Aligning technology and business: Applying patterns for legacy transformation

H. M. Hess

Two key goals for aligning technology and business are to increase an organization's ability to change rapidly and to reduce the costs of technology. While many efforts are underway to improve application development, less emphasis has been placed on addressing key challenges posed by existing applications that resist rapid change. In this paper, we discuss techniques for accelerating change to legacy systems and for streamlining an application portfolio. Our approach takes business-driven application requirements, links them to analysis of an application portfolio, and identifies potential sequences of transformations to realize the targeted improvements. This paper describes our approach for mapping business requirements to application software, for using patterns to help translate business requirements to software requirements, and for using patterns to translate software requirements into potential solution designs. The paper describes how these techniques are applied to two stages of the software life cycle—initial analysis and detailed analysis—and summarizes experience gained from projects working with IBM clients.

Businesses often depend on information systems that were built with traditional transactional and batch technologies. These information systems—commonly called legacy systems—were built to satisfy high demands for throughput and scale at a time when systems software and hardware were far less capable than they are today. These legacy systems are often very large and complex by any standard, and relatively closed and inflexible by the standards of today. The application portfolio of an enterprise typically contains many applications, often developed independently, operating in *silos* (isolated, often vertically integrated structures) with overlapping and redundant function and data. For example, an IBM Business Consulting Services report on financial institutions noted, "From our extensive studies of performance improvement and cost reduction, we estimate that as much as 60 percent to 80 percent of the functionality in silos may be redundant or duplicated in other parts of the business. This weakens the performance of financial

©Copyright 2005 by International Business Machines Corporation. Copying in printed form for private use is permitted without payment of royalty provided that (1) each reproduction is done without alteration and (2) the Journal reference and IBM copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free without further permission by computer-based and other information-service systems. Permission to republish any other portion of the paper must be obtained from the Editor. 0018-8670/05/$5.00 © 2005 IBM

institutions and makes it harder for them to respond to fresh demands for change."[1]

Over the past two decades, significant efforts have been made to improve interoperability of legacy systems by wrapping them first with client/server solutions, then with messaging, and now with Web technology. These changes have added capability and increased interoperability, but have come at a cost of increased application size and complexity. This was also described in the IBM Business Consulting Services report:

> Recent advances in integration middleware technology have provided some relief by making it possible for financial institutions to move customer information across channels. But in many cases the technology has been laid over flawed legacy architecture and has merely created more duplication. The cost of such an approach is reflected in our research, which shows that up to 70 percent of IT [information technology] spending by financial institutions goes toward maintenance and redevelopment. Furthermore, the problem seems to grow with size: The largest banking institutions (with assets more than $60bn) spend 50 percent to 100 percent more on IT relative to their smaller (under $2bn) counterparts. Despite this [expenditure], virtually every business manager in large financial institutions bemoans the organization's lack of speed and institutional inflexibility.[1]

The IBM Business Consulting Services report focuses on financial institutions, but similar results can be found in other industries. The inertia of legacy systems creates significant challenges that remain even when new layers of more flexible technologies have been added. Emerging approaches for creating applications—such as model-driven development and service-oriented architecture (SOA)—sidestep the challenges of legacy systems, treating the applications as black boxes to be reused through connectors and adapters.

There is an opportunity to make these emerging approaches more valuable by augmenting them with solutions to some of the deep-rooted problems of legacy systems. This paper describes research on analysis and transformation techniques to face directly the problems of duplication and inflexibility of legacy systems, and to integrate top-down approaches for creating applications with bottom-up analysis of existing applications. Our approach combines three techniques:

1. *Business-to-technology model mapping*—A *to-be* model of business (i.e., a model of the business as we wish it to be) is mapped to an *as-is* model of applications (i.e., a model of applications as they are today) to identify areas of redundancy and overlap, and to provide a basis for tools that can help derive technical requirements from business requirements.

2. *As-is pattern discovery*—A to-be model of software interfaces is compared with an as-is model of the composition and flow of legacy applications. The analysis identifies instances of structural *patterns* in the architecture of legacy applications, the programming, and the data representation that define the gap between the as-is implementation model and the to-be implementation model. (We discuss patterns in substantially greater detail in the section "The role of patterns for legacy transformations" later in this paper.)

3. *Transformation pattern selection*—A set of patterns for transforming legacy applications is compared with the model gaps and the as-is patterns to identify approaches and techniques for closing the gap. The transformation patterns provide solutions that allow the gap to be closed over several iterations. The transformation patterns include structural patterns for changes to application architecture, programming, and data representation, as well as process patterns for the transformation work itself.

The approach is founded on experience gained with IBM clients in projects that needed to analyze and change large portfolios of legacy applications. This paper describes some of the relevant characteristics of legacy applications and the motivation for moving beyond legacy integration to legacy transformation. It describes the motivation for mapping business models and technology models and the processes by which this mapping is performed at two stages in the life of a project: the initial analysis stage and the detailed analysis stage. The models used in detailed analysis of a project are then discussed (the design of a model is often referred to

as a metamodel). The paper concludes with two summaries of project experience: one for the initial analysis of a project to consolidate redundant applications, and one for the detailed analysis of integrating a batch legacy program with a service-based interface.

CHARACTERISTICS OF LEGACY APPLICATIONS

Two broad definitions of legacy applications are "anything that is running in production" or "anything that is not end-user computing." This could include batch and transactional applications running on z/OS* or OS/400*, C++ programs running on an open standards-compliant operating system, client/server applications based on personal computers, and others.

Our research is focusing on a significant subset of these broader definitions: batch and transactional applications running on z/OS. Even this constrained scope is quite large. For example:

- An enterprise's mainframe application portfolio can comprise tens of thousands of programs, tens of millions of data declarations, and more than 100 million lines of source code.
- The application portfolio may contain hundreds of business applications, which in turn may be architecturally and technically diverse.
- There are numerous interfaces between applications, many of which have been developed as needed, without a consistent underlying architecture.

These issues of portfolio-level scale and complexity have traditionally led to an approach that focuses on analysis of individual applications. Conducting analysis one application at a time has at least two significant shortcomings:

1. It does not enable analysis of the duplication and redundancy that exist for different applications—where there may be an overlap of 60 percent to 80 percent. Significant opportunities to consolidate and streamline may be missed.
2. It does not expose interfaces into the single application from other applications in the portfolio, increasing the risk of making a change that breaks one or more unseen interfaces.

Scale and complexity are issues even at the application level and program level. There are also systems and programs known to be brittle and error-prone, for which only the most critical changes are attempted.

MOVING FROM INTEGRATION TO TRANSFORMATION

The challenges and risks in making changes to legacy systems have played a large role in shaping current integration architectures and solutions in the marketplace. These solutions treat z/OS applications as black boxes to be integrated through Java** connectors, messaging solutions, and data access.[2]

Families of patterns for legacy integration have been developed to build needed capabilities on top of the connectors and message interfaces. For example, Reference 3 describes a technique to support requirements for routing a response from a legacy system to the appropriate end point. The legacy system returns a data structure to its caller; the integration solution is responsible for managing the routing to a Return Address:

> One difficulty in exposing systems as services results from the fact that many legacy systems were not built with features such as Return Address . . . in mind. Therefore we "wrap" access to the legacy system with a Smart Proxy. . . . This Smart Proxy . . . enhances the basic system service with additional capability so that it can participate in an SOA.[3]

SOA is an important element of both legacy integration and legacy transformation. For integration, SOA through messaging or Web services offers well-accepted tools and protocols for interoperability.[4] For transformation, SOA provides a formal interface to the legacy systems, separating the interface from the implementation of the legacy system or systems. After the interface is established, the legacy systems underneath the interface can be changed, consolidated, or replaced with significantly reduced impact on the other parts of the application portfolio.

TO-BE MODEL MAPPING

As with most information technology projects, legacy transformation projects begin with a goal and a general plan that are iteratively refined to detailed requirements, specifications, and plans. While some legacy transformation projects—such as data name standardization—have information technology is-

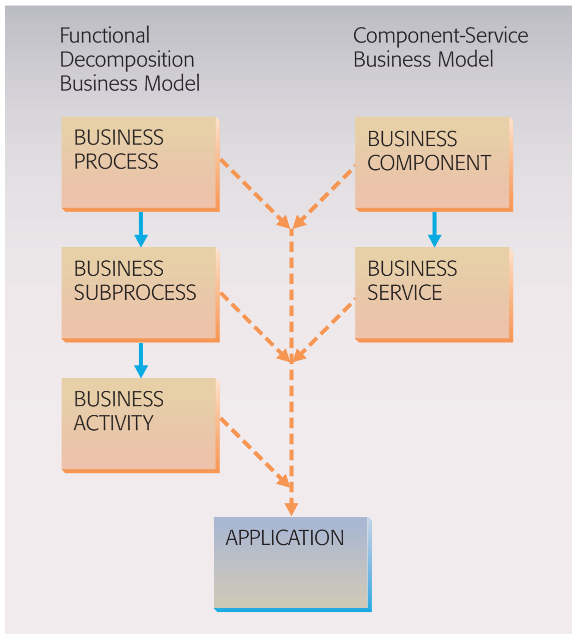

Figure 1
Business models mapped to applications

sues as their primary drivers, our work focuses on legacy transformation driven by business process change. In some cases, the project begins with a well-defined scope and business objective. In other cases, analysis of business strategy and the application portfolio are performed first, to define the scope and objectives of a set of related legacy transformation projects.

Understanding the relationships between the business processes and applications is a prerequisite to specifying and planning such a legacy transformation project. To record these relationships, we use two types of model: a model of the business and a model of the legacy applications. These models are used and refined throughout the life cycle of a project. Our approach starts with a broad, shallow model of the business and a high-level model of the legacy applications.

In the initial analysis we capture coarse-grained mappings—between high-level business processes and applications, for example—to provide the basis for identifying potential redundancies and overlaps in the application portfolio, estimating the size and complexity of the applications that support a business process, and identifying applications that perform common services.

In later iterations we capture finer-grained mappings, such as those between a business service and specific on-line transactions or batch jobs, to provide the basis for selecting integration and transformation strategies, identifying technical risks, and creating transformation plans. Our work addresses both the coarse- and fine-grained mappings.

Business modeling is itself a broad and complex subject. There are many types of business models, incorporating static and dynamic views of organizations, processes, and information. There are a number of tools for business process modeling, some of which link to tools for application development and infrastructures for business activity monitoring. There are also efforts, such as the Object Management Group's Model Driven Architecture**[5] initiative, working on the complex problems associated with comprehensive, end-to-end modeling and transformation and traceability between models.

At this point in our work, we have opted to use simple models that provide a static view of business processes. Two variations of static business process models have proven useful to our work:

1. *Functional decomposition*—In the long-accepted form of static process model, we use three levels of decomposition. *Business processes*, the highest level of the model, are composed of *business subprocesses*, which in turn are composed of *business activities*.
2. *Component-service model*—This is a form used within the component business modeling (CBM) approach developed by IBM Business Consulting Services. In this variation, a *business component* provides *business services* that can be used by other business components.

An illustration of the two forms of business model is shown in ***Figure 1***. In many cases, an organization already has a documented functional decomposition that can be used to populate this model. In other cases, a good initial model can be prepared in a few weeks. Anecdotal experience from consulting engagements suggests that the coarse-grained model is likely to contain between 150 and 300 business activities.

Initial analysis and coarse-grained mapping

One key objective of the initial analysis is to identify the context and scope of a proposed change. For example, the initial analysis for a system consolidation project needs to expose the areas of redundancy and the interface dependencies that must be considered in a migration plan. We use coarse-grained models of business and legacy applications for this stage.

A coarse-grained model is well-suited to the initial mapping between the business and legacy systems models. The coarse-grained mappings are made between the business activities or subprocesses and the legacy applications. This mapping is typically performed in the first several weeks of a project. It is a broad, shallow analysis. This mapping is typically performed at this coarse-grained level for the entire enterprise and the entire application portfolio.

The mappings help us gain insight from two viewpoints:

1. *The business model point of view*—The mapping allows us to understand the degree to which business functions or business services depend on multiple applications or have redundant implementations.
2. *The application point of view*—The mapping allows us to understand the business functions or business services supported by each application.

The development of the business model is usually performed in parallel with an application portfolio analysis. The application portfolio analysis uses manual techniques such as interviews, questionnaires, and workshops to document key attributes of the applications. Those attributes include measures of size, technologies used, and interfaces with other applications and external entities. The portfolio analysis may also estimate each application's development and operational costs, business value, and technical quality.[6] The application portfolio analysis and mapping results are typically manually entered into a spreadsheet, database, or other tool.

Initial application portfolio analysis projects typically avoid the use of automated tools for scanning source code and other application artifacts. This is true for two reasons. First, significant time and effort are required to scan large numbers of application artifacts. Second, even if the artifacts are scanned, it is difficult to sift through the detailed meta-data collected by automated tools to find information that is relevant to the early decision-making and estimating processes. Our research is exploring ways to make detailed application meta-data useful in the early stages of a project. Some examples of the ways in which the detailed meta-data could improve the initial analysis are:

- Detection and categorization of interfaces between applications to validate and extend the list of interfaces gathered in interviews and workshops
- Identification of technical dependencies, such as use of specific application programming interfaces or obsolete language versions or features
- Detection of implementation practices that have an impact, positive or negative, on the integration or transformation of an application (e.g., separation or coupling of user interfaces with business logic, or sequential batch processing)
- Determination of ways to group related applications together based on their interrelationships
- Improved accuracy of initial estimates through use of automated tools to compute software metrics

Automated tools, such as WebSphere* Studio Asset Analyzer (WSAA),[7] can be used to analyze the software artifacts associated with a portfolio of applications and to create detailed meta-data on the composition, size, and complexity of each application. The mapping between the coarse-grained business and application models, when combined with the linkage between the manually gathered application model and the tool-gathered application meta-data, gives us the foundation for assessing the technical impact of the desired business change.

The tools may supplement the initial analysis, but the purpose of the initial analysis remains the identification of the context, scope, and requirements for a proposed change. The initial analysis relies on a high-level model of the business, a high-level model of the application portfolio, and the mapping between them. The output of this initial analysis is thus a high-level specification of the business functions or services to be changed and the applications and interfaces that need to be changed, integrated, replaced, or retired. This specification is used as input to the detailed analysis.

Detailed analysis and fine-grained mapping

The detailed analysis identifies all of the software assets affected by a change and their dependencies

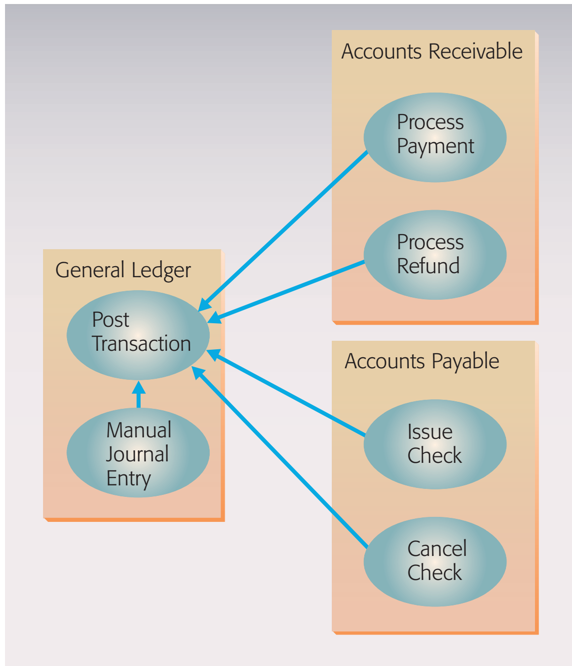

Figure 2
Legacy application interfaces

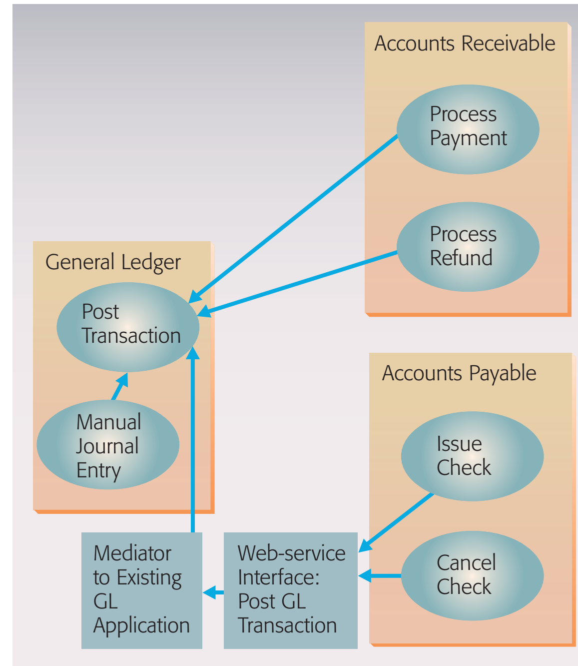

Figure 3
General Ledger application replacement: Release 1

and interfaces throughout the application portfolio. For example, detailed analysis for a project to create a Web services interface to a legacy implementation of a business function needs to examine the composition and flow of online transactions and batch jobs, as well as the persistent data stores that help perform that function. The analysis must identify any barriers to creating a Web services interface with the desired quality of service—such as a monolithic batch implementation of a business function that may require subsecond response time when invoked as a service.

The mapping between business and legacy systems models is also an important aspect of this subsequent detailed analysis. Our work on fine-grained mapping focuses principally on the scenario for transforming legacy systems to participate in SOAs, particularly those legacy systems that are not amenable to straightforward integration. The type of analysis required can be shown with an example: replacing an existing application with a software package, using Web services for interfaces. Consider the simplified example in ***Figure 2***. Assume that the project objective is to replace a company's General Ledger (GL) application with an off-the-shelf package. The interfaces between the GL application and the rest of the portfolio need to be considered. In our simple example, the GL application's Post Transaction function is used by two other applications: Accounts Receivable and Accounts Payable.

In this example, let us assume that Accounts Receivable is undergoing significant enhancement to meet new business requirements and that changes to its interfaces cannot be made for several months. The package replacement could then be implemented in three releases:

1. A first release might create a new Web-service interface to the existing code for the Post Transaction business function. The Accounts Payable application would be modified to invoke the Web service. Two of Accounts Payable's functions would need to be changed: Issue Check and Cancel Check. This is shown in ***Figure 3***.

2. After the enhancements to Accounts Receivable are complete, the second release in the GL replacement would be prepared. Accounts Receivable would be modified to use the Web services interface. Two of Accounts Receivable's functions would need to be changed: Process

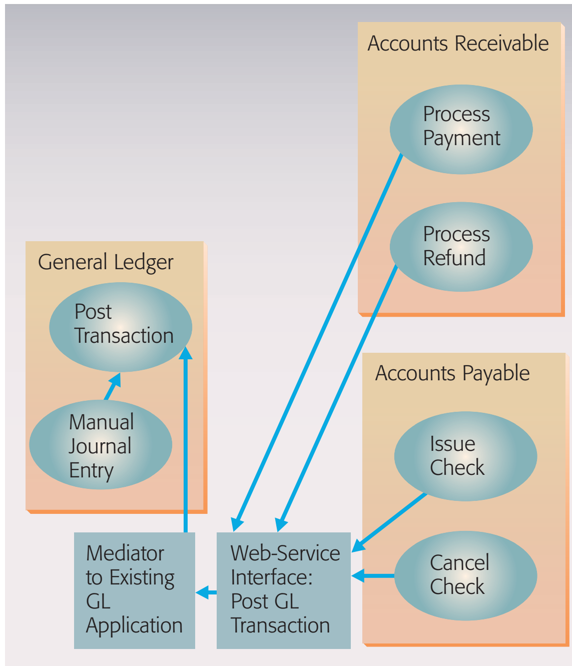

Figure 4
General Ledger application replacement: Release 2

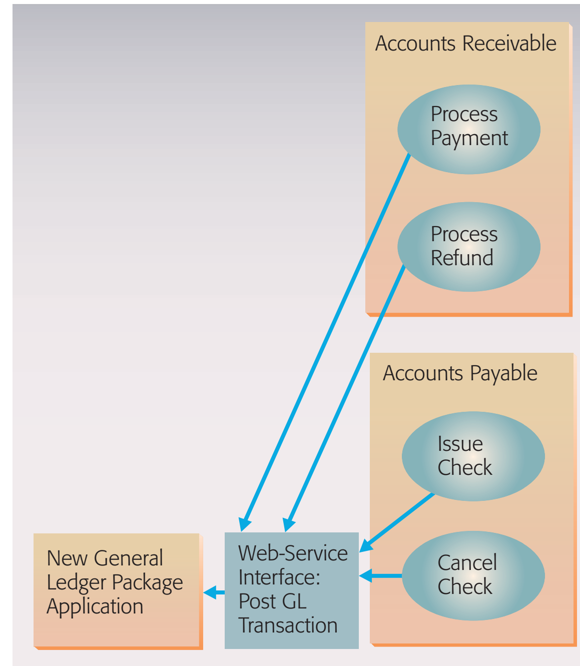

Figure 5
General Ledger application replacement: Release 3

Payment and Process Refund, as shown in *Figure 4*.

3. The third release would preserve the Web services interface, but replace the existing GL application with the new package. *Figure 5* depicts this final release.

It should be noted that the use of Web services is not a requirement of this encapsulation; the essential requirement is to create a well-defined interface that hides the implementation of the GL functions from other applications.

One set of project planning problems revolves around the implementation of each of the interfaces to the GL. For each, a decision needs to be made whether to convert the other application to use the Web service directly, or whether to create a "reverse adapter" around the Web service that implements the existing legacy interface.

Although simplistic, this scenario demonstrates that the planning of a legacy transformation project requires integration of knowledge about the business model, its mapping to the legacy systems model, and a legacy systems model of software assets and their interdependencies. This analysis requires mapping at a finer-grained level for both the business model and the legacy systems model. Given the orientation of our work toward aiding the transformation to SOA, it is natural that the fine-grained representation of the business model may be that of a service interface.

METAMODELS FOR DETAILED ANALYSIS

Figure 6 is a Unified Modeling Language** (UML**)[8] class diagram that is a conceptual-level model of a service interface and implementation, based on prior work in modeling and specification of services.[9] The ComponentFacade stereotype in the diagram represents the interface to the service: in this example a service named doService.

The Mediator behind the facade provides the implementation of the service by coordinating the execution of BusinessComponents, which could be new code or interfaces into legacy systems.

Our detailed analysis identifies the points of interface into legacy applications and helps determine whether the legacy application is well-suited for integration through standard connectors or mes-

saging. When a legacy system is already well-suited for integration, transformation is not needed to create services. An example would be a legacy system that already has callable interfaces to its transactions.

Our model for detailed analysis of legacy systems includes a model of the z/OS application runtime environments and a model of application programming artifacts. ***Figure 7*** is a conceptual-level UML model of z/OS runtime artifacts. Analysis of runtime artifacts is important, because it provides essential information about how programs are invoked in batch and online interactions, and it also reveals the binding between programs and the physical files and databases they use. ***Figure 8*** is a conceptual-level model of a compilation-unit application program and its interfaces. The model captures information about the internal details of each program—the data

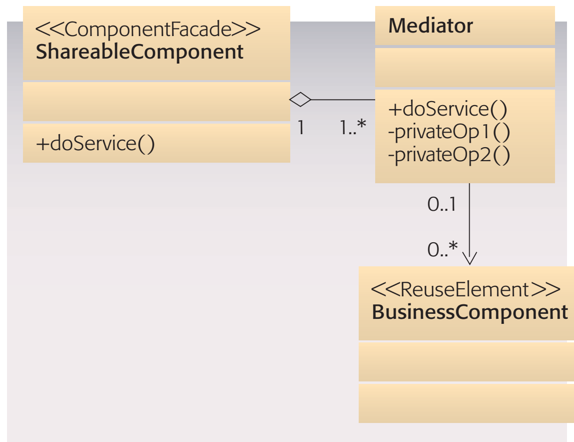

Figure 6
Template for service interface

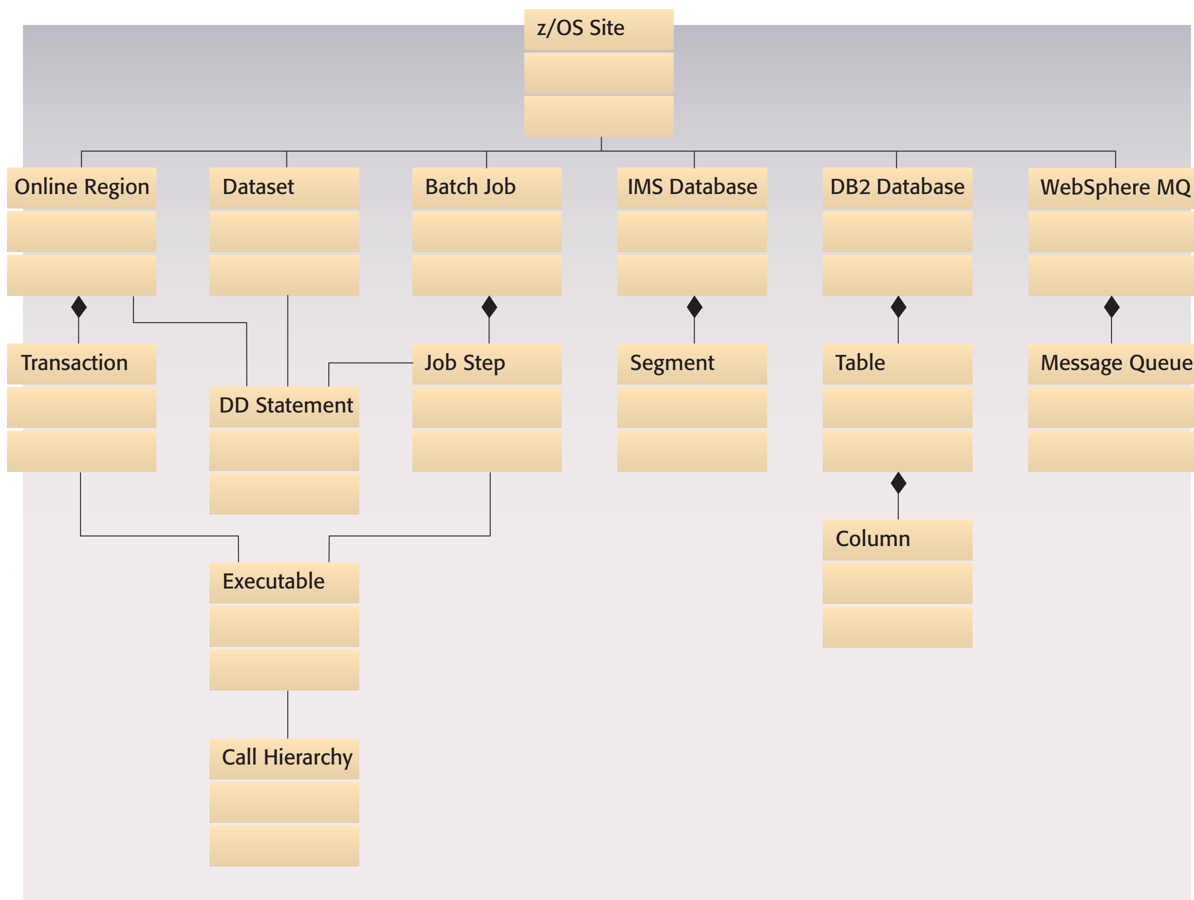

Figure 7
Conceptual model of z/OS runtime artifacts

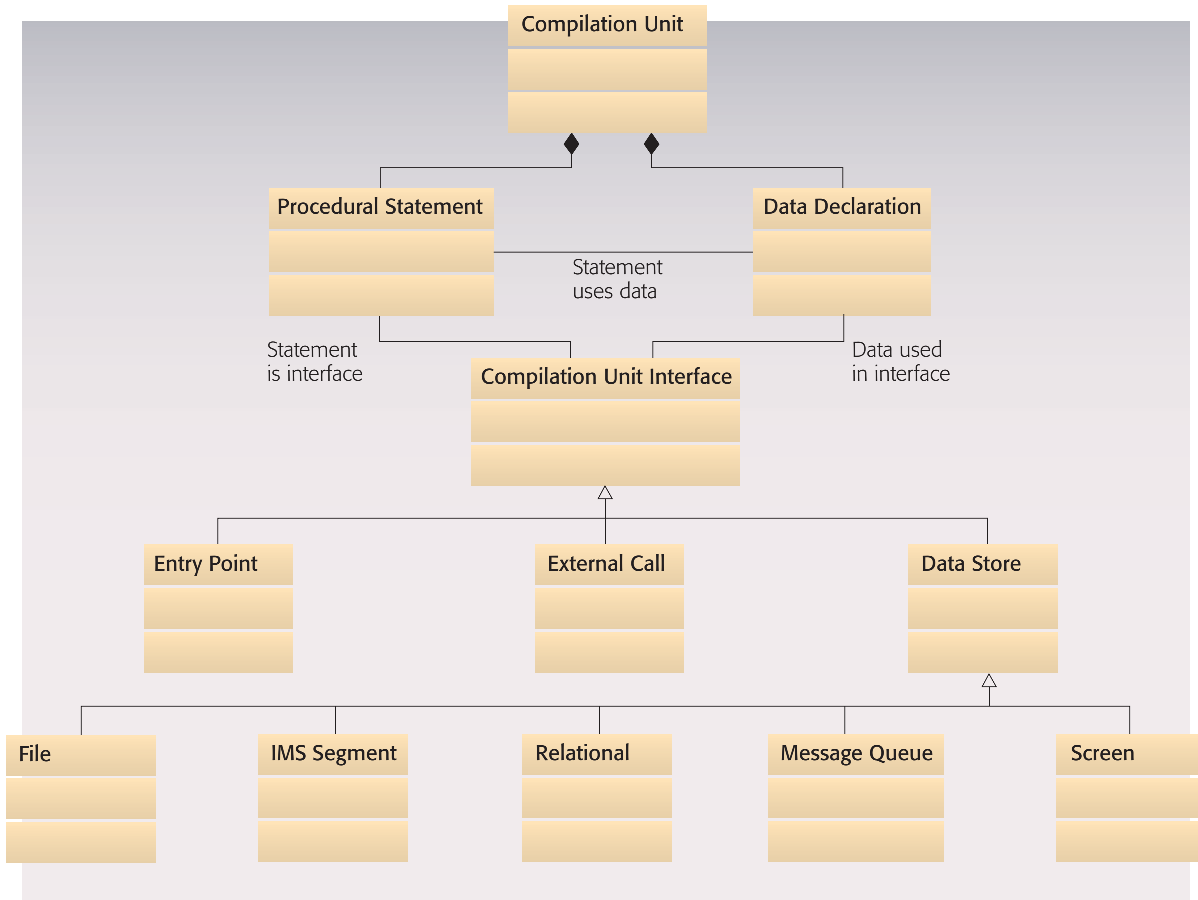

Figure 8
Compilation unit conceptual model

elements declared and the procedural statements—and about the external interfaces of each program.

The runtime and program models are used by generalized impact-analysis tools that trace control and data flow within and across programs. This automated impact analysis can be performed starting with a scenario and a *seed* (described in more detail in the next section). Example scenarios include:

- Maintenance and enhancement changes to data definitions, procedure interfaces, and programming logic
- Creation of Java 2 Platform, Enterprise Edition (J2EE**) Connector Architecture connectors[10] to CICS* (Customer Information Control System) and IMS* (Information Management System) transactions[11]
- Creation of a subroutine from procedural logic contained in a COBOL (Common Business Oriented Language) program

The impact-analysis tools trace control and data flow within programs, across program interfaces, and through files and databases. Building on top of the control and data-flow tracing, the impact-analysis tools determine the impact of a potential change upon runtime and program artifacts at all levels of granularity, from programming statements or data items up to the level of the overall application. This impact information is stored in the meta-data repository.

CONNECTING THE BUSINESS AND LEGACY SYSTEM MODELS

Whereas the initial coarse-grained mapping is typically performed broadly for the enterprise and its application portfolio, the fine-grained mapping is

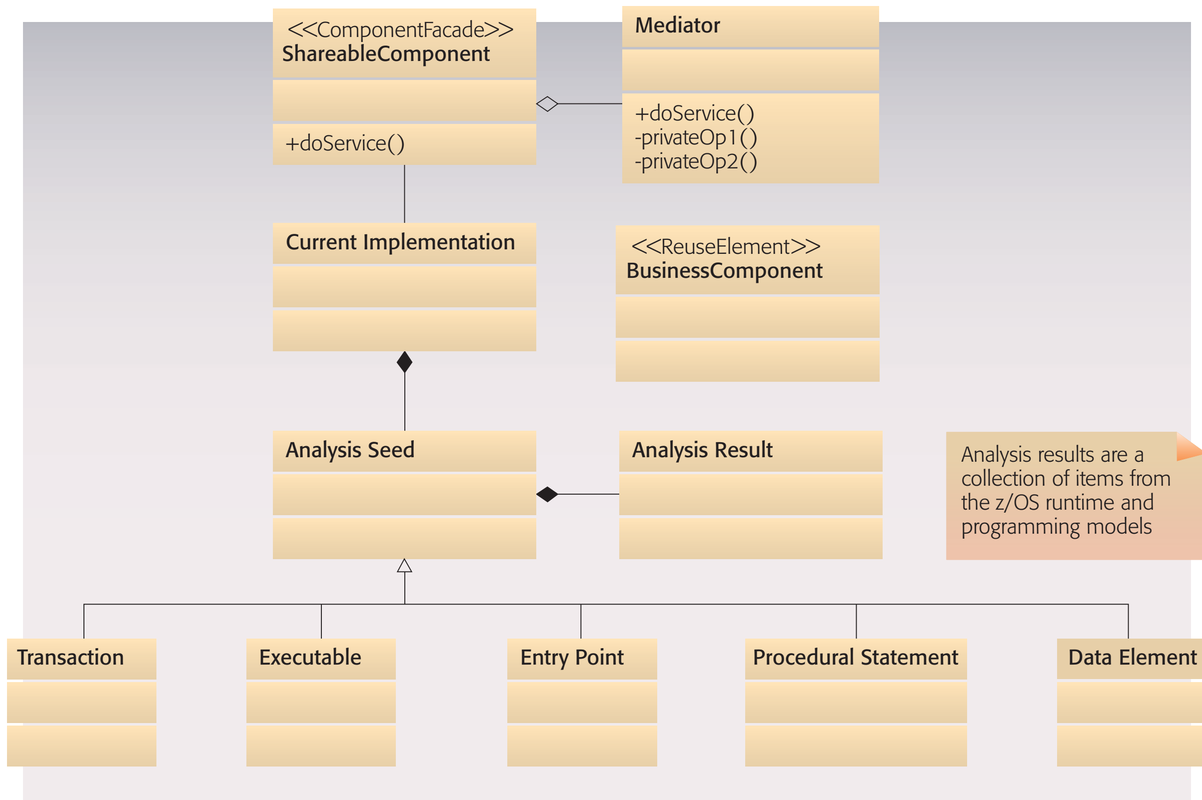

Figure 9
Conceptual model for fine-grained mapping and analysis

performed selectively for the subset of the application portfolio that is relevant to the project. The fine-grained mapping is performed with a combination of user input and tool assistance. ***Figure 9*** shows a conceptual view of the metamodel for this fine-grained mapping and analysis.

The selective mapping and the subsequent analysis use the following steps:

- Define the service to be created and its proposed interface. This is done from the top down in the traditional way, using the business and functional requirements. The service interface is represented by the ShareableComponent.
- Identify one or more points in the legacy system that are currently performing processing that needs to be included in the service. These points could be existing CICS or IMS transactions, batch jobs, programs, or even statements within a program. These are represented by the Analysis Seed. When a subject matter expert is available—a business user, analyst or developer who can describe the systems and transactions used to perform a process—this is a straightforward process. When the desired expertise is not available, other techniques are employed, such as documentation review, observation of system users, and the use of tools to search and analyze source code.
- For the touch points identified above, use generalized application understanding and impact analysis tools that follow control and data flows to identify the parts of the application portfolio that may be part of the implementation of the service. These are represented by the Analysis Result.
- Use specialized analysis tools to examine the affected parts of the portfolio, looking for instances of architecture, coding, and data-structure patterns that may affect the integration of the legacy system with the service interface.

Impact-analysis tools are used to determine the application software assets related to the initial seeds. The tools analyze control and data flow within a program to identify affected statements and data elements, and analyze control and data flow across interfaces to identify other programs and data stores that are affected. The tools use the z/OS runtime model to analyze impact through shared data within and across applications.

The ability to do global impact analysis simplifies the job of an analyst who needs to determine how to integrate the legacy system with the service. It also provides flexibility to the analyst performing the fine-grained mapping, since it allows the analyst to choose from many types of seed based on the information that is available. The analyst does not need to know the precise location within the application where the current business process begins or ends. Rather, the analyst just needs to know a location where the business process is performed; the automated tools will help determine the boundaries of the process implementation.

Given a set of initial mappings, the role of the tools is to compute other relevant mappings and to provide a way to help a user iteratively refine these mappings to expose the alignment and gaps between the current legacy applications and the desired service interface.

THE ROLE OF PATTERNS FOR LEGACY TRANSFORMATION

Patterns have become recognized as a useful way to capture lessons learned and to help disseminate and apply practices that have proven successful. IBM's Patterns for e-business are an example of this concept:

> The Patterns leverage the experience of IBM architects to create solutions quickly, whether for a small local business or a large multinational enterprise. . . . customer requirements are quickly translated through the different levels of Patterns assets to identify a final solution design and product mapping appropriate for the application being developed.[12]

These patterns have been developed and refined with experience gained on more than 20,000 Internet-based engagements,[13] and they provide guidance for working top-down from business design to application runtime. In general, the patterns provide a description of proven practices (and thus, reusable assets) that identify how to satisfy a given set of objectives within a defined context. The patterns are typically used in a prescriptive fashion, guiding analysis and design processes that start from the top down.

The application of patterns to legacy transformation is somewhat different. The intent is the same: to capture lessons learned and help disseminate and apply proven practices. Legacy transformation adds significant bottom-up analysis and design efforts to the top-down analysis inherent to IT projects.

Patterns are a means to the end of legacy transformation. We want to use them to:

- Help us understand the current state of legacy systems, with *as-is legacy patterns* found through tool-based analysis of legacy application assets
- Model the desired end state of legacy transformation projects, with *patterns for z/OS applications* and the IBM Patterns for e-business
- Identify the sequence of incremental changes to close the gap between the current state and end state, using *legacy transformation patterns*

The definition of as-is legacy patterns is an ongoing process and results from analysis and transformation projects with IBM clients. As with the Patterns for e-business, our goal is to define and refine these patterns by using experience gained on a large number of projects. A taxonomy of patterns has been developed as part of this work. There are three basic as-is legacy pattern categories:

1. *Architecture patterns,* which span from the application portfolio to the flows invoked by mechanisms other than program calls (e.g., CICS pseudoconversations, batch schedules) and connected by shared data.

2. *Program patterns,* which address items found in a single compilation unit or in a collection of programs connected in a calling hierarchy.

3. *Data patterns,* which address the structure and navigation of data structures, typically files and databases.

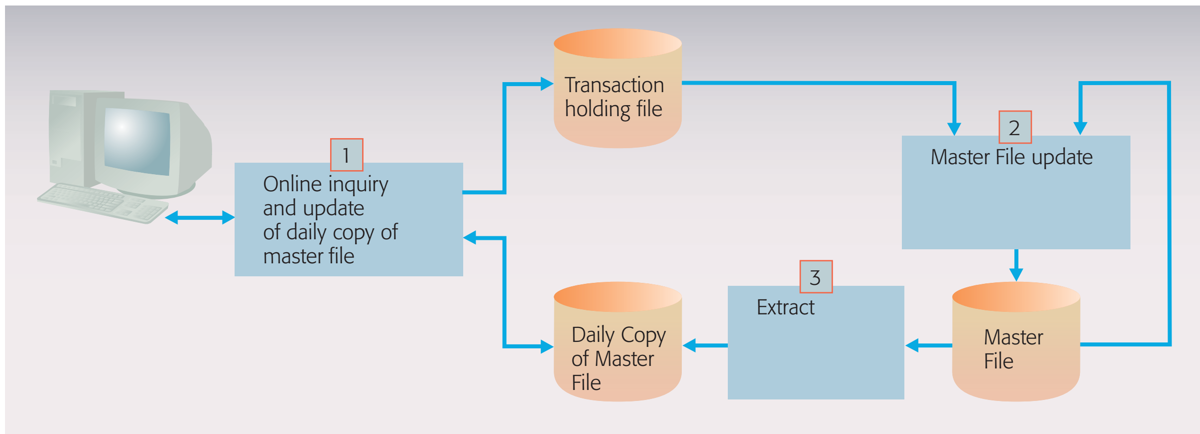

Figure 10
Online Data Capture With Batch Update legacy architecture pattern

An example of an as-is legacy pattern at the architecture level is Online Data Capture With Batch Update, shown in ***Figure 10***. For purposes of this example, we use the term *file* to represent either a file or a database.

This architecture pattern includes three key elements:

1. Online programs that store transactions for subsequent processing in a batch job. The on-line programs typically validate the transactions before storing them, to minimize the number of exceptions detected in the batch job.

2. Batch programs that read the stored transactions and update master files. The batch programs may repeat the validation performed online, and may perform additional validations as well.

3. Batch programs to create or update the replicated master file data used by the online programs.

This pattern is useful when analyzing options for exposing existing function as a service. The consequence of this pattern is that a straightforward connector to the online program cannot provide a synchronous update to the master file of record. If the service requires a synchronous update to the master file, then alternative approaches must be evaluated.

An example of an as-is legacy pattern at the program level is the Multiple Split/Merge On Transaction Data pattern. This pattern detects a coding idiom commonly found in COBOL programs that perform multiple business functions, such as opening a bank account, recording deposits, recording withdrawals, and closing a bank account. In this idiom, the program is organized with a common mainline routine that handles all business functions, with lower-level routines making tests of business function to perform specialized processing. This is in contrast to a program organized into a separate mainline routine for each business function, routines specific to a business function, and lower-level routines that perform logic shared by two or more business functions.

This program pattern includes three key elements:

1. A field or structure in an input record that is compared to multiple hard-coded values

2. A control flow graph that branches based on the comparisons to hard-coded values, then merges to perform common logic

3. Multiple instances of a test for a specific value

This pattern is useful when analyzing options to change a program to make it more extensible or to extract a subset of its function. The consequence of this pattern is that the organization of the program needs to be changed—for example, to make a single test of the transaction data at the top of the procedural code—before the program can be made more extensible or before the code for the business

function can be extracted to a separate module. The process of changing the organization of the code without changing its function is commonly known as *refactoring*.[14] Tools and patterns for refactoring J2EE applications are an important part of Java integrated development environments. Many of our legacy transformation patterns are intended to guide the process of refactoring legacy systems.

An example of an as-is legacy pattern at the data level is the Sequential Master File pattern. Some legacy applications still exist in which master files are organized or processed sequentially. The pattern has one key element: the way in which the master file's dataset is organized. This pattern is useful when analyzing options to expose existing business functions as services. The consequence of this pattern is that redesign of data structure is likely to be a prerequisite for refactoring the program or exposing any of its capabilities as a service.

As best practices for z/OS applications are created or identified, we are able to define and document z/OS application patterns. Emphasis is being placed on patterns that exploit newer capabilities available on z/OS, such as support in COBOL for XML (Extensible Markup Language) data, Java interoperability, and support for SOAP (Simple Object Access Protocol) and Web services interfaces in CICS and IMS.

We are able to define legacy transformation patterns as we identify discrete transformations that can be combined to close the gap between an as-is state and a to-be state. These legacy transformation patterns are associated with instances of the as-is legacy patterns. For example, the Online Data Capture With Batch Update as-is legacy pattern shown in Figure 10 could be addressed by transformations such as:

- Use Primary File Instead of Replica, which would modify the online inquiry and update application to use the master file directly and eliminate the daily copy of the master file. The result would be online transactions that could be wrapped to create a service interface.

- Run Subset Of Batch On Demand, which would preserve the existing online and batch architecture, but would make any changes needed to allow the batch process to be run as needed on a small set of transactions.

- Add Message Interface To Batch, which would modify the batch programs to accept transactions from a message queue, in addition to the current file-based transaction source.

Tools to support the use of the as-is legacy patterns and legacy transformation patterns are being built as extensions to existing analysis tools. This automation is important: tools have the ability to comb through models of large software portfolios searching for instances of as-is legacy patterns through multiple implementation variations and levels of indirection. These tools can extend the abilities of analysts who may lack experience in z/OS applications and methods commonly used years ago. Moreover, the tools are built upon models that can be the basis for project planning, change management, and testing. As we expand our set of patterns, we will look for opportunities to generalize the way we detect patterns and externalize their specification.

Our work has focused on capturing these patterns through engagements in two client scenarios: (1) consolidating data and applications to align with business processes and (2) integrating z/OS batch and online software with service interfaces. This work is in its early stages and will certainly evolve. Two case studies are summarized to illustrate these patterns and their use in legacy transformation. It should be noted that the use of patterns is a way to augment, not replace, the processes and methods already in use today.

The case study on application consolidation focuses on the use of patterns in the initial analysis phase. The case study on integrating with a services interface focuses on the detailed analysis phase.

Case study: Application consolidation

There are many reasons why application consolidation may be proposed. Mergers are one example; removing redundancy found by CBM is another. An application consolidation includes the following steps:

1. Initial analysis
    a. Identify existing functions and data in each application

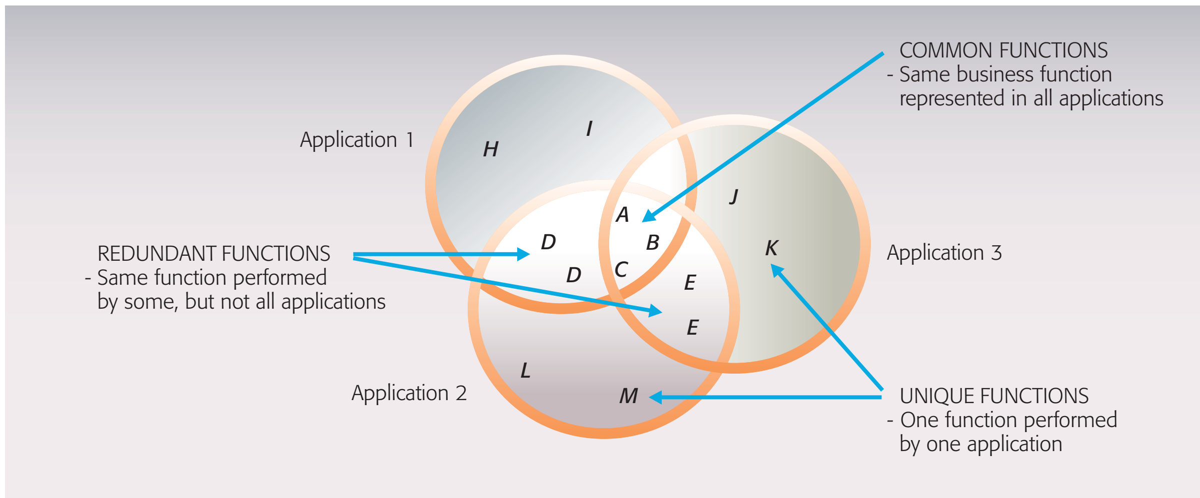

Figure 11
Venn diagram of legacy pattern at the application level

   b. Define criteria for choosing "best-in-class" implementation
   c. Assess application functional overlap
   d. Classify functions by degree of commonality: core (used by all), common, or application unique
   e. Select "best-in-class" application (or replacement package)
   f. Determine disposition for other common and unique functions
   g. Create initial project estimate
2. Detailed analysis
   a. Plan new user and external interfaces
   b. Plan code and data migration, testing
3. Design
4. Develop
5. Test
6. Deploy

The scope of application consolidation projects includes data as well as function. Our work on legacy patterns also includes that scope. In the interest of clarity, however, in this paper we will focus specifically on functions and the role of legacy patterns in their analysis.

Initial analysis is supported by the coarse-grained mapping between the business model and the applications defined in the high-level legacy systems model. Using that mapping data, as-is legacy patterns at the application level can assess functional overlap and classify functions based on the degree to which their use is repeated in the applications. If we were to stop at this point, we would be automating some of the data capture that is typically performed with spreadsheets and documented with reports and Venn diagrams such as the one shown in *Figure 11*. However, at this coarse-grained level, relying only on information gathered in a consultative process, we can go a bit further in two ways:

1. Use the information about the applications (including the application-to-function mapping, information gathered about technical quality, and business capability) as input to the analysis that selects the "best-in-class" application to be used as the base for consolidation. This analysis also determines the disposition of each common or unique function: whether it should be included in the consolidated application, reallocated to another application, or eliminated.
2. Capture coarse-grained information about application interfaces. This information is typically gathered with interviews and questionnaires that ask an application owner to identify the other applications with which interfaces are known to exist.

Patterns for coarse-grained analysis

With this additional information, we can apply high-level legacy transformation patterns to develop the initial work plan. Because we are working on the

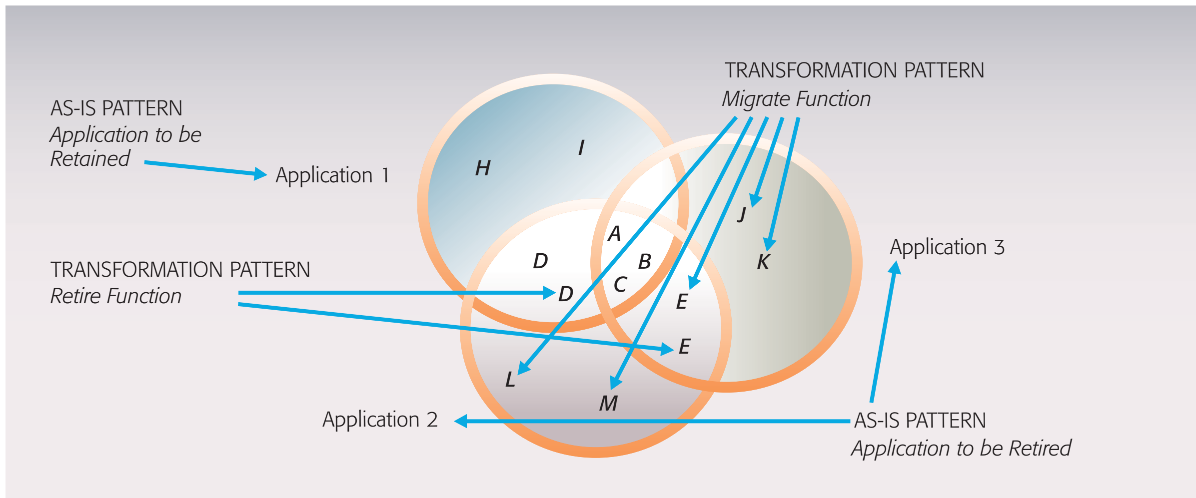

Figure 12
Venn diagram including legacy transformation patterns

initial plan, these patterns represent high-level activities to be completed in later detailed planning. These patterns can be thought of as refactorings at the level of applications, functions, and interfaces. The legacy transformation patterns include:

- Retire Application, which is instantiated for each instance detected by an as-is legacy pattern called Application To Be Retired.
- Retire Interface, which is instantiated for each instance detected by an as-is legacy pattern called Interface Connects Applications To Be Retired.
- Migrate Interface, which is instantiated for each instance detected by an as-is legacy pattern called Interface Connects Applications To Be Retained.
- Migrate Function, which is instantiated for each instance detected by an as-is legacy pattern called Function To Be Retained from Application To Be Retired.
- Retire Function, which is instantiated for each instance detected by an as-is legacy pattern called Application To Be Retained Implements Function To Be Consolidated.
- Create Common Interface, which is instantiated for each instance detected by an as-is legacy pattern called Interface Connects To Multiple Applications; this analysis occurs after the analysis for the Retire Interface and Migrate Interface legacy transformation patterns.
- Create Interface Adapter, which is instantiated for each interface consolidated by the Create Common Interface legacy transformation pattern.

A simple illustration of how these patterns apply is shown in ***Figure 12***. In this example, Application 1 was chosen to be the base for consolidation; Applications 2 and 3 are to be retired.

Coarse-grained analysis with fine-grained legacy model

If automated analysis of the application portfolio has been performed, legacy patterns can be applied to this finer-grained information, even in the context of initial analysis. In this case, the finer-grained application model gives a more detailed understanding of the interfaces between applications. We can detect instances of as-is legacy patterns such as:

- Shared File/Database, which looks for files or databases used directly by multiple applications (see ***Figure 13***).
- Custom Extract, in which a file is created by one application and processed by one other application (see Figure 13).
- Common Extract, in which a file is created by one application and processed by more than one other application (see Figure 13).
- Custom API, in which a transaction or message-driven program belonging to one application is invoked by one other application.
- Common API, in which a transaction or message-driven program belonging to one application is invoked by more than one other application.

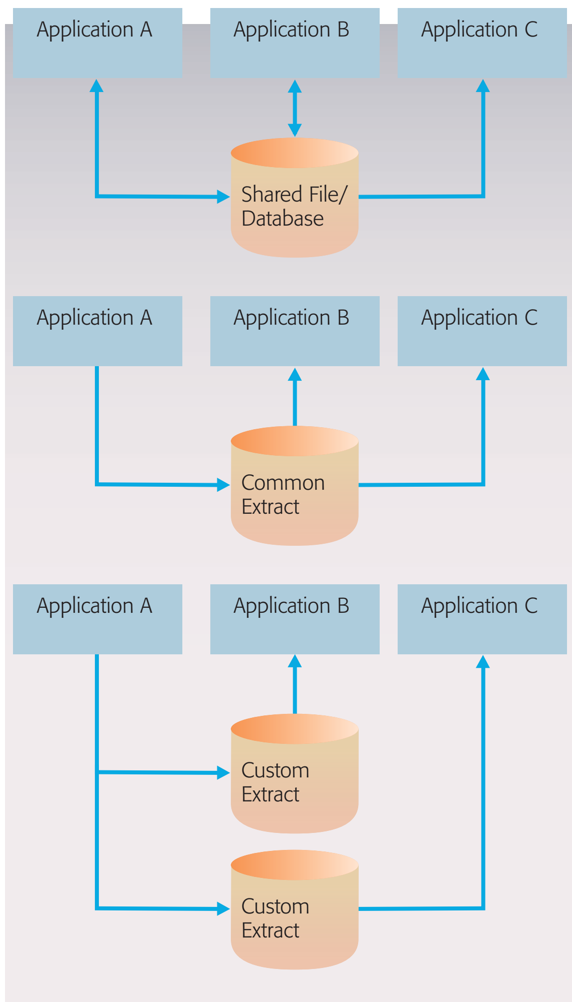

Figure 13
Example as-is legacy pattern

These as-is legacy patterns lead to corresponding legacy transformation patterns. For example, the Custom Extract as-is pattern shown in ***Figure 14*** leads to potential legacy transformation patterns for the data provider, Application A, such as Replace With Standard Extract/Transform/Load Tool, and for data consumers, applications B and C, legacy transformation patterns such as Replace Use Of File With Access To Common Warehouse.

While these patterns can provide insight to architects performing initial analysis and can also help with analysis of modest amounts of information, they represent an incremental improvement in the way these projects are performed today. It is in the transition to detailed analysis that patterns can enable a significant change in the size and scope of projects that can be handled effectively.

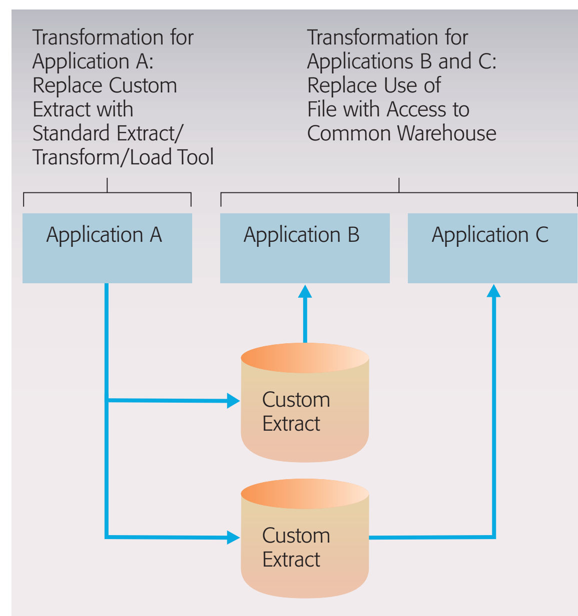

Figure 14
Legacy transformation patterns for Custom Extract legacy pattern

The detailed analysis requires that the automated analysis of the application portfolio be complete. For application consolidation, the legacy patterns can help identify ways to migrate functions and interfaces. In the consolidation scenario, we may be doing the detailed analysis of consolidation requirements by using a business process model that is based on a functional decomposition or on a business component and service model.

Once we focus our attention on the transformations required for a single function in a single application, our analysis for the application consolidation scenario converges with that needed to integrate z/OS batch and online applications with a services interface. For the purposes of this paper, we take up the discussion of detailed analysis in the second case study that follows.

Case study: Integration with services interface

Our work on legacy patterns focuses on integration and transformation requirements that are not met by existing integration approaches. As an example, consider the application architecture shown in

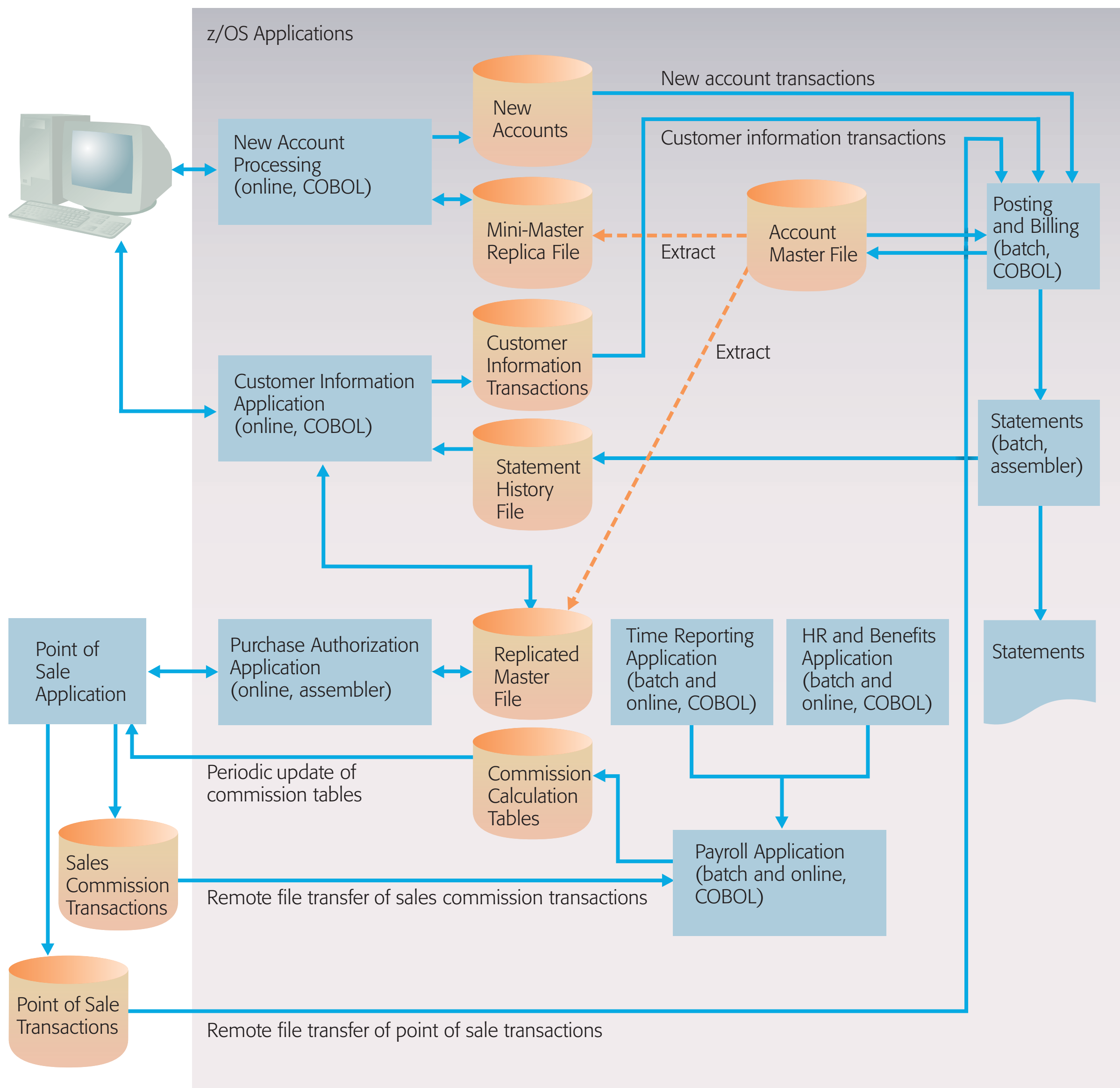

Figure 15
Application architecture for detailed analysis case study

Figure 15. This architecture is a composite created from two similar client engagements. It includes a set of batch applications that perform the core business processes of posting and billing. These batch applications are the current systems of record and thus manage the master files. Over time, these batch applications have been augmented with online systems for customer-service and new-account processing, as well as interfaces to point-of-sale systems. Each of these systems maintains its own data, which is synchronized with the master files on a nightly basis.

The business process of creating a new account spans two COBOL applications: the online application for new-account processing and the batch posting application. If the requirement for business change is to create a service interface to the current Create New Account process, it might be sufficient to wrap the existing new-account-processing online transactions. If, however, there is a requirement that the account be created immediately, and not by overnight batch processing, the straightforward wrapping of the online new-account-processing application will not suffice.

This requirement drives deeper analysis that can be improved by using patterns. The existing implementation needs to be examined to determine whether some portion of it can be reused, or whether it needs to be refactored or even replaced.

Our approach to detailed analysis comprises the following steps:

1. Use the mapping between service interface and legacy system to perform automated data and control-flow analysis to identify the potential scope of change.
2. Within that scope, and the type of change, perform automated analysis to see which as-is legacy patterns apply.
3. Review the results, to identify any significant barriers to project success.
4. Use legacy transformation patterns to identify and estimate alternative solutions to the requirements and constraints.

As-is legacy patterns

We have discovered several as-is legacy patterns that are relevant to the service interface scenario. At the architecture level, they include:

- Online Entry/Batch Update, in which transactions are collected and edited for subsequent batch processing to re-edit and apply the transactions.
- Edits Repeated, in which the transaction data undergoes the same or similar validation at different times in the process. An example is online edits that are similar to those in the batch process.
- Job Step Handles Multiple Processes, in which many business processes are handled in a single job step. This is typically a performance optimization, to avoid reading the same file multiple times.

At the data level, as-is legacy patterns include:

- Replicated Master File, in which batch systems create a copy of their master files for use by online systems.
- Sequential Master File, in which the physical organization of the file permits only sequential access. Direct access to a given record is not possible without reading the file from the beginning.
- Home-Grown Database, in which access to data—typically master file data—is controlled through an API (application programming interface) that supports positioning, reading, writing, updates, and deletions, but hides details associated with underlying data formats, compression, and physical access.

At the program level, as-is legacy patterns include:

- Abend On Exception, in which the program intentionally executes an instruction or invokes a system API to terminate execution of the program and batch job step, rather than attempting recovery.
- Screen Handling Combined With Business Logic, in which online programs have business logic tightly coupled to presentation logic, making it harder to create a service.
- Screen Handling Separated From Business Logic, in which online programs are well-suited for invocation of business logic from a wrapper or service interface.
- Multiple Split/Merge on Transaction Data, in which a single program handles multiple business transaction types and has a single mainline path that contains tests for transaction type in multiple locations, in contrast with testing for transaction type once and performing the processing of each transaction in a cohesive routine.
- Multiple Statements For File I/O, in which a single program uses multiple identical I/O statements (such as read or write), instead of encapsulating I/O in a common internal procedure or external subroutine.
- Deferred Write Of Master File, in which a program is sequentially processing an input transaction and input master file, and writing a new copy of the master file as an output. The transaction and master files are sorted on some common key. In the classic batch update program, an in-memory copy of the master file record is updated for each applicable transaction. This in-memory copy is written only after all applicable transactions have been applied to it. The write is not triggered by any of the transactions that update the record; rather, it is triggered when the program reads a transaction that applies to a different master record.

How do these patterns help create a service interface to the new account processing? First, they help us

understand the architecture of the current implementation, which imposes constraints on the solution.

Recalling Figure 9, analysis starts with one or more user-specified links between the business model, in this case the service definition, and the legacy systems model. The analysis approach accommodates multiple starting points. The case study application shown in Figure 15 offers two examples:

1. If the analyst were to specify the online transaction that initiates the business process, analysis of control and data flow would lead to the New Accounts transaction file, which would in turn lead to the batch job steps and programs in the posting application. In turn, additional control and data-flow analysis would detect the data insertion into the master file, and the copying of the master file to the mini-master file used by the online new-account-processing application.

2. If the analyst were to specify the statements in the batch posting program, the control and data flow analysis would work backward from the batch program, discovering the New Accounts transaction file, and leading to the online system and relevant transactions.

In either case, this analysis creates a graph structure of the control and data flows. That graph can be analyzed to detect the instance of the Online Entry/Batch Update as-is legacy pattern. It can also be analyzed to detect the creation of the mini-master file as an instance of the Replicated Master File pattern. Analysis of the batch job's dataset attributes reveals the Sequential Master File pattern. Analysis of the online programs exposes any instances of Screen Handling Combined With Business Logic, while analysis of the batch programs exposes Abend On Exception, Multiple Split/Merge On Transaction Data, Multiple Statements For File I/O, and Deferred Write Of Master File. By examining and comparing the statements used in the online and batch programs, Edits Repeated may be identified.

Having found these instances of as-is legacy patterns, what can we do with them? In our current engagements, the pattern instances have been used by the architects developing the solution approach and plan. Certain of these patterns—such as Sequential Master File—cannot be replaced by a programmed work-around. In fact, this data structure is a dominant factor shaping the application architecture. Other patterns, such as Edits Repeated, can have a programmed work-around or can even be tolerated.

In the scenario for integrating with a services interface, the as-is legacy patterns help the architect identify barriers to creating a service and explore implementation options. For example, it may be possible to split this effort into subprojects. In a first release, one might initially create the service interface on top of the existing online system, realizing the benefit of interoperability without achieving the benefit of straight-through processing. A second release might convert the sequential file to a database or direct access file. A third release could consolidate the processing to create the straight-through service.

Legacy transformation patterns

Whereas these as-is legacy patterns help discover constraints and requirements, legacy transformation patterns help identify potential solutions. For each as-is pattern, there may be one or more potential transformation patterns. For example, two transformation patterns for Sequential Master File are Convert Sequential File To Direct Access File and Externalize Program I/O Statements In I/O Module. In these transformations, the underlying data structure can be changed to permit keyed access, even if batch programs continue to access the file sequentially. The transformation also decouples the programs using the file from the file's physical characteristics, facilitating future changes in data structures.

Another example involves transformation patterns associated with the Multiple Split/Merge On Transaction Data pattern. Two transformation patterns could apply:

1. Refactor By Transaction Type, which creates a separate subroutine to handle each type of transaction.

2. Refactor Common Code Into Shared Module, which further refactors the subroutines created for the transactions.

In one client engagement, a batch job step for transaction posting was refactored. The original implementation handled 48 unique transaction types for 10 account types and included 38,000 lines of code in nine source programs. The refactored implementation totaled 42,000 lines of code in 45 programs. The number of lines of code increased slightly, but the average program size was reduced from approximately 4,200 to approximately 900 lines of code. The resulting programs were functionally cohesive and well-structured for single-entry, single-exit processing. The transformation offered two benefits: (1) improved maintenance because the application could now easily be extended to handle new transaction and account types, and (2) improved interoperability, because the processing for each transaction type was exposed for reuse.

We continue to explore ways to link the legacy transformation patterns, the legacy system models, and impact-analysis tools to provide "what if" simulation capability in developing transformation alternatives.

CONCLUSION

The design trade-offs made in creating application software have changed dramatically over the past four decades. Where we once sought to automate clerical processes and squeeze programs and data into the smallest possible space, we now seek to automate business processes within and across enterprises, with an eye toward maximizing flexibility and responsiveness to change. Where software was once designed with flowcharts, written on coding sheets, and punched into cards, we now see business processes designed with visual models that are transformed into executable flows, which in turn choreograph reusable components and services.

At the core of many application portfolios, however, are legacy systems that were designed, quite appropriately, to meet the constraints of their time and that cannot easily adapt to the types and pace of change required today. A variety of techniques exist to wrap these legacy systems, delivering the benefits of interoperability. In many cases these techniques are sufficient. In other cases, additional techniques are needed to achieve interoperability and to streamline the application portfolio and align it with business needs.

This paper has described an approach that is grounded in past experience with legacy transformation processes and tools, that seeks to align with other work in business strategy consulting and information technology, and that appears to offer promise in making legacy transformation a more scalable, repeatable process. There is much to be learned about useful patterns and refactorings for legacy systems, and much work remains in the areas of managing patterns and of supporting a community that can use and extend the legacy patterns.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Object Management Group, Inc., or Sun Microsystems, Inc.

CITED REFERENCES

1. S. Ramamurthy and M. Robinson, *Simplify to Succeed,* G510-9109-00, IBM Corporation (2003), http://www-1.ibm.com/services/us/imc/pdf/g510-9109-00-simplify-to-succeed-retail-banking-in-2005-full.pdf.
2. F. Injey, K. Findeis, Y. Tang, and M. Zelbel, *Patterns on z/OS: Connecting Self-Service Applications to the Enterprise,* SG24-6827-00, IBM Corporation (March 2003), http://www.redbooks.ibm.com/abstracts/sg246827.html.
3. G. Hohpe and B. Woolf, *Enterprise Integration Patterns: Designing, Building and Deploying Messaging Solutions,* Addison-Wesley, Boston, MA (2003).
4. M. Endrei, J. Ang, A. Arsanjani, S. Chua, P. Comte, P. Krogkahl, M. Luo, and T. Newling, *Patterns: Service-Oriented Architecture and Web Services,* SG24-6303-00 IBM Corporation (July 2004), pp. 31–37, http://www.redbooks.ibm.com/abstracts/sg246303.html.
5. *OMG Model Driven Architecture,* Object Management Group, http://www.omg.org/mda/.
6. R. Seacord, D. Plakosh, and G. Lewis, *Modernizing Legacy Systems,* Addison-Wesley, Boston, MA (2003).
7. *WebSphere Studio Asset Analyzer for Multiplatforms,* IBM Corporation, http://www-306.ibm.com/software/awdtools/wsaa/.
8. *Unified Modeling Language,* Object Management Group, http://www.uml.org/.
9. A. Arsanjani, J. Alpigini, and H. Zedan, "Externalizing Component Manners to Achieve Greater Maintainability through a Highly Re-Configurable Architectural Style", *Proceedings of IEEE International Conference on Software Maintenance (ICSM'02),* Montreal, Canada, October 3–6, 2002, IEEE, New York (2002), pp. 628–639.
10. *J2EE Connector Architecture,* Sun Microsystems, Inc., http://java.sun.com/j2ee/connector/index.jsp.
11. F. Injey, J. Lastra, D. Hoer, and D. Carmona, *XML on z/OS and OS/390: Introduction to a Service-Oriented Architecture,* SG24-6826-00, IBM Corporation (June 2003), http://www.redbooks.ibm.com/abstracts/sg246826.html.
12. *IBM Patterns for e-business,* IBM Corporation, http://www-106.ibm.com/developerworks/patterns/.

13. Endrei et al., *Patterns: Service-Oriented Architecture and Web Service,* p. 46.

14. M. Fowler, *Refactoring,* Addison-Wesley, Boston, MA (1999).

Accepted for publication August 9, 2004.
Internet publication January 7, 2005.

Howard M. Hess
IBM Research Division, Thomas J. Watson Research Center, 330 North Wabash Avenue, Chicago, Illinois 60611 (h2@us.ibm.com). Mr. Hess is a Distinguished Engineer in the Software Technology department at the Thomas J. Watson Research Center. He received a B.A. degree in communications and theater arts from the University of Iowa in 1981. After working for systems integration firms and the software re-engineering practice of a Big Six consulting firm, he joined IBM in 1992 as part of an application redevelopment services practice. He worked on solutions for legacy transformation in IBM Global Services and the IBM Software Group prior to joining IBM Research in 2003. ■