Integration of Legacy Systems in Software Architecture

Maria Wahid Chowdhury
Department of Computer Science
University of Victoria
PO Box 3055, STN CSC
Victoria, BC, Canada V8W 3P6
Email: mwchow@uvic.ca
Phone no: 250-477-9420

Muhammad Zafar Iqbal
Professor & Head,
Computer Science and Engineering Department,
Shah Jalal University of Science and Technology
Sylhet-3114, Bangladesh
Email: i_am_zafar@yahoo.com
Phone: 880-821-713491(Ext: 154)

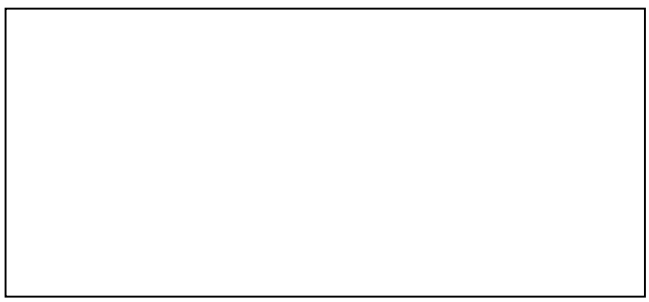
ABSTRACT
Most Companies have an environment of disparate legacy systems, applications, processes and data sources. Maintaining legacy systems is one of the difficult challenges that modern enterprises are facing today. The commercial market provides a variety of solutions to this increasingly common problem of legacy system modernization. However, understanding the strengths and weaknesses of each modernization technique is paramount to select the correct solution and the overall success of a modernization effort. This paper examines the strengths and the weaknesses of several modernization techniques in order to help engineers to select the right technique to modernize a legacy system.

Categories and Subject Descriptors
D.3.3 [Management]: Life Cycle

General Terms
Design.

Keywords
Legacy system, reengineering, integration.

1. INTRODUCTION
Software systems are critical assets for companies and incorporate key knowledge acquired over the life of an organization. Companies spend a lot of money on software systems. To get a return on that investment, these software systems must be usable for a number of years. The lifetime of software systems is very variable though many large systems remain in use for many years. Organizations rely on the services provided by these systems and any failure of these services would have a serious effect on the day to day running of business. These old systems have been given the name legacy systems.



Legacy systems incorporate a large number of changes continuously to reflect evolving business practices. Repeated modification has a cumulative effect on system complexity. Usually, a legacy system has to pass through many developers evolving over decades to satisfy new requirements. These systems are matured, heavily used, and constitute massive corporate assets. Today, legacy systems must be designed to be capable of integrating with other applications within the enterprise. However, scrapping legacy systems and replacing them with more modern software involves significant business risk. Replacing a legacy system is a risky business strategy for a number of reasons [4]: a) There is rarely a complete specification of the legacy system. Therefore, there is no straightforward way of specifying a new system, which is functionally identical to the system that is in use, b) Business processes and the ways in which legacy systems operate are often inextricably inter-twined. If the system is replaced, these processes will also have to change, with potentially unpredictable costs and consequences, c) Important business rules may be embedded in the software and may not be documented elsewhere, d) New software development is itself risky because there may be unexpected problems with a new system. In general, a legacy system has following characteristics:

1. High maintenance cost.
2. Complex structure.
3. Obsolete support software.
4. Obsolete hardware.
5. Lack of technical expertise
6. Business critical.
7. Backlog of change requests.
8. Poorly documented.
9. Embedded business knowledge.
10. Poorly understood.

2. ARCHITECTURE DESIGN CONSTRAINTS AND ISSUES
A legacy system significantly resists modification and evolution to meet new and constantly changing business requirements. Legacy systems have not been designed to accommodate changes because of the following reasons:

- The Legacy system was designed for the immediate needs. When constructed, it was not expected that it would be in service for many years.
- There may be some constraints (as for example: low

cost) that were satisfied by the development of legacy system.

Before making any change, it is necessary to assess the feasibility of making changes and to determine the impact of the changes on the rest of the system. Due to the complex structure of legacy systems, they require considerable effort to understand. The challenge in the integration of legacy systems is to understand the functionality, design, operation and performance of the system and to anticipate the types of changes that will be required over the integration steps. After years of maintaining, upgrading and enhancing the legacy system, the user manuals and system design documentation are often out of date, inaccurate, and fail to reflect the current system's capabilities and operations. As a result legacy system architectures are often poorly documented. This emerges as a new kind of problem when integrating a legacy system into an overall system architecture design and specification.

Architectural issues include gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among alternatives (an architectural style). [1]. Some of the constraints found in integrating legacy system are on how to deal with components, connectors, semantics or topology.

Perhaps the most obvious component constraint relates to the component types allowed by the architectural style. There are also a number of constraints when one component needs or manages to collaborate with other components. This includes collaborator location and availability, information such as transfer protocol, data format, schema and content (including method signatures and interfaces) as well as architectural assumptions. Furthermore, there may be constraints on the types of access that the components must provide, e.g. interface access to the database, to the application logic, or to internal objects of the component. We must be careful about incompatible data and file formats, hardware incompatibility, software dependency on hardware, proprietary protocols and networks when integrating a legacy system. Other problems posed by legacy systems are the absence of clear interfaces, and insufficient encapsulation.

In summary, the most important issues to consider when integrating a legacy system are:

1. Data: how data is going to be integrated. That is, identify and link records on the same subject or other entity in disparate systems. One solution is to use metadata. However, this leads to one problem; because the same metadata can have different meanings in different applications, companies must develop custom interfaces between applications. Another approach is to perform data integration at the semantic level (based on actual content, not the metadata).
2. Connectivity to each component in the architecture.
3. Routing of messages between components.
4. Validation and transformation of data into and out of each application.
5. Interfacing with each application based on its own syntactical and semantic requirements.
6. There exist some security issues that must be addressed. We must pay attention to the legacy system's security mechanisms.
7. Conformity to organizational and business process structures. The legacy system must be adapted to new business policies.

The above issues make us wonder: What is the format in which data is interchanged? How does the application interpret a message it receives? What is the impact of changing a message definition?

3. ARCHITECTURE DESIGN STRATEGIES

There are basically two approaches to reuse legacy systems: reengineering and integration. Re-engineering means re-structuring. Re-structuring a legacy system's code requires that the system and code are well documented and/or can be automatically analyzed and transformed by an automatic process. Re-engineering a system is slow. Integration is faster and cheaper than re-engineering. To integrate a legacy system, we must define the role of each subsystem, define interfaces for each subsystem, and build an object wrapper for each subsystem.

An integration strategy can be intrusive or non-intrusive. An integration strategy that requires knowledge of the internals of a legacy system is called intrusive (white box) integration, while integration strategy that requires the knowledge of external interfaces of a legacy system is called non-intrusive (black box) integration. A connection to an application system is considered non-intrusive if an existing entry or exit point is used. If application source code is modified, the connection is considered intrusive. Intrusive connections are used when custom coding is developed to handle specific application needs or to increase performance. Non-intrusive connections are recommended for use if the information required from the application is already available from an existing interface and the transaction volume is low to moderate. Intrusive integration requires an initial reverse engineering process to gain an understanding of the internal system operation. After the code is analyzed and understood, intrusive integration often includes some system or code restructuring. There are two major approaches for legacy systems integration: application integration and data integration.

3.1. Application integration

The guiding philosophy behind this approach is that applications contain the business logic of the enterprise, and the solution lies in preserving that business logic by extending the application's interfaces to interoperate with other or sometimes newer applications. There are some major classes of application integration solutions given below:

User Interface Modernization: The user interface (UI) is the most visible part of a system. Modernizing the UI improves usability and is greatly appreciated by final users. A common technique for UI modernization is Screen scraping, as shown in Figure 1, consists of wrapping old, text-based interfaces with new graphical interfaces. [2]

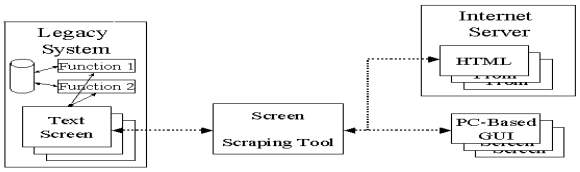


Figure 1. Legacy System Wrapping Using Screen Scrapping

Point-to-point integration: In Point-to-point integration, communication channels are developed between each pair of applications. Such a solution is expensive, because the number of

interfaces required grows exponentially. With n applications, n*(n - 1) interfaces may be required since each application may need an interface with other application. The impact of minor changes in communication requirements and that of adding a new application is significant. Maintenance is clearly a problem due to the number of nodes.

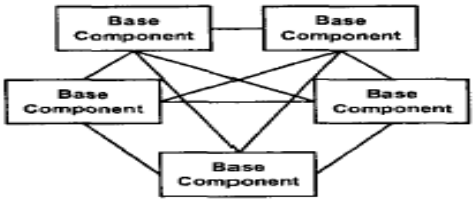


Figure 2 (a). point-to-point integration

Message routers: Point-to-point integration exponentially increases the number of interfaces. This can be reduced to a linear increase through the use of middleware – message-oriented or based on the Common Object Request Broker Architecture (CORBA).

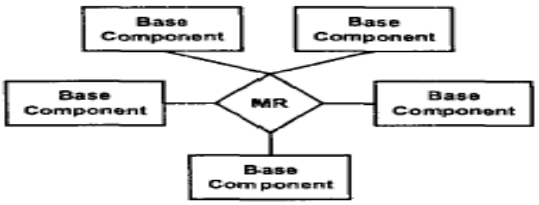


Figure 2(b). message router

The solution requires interfacing each application to the message bus through an adapter. Each application has only one programmatic interface, the message bus. Applications communicate by publishing a message to the bus, which delivers message to those who subscribe. Subscription topics of queues let subscribers receive only messages they are interested in. The Middleware product may also provide value-added services such as guaranteed delivery, certified delivery, transactional messaging, message transformation (using brokers) and so on. [1]

CGI integration: The Common Gateway Interface (CGI) is a standard for interfacing external applications with information servers, such as HTTP or Web servers. Legacy integration using the CGI is often used to provide fast web access to existing assets including mainframes and transaction monitors. [2]

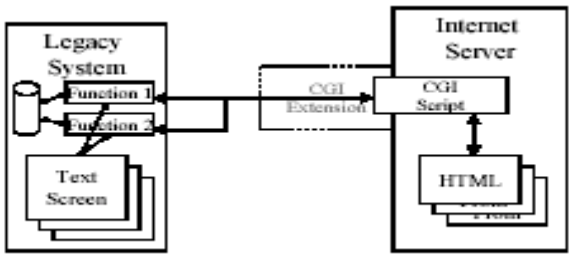


Figure 3. CGI Integration

3.2. Data Integration

The guiding philosophy behind integration of data is that the real currency of the enterprise is its data. The implied business logic in the data and metadata can be easily manipulated directly by applications in the new architecture of the enterprise. Some data integration solutions are described below:

XML Integration: The Extensible Markup Language (XML™) is a broadly adopted format for structured documents and data on the Web. [2]

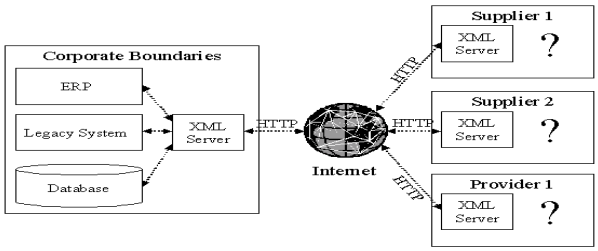


Figure 4. XML Wrapping

XML is a simple and flexible text format derived from standard generalized markup language (SGML) (ISO 8879) and developed by the World Wide Web Consortium® (W3C). XML is expanding from its origin in document processing and becoming a solution for data integration.

Data replication: Database replication is the process of copying and maintaining database objects in multiple databases that make up a distributed database system.

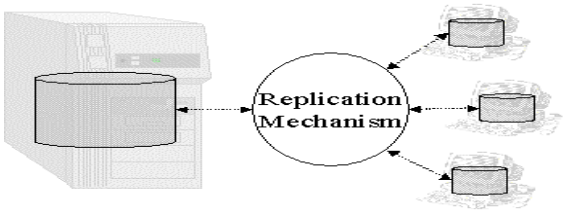


Figure 5. Data Replication

Replication provides users with fast, local access to shared data and greater availability to applications because alternative data access options exist. Even if one site becomes unavailable, users can continue to query, or even update, data at other locations. Database replication is often used to enable decentralized access to legacy data stored in mainframes.

4. EXAMPLE OF GENERIC ARCHITECTURES

Java J2EE Connector architecture: Java J2EE Connector architecture defines a standard set of services that allow developers to quickly connect and integrate their applications with virtually any back-end enterprise information system. These services are supplied as "plug-in" connectors.

Sun ONE: Sun Open Net Environment (Sun ONE) is Sun's standards-based software vision, architecture, platform, and

expertise for building and deploying Services on Demand. The network is all about servicing the communities, stockholders, customers, and employees.

OMG MDA: Computing infrastructures are expanding their reach in every dimension. New platforms and applications must interoperate with legacy systems. MDA is a new architectural approach that provides companies with the tools necessary to integrate all the various middleware technologies (such as CORBA, EJB, XML, SOAP and .NET). MDA addresses the complete life cycle of designing, implementing, integrating and managing applications and data using open standards. MDA provides an architecture that assures portability, cross platform interoperability, platform independence, domain specificity, and productivity.

B2B: B2B integration or B2Bi is basically about the secured coordination of information among businesses and their information systems.

EAI: As the need to meet increasing customer and business partner expectations for real-time information continued to rise,

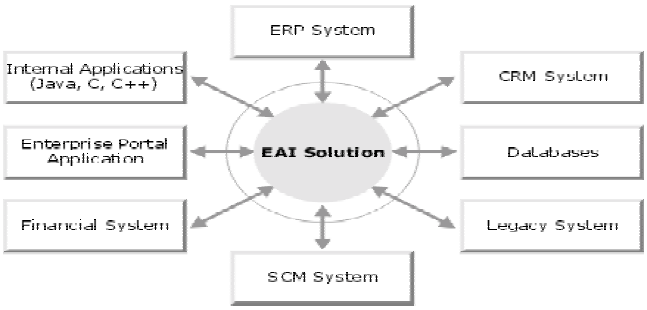


Figure 6. Enterprise Application Integration

companies are forced to link their disparate systems to improve productivity, efficiency, and, ultimately, customer satisfaction. EAI is the process of creating an integrated infrastructure for linking disparate systems, applications, and data sources across the corporate enterprise.

CORBA: CORBA allows applications to communicate with one another no matter where they are located or who has designed them. With CORBA, users gain access to information transparently, without them having to know what software or hardware platform it resides on or where it is located on an enterprises' network. This characteristic makes CORBA an excellent technology to integrate legacy systems.

XML: XML improves the web functionality by providing more flexible and adaptable information identification (tags).

SOAP: The Simple Object Access Protocol (SOAP) is a standard that specifies how two applications can exchange XML documents over HTTP.

Java RMI: Java Remote Method Invocation (RMI) enables the programmer to create distributed Java technology-based applications in which methods of remote java objects can be invoked from other Java virtual machines, possibly on different hosts. Java RMI is well suited to be used in the application level of integration.

JDBC: JDBC technology is an API that lets user access to virtually any tabular data source from the Java programming language. The JDBC API allows developers to take advantage of the Java platform's "Write Once, Run Anywhere™" capabilities for industrial strength, cross-platform applications that require access to enterprise data.

DCOM: The Distributed Component Object Model (DCOM) is a protocol that enables software components to communicate directly over a network in a reliable, secure, and efficient manner.

5. CONCLUSION

There are different approaches to the modernization of legacy assets including reengineering (white-box) and wrapping (black-box). Before starting any legacy modernization effort, every possible option should be considered and business and strategic factors also need to be considered for ensuring long-term success. Present-day systems are the potential source of future legacy problems. To eliminate future legacy problems from present-day systems, systems should be built by using modular engineering and configurable infrastructure.

6. REFERENCES

[1] Architectural Integration Styles for Large-Scale Enterprise Software system, By Jonas Anderson, Pontus Johnson, Department of industrial and Control Systems. Royal Institute of Technology, Sweden.

[2] A Survey of Legacy System Modernization Approaches: http://www.sei.cmu.edu/publications/documents/00.reports/00tn003.html

[3] OMG Model Driven Architecture, http://www.omg.org/mda/

[4] Software Engineering, Sixth Edition, By: Ian Sommervile.