice to be at about the fourth level of decomposition from the root of the hierarchy. At this level it usually possible to formulate CRUD (Create, Read, Update and Delete) relationships with data entities held in the Data sphere.

We can tackle the definition of business functions either bottom-up or top-down or in a mixture of the two.

Bottom-Up analysis

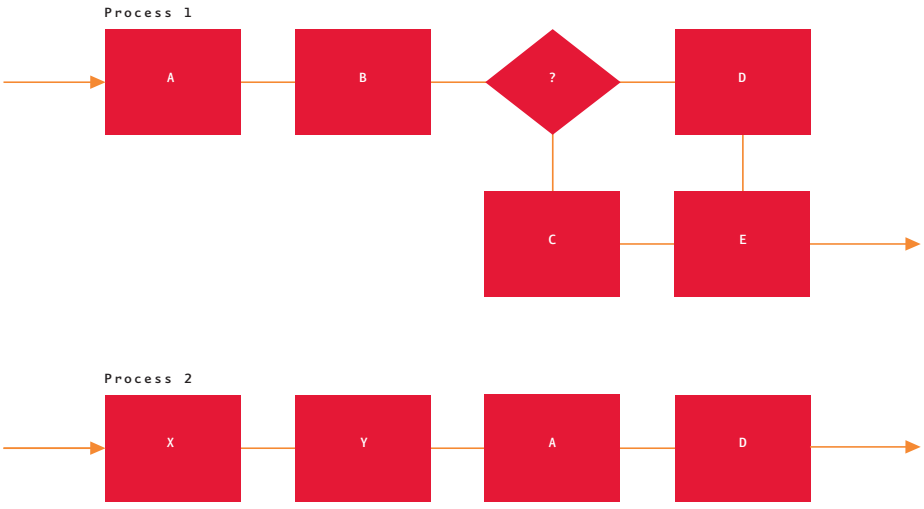
In this approach the analyst works with a representative set of users from the business domain, to analyse their view of the business processes that they perform. This may be done using use case diagrams or any technique that can show the sequential and parallel tasks carried out in the process. The collection of processes is then catalogued and analysed and it is usually noticed that many of the steps or low-level tasks carried out in these processes are repeated many times in different processes. The redundant tasks are identified and eliminated a non-redundant set of primitive functions.

This is shown in *Figure 1* using a simple flowcharting technique. It can be seen that task A is redundant and so will appear only once in the primitive set.

Then having derived the non-redundant primitive set by this analysis, we can now iteratively aggregate the functions into a provisional hierarchy, such as that shown in *Figure 2*.

From the point of view of a business pattern, there are issues with using a bottom-up approach:

Figure 1. Process discovery & synthesis



[3] Problem Refinement Model – please see Article 1 in JOURNAL2.

1. This can be a very labour-intensive process for a large business area, involving many users. (This can be mitigated by compromising on a scenario + use-case approach rather than full process analysis).

2. The nature of the process leads to an analysis of the 'as-is' situation. This is acceptable as long as this is clear and the results are used accordingly.

3. In order to be sure that you are documenting a *pattern*, you would have to repeat or at least verify the analysis in several similar enterprises. This could present many practical problems.

The benefits of the bottom-up approach are that the fundamental analysis has been now done and can readily be applied as a foundation for a solution derived from the pattern. The point of a *pattern* is to document successful practice, and given that the example has been well chosen, this will naturally occur.

Top-Down Analysis

The functional decomposition in *Figure 2* could be derived another way. This could involve hypothesising the upper levels of the hierarchy and 'filling in' the levels below until a satisfactory degree of detail has been obtained. Clearly it is necessary to verify that the results are accurate and reflect reality.

Thus, a Top-Down analysis starts by describing the business problem domain at its most abstract, and then iteratively decomposing until the primitive level is reached. This can be done by using a standard approach such as IDEF0[4] for functional modelling. Alternatively use the UML extensions for business process modelling.

In fact, the 'top-down' approach need not address business processes at all. There are pros and cons to this. A benefit of using this approach in pattern definition is that the work can be done with industry consultants who have worked with many clients in the problem domain. In essence, one is performing 'knowledge mining' with experts and this can be much faster. It reduces the number of examples you need to work with directly, and alters the analyst's role to that of reviewer rather than worker.

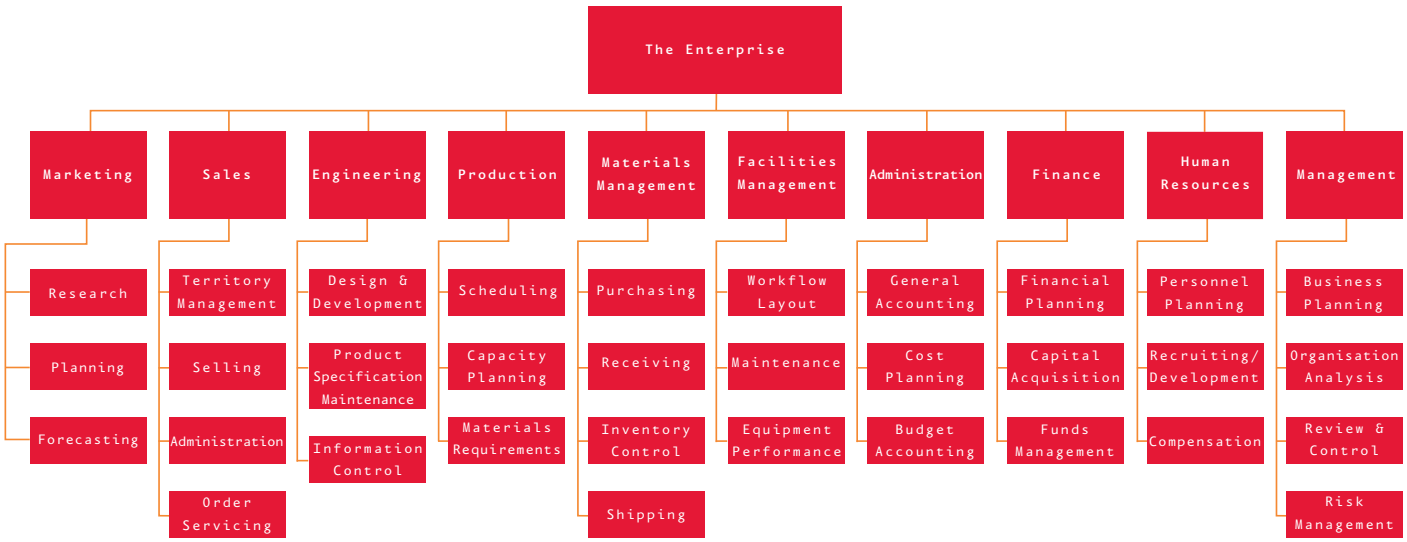
The Mixed Approach

In practice, it is often the case that we carry out the top-down and bottom-up approaches together, iteratively switching from process synthesis to hierarchy construction to decomposition to the next lower level. This has the benefit of verifying the hypothetical top-down decomposition with real-world functions gleaned bottom-up. In practice this can be the quickest and most reliable method.

Functional Decomposition Example

The following example, and those that follow, are based on a real world situation in Patient Healthcare and show the deliverables for a core business pattern, using a process analysis approach.

Figure 2. Example of typical result of business function analysis



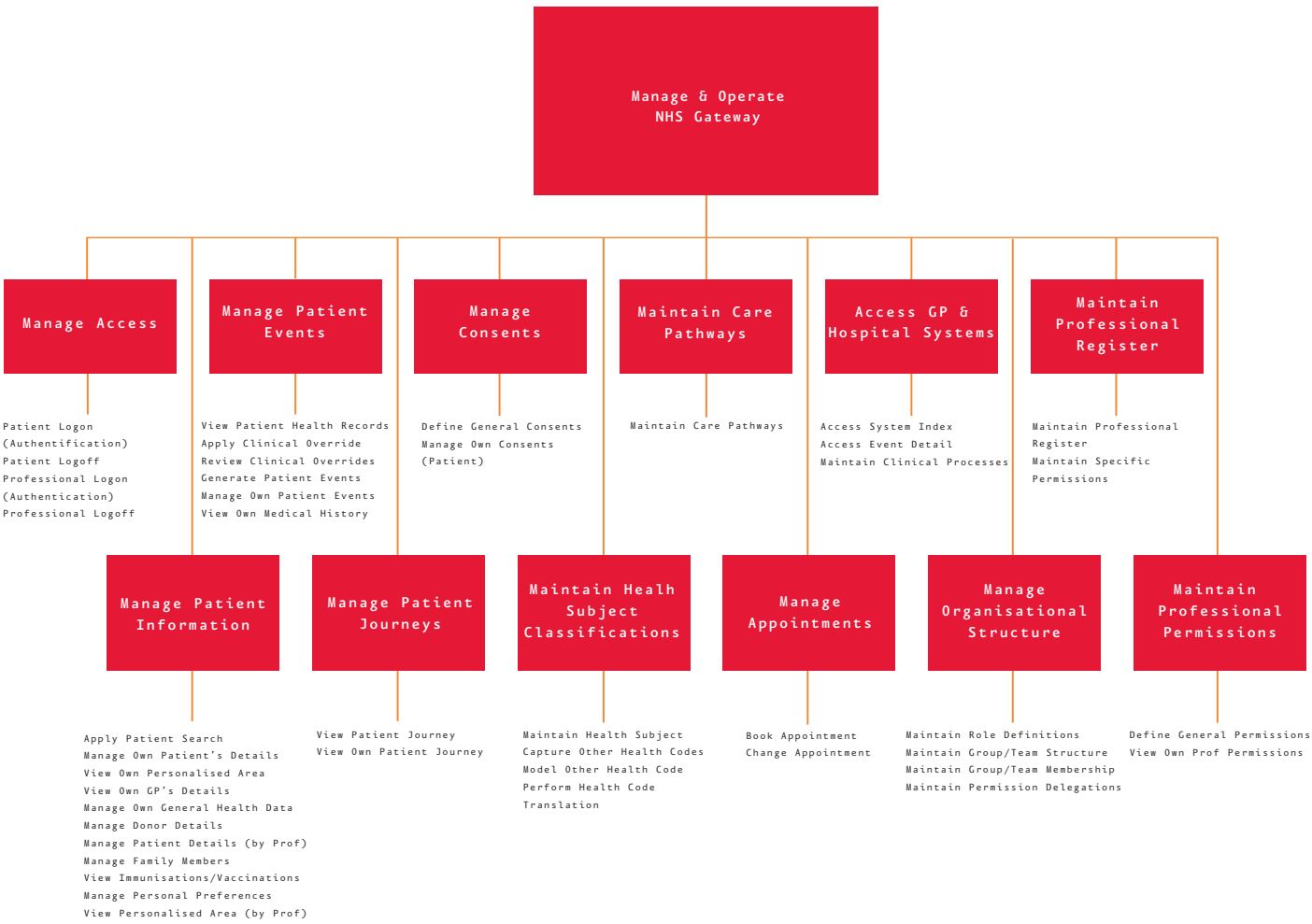
[4] A member of the IDEF family of FIPS standards from NIST. See http://www.itl.nist.gov/fipspubs/by-num.htm number 183.

Firstly, we carried out an analysis of the functional requirements of the problem domain. Working from provided scenarios such as that for breast cancer care, we have derived over 40 processes carried out by patients, professionals, system administrators and the 'confidentiality guardian'. This latter role is charged with the stewardship of confidentiality of information.

These processes were documented using UML use cases. These proved to be highly repetitious in that the same or similar sub-activities re-occurred within many use cases. Thus we extracted and consolidated these activities into a single list as follows:

Patient Logon (Authentication)
Patient Logoff (Audit)
Apply Patient Search
Manage Own Patient's Details
Manage Own Patient's Preferences
View Personalised Area
View Own GP Details
Manage Own General Health Data
Manage Donor Details
Manage Patient Details
Manage Family Members
View Immunisations/Vaccinations
Manage Personal Preferences
View Patient Health Records
View Patient Journey
View Patient Personalised Area
Apply Clinical Override
Review Clinical Overrides
Generate Patient Events
Manage Own Patient Events
Construct Patient Journey
View Own Patient Journey
View Own Medical History
Define General Consents

Figure 3. Healthcare Example – Functional Decomposition



Manage Own Consents
Maintain Health Subjects
Perform Health Code Translation
Capture Other Health Code
Model Other Health Code Structure
Maintain Care Pathways
Manage Book Appointment
Change Appointment
Access Event Detail
Access System Index
Maintain Clinical Processes
Maintain Role Definitions
Maintain Group/Team Structure
Maintain Group/Team Membership
Maintain Permission Delegations
Maintain Professional Register
Professional Logon (Authentication)
Professional Logoff (Audit)
View Own Prof Permissions
Maintain Specific Permissions
Define General Permissions

These processes may be represented in a hierarchy as shown in *Figure 3*. In so doing we have derived a higher level which summaries and contains these activities according to their subject and functional similarity.

Data Model

The Healthcare example is underpinned by a comprehensive data model.

We carried out an analysis of the data created and managed within the scope of our problem domain. This revealed 31 main entities such as Patient, Healthcare Professional, Patient Event, Care Pathway, and so on. These were defined. We identified candidate primary keys and principal attributes for these entities and mapped the relationships between the entities, including resolving any many-to-many relationships. The identification of these entities and their relationships was driven by the functional analysis being carried out in parallel. As far as we are aware, all identified functional requirements are supported by the data model and vice versa. The 31 entities have been allocated to eight 'data subjects' based on the cohesion of the entities in terms of the relative strengths of the mutual relationships.

These groups are:
– Patients
– Patient Consents
– Care Pathways
– Health Subjects
– Clinical Processes
– Roles, Teams and Organisations
– Professionals and Permissions
– Local Systems

These data subjects form the first pass definition of the required data bases and their provisional content. The Patients' data subject is shown in *Figure 4*. This takes the form of a conventional Entity-Relationship model showing data entities named and identified by their primary keys. The entities within the coloured area belong to the Patients data subject. The entities outside the coloured area belong to other data subjects but have significant relationships with entities within the Patient data subject.

A 'Crow's Foot' relationship notation has been used but an IDEF1x notation could have been used if preferred. All many-to-many relationships have been resolved. This is necessary because M:M relationships usually conceal further entities and relationships.

This approach allows us to form a shallow hierarchy for data (Root > Subject Area > Entity > Attribute) and then to form relationships between the members of Data and the members of Business Function. This normally takes the form of a CRUD Matrix showing the actions of specific Business Functions upon specific Data Entities.

Mapping Relationships

Having defined the required functionality in the form of a functional decomposition hierarchy and also defined the data required in an entity relationship data model, we can now derive a first cut component architecture by comparing the identified functions and data.

We do this by forming a matrix, the rows of which are the identified functions and the columns the identified data entities. In the cells we place a value:

– 'C' meaning this function CREATES an instance of this data entity

– 'R' meaning this function READS an instance of this data entity

– 'U' meaning this function UPDATES an instance of this data entity

– 'D' meaning this function DELETES an instance of this data entity

The result is shown in *Figure 5*.

Clustered Matrix

We have ensured that every column (data entity) has at least one *create* operation and that each row (function) has some activity. Please note that the values are not absolutely precise, particularly with *regard* to read operations. We have not specified *delete* operations.

We now analyse the matrix by using the affinity analysis and clustering

Figure 4. Example Data Model – Patients' Data Subject

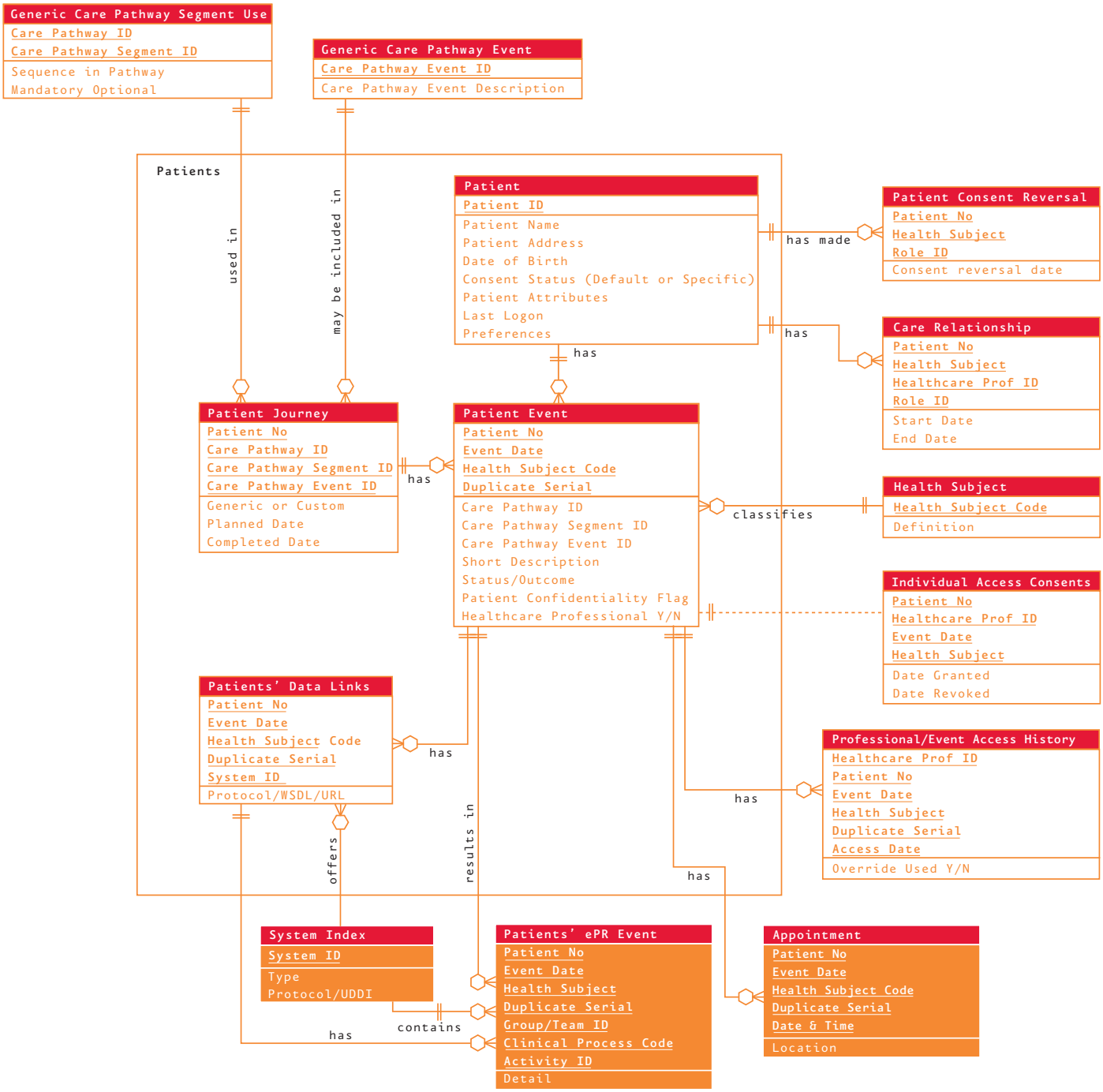


Figure 5. Function vs Data 'CRUD' Matrix

NHS Gateway Component Architecture	Appointments	Care Relationship	Clinical Pathway (Local)	Clinical Pathway Activity	General Consents Table	Generic Care Pathway	Generic Care Pathway Event	Generic Care Pathway Segment	Generic Care Pathway Segment Use	GroupTeam Membership	GroupTeam Roles	GroupTeam Structure	Health Subject	Health Subject Code Translation	Individual Access Consents	NHS Groups & Teams	NHS Professional	NHS Professional's Roles	NHS Roles	Other Health Code	Pathway Segment Event Use	Patient	Patient Consent Reversal	Patient Event	Patient Journey	Patients' Data Links	Patients' ePR Events	Permission Delegations	Professional Permissions	Professional/Event Access History	System Index
Patient Logon (Authentication)																						C									
Patient Logoff																						U									
Apply Patient Search		R																				R							R		
Manage Own Patient's Details																						U									
Manage Own Patient's Preferences																						U									
View Personalised Area																						R									
View Own GP Details																						U									
Manage Own General Health Data																						U									
Manage Donor Details																						U									
Manage Patient Details																						U							R		
Manage Family Members																						U							R		
View Immunisations/Vaccinations																						R									
Manage Personal Preferences																						R									
View Patient Health Records													R		R		R	R				R	R	R		R			R	C	
View Patient Journey						R	R	R	R								R	R			R	R		R	R				R	C	
View Patient Personalised Area																	R					R							R	C	
Apply Clinical Override																	R					R		R					R	C	
Review Clinical Overrides																	R	R											R	R	
Generate Patient Events	R		R	R									R	R						R		R		C		C	R		R		R
Manage Own Patient Events																						R		U							
Construct Patient Journey						R	R	R	R												R	R		U	C				R		
View Own Patient Journey																						R		R	R						
View Own Medical History																						R		R							
Define General Consents					C								R		U								U								
Manage Own Consents					R								R		C			R				R	C								
Maintain Health Subjects													C	C						C											
Perform Health Code Translation													R	C						R											
Capture Other Health Code																				C											
Model Other Health Code Structure																				C											
Maintain Care Pathways						C	C	C	C												C										
Manage Book Appointment	C					R			R													R		R	R				R		
Change Appointment	U																					R		R							
Access Event Detail				R																							R				
Access System Index																															R
Maintain Clinical Processes			C	C																											
Maintain Role Definitions											R								C												
Maintain Group/Team Structure										R	C	C				C			R									R			
Maintain Group/Team Membership										C	R	R				R	R	R										R			
Maintain Permission Delegations										R	R	R																C			
Maintain Professional Register																	C	C	R												
Prof Logon (Authentication)																	U												R		
Prof Logoff																	U														
View Own Prof Permissions										R			R		R		R	R											R		
Maintain Specific Permissions		C								R	R		R			R			R			R						R	C		
Define General Permissions													R																C		

technique. The objective is to deduce groups of functions and entities that share *create* and *update* operations. We are exploiting the 'loose coupling' and 'tight cohesion' notions used in the functional decomposition with the inter-entity relationships (some of which are vital and others merely transient) in the data model.

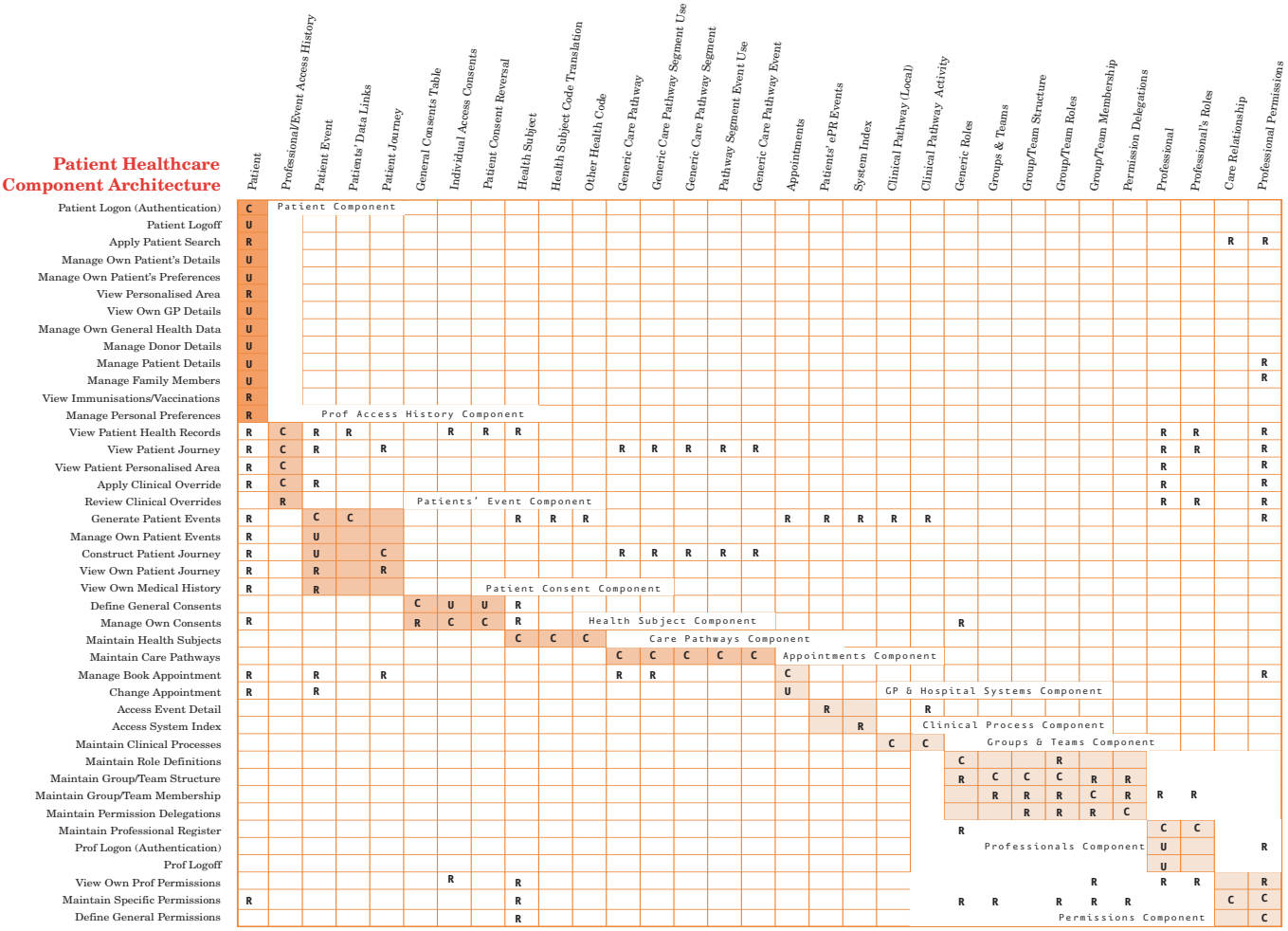
We adjust the model to bring together functions and entities with strong affinity. Description of the detailed algorithm used (the 'North West' method) is beyond the scope of this paper, however the result is the emergence of mutually exclusive groups of functions and entities formed round *create* and *update* actions. These groups are our candidate business components as shown in *Figure 6*.

Business Component Derivation

Firstly we should clarify what we mean by business component.

A business function creates, reads, updates and deletes data. Grouping together all the functions that create and update the same data entities, using a technique such as commutative clustering, defines non-redundant 'building blocks' – business components – that can be used to construct patterns, systems or applications that in turn support particular business processes. The *Business Components* sphere is an example of a 'derived sphere' – one which is deduced from

Figure 6. Clustered Matrix



the relationships between two other spheres. This is a powerful technique that exploits hidden value in an enterprise architecture. By encapsulating functionality and data into components, software reuse and replaceability become practical. Further, components offer 'services' that may be 'orchestrated' in conjunction with the services offered by other components to create an application.

The business components resulting from our cluster analysis include service interfaces, business entity components, data access components and perhaps service agents. These artifacts align with the .Net Application Architecture[5]. The business component does not contain 'agile' elements such as UI Components and UI process components, business workflows, or elements such as security, operational management and communication.

We think this coarse grained definition is very useful. It provides the stable part of the overall solution architecture which is then rounded out with the agile elements – like UI, UI Processes and Business Workflows. *Thus we can provide an agile solution based on the foundation of a stable pattern.*

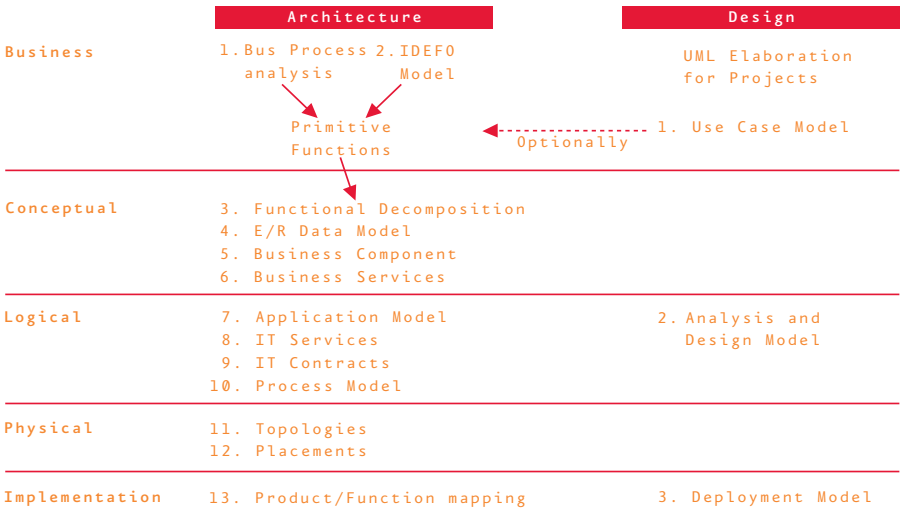
The coarse-grained business components and their services are discovered through an affinity and

[5] Application Architecture for .Net: Designing Applications and Services – Patterns and Practices Guide – Microsoft Corporation 2002

Figure 7. Sample Business Component.

Business Component Specification		
Patient Component		
Description		
The Patient component supports storage and access to data regarding a patient. Functions are provided to input, validate, maintain, store and output standing patient data such as name and address, personal details and limited medically-related data should this be available.		
Functions supported (Business Logic)	**Data maintained (Business Entities)**	
EXAMPLES: Patient Logon (Authentication) Patient Logoff (Audit) Apply Patient Search Manage Own Patient's Details View Own GP Details Manage Own General Health Data Manage Donor Details Manage Patient Data (by Professional) Manage Family Members View Immunisations/Vaccinations Manage Personal Preferences Etc	EXAMPLES: Patient Number Patient Name Patient Address Date of Birth Patient Attributes: Phone Number Occupation Sex Special Needs Marital Status Weight History Etc…	PK AK AK AK Atts
Services and Features provided:		
EXAMPLES: • Patient Logon and Logoff records • Search for Patient by Patient Number or Name/Address/DoB etc. • Maintenance of Patient Details and Preferences by the Patient • Provision of GP assignment for each patient • Provision of patient medical attributes (blood group, etc.) - GP supplied • Provision of immunisation/vaccination information - GP supplied • Etc		

Figure 8. PRM with Architect and Design views



clustering analysis performed on the relationships between business function and data. After these spheres are defined, a CRUD matrix is created to determine the component-data relationship.

In SAM, business component forms a hierarchy too. Thus the business component can decompose one level to specify the services, business entity and data access sub-components. At this level, business components will also specify the business services that we intend to expose. It is useful to explicitly call these out, so that decisions can be made around whether these will be internal services or exposed as web services.

Components and Services

The list of components is as follows:

– Patient Component
– Professional Access History Component
– Patients' Events Component
– Patient Consents Component
– Health Subject Component
– Care Pathways Component
– Appointments Component
– GP & Hospital Systems Access Component
– Clinical Processes Component
– Groups & Teams Component
– Professionals Component
– Permissions Component

We think that this set of components represents the essence of a Business Pattern definition.

One of these components - the *Patient* component is shown in *Figure 7*. This indicates the functionality, data managed and services and features offered by the component.

Figure 9. Business patterns sweet spot in the PRM

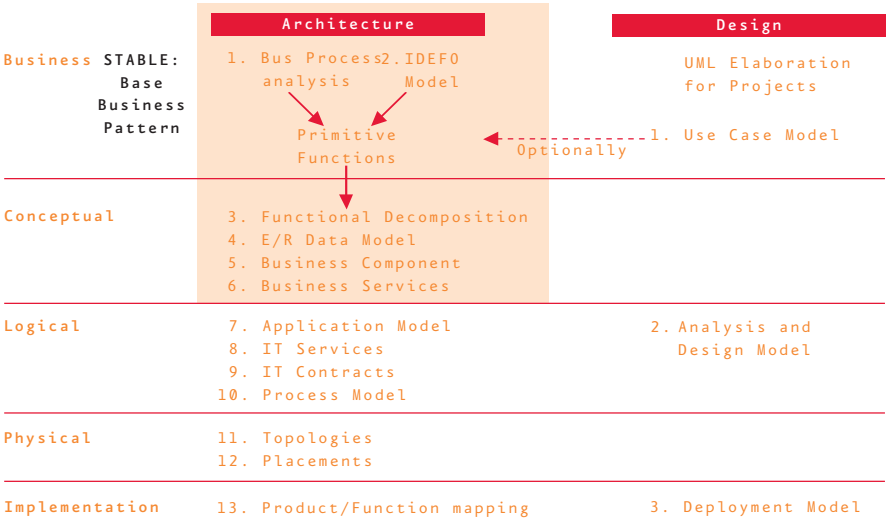
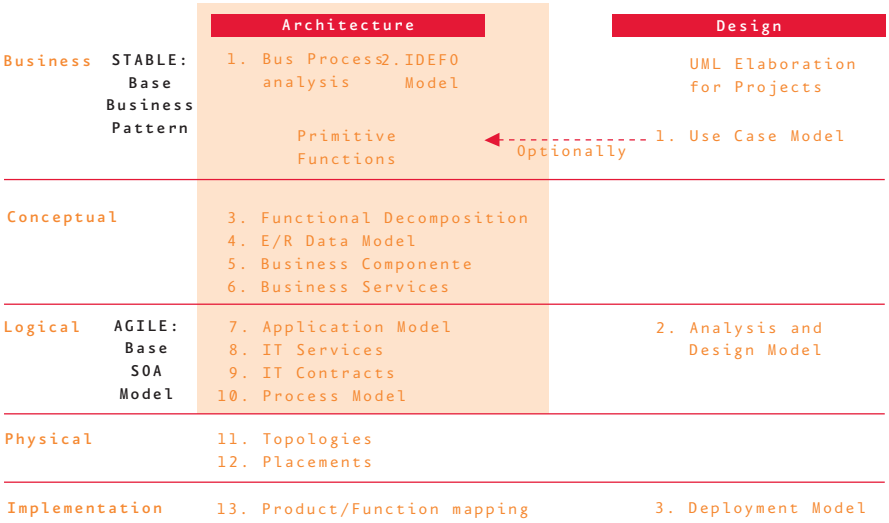


Figure 10. Adding an IT architecture to the business patterns



How the Business Patterns are useful for Software Engineering

Now we've reached the second part of this article where we talk about the roadmap for how to use the above descriptions of business patterns to engineer either other types of patterns, or actual software systems.

We illustrate this using the five layers of the PRM as shown in *Figure 8*. You'll note that the PRM distinguishes between the architect's view (a higher-level view, for example guiding a programme of projects) and a designer's view (for example, design for a project or subset thereof). *Figure 8* intends to indicate the different techniques and notations appropriate to these different views, just to give you a flavour for how the refinement is done.

What is key to note is that the PRM simply identifies the elements needs to get right through to an implementation – it makes no assumptions or judgements about how best to achieve this! You can use whatever approach you favour to fill out the set of deliverables and in whatever order you want (after the business patterns are defined). You chose the best way for the organisation that you are working in.

When considering business patterns we need to recognise this essential difference between architecture and design in the context of the side-bar about the *right level of abstraction*.

Where the Business Patterns fit

The sweetest spot for the business patterns is shown in *Figure 9*. Here they define those stable elements of a business, the business functions, data and business components that are in scope for the effort. Used this way they can be used to guide large programmes of work, which leads to greater consistency across projects, with less overall effort.

Business Patterns and Service Oriented Architecture

For those that want to provide either other IT patterns, or further guidance

[6] See http://www.microsoft.com/resources/practices/ or Amazon.
[7] http://www.martinfowler.com/books.html#eaa
[8] Design Patterns Elements of Reusable Object-Oriented Software by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. http://hillside.net/patterns/DPBook/DPBook.html

Figure 11. Adding Microsoft technology elements

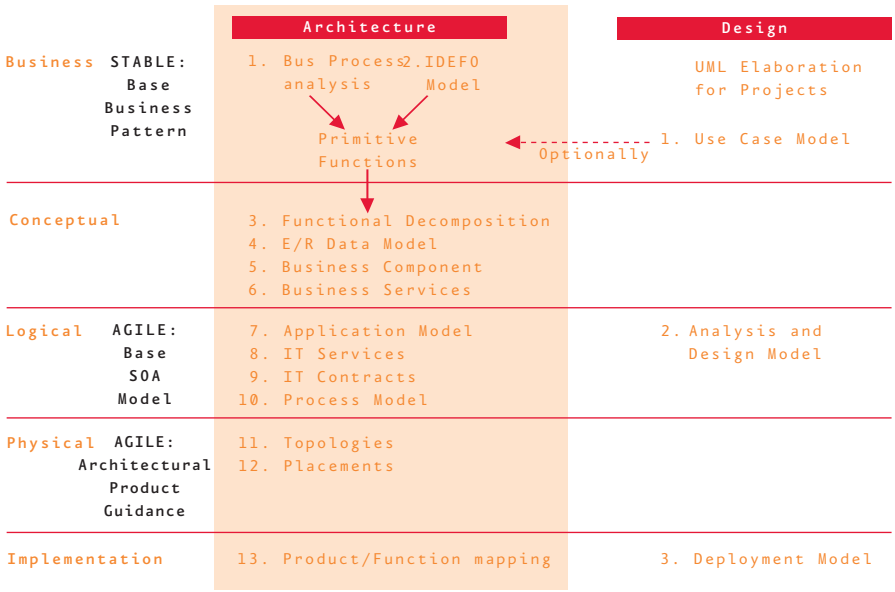
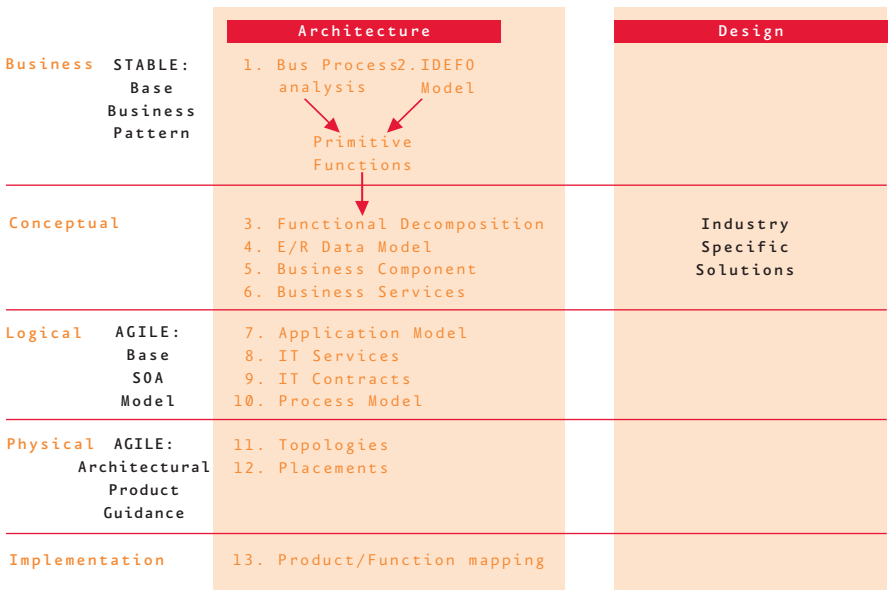


Figure 12. Refinement for solution design



related to providing an IT solution for the business problem, the business patterns can be refined further with the elements of a Service Oriented Architecture (SOA is not required, but is an excellent fit; what is needed is an architectural elaboration to transform into an IT solution).

To do this we need to add the topics that we said provided the opportunity for business agility, as shown in *Figure 10*.

The elaboration that we are describing here and in the next part could still be described in the form of patterns. The addition of these two sets of elements changes a business pattern into a business solution pattern. Alternatively, we could build a specific business solution architecture for the business pattern. We'll revisit this in a moment.

Note that at this stage all the work is still technology product-independent.

Business patterns, SOA and (Microsoft) Product details

Finally we can elaborate yet further with product and practice recommendations for successfully implementing the solution on Microsoft technology, as in *Figure 11*.

At this point we have delivered business patterns, a service-oriented application architecture, and a set of Microsoft patterns for implementation. Examples of the Microsoft technology patterns can be found on

http://www.microsoft.com/resources/practices/

Robert Jarvis,
Director, SA Ltd
v-rjarvi@microsoft.com

Robert Jarvis is a Director of Systems Advisers Limited, a UK consultancy specialising in the development of Strategic Systems Architectures for major international enterprises. He is also an Associate Architectural Consultant with Microsoft Ltd. Bob has over 30 years experience as an International Systems Consultant and Architect advising business and governmental organisations in the UK, Continental Europe and the Americas. He specialises in Enterprise Architecture working particularly on the business/technology intersection. He is the author of 'Enterprise Architecture – Understanding the Bigger Picture', a best practice guideline published by the UK's National Computing Centre in 2003.

At this stage we would have completed the delivery of a full set of linked patterns, or an IT solution, to solve a common business problem at the architecture level. But what if you'd like to actually build a system? You probably need more detail, which is what the design view will deliver.

Business Patterns and Industry Solutions

The above deliverables provide a guiding architecture which can be used to scope, costs and govern the IT projects that implement it. However, it is not detailed enough for solution implementation. If we want to provide a solution, we need further elaboration at the design level, which is guided by the architecture provided. It is probable that the solution will be implemented in several projects, and in this case the architecture is what keeps all the projects on the right track for cross-project consistency.

Figure 12 shows this next refinement. In the figure we show UML as it is a common way to drive a design phase. Again though we want to be clear that this is an illustration and there is nothing to say that is has to be done this way. All we are saying is that this is a common way to achieve the refinement of the architecture into a design for implementation.

While we are performing this refinement, we can again be creating patterns – IT design ones this time. In fact we are now getting (finally) to the territory that most of today's software pattern literature describes! Examples include Microsoft's Enterprise Solution Patterns Using Microsoft .NET[6], and Martin Fowler's Patterns of Enterprise Application Architecture[7].

Or, rather than patterns we can be creating an actual solution design.

This is the end of the roadmap for this article. Clearly the last stage would be to implement the design, and that is where patterns such as those of the Gang of Four[8] and the Patterns-Oriented Software Architecture set are very relevant, as well as the Microsoft and Martin Fowler patterns already mentioned.

Summary – Framework and approach for Business Patterns and Industry Solutions

What we have described in this article is a framework and approach for creating and then using business patterns through a guiding architecture that can be used to scope, cost and govern IT projects to implement it. To provide a solution, you need further elaboration at the design level, and this is guided by the architecture provided.

Disclaimer:
The opinions expressed in this paper are those of the authors. These are not necessarily endorsed by their companies and there is no implication that any of these ideas or concepts will be delivered as offerings or products by those companies.

Philip Teale,
Partner Strategy Consultant,
Microsoft UK
pteale@microsoft.com

Philip Teale is a Partner Strategy Consultant working for Enterprise & Partner Group in Microsoft UK. Previously, he worked for the Microsoft Prescriptive Architecture Group in Redmond, and for Microsoft Consulting Services before that. He has 29 years of Enterprise IT experience of which four years have been with Microsoft and 16 with IBM, in both field and software development roles. His international experience includes nine years working in the USA, three years in Canada and 17 years in the UK. Phil's background is in architecting, designing and building large complex distributed commercial systems. His most recent contribution to industry thought-leadership was to drive Microsoft in the creation of patterns for enterprise systems development. He is a Fellow of the RSA.

A Strategic Approach to Data Transfer Methods

By E G Nadhan and Jay-Louise Weldon, EDS

Introduction

Today, business is driven by having access to the right information at the right time. Information is the lifeline for the enterprise. However, timely access to the right information is complicated by the number and complexity of business applications and the increased volumes of data maintained. Data needs to be shared in order for business processes to be effective across the enterprise. Organizations have a variety of ways in which they can share data. A fully integrated set of applications or access to common databases is ideal. However, if these alternatives are not practical, data must be moved from one application or database to another and the designer must choose from the range of alternatives that exist for data transfer. As a result, data sharing can pose a significant challenge in the absence of an established data transfer strategy for the enterprise.

Most enterprises have acquired or built applications that support the execution of business processes specific to autonomous business units within the enterprise. While these applications serve the specific need of the business units, there continues to be a need to share data collected or maintained by these applications with the rest of the enterprise. In cases where the applications serve as the Systems of Record, the volume of data to be shared is relatively high. Further, enterprises have accumulated huge volumes of data over the past few decades as storage costs have decreased and the amount of activity tracked in the e-Commerce environment has grown beyond that of the mainframe-centric world.

Modern IT organizations face the challenge of storing, managing, and facilitating the exchange of data at unprecedented volumes. While data base management systems have evolved to address the storage and management of terabytes of data, the issue of effectively exchanging high volumes of data between and among enterprise applications remains. A sound, enterprise-wide data transfer strategy is necessary to guide IT practitioners and to enable consistent representation of key business entities across enterprise applications.

Background

The mainframe world of the 70's consisted of punch card driven monolithic applications, many of which continue to be the systems of record in organizations today. The advent of the Personal Computer in the 80's fostered the client-server world of the early 90's where the PC evolved into a robust client workstation. Processing power continued to grow exponentially resulting in the introduction of mid-range servers that were employed by key business units within the organization. Client-server technology gave these autonomous business units the power to store and use data in their own world. Such autonomy gave birth to many repositories that did a good job of storing isolated pockets of data within the enterprise. N-tier distributed computing in the late 90's resulted in the creation of additional layers that stored data specific to these business units.

While departmental business processes are not impacted by the proliferation of data across multiple repositories, there exists a critical need to leverage data at an enterprise level – as well as at the business unit level. For example, organizations need to have an enterprise level view of the customer and serve their customers as a single logical entity. In today's world of real-time online interaction with customers, end-to-end system response has become a critical success factor as well. The fundamental requirements to provide basic customer service have not changed over the years. However, maintenance and retrieval of the data expediently to provide such service has become a much more complex process.

In spite of this complexity, enterprises of today need to have access to all these pockets of data as well as the original systems of record. Such access is accomplished either by building connection mechanisms to the various systems or by transferring data between systems at periodic intervals. Enterprise Application Integration (EAI) tools can be applied to move transactions and messages from one application to another. Extract Transformation and Load (ETL) tools perform the same task but usually move data in bulk.

This article describes the options available to address this problem of data sharing. While the options are not mutually exclusive, they represent logically different design and execution principles.

Target Audience

IT personnel who are faced with the challenges of sharing data between multiple applications within the enterprise would benefit from the

"A sound, enterprise-wide data transfer strategy is necessary to guide IT practitioners and to enable consistent representation of key business entities across enterprise applications."

contents of this article. Such personnel include IT Enterprise Architects, Data Architects, Integration Architects as well as Subject Matter Experts for the key enterprise applications. Process and Functional managers within the enterprise who work closely with the IT architects will develop an appreciation for the complexities of data sharing driven by business process changes.

Problem Definition

Applications often need to make their data accessible to other applications and databases for various reasons. Data may need to be moved from one platform to another or from one geographic location to another. Data may need to be moved to make it accessible to other applications that need it without impacting the performance of the source system. Changes in data may need to be moved to keep two systems in sync. Often, firms will create a shared repository, called an Operational Data Store (ODS), to collect data from source systems and make it available to other systems and databases. Data must then be moved from the originating application to the ODS.

There are many ways to accomplish data transfer and many factors to consider when choosing the alternative that best fits the situation at hand. Efficiencies become critical when the data volume is large. Bulk data transfer may not be a viable alternative due to time constraints. At the same time, identification of changed data can be a challenge.

The example below represents a realistic scenario where this situation manifests itself.

Sample Scenario

A customer-facing website allows subscribers to a service to enroll online. The processes involved in this activity will capture a number of relevant data elements about the subscriber. The captured data will be immediately housed in a Sales system (e.g. an Order Management System). In this example, the Sales system would be considered the System of Record for these data elements. Subscriber data is, no doubt, critical to the Sales department. At the same time, it is also important to several other business units. For instance, the Billing department will need it to make sure that financial transactions with the subscriber are executed. And, the Marketing department may want this data to help design campaigns to cross-sell and up-sell products and services to the subscriber. Therefore, it is crucial that subscriber data be placed, as soon as possible in an enterprise accessible Operational Data Store from which it can be made available to those applications that need it.

From a systems standpoint, *Figure 1* represents the scenario just described. It illustrates a front end application storing data into its proprietary System of Record and receiving an acknowledgement of a successful update. This System of Record is constantly populated with high volumes of data that need to be transferred to an Operational Data Store so that they may be shared with the rest of the enterprise. The subsequent sections illustrate the different ways of accomplishing such a transfer.

Figure 1 illustrates the following steps:

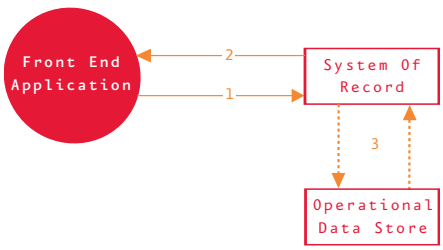
1. Front End Application updates System of Record.
2. System of Record acknowledges successful update.
3. Transfer of data to the Operational Data Store.

Depending on the context of the specific problem domain for a given enterprise, there are multiple approaches to effect the transfer of data to the ODS in this scenario. The various approaches involved are described in the sections that follow.

The approaches presented are based on the following assumptions:

1. For simplicity's sake, we have assumed that there is only one System of Record within the Enterprise for any given data element. Propagation of data to multiple Systems of Record can be accomplished using one or more of these options.
2. An *update* to a System Of Record can mean the creation, modification, or even the deletion of a logical record.
3. The *acknowledgement* step is the final message to the Front End Application that indicates that all the intermediate steps involved in the propagation of data to the System of Record as well as the Operational Data Store have been successfully completed. Additional acknowledgement steps between selected pairs of nodes might be necessary depending on the

Figure 1. Sample Scenario



implementation context for specific business scenarios.

4. Metadata, while not directly addressed in this paper, is a crucial consideration for data transfer[1]. It is assumed that all the options discussed entail the capture, manipulation and transfer of metadata. However, the discussion in this paper is limited to the logical flow of data between the different nodes within the end-to-end process.

Business Process Review

There are various options available for engineering the transfer of data within the Sample Scenario defined in *Figure 1*.

However, prior to exercising any given option, it is prudent to take a step back, review and validate the business need for the transfer of data. The actual transfer of data between systems could be a physical manifestation of a different problem at a logical business process level. A review of the end-to-end processes may expose opportunities to streamline the business process flow resulting in the rationalization of the constituent applications. Such rationalization could mitigate and in some cases, eliminate the need for such transfer of data. Some simple questions to ask would include:

– *Why does the data need to be transferred?*
– *Why can't the data stay in a single system?*

A viable answer to these questions could eliminate the need for such transfer of data.

If there is still a clear need for this transfer of data even after a review of the end-to-end business process, there are multiple options available that broadly conform to one or more of these approaches:

– EAI Technologies
– ETL Technologies
– Combinations

The remaining options explore these different possibilities.

Data Transfer Options

This section describes the architectural options that are available for sharing data between discrete applications. The discussion here is independent of commercial solutions; rather it focuses on genres of technologies that are available in the market today.

The options discussed in this section are:

– Option 1: EAI Real-time Transfer
– Option 2: EAI Propagation of Incremental records
– Option 3: Incremental Batch Transfer (Changed Data Capture)
– Option 4: Native Replication
– Option 5: Bulk Refresh using Batch File Transfer
– Option 6: ETL/ELT Transfer
– Option 7: Enterprise Information Integration

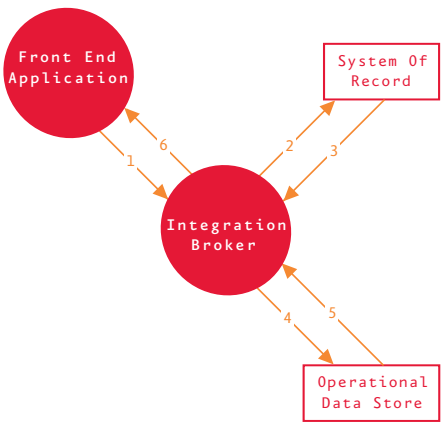
Option 1: EAI Real-time Transfer

Figure 2 illustrates the manner in which an EAI Integration Broker[2] can facilitate this transfer.

This option is application-driven and most appropriate for data transfers where the updates to the System of Record and the ODS are part of the same transaction. An Integration Broker receives the transaction initiated by the Front End Application after which it assumes responsibility for the propagation of the data to the System of Record as well as the Operational Data Store. The steps are executed in the following sequence:

1. Front End Application initiates update to the System of Record.
2. Integration Broker receives this update from the Front End Application and sends it to the System of Record.
3. System of Record acknowledges the update.
4. Integration Broker immediately initiates a corresponding update to the Operational Data Store, thereby effecting an immediate, real-time transfer of this data.
5. ODS acknowledges the receipt of this update.

Figure 2. EAI Real-time Transfer



[1] Effective data sharing requires a common understanding of the meaning and structure of data for the provider and the receiver. Metadata – data about data – is the vehicle for achieving that understanding. When data is shared or physically transferred between parties, metadata also must be exchanged. It is the designer's responsibility to ensure the appropriate metadata is captured and transferred in all data transfer situations.

[2] An integration broker is a component that routes the messages exchanged between applications. It facilitates the conditional transfer of messages between applications based on predefined rules driven by business logic and data synchronization requirements.

6. Integration broker sends the acknowledgement of these updates to the Front-End Application.

Usage Scenario – Financial Institution
A front end CRM application captures data about prospects calling into the Contact Center. The CRM application propagates the prospect data to an Operational Data Store that contains the basic customer data for enterprise-wide reference. This data needs to be propagated to the ODS immediately so that the most current data is available to all the other customer-facing applications like the Automated Teller Machine (ATM), financial centers (branches) and online banking access.

Option 2: EAI Propagation of Incremental records
This option is application-driven and appropriate for lower priority data. The Front End Application updates the System of Record after which this data is propagated to the ODS through the Integration Broker. This is characteristic of scenarios where there is a tightly coupled portal to an ERP or CRM system. There are two different mechanisms to effect this transfer of data to the ODS:
– *Option 2a: Push to Integration Broker:* System of Record initiates the notification of the receipt of this data to the Integration Broker. The 'push' is frequently triggered by a scheduled requirement, for example, daily update.
– O*ption 2b: Pull from Integration Broker:* Integration Broker continuously polls the System of Record for receipt of this data. The 'pull' is frequently triggered by a business event in the application using the ODS, for example, a service transaction that requires up-to-date customer data.

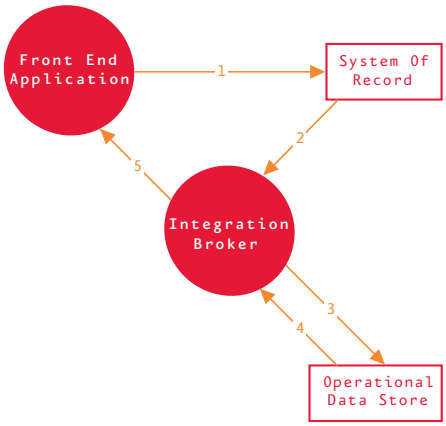
Usage Scenario – Manufacturing Organization
An order entry ERP application is used by Customer Service Representatives to enter orders every hour directly into the backend orders database. New orders received must be transferred to the enterprise service dashboard repository on a daily basis. The enterprise service dashboard provides management a holistic view of the order volume as of the previous business day. The first option could be a daily 'push' of new orders from the ERP application to the dashboard repository. Or, the dashboard could initiate a 'pull' from the orders database through the Integration Broker to provide this data when management requires the latest view of the order volume.

Each of these options is explained in further detail below.

Option 2a: Push to Integration Broker
Figure 3 illustrates the EAI propagation of incremental records by having the System of Record push this data to the Integration Broker. The steps are executed in the following sequence:
1. Front End Application initiates update to the System Of Record
2. System of Record notifies Integration Broker about receipt of this data after completing the update.
3. Integration Broker receives this update and sends it to the ODS.
4. ODS acknowledges the update.
5. Integration Broker sends an acknowledgement of the successful propagation of this data to the Front End Application.

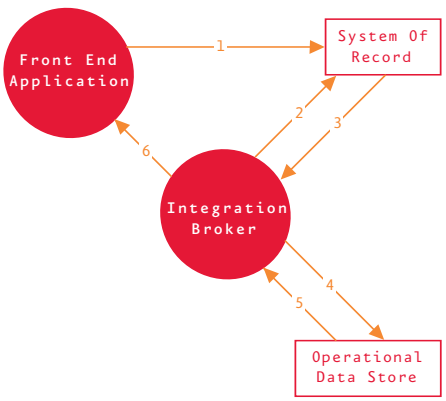
Option 2b: Pull from Integration Broker
Figure 4 illustrates the EAI propagation

Figure 3. Push to Integration Broker



of incremental records by having Integration Broker poll the System of Record on a regular basis and propagate this data to the ODS. The steps are executed in the following sequence:
1. Front End Application initiates update to the System of Record.
2. Integration Broker polls the System of Record to check if any new data has been received.
3. System of Record responds to the poll.
4. If there is new data to be propagated, Integration Broker sends an update to the ODS.

Figure 4. Pull from Integration Broker



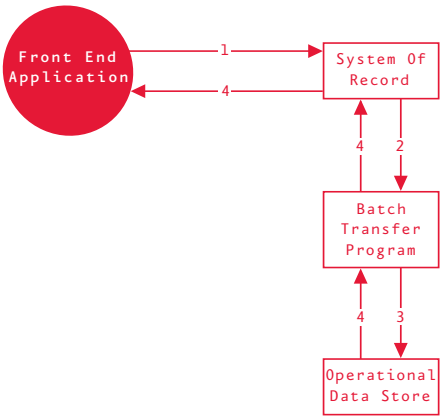
"While departmental business processes are not impacted by the proliferation of data across multiple repositories, there exists a critical need to leverage data at an enterprise level as well as at the business unit level."

5. ODS acknowledges the update.
6. Integration Broker sends an acknowledgement of the successful propagation of this data to the Front End Application.

Option 3: Incremental Batch Transfer (Changed Data Capture)

Option 3 is data-driven and is used to periodically move new or changed data from the source to the target data store. This option is applicable to scenarios where it is acceptable for the data updated in the System of Record to be provided to other applications after a finite time window (e.g. one day). In such scenarios, the data is transferred on an incremental basis from the System of Record to the ODS. This data sharing option involves capturing changed data from one or more source applications and then transporting this data to one or more target operations in batch. This is graphically depicted in *Figure 5*. Typical considerations in this option include identifying a batch transfer window that is conducive to both the source and target system(s) to extract and transport the data.

Figure 5. Incremental Batch Transfer



There are two ways to accomplish this:

1. Change Log: System of Record maintains the changed data in dedicated record sets so that the Batch Transfer Program can directly read these record sets to obtain the delta since the last transfer. In this case, the System of Record is responsible for identifying the changed data in real-time as and when the change happens.
2. Comparison to previous: Batch Transfer Program leverages the data in the base record sets within the System of Record to identify the changed content. In this case, the Batch Transfer Program has the responsibility of comparing the current state of data with earlier states to determine what had changed in the interim.

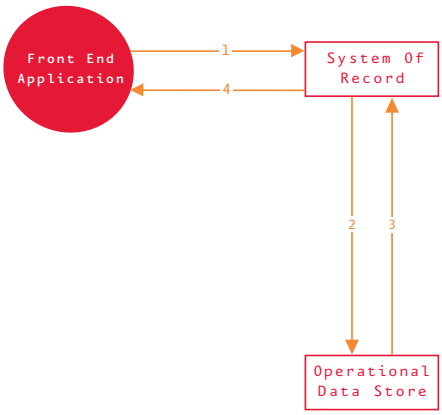
The typical sequence of events for this kind of data sharing is as follows:

1. Front End Application initiates update to the System of Record.
2. Batch Transfer Program fetches changed data from System of Record.
3. Batch Transfer Program updates Operational Data Store.
4. An acknowledgement is sent to the Front End Application, System of Record and/or the Batch Transfer Program after the Operational Data Store has been successfully updated.

Usage Scenario – Service Provider

The sales force uses a sales leads database that tracks all the leads that the sales representatives are pursuing. The project delivery unit tracks the resources required for sales and delivery related activities. The project delivery unit maps resource requirements to existing projects as well as leads currently in progress. To that end, the leads data is transferred on a daily basis from the sales leads database to the project delivery database through the incremental batch transfer option.

Figure 6. Native Replication

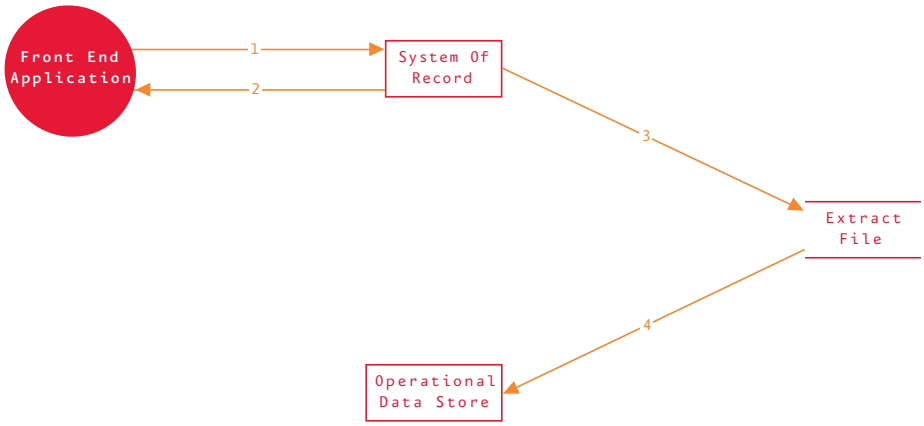


Option 4: Native Replication

Option 4 is a data-driven option that is especially relevant for high-availability situations, e.g., emergency services, where the source and target data stores need to stay in sync virtually all the time. This data sharing option involves the use of native features of database management systems (DBMS) to reflect changes in one or more source databases to one or more target databases. This could happen either in (near) real-time or in batch mode. The typical sequence of events for native replication is:

1. Front End Application initiates update to the System of Record.
2. Native Replication transfers data from System of Record to Operational Data Store.
3. Operational Data Store sends an acknowledgement of receipt of data back to the System of Record.
4. The System of Record sends an acknowledgement of the success of the operation to the Front End Application.

Figure 7. File Extract



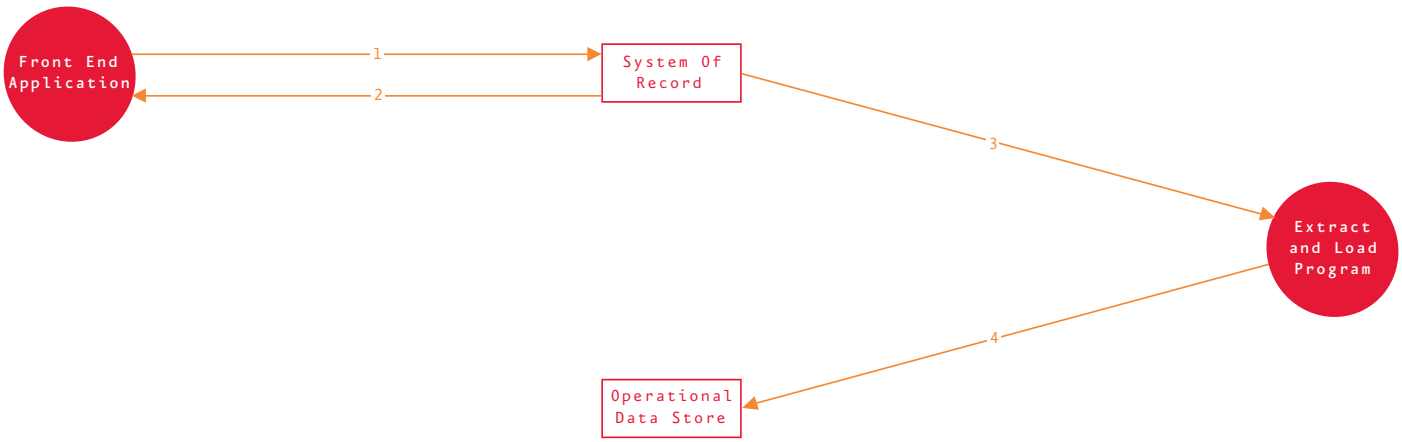
Usage Scenario – Health Care Payer
Claims data is being entered through a two-tier Client Server application to a backend RDBMS by Customer Service Representatives. Updates to the Customer Profile are also made in the System of Record while entering data about the claims. Customer Profile updates are directly replicated into the ODS which serves as the Customer Information File for all the other enterprise applications.

Option 5: Bulk Refresh using Batch File Transfer
This option is data-driven and appropriate when a large amount of data, for example, a reference table of product data, needs to be periodically brought into sync with the System of Record. This option transfers all the data inclusive of the latest changes on a periodic basis. All the records are extracted from the System of Record and refreshed into the ODS. Existing records in the ODS are purged during each transfer. Such transfers are typically done in batch mode overnight.

Bulk Refresh is well suited for scenarios where there is significant overhead involved in identifying and propagating the incremental changes. The incremental approach can be more error prone and therefore, maintenance intensive.
These types of transfers can be accomplished in one of two ways:
– *Option 5a: File Extract:* A program in System of Record extracts all the records into an intermediate file. This file is subsequently loaded into the ODS by another program.
– *Option 5b: Program Extract:* A separate program queries the System of Record and transfers each record in real time to the ODS. There is no intermediate file created.

Option 5a: File Extract with Full Refresh
Figure 7 illustrates the file based extraction process for bulk transfer of data. The following steps are executed in this process:

Figure 8. Program Extract



"The enterprise-wide data model functions like a virtual database. In some respects, it is a view, in relational database terms, on tables spread across multiple physical databases."

1. Front End Application initiates update to the System of Record.
2. System of Record acknowledges the update.
3. All records are extracted into an Extract File from the System of Record.
4. Extract File is refreshed into ODS.

Option 5b: Program Extract with Full Refresh

Figure 8 illustrates the program-based extraction process for bulk transfer of data. The following steps are executed in this process:

1. Front End Application initiates update to the System of Record.
2. System of Record acknowledges the update.
3. Extract and Load program retrieves and updates all the records from the System of Record into the ODS.

Unlike the File Extract, retrieval from the System of Record and updates into the ODS are part of a single transaction with no intermediate persistence of the data. The Extract and Load program can be triggered at fixed time intervals or on the occurrence of specific events. For instance, it can run four times a day, or on updates to a critical master table. While this is architecturally similar to *Option 3: Incremental Batch Transfer* (see *Figure 5*), the scope is different: here, all data from the System of Record is transferred to the ODS, rather than just an incremental change.

Usage Scenario – Large Enterprise HR Department

Large international enterprises with thousands of employees have an organizational hierarchy that is spread wide and deep across the globe. A minor change to this hierarchy can have a ripple effect across the organizational layers. While the organizational structure is maintained in a single repository, it is used in a read only mode by other applications from the Operational Data Store. The organizational structure, thus, must be fully refreshed on a regular basis in the Operational Data Store.

Option 6: ETL/ELT Transfer

Option 6, illustrated in *Figure 9*, is data driven and most appropriate where substantial data scrubbing and transformation are required as the data are moved, e.g., for integration into a data warehouse or data mart. This option overlaps with both *Option 3: Incremental Batch Transfer* and *Option 5: Bulk Refresh* transfers. The difference is that business logic is applied to the data while it is transported from source to target systems. An ETL tool is often used for this kind of data transfer. Source data is extracted, transformed en route, and then loaded into one or more target databases. The transformations performed on the data represent the business rules of the organization. The business rules ensure that the data is standardized, cleaned and possibly enhanced through aggregation or other manipulation before it is written to the target database(s).

ETL transfer involves the following steps:

1. Front End Application initiates update to the System of Record.
2. ETL Transfer Program fetches changed or bulk data from System of Record.
3. ETL Transfer Program updates Operational Data Store.
4. An acknowledgement is sent to the Front End Application, System of Record and/or the ETL Transfer Program after the Operational Data Store has been successfully updated.

The same applies to ELT transfer as well. The difference between ETL and ELT lies in the environment in which the data transformations are applied. In traditional ETL, the transformation takes place when the data is en route from the source to the target system. In ELT, the data is loaded into the target system, and then transformed within the target system environment. This has become a popular option recently with since there are significant efficiencies that can be realized by manipulating data within database environments (for example by using stored procedures).

Usage Scenario – Health Care Provider

Employers send Entitlement information for Employees and their dependents to Health Care Insurance Payers on a weekly basis recording all the changes that happened each week. The incoming data is in a format proprietary to the Employer that needs to be converted into the Health Care provider's backend mainframe system's format. Summary records have to be

Figure 9. ETL Transfer

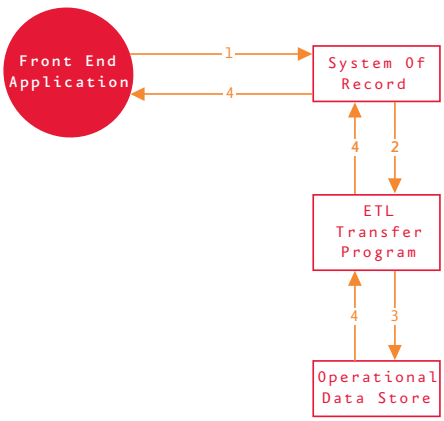
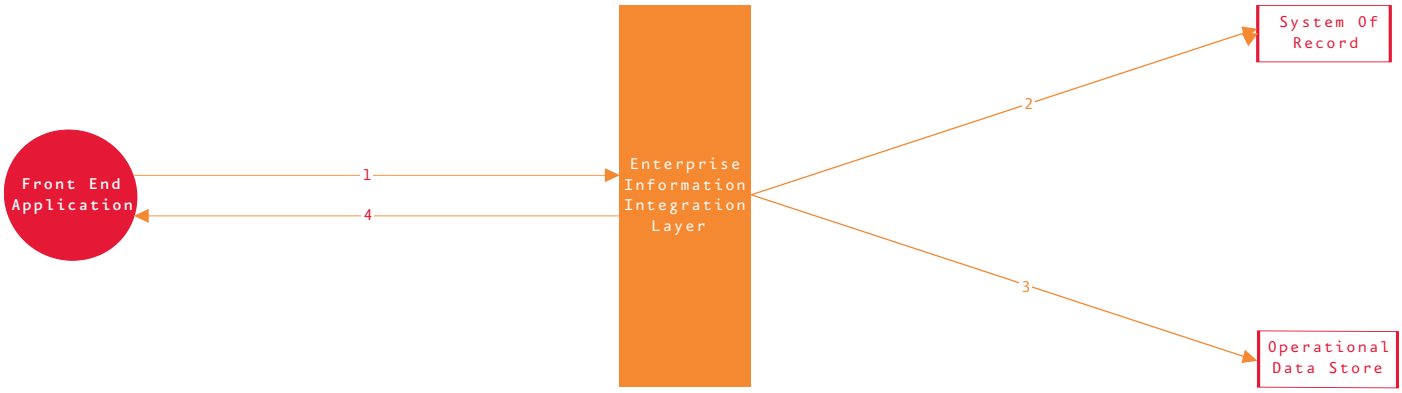


Figure 10. Enterprise Information Integration



created that list the number of dependents and children that each employee has. ETL tools can be used to perform these format and content transformations in batch mode.

Option 7: Enterprise Information Integration

This option is an emerging one and is similar to *Business Process Review*. It involves the creation of a logical enterprise-wide data model that represents the key business entities and their relationships in a consistent, standardized fashion. The Enterprise Information Integration layer where this model resides has the business intelligence to do the following:

– Determine the repository that has the most accurate value for each data element.
– Construct the result set by fetching the right information from the right repository.
– Propagate updated information to all the affected repositories so that they are in a synchronized state all the time.
– Provide an enterprise-wide view for all the business entities.

The enterprise-wide data model functions like a virtual database. In some respects, it is a *view*, in relational database terms, on tables spread across multiple physical databases.

As part of its information integration responsibilities, the Enterprise Information Integration (EII) layer can propagate the information to the ODS and the System of Record ensuring that they are synchronized. This is illustrated in *Figure 10*.

The following execution steps are involved when the EII option is exercised:

1. Front End Application initiates update to the System of Record through the EII layer.
2. EII layer updates System of Record.
3. EII layer updates the Operational Data Store.
4. Upon successful completion of both updates, the EII layer sends the acknowledgement back to the Front End Application.

Compound Scenarios

Apart from the *Sample Scenario* described at the beginning of this paper and the usage scenarios described under each option, there are complex situations where the various options for data transfer need to be evaluated carefully and a combination of the relevant ones applied. These scenarios include, but are not limited to:

– Populating a DW or an ODS with data from operational systems
– Populating data marts from a DW or an ODS
– Back propagating integrated data into applications
– Combinations of application-to-application and application-to-ODS data transfers

The first three of these scenarios can be handled using *Business Process Review* and/or *Option 1: EAI Real-time Transfer* through *Option 7: Enterprise Information Integration* described above. Application to application scenarios involve a mix of the above options and two types are discussed here in detail.

"The most appropriate option for an environment is based on the data transfer requirements and constraints specific to that environment."

Option 8a: Application-to-Application Transfer with Cross-reference

Option 8a is appropriate when the EAI tool must perform a simple lookup during data transfer. For example, while transferring data from a Sales application (X) to a Finance application (Y), current account code based on the transaction type in the Sales transaction must be looked up and added to the transaction during transfer.

The business requirement in this scenario, graphically depicted in *Figure 11*, is to transfer data from application X to application Y. As part of this transfer, there must be manipulations performed on the data that require cross-reference tables (like looking up codes and translating into meaningful values in the target system). While real-time EAI transfer can effect the transfer of data from application X to application Y, ETL transfer can be used to transfer cross-reference data from these systems into a cross-reference data construct (represented as XREF in the diagram).

Note: *Option 5a: File Extract with Full Refresh* or *Option 5b: Program Extract with Full Refresh* could also be used to update the XREF table.

Option 8b: Application-to-Application Transfer with Static Data

Option 8b represents a situation where the data from application X must be augmented with data from application Z during transfer to application Y. For example, a transaction from the Sales application (X) must be augmented by product cost data from the Inventory application (Z) during transfer into the Finance application (Y).

In this scenario, depicted in *Figure 12,* data is transferred from application X to Y. At the same time, updating application Y also involves receiving other data from secondary applications that are static – or at least relatively static compared to the real-time nature of transfer from X to Y. Here, EAI is used to achieve the transfer of some of the data from X to Y. ETL transfer is used to prepare and provide the additional data that application Y requires from a secondary application (Z) into an ODS. EAI then fetches the additional data from the ODS to populate application Y.

Note: Any one of *Option 6: ETL / ELT Transfer* through *Option 7: Enterprise Information Integration* could be used for the update of the ODS.

Options Analysis

The most appropriate option for an environment is based on the data transfer requirements and constraints specific to that environment. There are several procedural, architectural and financial criteria that have to be taken into account while determining the most suitable option for an environment. This section outlines the key criteria to be considered followed by a ranking of each option in the context of these criteria. While there may very well be other applicable options or combinations of these options as discussed under Compound Scenarios, this section focuses on the basic options (1 through 6) described earlier.

Figure 11. A2A Transfer with Cross-Reference

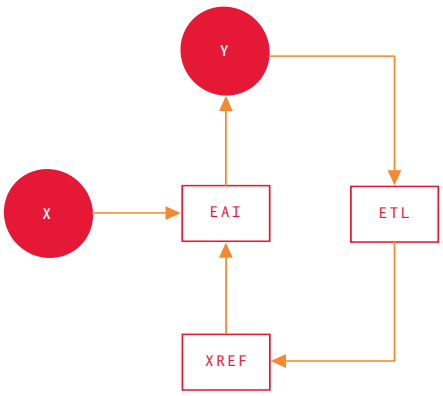
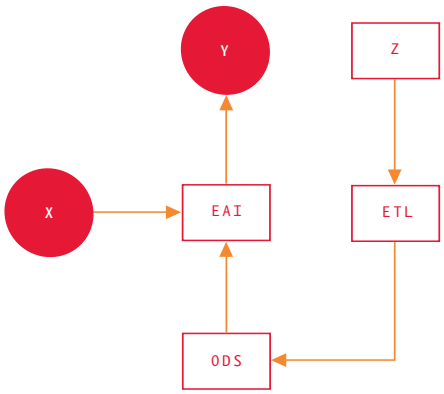


Figure 12. A2A Transfer with Static Data



Business Process Review and *Enterprise Information Integration* have been excluded from the analysis since they do not actually involve the transfer of data.

These criteria can be classified into *Requirements and Constraints* as shown in *Table 1*. Requirements are typically architectural in nature, driven by business needs. Constraints define the parameters within which the solution must be architected keeping the overall implementation and maintenance effort in mind.

Table 1. Evaluation Criteria

Criterion	Category	Description
Latency	Requirement	How quickly is the data to be transferred?
Transformation	Requirement	Complexity of the transformation to be performed as part of the transfer
Volume	Requirement	Quantity of data exchanged during each transfer
Intrusion	Constraint	Degree of change to existing applications in order to effect data transfer
Effort	Constraint	Effort required to build and maintain the solution

Table 2. Options Evaluation

Criterion / Option	1 – EAI Real Time Transfer	2 – EAI Propagation of Incremental Records	3 – Incremental Batch Transfer	4 – Native Replication	5 – Bulk Refresh with Batch Transfer	6 – ETL/ELT
Latency	Near Real Time	Near Real time	Batch	Near Real Time	Batch	Batch
Transformation	High	High	Medium	Not Applicable	Not Applicable	High
Volume	Low	Low	Medium	Low	High	High
Intrusion	High	Medium	High	Low	Low	Low
Effort	High	Medium	Medium	Low	Low	High

Table 2 outlines the characteristics of *Option 1: EAI Real-time Transfer* through *Option 6: ETL / ELT Transfer* in the context of these criteria. Please note that *Business Process Review* and *Option 7: Enterprise Information Integration* have not been analyzed in *Table 2*. *Business Process Review* is a revision to the existing business processes that may result in the implementation of any one of the other options. *Option 7: Enterprise Information Integration* has to do with the logical representation of information at an enterprise level. Any one of *Option 1: EAI Real-time Transfer* through *Option 6: ETL / ELT Transfer* may be used in conjunction with the EII model.

E G Nadhan
Principal, EDS
Easwaran.Nadhan@eds.com

E G Nadhan is a Principal with the EDS Extended Enterprise Integration group. With over 20 years of experience in the software industry, Nadhan is responsible for delivering integrated EAI and B2B solutions to large scale customers.

Conclusion

There are many approaches available to enterprises for effecting data transfer between and among their business applications. Enterprises should first review the Business Process to confirm the necessity of the transfer. Once confirmed, there are multiple options, enabled by EAI and ETL technologies, to effect the data transfer. In some cases, a combination of options might be needed to address the complete set of data transfer requirements within an enterprise. The process driving such transfers should establish the technology and the tool employed rather than have the technology define the process. Large enterprises typically employ an optimal mixture of all three strategies: *Business Process Review,* EAI and ETL. *Enterprise Information Integration* is emerging as another viable option in this space. The right option or combination of options to be used for a given scenario depends upon several criteria, some of which are requirements-driven while others are constraints. This paper presents the most significant criteria to consider and provides an evaluation of each option based on these criteria.

Special Acknowledgement:
The authors thank Carleen Christner, Managing Consultant with the EDS Extended Enterprise Integration group for her thorough review of the paper and the feedback she provided on the content and format.

Jay-Louise Weldon
Managing Consultant, EDS
Jaylouise.weldon@eds.com

Jay-Louise Weldon is a Managing Consultant with EDS' Business Intelligence Services group. Jay-Louise has over 20 years experience with business intelligence solutions and database and system design.

Messaging Patterns in Service Oriented Architecture – Part 2

By Soumen Chatterjee

Introduction
In part one of this paper published in issue 2 of JOURNAL we described how messaging patterns exist at different levels of abstraction in SOA. Specifically, Message Type Patterns were used to describe different varieties of messages in SOA, Message Channel Patterns explained messaging transport systems and finally Routing Patterns explained mechanisms to route messages between the Service Provider and Service Consumer. In this second part of the paper we will cover Contract Patterns that illustrate the behavioral specifications required to maintain smooth communications between Service Provider and Service Consumer and Message Construction Patterns that describe creation of message content that travels across the messaging system.

Contracts and Information Hiding
An interface contract is a published agreement between a service provider and a service consumer. The contract specifies not only the arguments and return values that a service supplies, but also the service's pre-conditions and post-conditions.

Parnas and Clements best describe the principles of information hiding:

"Our module structure is based on the decomposition criterion known as information hiding [IH]. According to this principle, system details that are likely to change independently should be the secrets of separate modules; the only assumptions that should appear in the interfaces between modules are those that are considered unlikely to change. Each data structure is used in only one module; one or more programs within the module may directly access it. Any other program that requires information stored in a module's data structures must obtain it by calling access programs belonging to that module".
(Parnas and Clements 1984)[8]

Applying this statement to SOA, a service should never expose its internal data structures. Otherwise it causes unnecessary dependencies (tight coupling) between the service provider and its consumers. Internal implementation details are exposed by creating an parameterized interface design mapped to the service's implementation aspects rather than to its functional aspects.

Contract Pattern
Problem:
How can behaviors be defined independent of implementations?

Solution:
The concept of an interface contract was added to programming languages like C# and Java to describe a behavior both in syntax and semantics. Internal data semantics must be mapped into the external semantics of an independent contract. The contract depends only on the interface's problem domain, not on any implementation details.

Interactions:
The methods, method types, method parameter types, and field types prescribe the interface syntax. The comments, method names, and field names describe the semantics of the interface. An object can implement multiple interfaces.

"A service should never expose its internal data structures."

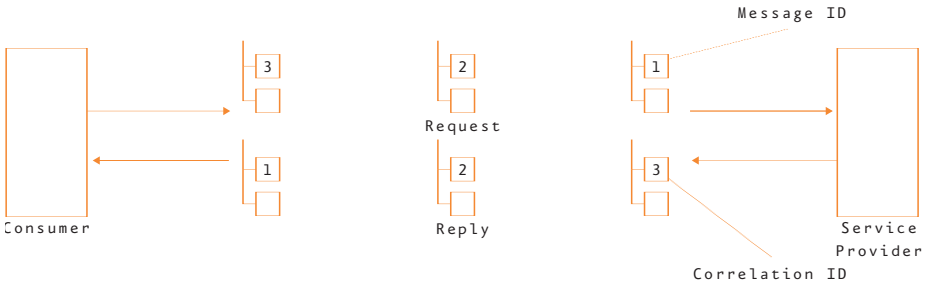
Message Construction
The message itself is simply some sort of data structure – such as a string, a byte array, a record, or an object. It can be interpreted simply as data, as the description of a command to be invoked on the receiver, or as the description of an event that occurred in the sender. When two applications wish to exchange a piece of data, they do so by wrapping it in a message. Message construction introduces the design issues to be considered after generating the message. In this message construction patterns catalogue we will present three important message construction patterns.

Correlation Identifier
Problem:
In any messaging system, a consumer might send several message requests to different service providers. As a result it receives several replies. There must be some mechanism to correlate the replies to the original request.

Solution:
Each reply message should contain a correlation identifier; a unique id that indicates which request message this reply is for. This correlation id is generated based on a unique id containing within the request message.

Interactions:
There are six parts to Correlation Identifier:
– ***Requestor*** – Consumer application.
– ***Replier*** – Service Provider. It receives the request ID and stores it as the correlation ID in the reply.
– ***Request*** – A Message sent from the consumer to the Service Provider containing a request ID.

Figure 25: Correlation Identifier



– ***Reply*** – A Message sent from the Service Provider to the Consumer containing a correlation ID.
– ***Request ID*** – A token in the request that uniquely identifies the request.
– ***Correlation ID*** – A token in the reply that has the same value as the request ID in the request.

Working Mechanism:
During the creation time, a request message is assigned with a request ID. When the service provider processes the request, it saves the request ID and adds that ID to the reply as a correlation ID. Therefore it helps to identify request-reply matching. A correlation ID (and also the request ID) is usually associated with the message header of a message rather than the body and can be treated as a metadata of the message.

Message Sequence

Problem:
Because of the inherent distributed nature of messaging, communication generally occurs over a network. You must utilize appropriate network bandwidth, maintaining best performance. In certain scenarios (such as sending a list of invoices for a particular customer) you might need to send large amounts of data (100 MB or more). In such cases it is recommended to divide the data into smaller chunks, and send the data as a set of messages. The problem is, how to rearrange the data chunks to form the whole set.

Solution:
Use a message sequence, and mark each message with sequence identification fields.

Interaction:
The three *message sequence* identification fields are:
– ***Sequence ID*** – Used to differentiate one sequence from other.
– ***Position ID*** – A relative unique ID to identify a message position within a particular sequence.
– ***End of Sequence indicator*** – Used to indicate the end of a sequence.

The sequences are typically designed such that each message in a sequence indicates the total size of the sequence; that is, the number of messages in the sequence (see *Figure 26*).

As an alternative, you can design the sequences such that each message indicates whether it is the last message in that sequence (see *Figure 27*).

Let's take a real life example. Suppose we want to generate a report for all invoices from 01/01/2001 to 31/12/2003. This might return millions of records. To handle this scenario, divide the timeframe into quarters and return data for each quarter. The sender sends the quarterly data as messages, and the receiver uses the sequence number to reassemble the data and identifies the completion of received data based on End of Sequence indicator.

Message Expiration

Problem:
Messages are stored on disk or persistent media. With the growing number of messages, disk space is consumed. At the end of messaging life cycle, messages should be expired and destroyed to reclaim disk space.

Figure 26: Message Sequence indicating size

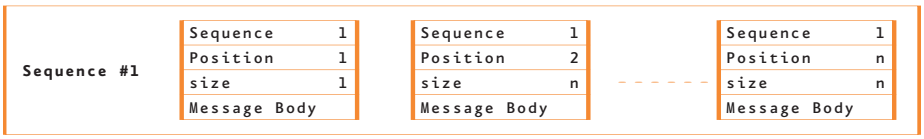
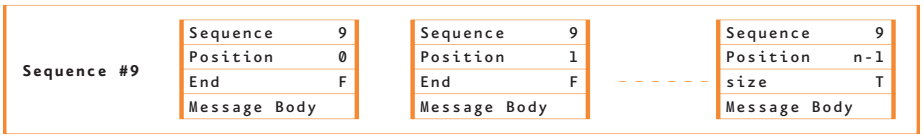
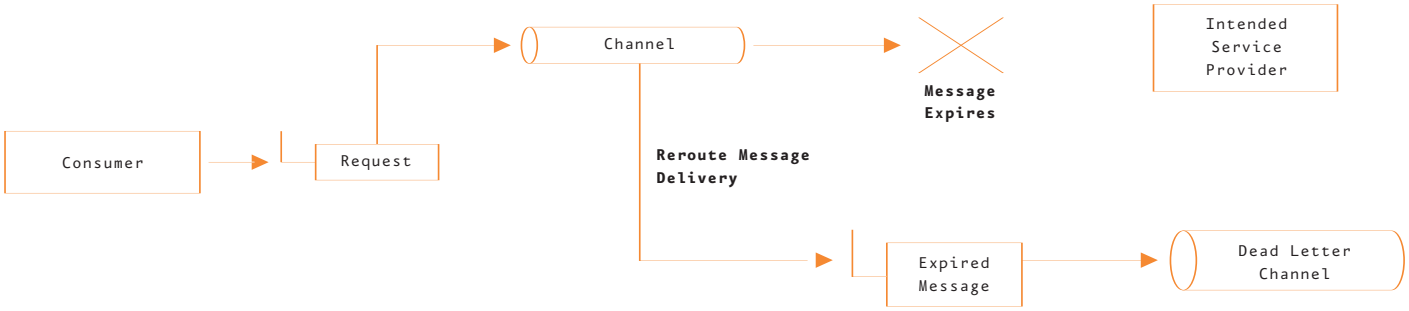


Figure 27: Message Sequence with message end indicator



"Message construction introduces the design issues to be considered after generating the message. Sender and receiver disagreements on the format of the message are reconciled by message transformation."

Figure 28: Message Expiration



Solution:
Set the *message expiration* to specify a time limit for preservation of messages on persisting media.

Interaction:
A *message expiration* is a timestamp (date and time) that decides lifetime of the message.

When a message expires, the messaging system might simply discard it or move it to a dead letter channel.

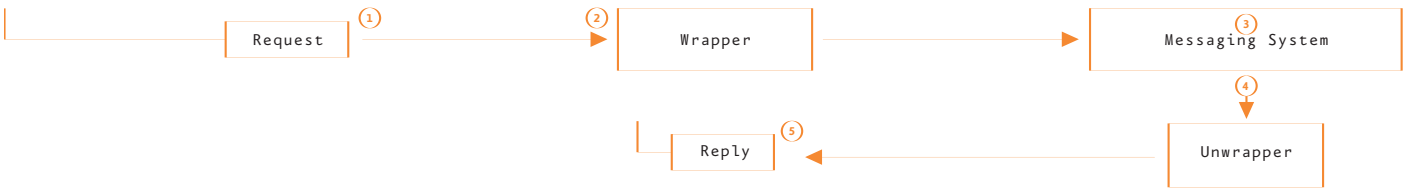
Message Transformation
Various applications might not agree on the format for the same conceptual data; the sender formats the message one way, but the receiver expects it to be formatted another way. To reconcile this, the message must go through an intermediate conversion procedure that converts the message from one format to another. Message transformation might involve data change (data addition, data removal, or temporary data removal) in existing nodes by implementing business rules. Sometimes it might enrich an empty node as well. Here we present few important message transformation patterns.

Envelope Wrapper
Problem:
When one message format is encapsulated inside another, the system might not be able to access node data. Most messaging systems allow components (for example, a content-based router) to access only data fields that are part of the defined message header. If one message is packaged into a data field inside another message, the component might not be able to use the fields to perform routing or business rule based transformation. Therefore, some data fields might have to be elevated from the original message into the message header of the new message format.

Solutions:
Use an *envelope wrapper* to wrap data inside an envelope that is compliant with the messaging infrastructure. Unwrap the message when it arrives at the destination.

Interactions:
The process of wrapping and unwrapping a message consists of five steps:
1. The message source publishes a message dependent on raw format.
2. The wrapper takes the raw message and transforms it into a message format that complies with the messaging system. This may include adding message header fields, encrypting the message, adding security credentials etc.
3. The messaging system processes the compliant messages.

Figure 29: Envelope Wrapper



4. A resulting message is delivered to the unwrapper. The unwrapper reverses any modifications the wrapper made. This may include removing header fields, decrypting the message or verifying security credentials.
5. The message consumer receives a 'clear text' message.

An envelope typically wraps both the message header and the message body or payload. We can think of the header as being the information on the outside of the envelope – it is used by the messaging system to route and track the message. The contents of the envelope are the payload, or body – the messaging infrastructure does not care about it until it arrives at the destination.

Content Enricher

Problem:

Let's consider the example. An online loan processing system receives information including a customer credit card number and an SSN. In order to complete the approval process, it needs to perform a complete credit history check. However, this loan processing system doesn't have the credit history data. How do we communicate with another system if the message originator does not have all the required data fields available?

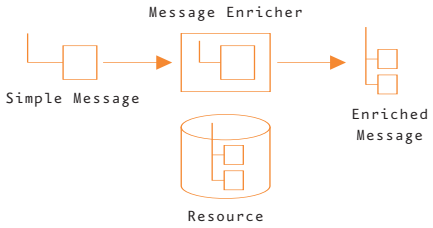
Solution:

Use a specialized transformer, a *content enricher,* to access an external data source in order to enrich a message with missing information.

Interactions:

The *content enricher* uses embedded information inside the incoming message to retrieve data from an external source. After the successful retrieval of the required data from the resource, it appends the data to the message.

Figure 30: Content Enricher



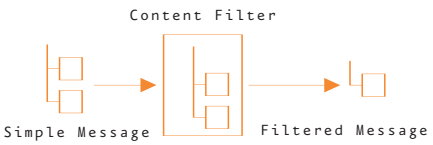
The *content enricher* is used in many occasions to resolve reference IDs contained in a message. In order to keep messages small, manageable, and easy to transport, very often we just pass simple object references or keys rather than passing a complete object with all data elements. The *content enricher* retrieves the required data based on the object references included in the original message.

Content Filter

Problem:

The *content enricher* helps us in situations where a message receiver requires more (or different) data elements than are contained in the original message. There are surprisingly many situations where the reverse is desired; the removal data elements from a message. The reason behind data removal from the original message is to simplify message handling, remove sensitive security data, and to reduce network traffic. Therefore, we need to simplify the incoming documents to include only the elements we are actually interested.

Figure 31: Content Filter



Solution:

Use a *content filter* to remove unimportant data items from a message.

Interactions:

The *content filter* not only removes data elements but also simplify the message structure. Many messages originating from external systems or packaged services contain multi-levels of nested, repeating groups because they are modeled after generic, normalized database structures. The content filter flattens this complex nested message hierarchy. Multiple *content filters* can be used as a to break one complex message into individual messages that each deal with a certain aspect of the large message.

Claim Check

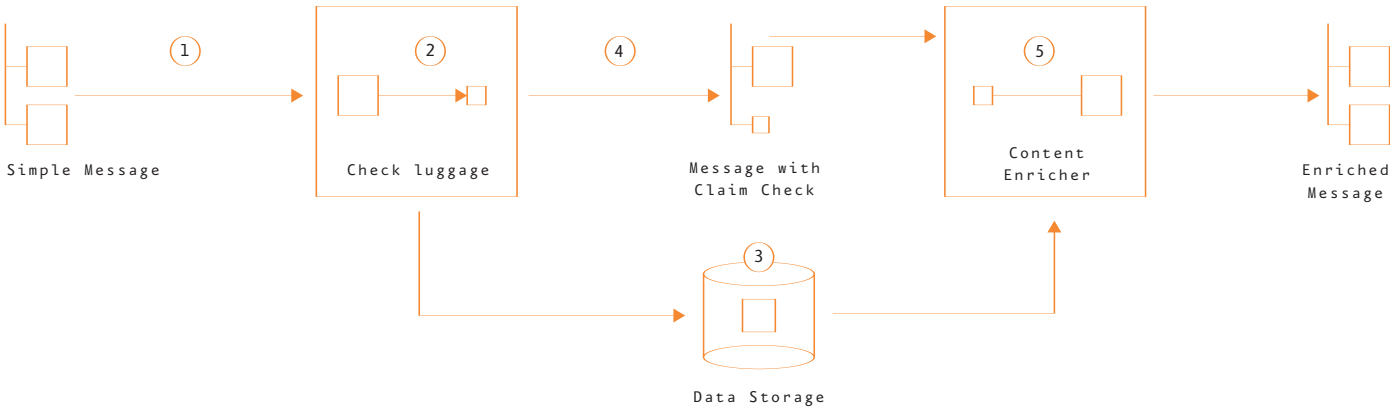
Problem:

A *content enricher* enriches message data and a *content filter* removes unneeded data items from a message. Sometimes however, the scenario might be little different. Moving large amounts of data via messages might be inefficient due to network limitation or hard limits of message size, so we might need to temporarily remove fields for specific processing steps where they are not required, and add them back into the message at a later point.

Solution:

Store message data in a persistent store and pass a *claim check* to subsequent components. These components can use the *claim check*

Figure 32: Claim Check



to retrieve the stored information using a content enricher.

Interactions:
The *Claim Check* pattern consists of the following five steps:

1. A message with data arrives.
2. The 'check luggage' component generates a unique key that is used in later stage as the claim check
3. The check luggage component extracts the data based on a unique key from the persistent store.
4. It removes the persisted data from the message and adds the *claim check*.
5. The checked data is retrieved by using a content enricher to retrieve the data based on the *claim check*.

This process is analogous to a luggage check at the airport. If you do not want to carry your luggage with you, you simply check it with the airline counter. In return you receive a sticker on your ticket that has a reference number that uniquely identifies each piece of luggage you checked. Once you reach your final destination, you can retrieve your luggage.

Conclusion
SOA stresses interoperability, the ability to communicate different platforms and languages with each other. Today's enterprise needs a technology-neutral fabricated solution to orchestrate the business processes across the verticals. The SOA, then, presents a shift from the traditional paradigm of enterprise application integration (EAI) where automation of a business process required specific connectivity between applications. According to Robert Shimp, vice president of Technology Marketing at Oracle:

"EAI requires specific knowledge of what each application provided ahead of time. SOA views each application as a service provider and enables dynamic introspection of services via a common service directory, Universal Description Discovery and Integration of Web services (UDDI)." [10]

Messaging is the backbone of SOA. Steven Cheah, director of Software Engineering and Architecture at Microsoft Singapore, states:

"We now finally have a standard vehicle for achieving SOA. We can now define the message standards for SOA using these Web services standards."[10]

Cheah considers SOA 'a refinement of EAI'. Specifically, SOA recommends some principles, which actually help achieve better application integration. These principles include the description of services by the business functions they perform; the presentation of services as loosely-coupled functions with the details of their inner workings not visible to parties who wish to use them; the use of messages as the only way 'in' or 'out' of the services; and federated control of the SOA across organizational domains, with no single party having total control of it.

We started at the ten thousand foot level with a vision of service-oriented enterprise. We then descended down through a common architecture (SOA) and proceeded by outlining messaging. Now, we are armed with the necessary messaging patterns valuable to attack the SOA complexities and to achieve the vision of dynamic process oriented service bus enterprise.

"SOA is 'a refinement of EAI'. Specifically, SOA recommends some principles, which actually help achieve better application integration."

References

1. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions,* Gregor Hohpe and Bobby Woolf, Addison-Wesley, 2004
2. *Service Oriented architecture: A Primer,* Michael S Pallos, EAI Journal, December 2001
3. *Solving Information Integration Challenges in a Service-Oriented Enterprise,* ZapThink Whitepaper, http://www.zapthink.com
4. *SOA and EAI,* De Gamma Website, http://www.2gamma.com/en/produit/soa/eai.asp
5. *Introduction to Service-Oriented Programming,* Guy Bieber and Jeff Carpenter, Project Openwings, Motorola ISD, 2002
6. *Java Web Services Architecture,* James McGovern, Sameer Tyagi, Michael Stevens, and Sunil Mathew, Morgan Kaufman Press, 2003
7. *Using Service-Oriented Architecture and Component-Based Development to Build Web Service Applications,* Alan Brown, Simon Johnston, and Kevin Kelly, IBM, June 2003
8. *The Modular Structure of Complex Systems,* Parnas D and Clements P, IEEE Journal, 1984
9. *Design Patterns: Elements of Reusable Object-Oriented Software,* Gamma E, Helm R, Johnson R, and Vlissides J, Addison-Wesley, 1994
10. *Computerworld Year-End Special: 2004 Unplugged,* Vol. 10, Issue No. 10, 15 December 2003 - 6 January 2004, http://www.computerworld.com.sg/pcwsg.nsf/currentfp/fp
11. *Applying UML and Patterns – An introduction to OOA/D and the Unified Process,* Craig Larman, 2001

Soumen Chatterjee,
schatterjee@ieee.org

Soumen is a Microsoft Certified Professional and Sun Certified Enterprise Architect. He's significantly involved in enterprise application integration and distributed object oriented system development using Java/J2EE technology to serve global giants in the finance and health care industries. With expertise in EAI design patterns, messaging patterns and testing strategies he designs and develops scalable, reusable, maintainable and performance tuned EAI architectures. Soumen is an admirer of extreme programming methodology and has primary interests in AOP and EAI. Besides software Soumen likes movies, music and follows mind power technologies.

JOURNAL3