Case Study Architecture Migration: Transition from Mainframe Batch Processing to Web Services

Spencer Rugaber

College of Computing

Georgia Institute of Technology

June 20, 2007

Outline

- Introduction
  - Problem statement
  - Motivation
  - Existing system
  - Web services
  - Running example
- Methodology
  - Requirements analysis
  - Domain analysis and model generation
  - Use case analysis
  - Services definition/identification
  - High-level design
  - Low-level analysis and design via - enterprise patterns
  - Validation of end result
- Lessons Learned

Problem Statement

- Status: Existing mainframe legacy applications
- Target: Web services
- Constraints
  - Incremental migration
  - Incomplete knowledge of application

Motivations for Migration

- Reduced platform and language dependence
- Improved accessibility
- Enhanced long-term flexibility and maintainability
- Improve customer-orientation
- Enable user interface enhancements

Mainframe Legacy Applications

- Cobol
- Batch processing
- Synchronous communications
- DBMS, proprietary and VSAM files
- Complexity
  - Multiple Cobol programs in separate files
  - Multiple data files
  - A job consists of a subset of the former accessing a subset of the latter

Web Services

- Internet access
- Standard data formats
  - XML, SOAP, HTML
- Structured around *services*
  - Described via WSDL
- Indexing (yellow pages) support
  - UDDI
- Service-oriented architecture

Situation

- Overall system
  - Payment reconciliation by customer service representatives (CSRs) comparing actual payments to invoiced amounts
- 90KLOC Cobol program in 64 files
- Interaction via CICS displays on PCs running 3270 terminal emulators
- Numerous VSAM and "flat" text files

Methodology

- After-the-fact organization of the steps we took to migrate two web services
  - Display of status codes
  - Current account display and update

Migration Project Constraints

- Incremental, non-disruptive migration
- J2EE (JEE) infrastructure running on IBM Websphere
- Partial documentation and test data
- No access to users (CSRs)
- No live execution; stand-alone prototyping only

Requirements Analysis

- Incremental (service at a time) approach meant we were not able to understand the entire system before beginning
- No access to CSRs meant we had to infer usability requirements
- Sources of requirements
  - CICS screens
  - Use cases
  - Source code
  - Documentation
- Existing system functionality acted as the ultimate source of requirements

Domain Analysis

- *Domain* is an application area (set of related applications)
- *Domain analysis* is system analysis on a set of related applications
- *Domain model* describes the vocabulary and relationship and architecture typical of applications in the domain

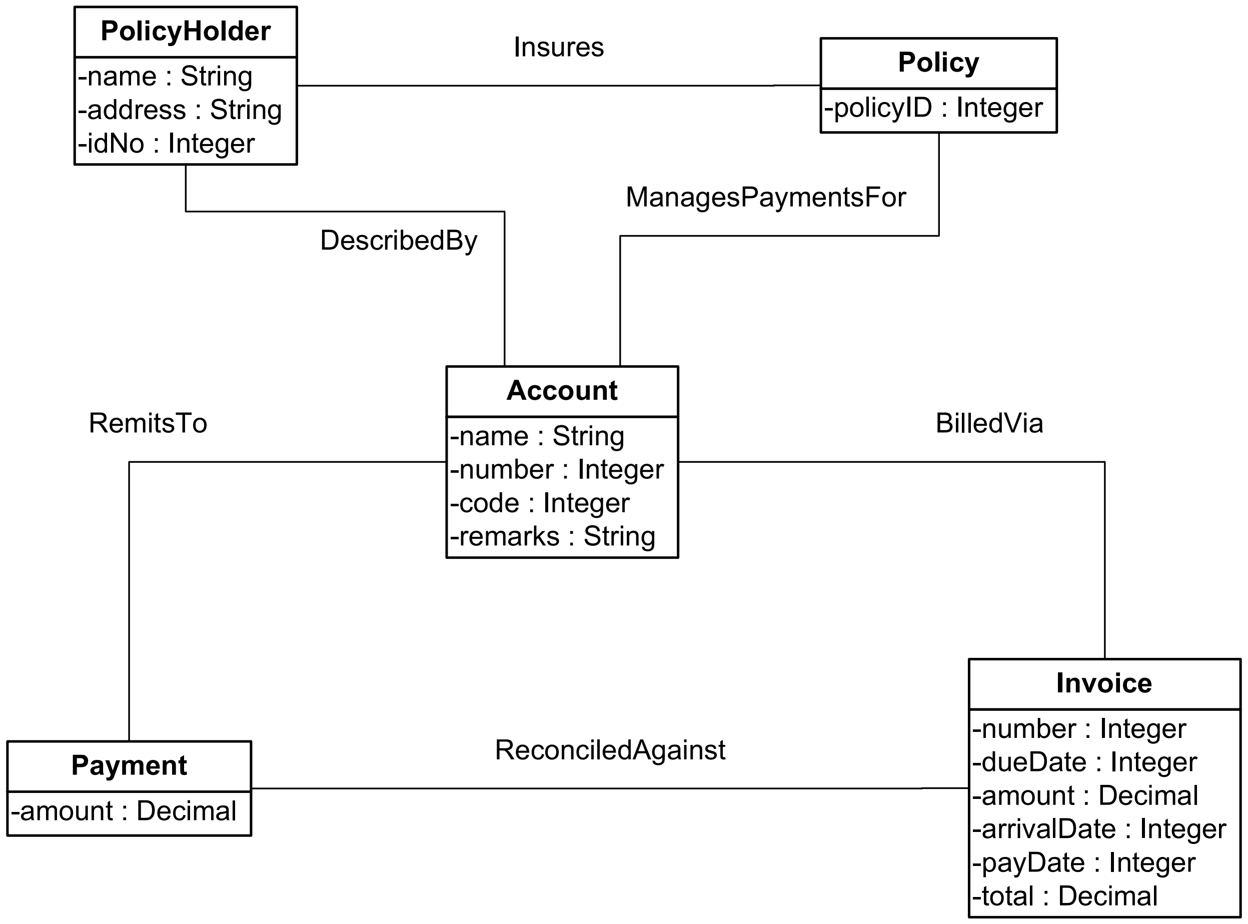
Domain Analysis - 2

- We used a glossary plus a UML class diagram to express our domain model
- We added to these documents as we learned more about the application

UML Class Diagrams

- We used a subset of the UML notation to capture the structural aspects of our domain model
  - Rectangles denote classes/concepts/actors/ entities
  - These may contain responsibilities/services/ operations
  - These may contain typed attributes
  - Lines between classes denote relationships (specialization/association/dependency)
- We intentionally did not use other UML features

Example Domain Model



Glossary

- **Billing System:** Generates invoices (batch process) for the payment of premiums
- **Group:** A payroll account; generally an employer with multiple employees insured under multiple policies. Thus, a group is associated with a group of policies, and is identified by a group number and name, with an employer address and other information.
- **Invoice:** A request, submitted to an account/group, for payment for premiums due under policies of the account. Contains a record for each policy number for which an amount (the premium) is being billed. The record specifies the amount billed, amount received, insured name, etc.
- **Invoice control data:** Includes header and summary data; how many policies on the invoice; amount due
- **Policy:** Each insurance policy insures a person against some risk. Policy record has an id number, employee number, insured name, etc.
- **Premiums:** Amount billed (on an invoice) under a policy; deducted from employees' paychecks and remitted by their employer. An employee may have several policies, and thus may owe several premiums on an invoice.
- **Reconciliation:** Comparison of the submitted amounts with the invoiced amounts.

Use Case Analysis

- A use case is a narrative description of the actual use of a system
  - Typically includes actors, actions and objects
  - May be structured or unstructured
- Enables the detection of dependencies and missing elements
- Provides a strong indicator of the existence of a service

Example Unstructured Use Case

- **Scan_Invoices** – George, a customer service representative, gets to work at 8am, sits down at his terminal and enters his desk code. His first job for the day is to begin working on a related group (0CP57) of customers, notorious for their unpaid/late/underpaid premiums. He invokes the Scan_Invoices service, specifying group 0CP57, and is presented with a list of invoices for that group. (Invoices over three months old and paid as billed are coded 00 “Out of Date,” and are not accessible.) There is a line of information on his screen for each of the first twelve invoices; he can scroll and look at the other invoices. The invoices are presented in chronological order, most recent first, and each line includes various pieces of information such as the total amount billed and total amount received on the invoice, and a status code. George stares at the screen a moment, then goes for his morning cup of coffee.

Example Structured Use Case

Use Case UC1: InquireAboutInvoice

Primary Actor: George, a customer service representative

Preconditions: None.

Postconditions: The invoice information is found and displayed

Main Success Scenario:

1. George requests an invoice enquiry for a group number.
2. The prototype displays the following information: Basic group information; Billing information
3. George selects an invoice by specifying a line number or invoice number
4. Service displays invoice information to George

Extensions:

a. If the service fails, the service issues a warning to George and exits.

Service Definition

- The single most important activity is deciding on exactly what the services are
  - They may already exist as modular units in the legacy system
  - They may be new
  - They may be determined by the underlying data model
  - They may have to be composed from various parts of the existing system

What is a Service?

- A *service* is a modular application with a defined and self-documented use protocol (interface) comprising one or more operations
- The service is published and its operations are dynamically invoked across the network, generally via XML messages sent over the SOAP protocol
  - XML stands for eXtendable Markup Language and is a cross-platform, extensible, text-based standard for representing data
  - SOAP is Simple Object Access Protocol and is an XML-based protocol that follows the HTTP request-and-response model

Four Tenets of SOA

(www.bpminstitute.org/articles/article/article/the-four-tenets-of-service-orientation.html)

- **Boundaries are explicit.** This means there is no ambiguity about whether the code or data resides inside or outside of the service
- **Services are autonomous.** This means each service has its own implementation, deployment, and operational environment. There is no presiding authority within a service-oriented environment. Services are autonomous in that they are isolated and decoupled; they are designed and deployed independently of one another and may only communicate using contract-driven messages and policies
- **Services expose to the world schema and contract, but do not expose to the world implementation.** Schema describes the format and the content of the messages, while contracts describes message sequences allowed in and out of the service
- **Service compatibility is based on a policy.** Formal criteria exist for getting making use of a service. The criteria are located in a document that outlines the rules for using the service

Service Design Principles

(msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/dataoutsideinside.asp)

- Service interfaces should accept a well-defined input message and respond with an output message
- Internal implementation details should not be leaked outside of a service boundary
- Contracts should be designed with the assumption that once published, they cannot be modified
- Isolate services from failure. From a consumer perspective, plan for unreliable levels of service availability and performance. From a provider perspective, expect misuse of your service

Service Design Principles

(msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/dataoutsideinside.asp)

- Contracts should be as explicit as possible to minimize misinterpretation. Additionally, contracts should be designed to accommodate future versioning of the service via the extensibility of both the XML syntax and the SOAP processing model
- A service’s internal data format should be hidden from consumers while its public data schema should be immutable
- Version services only when changes to the service’s contract are unavoidable

Issue

- One motivation for determining what is a service is indexing. That is, you want to organize services in such a way that potential customers can find them (using UDDI). These tend to be coarse-grained or *business* services
- Another motivation for organizing services is in units that are meaningful to the end user. These tend to be fine grained

Example Services and Operations

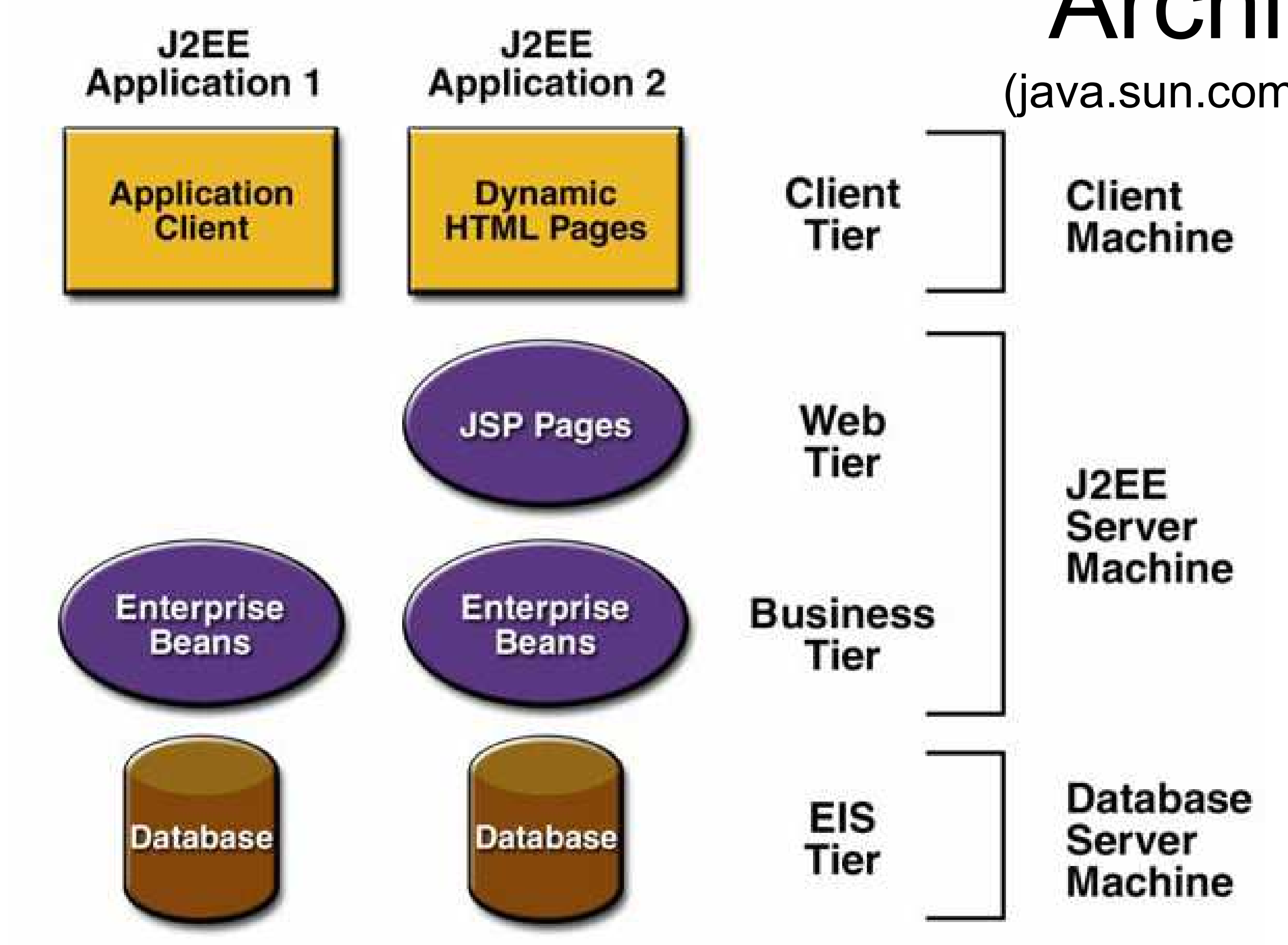
- Account (Group) Administration
  - CreateAccount, GetDetails, ModifyDetails
- Invoice Activity Helper
  - RecordInvoiceActivity, GetInvoiceActivity, GetInvoiceActivityByPolicyNumber, DeleteReconciliationActivity
- Invoicing
  - CreateInvoice, DeleteInvoice, ModifyInvoice, GetInvoicesForAccount, UpdateEmployeeNumberForInvoice, PolicyLine, UpdateAmountReceivedForInvoice, PolicyLine, UpdateMonthsForInvoicePolicyLine, PayInvoice

High-level Design – Hybrid Three+ Layered Architecture

- Legacy system architectural style
  - Batch processing; master-file update
- Traditional three-tier distributed system architecture
  - UI, business logic, database
- Modified three tier architecture
  - Break business logic into domain independent and domain-specific parts
  - Use OO style for business logic
  - Use Object-Relational Mapping (ORM) interaction with database

Typical JEE Architecture

(java.sun.com/j2ee/1.4/docs/tutorial/doc)



Advantages

- Layers (tiers) are loosely coupled, making it possible to divide up development work among available workers
- It is easier to understand the system and to locate desired functionality
- It is easier to modify the system, and modifications are less likely to affect other layers
- *Service orientation* achieves loose coupling among interacting software agents, with a defined, stable interface
  - For example, one module might be implemented in .NET, another in Java, and they can still talk
- Services encapsulate common interactions with applications, avoiding code duplication.

The Fourth Tier

- We have chosen to partition the business logic layer into two pieces: application logic and domain logic
- Application logic, sometimes called workflow logic, involves application-specific responsibilities such as a requirement to notify some other application about an event
- Domain logic relates purely to the problem domain, for example, a set of business strategies for calculating revenue recognition
- It is helpful to keep domain logic and application logic separate because there may be multiple applications with different workflow requirements that need to access and use the same domain logic
- Lower-level design patterns facilitate this separation

Enterprise Design Patterns

- To facilitate movement from high-level to low-level design, we made use of enterprise design patterns
- A *design pattern* is a solution to a problem in a context. Typically, an object-oriented design pattern guides you through the process of solving a common problem, such as how to visit all of the nodes in a complex data structure
- A multi-tiered architecture presents an opportunity to make us of enterprise design patterns. For example, business (domain) logic can be implemented via the object-oriented ("OO") domain model design pattern

Enterprise Design Patterns - 2

- Services are defined and exposed so disparate applications can integrate the functionality they need, from available services. The services are facilitated via the layered architecture that provides a service façade at the "top" of the business tier. The domain business logic itself is implemented via interactions between fine-grained domain-entity objects
- A helpful rule-of-thumb we followed is to *design* object-oriented, and *integrate* service-oriented

Issue: Distribution

- Designers may be tempted to distribute objects on web and business layers between two machines. We recommend against this architectural alternative
- Although JEE makes it possible to put the web tier on one machine and the business tier on another, this will result in slower response times due to expensive network communications between those layers

Distribution - 2

- It is better to scale up by clustering server instances instead of distributing tiers, so you can use local, not remote, interfaces between web and business layers
- Then, service requests can be directed to different computers in the cluster, as dictated by load balancing
- Use the service layer pattern with operation script approach, delegating to POJO (plain old Java objects) domain objects persisted by an ORM tool

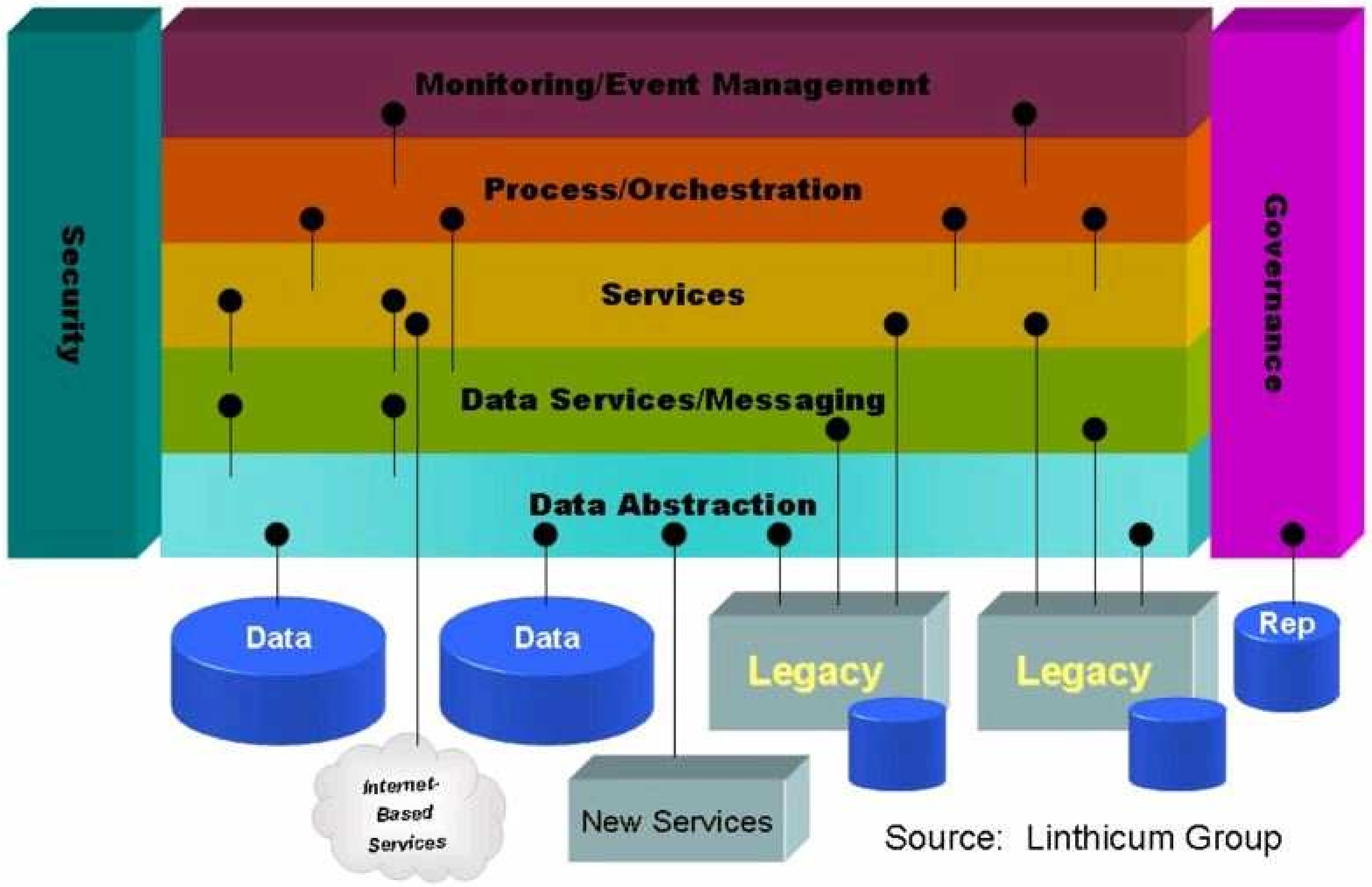
Distribution - 3

- By running all in one process, we also may avoid the necessity for tedious-to-construct value objects to conserve network bandwidth
- However, in an enterprise-wide service-oriented architecture, avoiding remote accesses cannot always be done without duplicating code, because needed services may already exist as web services exposed in disparate (remote) systems
- In this case it is probably best to use the existing services, paying the network connection cost and taking this cost into account in the design.

Service Oriented Architectures (SOA)

- Buzz word - many different definitions
- Common themes
  - Loosely coupled services
  - Platform independence
    - REST, RPC, DCOM, CORBA, Web Services, .NET
  - Well-defined interfaces; information hiding

SOA Meta-Model



Source: Linthicum Group

SOA Guiding Principles

(Wikipedia)

- Reuse, granularity, modularity, composability, componentization, and interoperability
- Compliance to standards
- Services identification and categorization, provisioning and delivery, and monitoring and tracking

Low-Level Analysis And Design Via Enterprise Patterns

- Outside in (UI, database, business logic)
  - Business logic (BL) is most complex, so leave it for last
- Strict recreation of screen appearance
  - UI enhancements allowed only after verification of identical behavior

UI Extraction

- Start with CICS screen
- Break out regions
- Categorize

Data	Computed business data
Label	Heading
Formatting	Boundary characters
Ganging	Related fields in the same record
Type-in	User input
Function keys	Access to other functions

- Find corresponding code units

Data Modeling

- Use identified code regions to determine data sources
- Backward dataflow analysis
- Incremental construction of logical data model
  - Also use documentation, file structure, etc.
- Candidate target objects

Business Rule Extraction

- Only study source code to answer specific questions
  - Reduces reverse engineering costs
- But, once a section is studied, identify it as to role (UI, DB, BL)
- Construct service flow diagram
  - Slice of original program related to screen population

BL Extraction Details

- Find reads/writes from/to the GUI
- Look for involved “if” statements, taking different actions based on data input from the GUI or read from the database
- Look for validation checks on data entered via the GUI
- Look for write/update to the database or data files
- Three forms of textual/spreadsheet representations (views) of business rules are
  - Procedure oriented: section-by-section documentation of Cobol code
  - Identifier oriented: name, description, expression as a business rule, where used
  - Controls oriented: program function key description

Low-Level Design Implications

- We used CMP (Container Managed Persistence) EJB Entity Beans initially
- We moved to POJO's and an ORM tool (Hibernate) as the business layer gets more complex
  – Less conceptual overhead
  – Closer mapping to domain and DB concepts

ECÓLE

- Light-weight static analyzer to visualizing software dependencies on system resources for a given vertical slice of interest
  - Program, data file, program function key
- *ECÓLE* stands for Enterprise COBOL Ligature Explorer
  - A ligature occurs where two or more letterforms are written as a unit
- Transitive closure on software resources (programs, data tables, and copybooks), a ligature symbolizes the graph node or the shared component for a particular software resource
- Differentiates program calls into three types
  - Direct program call
  - Call through an upper level navigator (manager of control flow)
  - Call via program function key as pressed by the user

ECÓLE - 2

- Once the source code has been analyzed, the extracted software dependencies are stored in a PostgresSQL database
- The visualization phase involves the tool console querying the Knowledge Base and writing out the representative GraphViz graph file
- A free third-party visualizer (ZGRViewer) is then invoked by the tool console to display the resultant software dependency graph

Enterprise Patterns Typical Scenario

- Creation of database tables
- Generation of entity and data access beans
- Addition of a session bean with some simple business logic
- Use of JEE web service wizard to create a web service client for the session bean

Patterns Used
Front Controller

- Front Controller (MVC variant), Service Layer, Domain Model, Business Delegate, Mapper and Data Mapper
- Model-View-Controller pattern involves both web and business layers
  - Separate the business logic, web layer, and the acceptance of inputs into the web layer
  - Front Controller uses one controller that handles all requests for a web site

Service Layer Pattern

- *D*efine an application’s boundary with a layer of services that establishes a set of available operations and coordinates the application’s response in each operation
- The recommended approach is to put the *application/workflow logic* into relatively thick Service Layer classes (e.g., stateless EJB session beans) that delegate to domain object classes for *domain logic*
- Putting application logic into pure domain object classes can make those classes less reusable across applications and can make it harder to re-implement the application logic in a workflow tool

Domain Model Pattern

- Use an object model of the domain, where objects correspond to possibly generalized entities in the domain, and incorporate both behavior and data
  - The Mapper pattern handles mapping the object's data to a row in a database table.
- Result is a web of interconnected, highly coherent, loosely coupled objects where business behavior is distributed

Business Delegate Pattern

- Provides a class in the web tier that duplicates methods to be invoked on some session bean in the business layer
- There is thus a one-to-one correspondence between web layer methods and business layer methods; a web layer method delegates to the corresponding business layer method
- This uncouples the layers; you can stub in some code in the delegate object in order to test the web layer before the business layer functionality exists

Mapper and Data Mapper

- A *Mapper* is an object that sets up communication between two independent objects (in our case, an object in the business layer and a table in the database)
  - Objects that the mapper separates are not aware of each other or the mapper
- A *Data Mapper*, implemented via an ORM tool, is a layer of Mappers that moves data between objects and a database
- A rich business layer implemented using the domain model pattern differs significantly from the (non-object-oriented) relational database design
  - It involves inheritance, collections, strategies and other patterns, and many small interconnected objects; therefore, it is hard to map to the database
  - ORM tools allow for a natural OO programming model for the business layer, with inheritance, polymorphism, composition, and collections
  - ORM tools support ultra-fine-grained object models, a rich variety of mappings, high scalability, and object oriented query languages similar to conventional SQL but including additional object oriented query functionality

Value Object

- Also called a data transfer object encapsulates attributes from an entity bean in a class so they can be passed around and used
- Useful when the information is being passed over an expensive network connection
- If the connection is local, use of coarse-grained value objects is generally not necessary or advised
- It is more clear and flexible to pass fine-grained objects corresponding to domain entities

Service Locator

- Makes web layer lookup of business layer services more efficient
- When you create an interface to a service, save the reference object in a map object, associating it with the name of the lower level service
- On subsequent accesses, instead of re-creating the local home interface, look up the service in the map and re-use it

Validation

- Use cases
  - Elicit feedback from customer
  - Use as a source of acceptance tests
- Code analysis
  - Have all sections of the code been assigned a role?
- Have all screen elements been accounted for?
- Have all data accesses been accounted for?
- Does the derived data model correspond to that indicated by existing files and database schema
- Verify extracted business logic with customer
- Bit-for-bit compatible with existing system on test cases

Lessons Learned

- High cost of upfront reverse engineering can be amortized
- Web services technology is intricate requiring significant effort to learn
  - Support tools, such as WASD, are necessary
- JCA is promising technology for incremental integration

Resources

- H. M. Hess. “Aligning Technology and Business: Applying Patterns for Legacy Transformation.” *IBM Systems Journal,* 44(1), 2005.
- William M. Ulrich. *Legacy Systems Transformation Strategies.* Prentice Hall, 2002.
- Martin Fowler. *Patterns of Enterprise Application Architecture.* Addison-Wesley, 2003.
- Olaf Zimmerman, Mark Tomlinson and Stefan Peuser. *Perspectives on Web Services.* Springer, 2003.

Resources - 2

- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.
- Michael C. Feathers. *Working Effectively with Legacy Code.* Prentice Hall, 2005.
- Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development.* Third Edition, Prentice Hall, 2005.