By Barbara Errickson-Connor

Truth or Consequences

Legacy Application Modernization

Cleaning, Transforming, & Promoting Reuse in Legacy Environments

With the growing recognition that modernizing legacy applications makes more sense than completely rewriting them, what are the consequences if we don't recognize the truth about the steps we should take?

The Conundrum

All too often, organizations are pressed for short-term speed at the expense of long-term efficiencies. Legacy application modernization projects face this same hazard, especially if the goal includes revamping the application architecture. To join the ranks of those moving in the direction of agile applications development, some initial investment of time is needed to accomplish the appropriate understanding, analysis, and restructuring of existing business logic embedded in legacy application code.

Although legacy application modernization may appear to be a newly highlighted IT requirement, it is here to stay. Because technology changes are cycling faster with every new paradigm shift, we have to separately identify and manage what is stable; a valuable outcome of a well-managed modernization project. That requires finding ways to separately manage business logic from the technology that provides access to that logic. The reward for this restructuring is that continuous modernization is easier. The speed of technology changes, coupled with the growing requirement for intra-organization software collaboration, makes modernization a must for survival. Think about the positive impact of reusing modernized legacy application code within a new architecture such as the highly touted Services-Oriented Architecture (SOA), or as the implementation code linked to a Model-Driven Architecture (MDA) project, or as the business logic now more easily

business integration journal takeaways

BUSINESS

- Modernizing mainframe applications can help future-proof an organization from technology changes and decrease total cost of software ownership.

TECHNOLOGY

- Modernizing legacy applications requires the separation of the business logic from the technology that provides access to that logic. There are several stages to achieve this goal: cleaning and restructuring existing code; transforming the restructured code; and managing the reuse of the business logic.

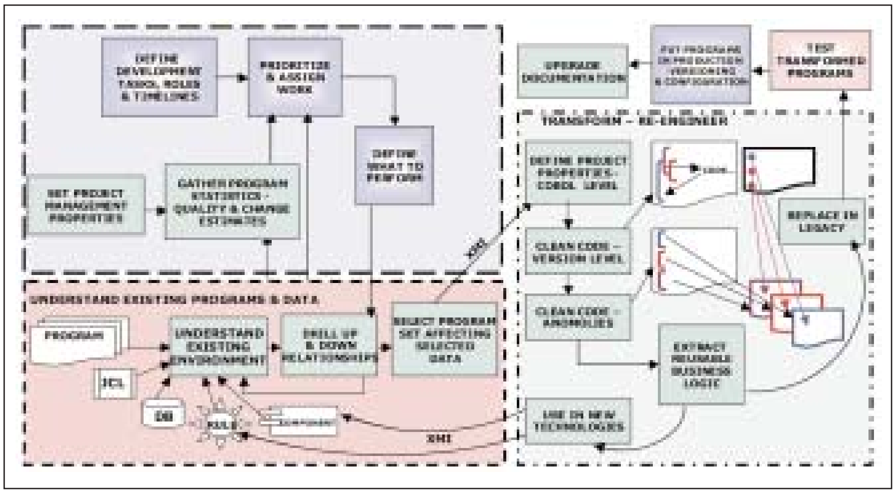
DEFINE DEVELOPMENT TASKS, ROLES & TIMELINES
PRIORITIZE & ASSIGN WORK
DEFINE WHAT TO PERFORM
SET PROJECT MANAGEMENT PROPERTIES
GATHER PROGRAM STATISTICS - QUALITY & CHANGE ESTIMATES
UNDERSTAND EXISTING PROGRAMS & DATA
PROGRAM
UNDERSTAND EXISTING ENVIRONMENT
DRILL UP & DOWN RELATIONSHIPS
SELECT PROGRAM SET AFFECTING SELECTED DATA
JCL
DB
RULE
COMPONENT
XMI
UPGRADE DOCUMENTATION
PUT PROGRAMS IN PRODUCTION VERSIONING & CONFIGURATION
TEST TRANSFORMED PROGRAMS
TRANSFORM – RE-ENGINEER
DEFINE PROJECT PROPERTIES-COBOL LEVEL
CLEAN CODE – VERSION LEVEL
CLEAN CODE – ANOMALIES
REPLACE IN LEGACY
EXTRACT REUSABLE BUSINESS LOGIC
USE IN NEW TECHNOLOGIES

Figure 1 — Legacy Application Modernization Process

plugged into an Enterprise Application Integration (EAI) framework.

How do we go about managing the transition from monolithic legacy applications to a more agile environment where we can mix and match business logic into new applications while maintaining business rule consistency and correctness?

Transitioning by Stages

Each company is unique. IT resources are scarce. Your priorities are decided by your company's business needs. As you move through the stages described here, you may choose to stop at any stage. You can derive significant value by implementing the entire architecture; however, not every company is prepared to accomplish all of the steps from modernization through re-engineering and transformation to new languages and technical environments in immediate sequence. What is critical is that you accomplish each stage with the correct, quality deliverables.

When you apply the right methods and techniques, your investment in each stage is secure. Each stage represents a significant step forward to improve your legacy environment, even with a substantial pause between steps. Additionally, because each stage can stand alone, there is no need for rework when a subsequent step is resumed after an interruption. Now let's look at the recommended stages, which are also highlighted in Figure 1.

STAGE ONE: CLEANING EXISTING PROGRAM CODE

This first stage of legacy application modernization leads to predictive, preventive maintenance that maximizes program quality. This stage includes understanding the existing applications, upgrading existing code to a consistent release level, and removing coding anomalies. For many organizations, accomplishing this stage is made easier through the use of automated application tools, especially when applications include programs that have been around for many years. Whether or not you choose automation, this stage is a prerequisite for the stages to follow.

Understanding Existing Applications

Before beginning the modernization process, the first task is to understand which applications need the most help. This task includes gathering statistics about size, complexity, the amount of dead or unused code, and the amount of bad programming for each program. In addition, in selecting which programs to improve together, selecting the ones that affect common data is a critical step when planning to move through all of the modernization stages, including re-engineering for reuse and/or migration.

The consequence of skipping this step is a lack of assurance that resources are applied to programs in the order of greatest return. Programs high in complexity, too big to manage, and that have had the worst programming habits applied should be cleaned first. They are likely to be the ones that cause the biggest problems. Even if they have not caused recent problems, if anything goes wrong in the future, they will be the hardest, and take the longest, to fix. This could be a costly oversight during a time when system downtime is measured in high dollars-per-minute, or -second.

As programs are understood, if they are not grouped into projects correctly, there will be more rework in the future. This is especially important when components and business rules are planned. All programs with logic that affects common data must be grouped together or it will be impossible to rationalize all redundant logic into newly reusable executables.

Upgrading Code Release Levels

Some organizations have avoided upgrading legacy programs that have been running in production. In some cases, the source code across applications may even have different language release levels. Before any program cleanup can begin, code must be at a consistent language release level. The consequences of skipping this step are:

- Verb and other language variances between releases could make it difficult to remove program coding anomalies according to consistent standards
- The inability to recompile all modernized programs with the same compiler
- Difficulty linking compiled programs together as cleaned applications because of compiler variances.

In some cases, there may not even be compilers available to recompile the cleaned programs.

Removing Program Anomalies

Each year, approximately 80 percent of most application budgets is allocated to software maintenance. In the past, the application's maintenance life cycle has been predominantly driven by defects identified in existing legacy applications. Often, these defects were isolated and corrections were made while avoiding any disruption to the code that was still working. The application was then tested and sent to production. This "patch" process often resulted in what we jokingly call "spaghetti" code. Today, we are paying the price of making "it work" — the key criteria for putting code back into production. We now must clean up the spaghetti.

Once the applications being modernized have all been upgraded to a consistent language release level, the task of

The major benefits continue to accrue once restructuring has been done correctly.

removing programming anomalies, or the results of bad programming habits, is easier. When resolving the confusion of deeply nested decision logic, for example, a standard verb approach can be used across all programs without having a different approach for each program's language release level. After all, we are looking for consistency, and, wherever possible, greater simplicity. And, as with the release level update step, there are application tools that can automate this step.

The consequences of not removing coding anomalies are:

- Difficulty in rationalizing existing logic to newly reusable code because of "noise" caused by dead code, or code that is not even being executed
- Difficulty, or even impossibility of, restructuring program code for ease of understanding. For example, if performed paragraphs overlap, or include branches out and back to a paragraph, it is not possible to recognize candidate reusable logic. Reusable logic must be complete and have a single entry point for it to be consistently reusable.

STAGE TWO: RESTRUCTURING EXISTING CODE

Isolating Business Logic

For most accurate reusable business logic identification, any programming logic that is used for managing file or database access, screen management, or for general system management, such as CICS or IMS interaction, should be filtered.

The consequence of not filtering non-business logic is increased difficulty in identifying common logic that should become component services, if for no other reason than the amount of code that would have to be manually excluded. It is a much more productive use of the analyst's time to review only true candidate business logic.

Rationalizing Common Logic

Once a project's program code is clean, any programming anomalies have been removed, and non-business logic has been filtered, it is possible to apply pattern-matching techniques across all of the project's programs to identify candidate common logic. This can be done either based on common data access for component or business rule services, or purely on common logic for reusable algorithms that could execute as component or business rule services.

The consequence of not performing this rationalization step to identify what is likely redundant, hard-coded logic within the same program, or across programs, would be:

- Added difficulty in simplifying future modernization efforts
- The inability to protect the organization from incorrect variances in system responses
- A reduction in reuse with an increase in complexity and cost of maintenance.

Extracting Business Logic as Reusable Services

When candidate reusable business logic has been rationalized to a subset of distinct, single occurrences of each service, it is then possible to determine whether each should become a component operation or a business rule.

Identifying Business Rules

As each reusable set of business logic is reviewed, there are alternative ways to determine business rule logic. In general, business rules capture an organization's policies, practices, and procedures. A business rule is a statement that describes how a business is organized, how a business decision is made, or how a business process works. Most business rules associate a condition to an action and naturally take the form of an "if ... then ..." statement. Business rules are typically used to solve various types of business problems, including: selection, assessment/scoring/prediction, classification, business threshold monitoring, configuration verification, diagnostic and prescription actions, or recommendation of a course of action based on synthesis of facts.

The significance of specifying whether the reusable logic will become a business rule or a component service is most important when a business rule's tool will be used to manage the business rules, outside of the standard application's space. In this case, business users can manage the rules themselves, and changes do not have to be scheduled through the application development area. However, these rationalized services can be reported to business users for them to work on in parallel with the creation of reusable component services.

The consequence of not utilizing a formalized business rules engine where appropriate is the loss of rapid reaction to changes in business policies and procedures that can be provided by their independent, user-managed control. The reliance on software enhancement/backlog rankings and programmer scheduling can slow reaction time for the implementation of competitive and/or mandated requirements.

Extracting Components

Once the candidate business rules have been isolated from the total candidate, reusable business logic set, the remaining candidates are extracted and associated with the appropriate component. Each component is responsible for a core set of common data, much like an object that owns its data. A key difference between a component and an object in object-oriented programming is that components are more loosely coupled with their data. In fact, any implementation that accomplishes what is promised by the component interface specification is a valid offering.

Extracted component services must include everything required for compiling fully executable code. When compiled, these component services become immediately reusable by the original programs that contained the source code and by new applications.

Whether the new reusable business logic is classified as a business rule, or as a component service, it is now future-proofed from rapidly changing technologies. The core business logic has been modernized as clean, well-structured, independent, non-redundant business logic that can be accessed by existing or new applications.

If we had not followed stage one and the steps of stage two correctly, we would potentially have had future modernization challenges. We would also not have gained the total return on investment for the work done. The major benefits continue to accrue once restructuring has been done correctly.

Summary Following Stages One and Two

There are companies that will have reaped the benefits they are prepared to invest in as early as stage one. Those companies simply have legacy applications that need modernizing to ease their maintenance, and there is currently no desire to do more than that. Some of those companies may even have outsourced their legacy applications to firms that specialize in managing application maintenance. This may even be done to relieve the owning company from the maintenance burden while new systems, either purchased or developed, are being put in place. When companies choose to outsource their applications' maintenance, it is often with a contractual requirement to statistically prove code quality improvement as part of the contract. For an outsourcing company, this means stage one is a minimum requirement for contract success. Whatever the reason, as I pointed out earlier, the companies that achieve stage one can stop there and reap the full benefits of that stage.

If, in the future, companies that stopped at stage one decide to move on and restructure their business logic for reuse, they can resume following the stages detailed here. And, even without accomplishing stage two, COBOL programming code cleaned to consistently follow enterprise COBOL rules can be migrated to new technologies such as .NET by compiling with specialized COBOL compilers created for that purpose.

For those who have accomplished stage two, it is now possible to move on to stage three, or, transformation. Transformation enables migration to new platforms, and often includes translation to new programming languages (see Figure 2).

STAGE THREE: TRANSFORM RESTRUCTURED PROGRAMS

If stages one and two have been followed correctly, and stage two has resulted in "well-behaved" components, those components can now be reused. They can be accessed in their original coding language through standard Application Programming Interfaces (APIs), or from different languages and/or environments through specialized interfaces, or even translated from legacy application languages such as COBOL (or Natural, PL/1, or FORTRAN) to newer object-based languages such as Java. This is where defining components correctly in stage two will lead to successful translation even when the new language follows a much different paradigm. Although I have not defined the rules for defining "well-behaved" components here, I would recommend that you become educated about successful methodologies to follow and, if you are new to this arena, contract for the services of expert consultants to assist in the first few application restructuring projects.

The speed of technology changes, coupled with the growing requirement for intra-organization software collaboration, makes modernization a must for survival.

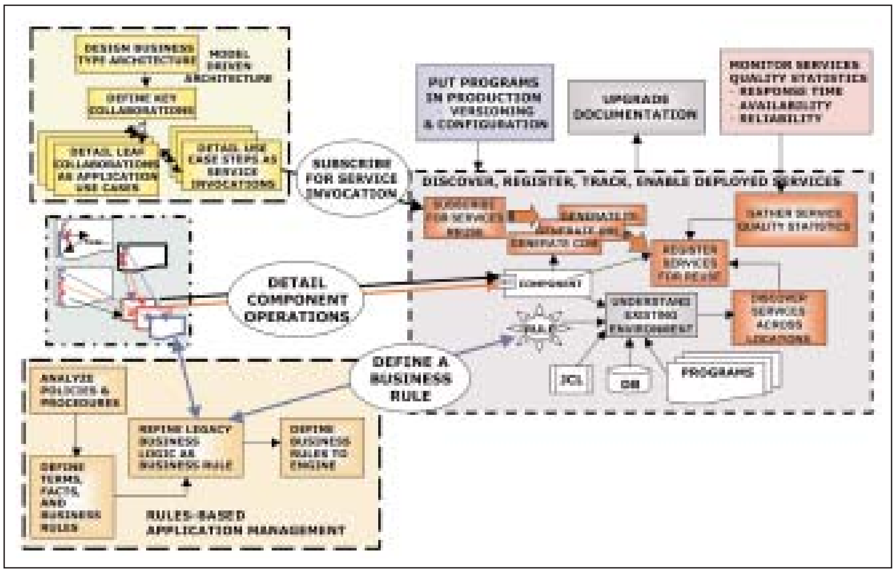
DESIGN BUSINESS TYPE ARCHITECTURE
MODEL DRIVEN ARCHITECTURE
DEFINE KEY COLLABORATIONS
DETAIL LEAF COLLABORATIONS AS APPLICATION USE CASES
DETAIL USE CASE STEPS AS SERVICE INVOCATIONS
SUBSCRIBE FOR SERVICE INVOCATION
PUT PROGRAMS IN PRODUCTION VERSIONING & CONFIGURATION
UPGRADE DOCUMENTATION
MONITOR SERVICES QUALITY STATISTICS
- RESPONSE TIME
- AVAILABILITY
- RELIABILITY
DISCOVER, REGISTER, TRACK, ENABLE DEPLOYED SERVICES
GATHER SERVICE QUALITY STATISTICS
REGISTER SERVICES FOR REUSE
DETAIL COMPONENT OPERATIONS
COMPONENT
UNDERSTAND EXISTING ENVIRONMENT
DISCOVER SERVICES ACROSS LOCATIONS
DEFINE A BUSINESS RULE
JCL
DB
PROGRAMS
ANALYZE POLICIES & PROCEDURES
REFINE LEGACY BUSINESS LOGIC AS BUSINESS RULE
DEFINE BUSINESS RULES TO ENGINE
DEFINE TERMS, FACTS, AND BUSINESS RULES
RULES-BASED APPLICATION MANAGEMENT

Figure 2 — Transformation and Reuse Management

Generating New Interfaces

If there is not a migration requirement and the original technology will continue, we can make component services available within almost any architecture. We can generate multiple, specialized interfaces such as:

- Interface Definition Language (IDL) for Common Object Request Broker Architecture (CORBA) access
- Web services-enabled by Simple Object Adaptor Protocol (SOAP), which consist of: XML, described by Web Services Description Language (WSDL), and discovered for reuse via Universal Description, Discovery, and Integration (UDDI). This provides a standard way to publish and discover Web services using XML, or as a Common Object Model (COM/DCOM) object directly accessible through Microsoft's object management technologies
- As a COM/DCOM object directly accessible through Microsoft's object management technologies.

The list of ways to access reusable business logic could, of course, continue. The key is that correct execution of stages one and two leads to many alternatives that are faster to implement in stage three.

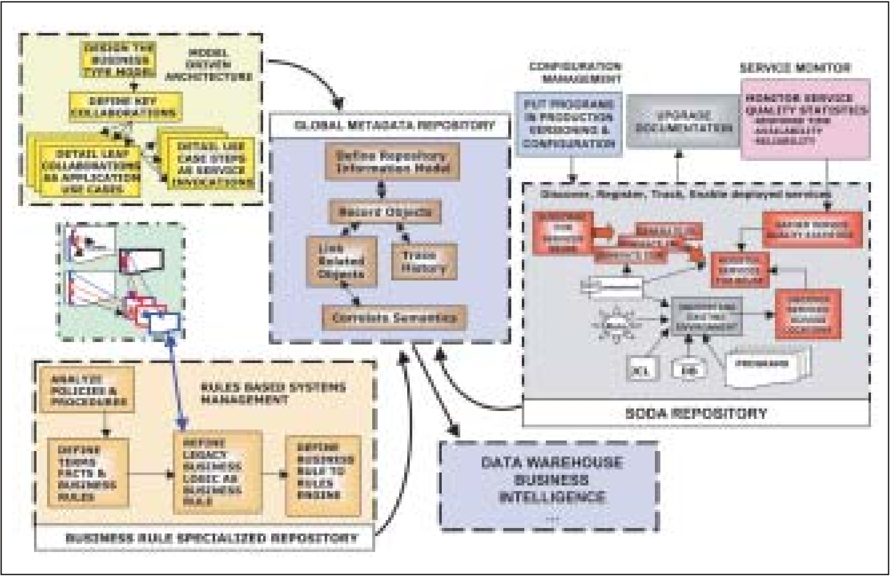
DESIGN THE BUSINESS TYPE MODEL
MODEL DRIVEN ARCHITECTURE
DEFINE KEY COLLABORATIONS
DETAIL LEAF COLLABORATIONS AS APPLICATION USE CASES
DETAIL USE CASE STEPS AS SERVICE INVOCATIONS
GLOBAL METADATA REPOSITORY
Define Repository Information Model
Record Objects
Link Related Objects
Trace History
Correlate Semantics
CONFIGURATION MANAGEMENT
PUT PROGRAMS IN PRODUCTION VERSIONING & CONFIGURATION
UPGRADE DOCUMENTATION
SERVICE MONITOR
MONITOR SERVICE QUALITY STATISTICS
SODA REPOSITORY
ANALYZE POLICIES & PROCEDURES
RULES BASED SYSTEMS MANAGEMENT
DEFINE TERMS FACTS & BUSINESS RULES
DEFINE LEGACY BUSINESS LOGIC AS BUSINESS RULE
DEFINE BUSINESS RULE TO RULES ENGINE
BUSINESS RULE SPECIALIZED REPOSITORY
DATA WAREHOUSE BUSINESS INTELLIGENCE
...

Figure 3 — Stage Four: Manage Reuse of Stable Business Logic

Translating Legacy Code to New Languages

Alternatively, if there is a migration strategy, such as moving from COBOL to Java, restructured code can be translated to Java and well-behaved component services into Java methods of Enterprise JavaBeans (EJB). Alternately, restructured code can be translated to other object-oriented code such as C++. In fact, restructured code eases the move to any other language or platform, and this is what we mean by future-proofing enterprise business logic.

The consequences of not following stages one and two correctly include an increase in the application's environment complexity, potential increase in business logic redundancy with its corresponding business risks, and a vastly more difficult time understanding the resulting deployment environment. Correcting a coding problem could become a nightmare when logic is redundantly hard-coded in numerous programs. In fact, without having re-engineered legacy code into well-behaved components, it is highly unlikely that an attempt to move to an object-oriented language would even be possible.

Companies that have added APIs to existing applications without modernizing the underlying programming code are beginning to realize that their clean-up and restructuring avoidance are coming back to haunt them. It may be possible to delay taking the plunge and fixing legacy applications, but eventually that avoidance will catch up and surpass the cost of doing it right the first time.

STAGE FOUR: MANAGE REUSE OF STABLE BUSINESS LOGIC

Having followed stages one through three, reaping the maximum rewards for your efforts is achieved by managing your newly modernized, reusable code (see Figure 3). This includes:

- Managing the life cycle of each reusable service from design through development, test, and production implementation
- Registering newly modernized and discovered executables
- Tracking correctly deployed reusable service versions
- Understanding shared service usage across platforms from distributed servers to mainframes
- Notifying interested subscribers of critical shared service details
- Discovering what exists and can be reused.

As distributed applications add the power of Web services to their solution sets, knowing what application is executing what service with global visibility becomes a must. If we thought a client/server environment was complex, the agility of an SOA magnifies that complexity exponentially. The rewards are great, and so is the requirement for following standards and applying the right methods and techniques along with powerful, enabling tools.

The consequence of not putting a software asset management system in place is the loss of control that can lead to reduced reuse and increased maintenance costs. If reusable services cannot be found and easily reused, or their stability, proof of quality, and correct configuration management are not in place, the motivation and capacity to benefit from the legacy application modernization effort will not be realized.

Bottom Line

Companies assess their IT software assets in the context of rapidly changing technologies. This requires new ways to manage business logic separately from the technology that provides access to that logic. The reward for restructuring correctly is that continuous modernization is easier. With the speed of technology changes, coupled with the growing requirement for intra-organization software collaboration, this is fast becoming a must for survival.

You can follow the right steps now to become more agile than your competitors and win more business. Additionally, you can future-proof your organization from technology changes. You can even improve your total cost of software ownership with your increased application quality. What you cannot do is achieve these benefits without doing the work the right way. Any other way is just putting Band-Aids on your current legacy applications.

Truth may seem an abstraction. The fact is that serious consequences will occur by ignoring legacy applications modernization. BIJ

This article is reprinted with permission from z/Journal *(August/September 2003).*

About the Author

Barbara Errickson-Connor is Director of Applicaions Product Management for ASG. Her programming and consulting background includes mainframe and distributed applications development, Information Engineering, Component-Based Development, Unified Modeling Language (UML), and legacy application transitioning methods and techniques. e-Mail: Barbara.Errickson-Connor@asg.com; Website: www.asg.com.