



We manage the technology that lets you manage your business.

# Stages of Legacy Application Modernization

## *Consequences of Transitioning Techniques*

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 identify and manage what is stable separately; a valuable outcome of a well-managed modernization project

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

leads to predictive, preventive maintenance that maximizes program quality and is a prerequisite for the stages to follow

includes

- understanding the existing applications,
- upgrading existing code to a consistent release level,
- removing coding anomalies

The consequences of incomplete accomplishment of Step One include

- a lack of assurance that resources are applied to programs in the order of greatest return
- more rework in the future if programs are not grouped into projects correctly
- difficulty in removing program coding anomalies according to consistent standards because of verb and other language variances between language releases
- the inability to recompile all modernized programs with the same compiler
- difficulty linking compiled programs together as cleaned applications because of compiler variances
- 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.

To successfully restructure existing code requires we:

- gather complete synonym sets across all programs
- filter programming logic that is used for
  - managing file or database access
  - screen management
  - general system management, such as CICS or IMS interaction
- rationalize likely redundant, hard-coded logic within the same program, or across programs
- discover what reusable logic should become business rules managed by a business rules engine or component services

## Stage Two – Restructuring Existing Code

The consequences of not identifying synonym sets includes

- incomplete recognition of common, redundant logic
- inability to fully recognize redundant data elements

The consequence of not filtering non-business logic is

- increased difficulty in identifying common logic that should become component services because of the “noise” caused by unused logic

The consequences of not rationalizing redundant logic include

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

The consequences of not extracting business logic into independently maintainable logic

- loss of rapid reaction to changes in business policies and procedures
- lack of future-proofing from rapidly changing technologies

Cleaned and re-architected code can be

- accessed in the existing coding language through standard Application Programming Interfaces (APIs)
- accessed from different languages and/or environments through specialized interfaces, like a COM, IDL, or WebService wrapper
- or even translated from legacy application languages such as COBOL (or Natural, PL/1, or Fortran) to newer languages, including object-based languages

The biggest impact on successful transformation is the correct application of the stages One and Two

Companies that have added APIs to existing applications without modernizing the underlying programming code are beginning to realize that their **cleanup 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

The **non-invasive** approach can offer **short term progress**, but eventually, the **invasive approach** of **physically recovering a quality architecture** will have to be tackled to attain long term application agility

## Reuse Management includes:

- life cycle management of each reusable service from design through development, test, and production implementation
- registration of newly modernized and discovered executables
- tracking correctly deployed reusable service versions
- understanding shared service usage across platforms from distributed servers to mainframes
- interested subscriber notification of critical shared service details
- Support for discovering what exists and can be reused

Consequences of not managing reuse of IT Assets includes:

- loss of control that can lead to reduced reuse and increased maintenance costs
- Unrealized motivation and capacity to benefit from the legacy application modernization effort
- Increased redundancy and related business logic inconsistencies over time
- Inability to reap application agility targets
- Implications of modernization project failure – when reuse is the real culprit

Application agility requires new ways to manage business logic separately from technology that provides access to that logic

The reward for restructuring correctly is that continuous modernization is easier

You can follow the right steps now to become more agile than your competitors and future-proof your organization from technology changes

You can even improve your total cost of software ownership with 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

## Market view

- **Business rule and component method mining support the move to .NET, Java and more agile application development**

## Business view

- **Managing rationalized business logic separately from applications**
  - Can keep content in synch with single source maintenance
  - gives business users power over their own business changes
  - frees IT for technology challenges

## Technical View

- **Software code modernization can be automated**
  - Understand code and clean coding anomalies, dead code, dead data
  - Restructure code – mine business logic as component methods and/or business rules
  - Transform code to new languages and/or to new technologies and platforms
  - Manage modernized & discovered reusable software

## Future View

- **Standards and managed process play a critical role**
  - ITIL, MOF, CWM, W3C, SOAP, XML, WSDL, OMG (Legacy Transformation group) ...
  - Work Flow, repositories, application portfolio management
  - ASG XMI standard and SOA tool framework