Indianjournals.com

Software Reuse in the Evolution of an E-Commerce System: A Case Study

Shihong Huang*, **James J. Mulcahy**

Computer Science & Engineering, Florida Atlantic University
shihong@cse.fau.edu, jmulcah1@fau.edu

ABSTRACT

The engineering and reengineering of software interfaces between dissimilar systems is a process that is constantly evolving with respect to the techniques applied and the lessons learned from their use. While many strategies and models for such development exist today, there are times that no single defined approach appropriately fits the requirement for all real-world situations. This is particularly true of the integration and sharing of data among modern e-commerce software systems and large, mature multi-channel legacy systems. This paper details the initial and subsequent evolutions of one such e-commerce software interface. Lessons learned from each evolution were used to improve upon the next, ultimately resulting in a highly maintainable and scalable product.

Keywords: software maintenance and evolution, software reuse, software systems integration, reengineering, legacy software systems, COBOL

1. INTRODUCTION

Integration of an increasing volume of online purchase data from relatively young web-based merchants into the databases of more mature multi-channel vendors can be a software engineering and maintenance challenge, particularly for those business-to-consumer (B2C) or business-to-business (B2B) vendors that use complex legacy enterprise resource planning (ERP) software systems. Bringing together an online merchant and an "offline" vendor in order to reconcile the details of online consumer purchases requires the transformation of data from one form to another. Business logic at the heart of the solution is required to audit and manipulate the data. The end result is ultimately the rendering of data in the vendor databases based on records received from one or more online merchants or payment providers. Automation of this process is the primary goal of any integration solution.

There is an inherent incompatibility between the modern "always on" real-time architecture of today's e-commerce storefronts and payment providers and the architecture of their more rigid counterparts. Older mainframe and minicomputer-based systems tend to process commerce and related financial data in a procedural, scheduled fashion, and do not necessarily operate in real time. This motivates the need for loose coupling of the interface between two or more systems to remove timing from the equation. Additional challenges arise in the process of the software engineering effort itself. Interfacing constantly-evolving online sales channels with legacy multi-channel enterprise systems often requires a rapid development effort geared toward the "time to market" of the solution. The legacy systems involved are often originally designed for direct marketing and mail-order sales and must therefore be modified to adapt to the rapidly changing landscape of commerce. Typically, legacy enterprise systems tend to evolve very slowly. However, the proliferation of competing online merchants and ever-changing web-based software technologies requires this paradigm to change. It also requires a certain amount of flexibility to be incorporated into any efforts to integrate dissimilar systems. This paper will address the software engineering and reengineering challenges of a particular software solution developed and enhanced over a three (3) year period. The solution was designed for use with the enterprise system of a major market leader in multi-channel direct commerce software solutions. The result of the solution was the integration of the legacy ERP system with several popular online e-commerce merchants. Each successive evolution of the solution revolved primarily around the re-implementation of the requisite software applications using new hardware and database architectures. Each evolution was performed for a different vendor using multiple versions of a common ERP.

The rest of this paper is organized as follows: Section 2 describes the background for the case study and related work in the area of software maintenance and evolution of legacy and modern systems. Section 3 outlines the requirement for the real-world solution that is studied by this paper, along with the circumstances that precipitated the need for the solution. Section 4 details the initial engineering effort, followed by two increasingly improved evolutions to scale the resulting software solution. Section 5 evaluates the engineering approach and identifies the strengths and weaknesses of the process. Finally, Section 6 summarizes the conclusions drawn from the series of evolutions and offers prospects for related future work.

2. BACKGROUND AND RELATED WORK

This paper focuses on two primary areas of the reengineering process: the encapsulation and reuse of available source code, and the creation of new source code to be highly platform-independent and fault-tolerant. Development techniques were used to produce a software solution that would be more easily maintained, enhanced and reengineered according to future requirements. Reuse of source code, particularly when working with large, complex legacy software systems, should be of paramount importance when engineering interfaces to other external systems. H. M. Sneed [25] asserted that fewer lines of new code and fewer changes to existing code reduced both the cost and risks of this type of reengineering. A decade or more ago, this concept was less of a concern. Software was expected to be obsolete within mere years of creation, requiring replacement. This assumption is now less feasible as software systems become more entrenched as successful business solutions, or so complex that they cannot be easily reengineered. It is no longer uncommon for legacy systems to contain working source code that is decades old [31].

The source code reuse technique serves multiple purposes. The primary purpose is to significantly speed development of new software interfaces by avoiding duplicate efforts when developing new source code [5]. A secondary, but no less important purpose, is to leverage the reuse of source code in increasing the reliability and maintainability of the final product [9][18]. The use of existing, “trustworthy” software components reduces the length and scope of many phases of the software development life cycle, including unit testing and acceptance testing. It can also reduce or eliminate the “breaking” of functionality that previously had no faults. When integrating legacy and modern systems, reuse may come in the form of entire standalone or callable applications. It may also entail reusing sections of existing source code that represent specifically useful logic. While such code may be cloned and “pasted” into new applications, there are times that it may be encapsulated and stored in separate external “copy libraries” or “includes”. While the terminology may vary by programming language or dialect, this commonly means that sections of source code are stored in individual files that are later merged dynamically by compilers into the programs that reference them. These code segments may include actual business logic or may include details related to internal and external data structures. They may also include other standardized variable storage areas that, in turn, may further map to particular database datasets or external file formats.

It is also important to consider that any new software should itself be coded in such a way that it is also encapsulated, “wrapped” [27] or otherwise organized to be reusable where possible. Canfora, Fasolino and Tortorella [6] studied the approach of creating a model or template when reengineering COBOL programs to be reusable in future contexts. The technique studied by this paper encapsulated source code at a much more granular level, within the individual applications themselves. Appropriate segregation and/or encapsulation of source code allows for an easier transition to new combinations of hardware and database architectures. This is especially true for source code related to machine-dependant activities. These activities include access to databases, file systems, user interfaces, and operating systems. This functionality can be identified and segregated from the core of the interface that contains the auditing and reconciliation (AAR) business logic (Figure 1: Segregating and Encapsulating by Functionality). Transitioning of systems to more modern business models and architectures benefit from these techniques [25]. While designing or modifying systems to be evolvable can add development expense, it is a necessity for legacy systems that are expected to be in service for the foreseeable future [24].

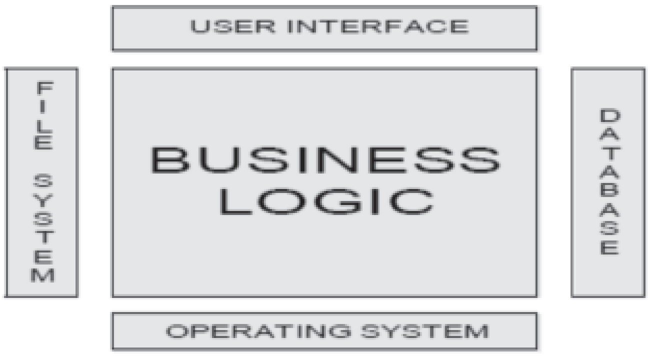


Figure 1: Segregating and Encapsulating by Functionality

M. Lehman has addressed the topic of future maintenance requirements in this context by mentioning that anticipated requirements of solutions, once developed, include the "changing properties of the operational domain, advancing technology, desire for business growth or the emergence of competitive products" [16].

3. REAL-WORLD REQUIREMENT FOR INTEGRATION

The type of e-commerce transaction given focus by this paper is not that between the consumer and product vendor, but rather between the online merchant or payment provider and the actual vendor fulfilling

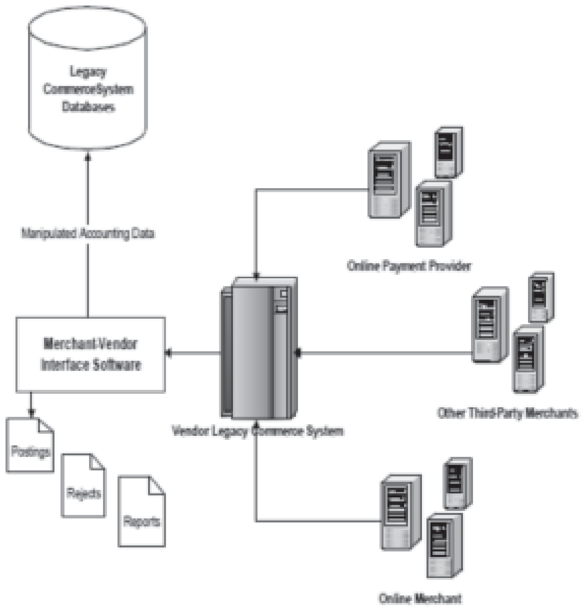

Legacy CommerceSystem Databases
Manipulated Accounting Data
Merchant-Vendor Interface Software
Postings
Rejects
Reports
Vendor Legacy Commerce System
Online Payment Provider
Other Third-Party Merchants
Online Merchant

Figure 2: Flow of Data from Merchant to Vendor

the order to the consumer (Figure 2). It is increasingly common for initial purchase information and subsequent financial transactions to be handled by a party other than the vendor who ships the goods to the consumer [33]. This section describes one such real-world merchant-vendor relationship. The solution was implemented for a well-known market leader in multi-channel direct commerce enterprise software systems. There was a specific requirement for the interface to automatically handle the reconciliation of a high volume of financial data exchanged between the legacy enterprise system and online providers that included Amazon.com, Google Checkout™ and Paypal.

3.1 Merchant and Vendor Relationship

With current technology, vendors are able to market to and derive sales from a variety of external commerce channels while paying online or otherwise remote providers a commission-based income for the handling of the initial interaction with and payment by the online consumer. This type of interaction typically takes the form of an internet website used by the consumer to shop for and purchase products. Examples of this type of interaction include Amazon.com and eBay.com. Another type of interaction occurs when an online payment provider collects purchase and credit card payment data from the consumer and handles the actual billing process. Examples of this type include Google Checkout™ and PayPal [22].

3.1.1 *The Online Merchant*

Internet merchants and online payment providers typically use more modern, flexible software and hardware systems. This is due largely to the relatively young age of the e-commerce community. The use of modern technology provides for the ready exchange of data between vendors and fulfillment providers, often with files composed in the now-familiar eXtensible Markup Language (XML) and comma-delimited (CSV) formats. The data is delivered between the merchant and the vendor in a variety of ways. One example is the use of a Simple Object Access Protocol (SOAP) server that is owned and maintained by the vendor. This architecture provides a framework for the delivery and validation of XML documents with appropriate order, fulfillment, confirmation and financial information. A standard XML schema is established that allows the vendor and online provider to efficiently communicate such data. In this case study, the SOAP/XML solution was used to establish a connection between Amazon.com and the vendor(s) using the interface. Another technique is the use of one or more FTP servers from which reconciliation documents can be retrieved "on demand" or at scheduled intervals by the vendor. These documents are typically created in comma-delimited (CSV) form. Paypal/eBay and Google Checkout™ are examples of entities that use this format. The implementation of the solution studied by this paper used the FTP technique to exchange data with these particular providers. Data-sharing techniques such as SOAP and FTP serve to decouple the "always-on" provider/merchant from the actual vendor, who may or may not have a software system waiting for transaction activity in "real time". In this manner, it is not necessary for the vendor and merchant to directly be aware of the internal proprietary data format of the other party when trading information [21].

3.1.2 *The Legacy Vendor*

Vendors that provide purchased items to the consumer are often mature multi-channel commerce businesses with large legacy software systems, using business models and data organization techniques that can be far less flexible than their expanding internet-based partners. Many of these vendors are existing direct marketing and catalogue-based companies that have expanded into the ecommerce arena. When a merchant transfers sale revenue to the vendor, it typically withholds an agreed-upon commission or processing fee in addition to other consumer-originated adjustments. These adjustments can take the form of credits for returned products or cancelled orders. The vendor is typically supplied with a reconciliation file containing the monetary details of transactions handled by the merchant on behalf of the vendor. This file may be available at regular intervals that can vary from hourly to weekly, or any other frequency agreed upon by the merchant and vendor. From the reconciliation data, the vendor is able to audit the amounts supplied by the merchant, to verify accurate calculations. The vendor is then able to reconcile the order information with records in the vendor's own database. The final step is the posting of the payments and commission write-offs to the appropriate financial datasets, thus completing the reconciliation of the order.

3.2 Requirement for Integration

Increasing sales volume causes the manual process of reconciling financial data to become quickly cumbersome, time-consuming and error-prone for the vendor. The increased volume can come from the addition of new commerce channels beyond the typical brick-and-mortar, telemarketing and catalog sales venues. Internet-based storefronts, for example, are open for business around the clock, vastly increasing marketing exposure and the potential for sales. From the increase in processing requirements arises the need for an automated auditing and reconciliation interface to make the process more efficient and less error-prone. The stakeholder goal is for the integration solution to accurately carry out the tasks of auditing and posting of the appropriate values to the accounting areas of the vendor's enterprise system, as illustrated in Figure 2. Further, it is important to automatically detect and single out any transactions that fail the auditing process. These problem transactions cannot be automatically settled by the interface and thus require manual attention by accounting personnel. The interface described by this case study was ultimately required to handle several different online merchants while remaining anchored on one side of the transaction to a particular legacy enterprise system. The enterprise system referred to by this paper is over fourteen (14) years old as of this writing. It includes at least eleven (11) distinct databases, and the software repository contains over one million lines of COBOL and C source code, making up more than 1800 individual software applications.

4. IMPLEMENTING THE SOLUTION

To meet the evolving demands and requirements of the vendor and its online partners, the software solution studied by this paper progressed through a series of development iterations after its initial deployment. While largely related to the reengineering of the software components to new environment architectures, maintenance tasks were also performed to adjust and improve the business logic and software design model with future maintenance and scalability in mind. This section will further detail the procedures followed by the software engineers tasked with developing the initial solution. It will also provide details of each subsequent evolution, describing the outcomes and their effects on the next iteration.

4.1 First Evolution – Single Merchant and Vendor

The first evolution of the merchant-vendor interface was to be implemented as a custom solution for a single vendor requiring an interface for a single online merchant (Amazon.com). The solution was specifically designed for the Microsoft Windows operating system using Microsoft's SQL Server database. The solution was to be developed without the protracted and thorough specification and analysis phase that is typical of established software development models. It was anticipated that live data files from the online merchant would adequately expose any unexpected conditions requiring modification to the software. Exceptions to the business model found in this manner were expected to be relatively minor. Little documentation was available from the onset describing the expected inbound data stream, and thus no significant effort was spent designing a formal specification. This shortcoming is typical of the integration of dissimilar or previously unrelated software systems. This arises largely from the opportunistic nature of many integration projects. Data that is generated or consumed by a system may not be officially published or documented for external use. Instead it is often an internal artifact of the system that is expected to experience changes in form and content over time. “Most of the software in regular use in businesses and organizations all over the world cannot be completely specified.” This can be especially true of high-volume merchant systems, both legacy and modern. As these systems evolve and expand, there can often be a lag or disconnect between how software actually behaves and how it is documented or expected to behave [16]. Financial interfaces integrating merchant and vendor systems are not immune to this “disconnect”. These types of interfaces often require regular maintenance and modification the longer they are in service. One or more of the parties independently evolve their own business systems over time, which may at times affect the interfaces between them. There were no initial plans to implement the software solution for other operating systems or databases. Indeed, the first evolution was considered a one-time custom solution for a particular merchant-vendor combination.

4.1.1 *Engineering the Solution*

The solution was coded using the COBOL programming language. The choice of language enabled the reuse of source code available from the existing software repository of the legacy commerce system (Figure 3). This choice also took advantage of the well-established data processing strengths of the COBOL language. Much of the reusable source code was related to the mapping of data to legacy datasets that were to be queried and modified by the auditing and reconciliation (AAR) component of the solution. Code reuse created a tight coupling with the legacy enterprise system, but this was considered an acceptable dependency according the custom nature of this software solution. Many of the details related to the incoming data stream were reverse engineered by the developers due to the lack of available official documentation. Reverse engineering in software is conceptually the same as the more familiar application in hardware, with similar approaches and goals [8]. In this case, details of the data format were derived from the data itself. The upstream provider of the reconciliation transactions supplied several files containing current “live” data, allowing the developers to derive the information needed to develop the solution.

Development included more than three thousand (3,000) lines of new source code, in the form of one new application and several new “copy libraries”. The copy libraries contained source code specifically related to the seven (7) legacy system datasets that were to be accessed and modified by the AAR component. Copy libraries reused from the enterprise repository added approximately 3,700 additional lines of source code to the resulting application.

4.1.2 *Documenting the Process*

The software development process from inception through the final release was documented by the software engineers. Details that were recorded included the amount of time consumed by each development phase. Other more granular details were also recording, including data related to specific unit tests performed to verify the business logic, lists of data and source code dependencies and a log of all problems and enhancements that required changes to the software.

Documentation was further augmented by the coding of a logical “trace layer” into all software components of the solution. This diagnostic coding was initially imbedded to visually mark key locations, variable values and logic paths while developing the software. The data produced by this coding was displayed to the console and written to a log file at execution time. This allowed for instantaneous or delayed review and diagnosis. The functionality was easily activated and deactivated by developers and users alike. It was designed to be left imbedded in the deployed solution without affecting its performance or complexity. The tracing logs would later provide detail enabling efficient identification and re-creation of data and logic errors. They resulted in fewer resources required to research and address logic problems found during the

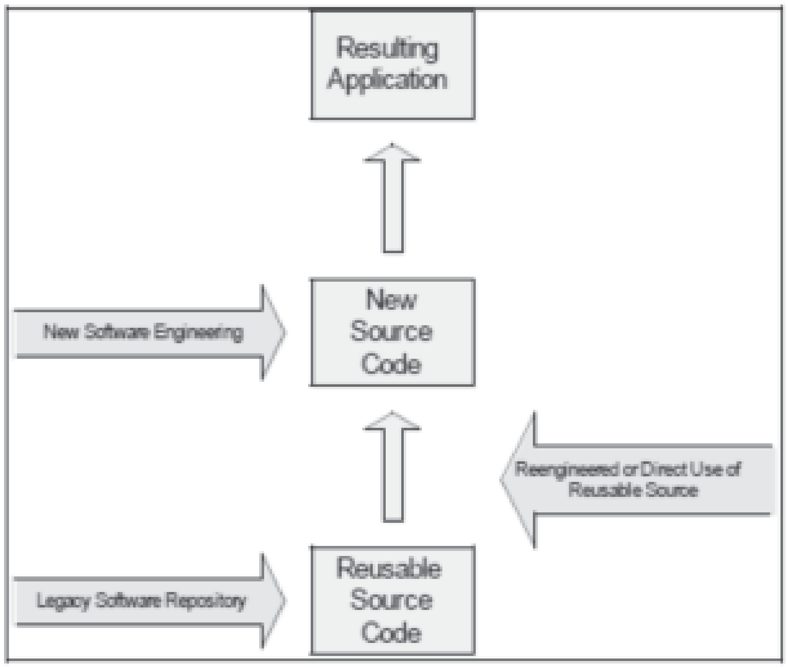


Figure 3: Combining New and Reusable Code

development, unit testing and integration testing phases of the project. The trace layer, coupled with source code designed to trap all possible logic error conditions, greatly increased the fault tolerance of the product by significantly reducing program crashes and data corruption issues.

Tracing logs provided additional documentation benefits. Capturing of logs during unit and integration testing added to the useful documentation of the software components of the solution. Any future maintenance to the software would benefit from these logs, especially during regression testing and redeployment phases.

The total development effort of the first evolution of the software spanned 175 hours over an eleven (11) month period. The level of detail of the documentation generated during this evolution allowed for future measurement of the successes and failures of the project.

4.1.3 *Understanding the Results*

Post-release analysis of the project showed that the initial development stages of the project were accelerated due to the minimal time spent planning and specifying project requirements. However, there was a price to pay for these shortcuts, in the form of a significant number of post-release modifications. The lack of resources expended by both the vendor and the developers during the early stages of the project ultimately resulted in significant delay of the vendor acceptance and deployment of the final product.

The development time consumed delivering the initial application accounted for only 37% (65 hours) of the total development effort. However, in excess of 110 hours were invested after the official release (Figure 4) due to the many change requests that followed. Most of the overage was found to have been spent addressing and resolving deficiencies in the initial business logic model. The resulting faults had not been anticipated by the parties involved in the project. The interface was subsequently tested with "live" data files as they became available from the merchant. More unanticipated conditions were identified as a result of testing with this data. These faults required frequent changes to the business logic after the initial release.

The additional effort was mitigated by the relatively quick release of patched versions of the code as each new fault was identified. This was made possible by the previously described logic tracing functionality. While the successful implementation of the solution was delayed several times, the end result was ultimately the deployment of a software interface able to automate the auditing and reconciliation of several thousand Amazon-originated orders per week for the vendor.

4.1.4 *Evaluating the Results*

The initial evolution of the product can be technically considered a failure, if based on the extended post-release acceptance period. In excess of nine (9) months elapsed between initial release of the solution and the final version accepted by the client. During this time, the software was modified a dozen (12) times, each to repair an unanticipated logic condition. Development of solution consumed nearly three (3) times the effort originally estimated, and the target date was pushed and missed several times.

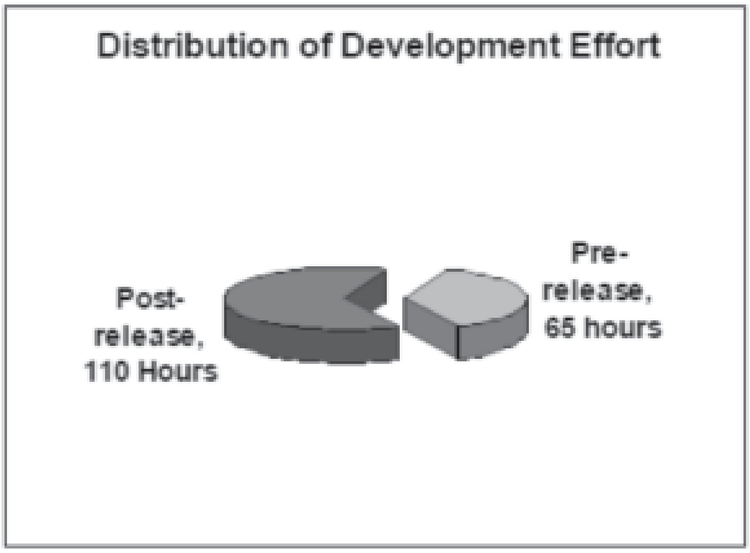


Figure 4: Distribution of First Evolution Resources

The resulting implementation of the software solution was, however, deemed successful. The project was neither canceled nor scrapped in favor of another solution. The resulting software solution is still in use by the vendor as of the writing of this paper. Mitigating the extended development and release period was the overall stability of the resulting software. No problems have been reported subsequent to the acceptance of the solution by the vendor. Several hundred thousand orders have been processed during this period, demonstrating the successful scalability of the design to “live”order volumes.

4.2 Second Evolution – Scaled and Migrated

The software solution provider was later tasked with producing a newer version of the merchantvendor interface to be based on the successful deployment of the first. The new iteration was to be implemented for a different vendor than the first evolution. The vendor was using the same legacy multi-channel enterprise system and communicating with the same online merchant. The solution was to be deployed upon a different hardware and database architecture. The vendor’s enterprise system was installed and in operation on an HP3000 platform using the TurboImage database and the MPE-iX operating system. This was a significant departure from the initial implementation, which had been deployed upon the Microsoft Windows platform and using the SQL Server database. The new solution was to be scaled to add support for multiple online merchants and providers (Google Checkout™, Paypal), in addition to the existing support for Amazon. It was crucial that the existing interface be able to process reconciliation data from additional merchants without extensive reengineering. The added functionality was to be implemented in multiple phases spread out over several months of the project.

4.2.1 *Reengineering the Solution*

The initial focus of the second evolution was whether to scrap the original custom solution and start anew with a new software design, or to salvage as much of the logic as possible from the original Windows-based source code. The decision was made to reengineer the previously developed solution using the original Windows-based source code as a base. Part of the decision was based on the high expertise level of the project members with the newer platform. The members of the project had a strong grasp of what was required for the new evolution and what had already been accomplished in the first. This domain knowledge allowed for easier identification of additional reusable source code, without extensive research and analysis. It also enabled the creation of new source code based on the previously successful design model.

The original business logic was automating the auditing and reconciliation of several hundred thousand online purchases for the original vendor by the time development of the second evolution began. Therefore, much of the source code was expected to be reusable in the newer version with only minor changes. The user interface logic for the main auditing and reconciliation (AAR) component was decoupled from the application itself and handled at the Job Control Language (JCL) level that is native to the MPE-iX operating system. This allowed the application to behave more like a “black box” than the previous evolution, with required component input and output being handled generically by the JCL at execution time. The developers designed the original Windows-based version using a method that segregated database-related source code into copy libraries. These copy libraries were in turn referenced and used by the AAR component. This design for reuse was repeated by the second evolution of the merchantvendor interface. Copy libraries from the enterprise software repository were reused in the same manner as in the original evolution of the interface. Logic and variables not encapsulated in the above manner were otherwise segregated and documented with programmer comments imbedded into the source code. This allowed the logic to be more easily copied, modified or replaced in the future without affecting the actual core business logic of the interface. Support for multiple merchants caused some redesign of the original software model. The changes were largely related to establishing a generic, internal format for input data that was to be processed by the AAR component, decoupling the various merchant formats from the main AAR process.

4.2.2 *Anticipating Future Evolutions*

The interface software was designed to be as generic yet flexible as possible. Input and output files were designed to be simple in both format and complexity, as was the user interface. A standard file format containing all data fields required by the interface had been established, and any non-report output files were written in this format. Isolation of the business logic into a largely generic module theoretically assured that the interface would require fewer future changes, even if the solution was again to be reengineered to other platforms or scaled to handle additional merchants. The stable, “black box” nature of the main AAR component was preserved in this manner. Establishing a standard input format enabled future support for other merchants to be more easily added to the interface without the need for complicated changes to the AAR component. Any number of data mapping programs could be created as needed to transform various inbound merchant formats into the required standardized input format.

This particular implementation supported Amazon.com input. Data was transformed by a separately coded data transformation module (DTM) from a standard XML format determined by Amazon into the generic internal format required by the AAR component. Also supported by the interface were Google Checkout™ and eBay/Paypal. The input files for these merchants required two additional DTMs to map comma-delimited (CSV) data to the same standardized internal format. The data in this format could then be generically processed by the AAR component using appropriate business rules that apply to all merchants. Figure 5 illustrates the general flow of data into the enterprise database once it is received from the merchants by the vendor's web services servers. These servers contain the FTP and SOAP components of the architecture. The design of this evolution of the interface allowed for the future scalability and enhanced business value of the solution. Other merchants are rather easily added to the solution with the creation of additional DTMs where necessary. Any new DTMs would perform data transformation tasks similar to existing modules. Indeed, an existing DTM could be used as a basis or framework for developing a new DTM to support a merchant using a similar data format (i.e. XML or CSV). These techniques also allowed for the efficient future reengineering of the product. The primary interface was expected to change rarely, if at all. Any changes to this area of the solution would likely be related to the migrating of the solution to a different database or operating system. Due to design of the interface application, any changes are easily identified and implemented. Future changes are also unlikely to affect any of the individual DTMs, further adding to the efficiency of subsequent evolutions.

4.2.3 *Evaluating the Results*

Implementation of the interface application for the new platform and merchant combination increased the source code of the solution to over 9,200 lines of new or reengineered code. Approximately 3,700 lines were reused from copy libraries already available from the legacy enterprise system repository. The individual copy library files, as in the previous evolution of the project, were largely related to the mapping of values to datasets in the legacy ERP system.

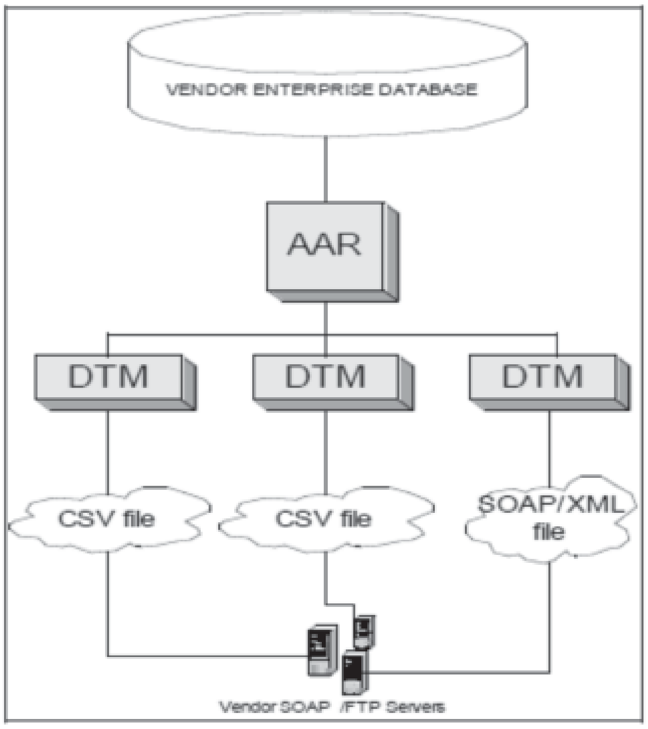


Figure 5: Multiple Merchant Architecture

While the evolution of the merchant-vendor interface consumed more resources than initially planned - even more so than the original evolution of the application - it is difficult to characterize it as a failure due to the additional forward-looking efforts applied to the programming model during development. While these efforts added to length of time consumed by this iteration of the solution, they were undertaken to insure future efficiency of any maintenance requirements, adding to the business value and longevity of the solution. Isolation of static source code added to the future maintainability of the application, as did the inclusion of the self-documenting logic trace layer that had been designed and implemented during the first iteration of the solution. The source code also contained extensive programmer comments describing the business logic, to aid in future maintenance and evolution requirements. Any lost technical documentation could theoretically be recreated by the examination of the source code. The implementation was ultimately considered a success based on the stability exhibited by the product subsequent to acceptance by the client. No software bugs or change requests have been reported since deployment of the solution, and the core application has processed tens of thousands of purchases as of this writing. The successful result of the second evolution of the merchant-vendor integration solution was to be more significantly realized with the yet another evolution of the product.

4.3 Third evolution – Combination of Efforts

The third evolution of the interface required a combination of the previous two iterations. That is, the more recent business logic model deployed in the version based on the HP3000 MPEiX/ TurboImage platform would be re-implemented on the original Microsoft Windows/SQL Server platform. The most pressing requirement for this evolution was that the software solution would be due for integration testing within approximately thirty (30) days. This again left little time to develop a detailed specification or software prototype. The decision to replace, outsource or reengineer the project would in this case rely heavily on the expert opinion of the developers and project managers and less on research efforts to identify the most cost-effective solution. The resulting solution would not merely be a direct conversion from one hardware/ database architecture to another. Instead, it would combine the newest business logic from the second evolution of the software with the database-specific and operating system-specific details of the original evolution. New stakeholder requirements were also desired for the AAR component, and would need to be integrated into the solution.

4.3.1 *Integrating New Requirements*

The primary thrust of the third evolution was to reengineer the HP 3000/MPE-iX version of the software to the Microsoft Windows operating system and SQL Server database architecture. The auditing and reconciliation business logic would need no major changes for the new evolution. The solution would also support the same three merchants (Amazon.com, Google Checkout™, eBay/Paypal) as supported by the previous evolution. Additional user requirements were to be integrated along with requirements typically related to a change in platform. The first new requirement was that all input and output files processed by the auditing and reconciliation (AAR) component were to be read and written in a standard commadelimited (CSV) format rather than the fixed-length format implemented in previous iterations. This new format would allow data to be more easily prepared, viewed or manipulated as needed by accounting department personnel using popular spreadsheet programs like Microsoft Excel. It would also allow for upstream processes to be developed to generate the input files consumed by the interface. Additionally, downstream processes could be developed to consume the output files for reprocessing, printing, reporting or forwarding purposes. A second additional requirement was related to the user interface for the new platform. The new AAR component would be required to prompt the user for the name of the input file to be processed. This was a departure from the previous version, which accomplished such interaction at the JCL level on the HP3000/

MPE-iX platform. On the Windows platform, this would at least initially be an interactive process, with the user entering a file name that could reside anywhere accessible by the machine acting as the application server. While the new evolution would re-couple the user interface to the new component, it was designed to be a simple and generic interaction. Entry of the requisite information could later be automated with "batch files" or timed scripts without affecting the AAR component, successfully allowing the user interface to be later decoupled from the main component.

The decision was made to reengineer the software using previous evolutions as the source code base. Code was merged from each of the prior evolutions to produce the new solution. It was anticipated that this choice to reuse and reengineer would allow for rapid development of a highly stable product. While the actual database access techniques were to follow the model established by the first evolution of the application (Microsoft Windows/SQL Server), the business logic and structure of the solution would follow the model established by the most recent evolution (HP 3000/MPE-iX.) The new version would reside on the Microsoft Windows platform, using the SQL Server database. The vendor requesting the software solution was using the same legacy multi-channel enterprise system as the prior two vendors, and would be receiving reconciliation data from all three of the merchants supported by the previous iteration of the solution.

4.3.2 *Reengineering the Solution*

The first step in the implementation stage was to identify each of the datasets to be accessed in the legacy commerce system's database. Many of the same datasets were accessed in the original Windows solution deployed by the first evolution. The source code related to these datasets could thus be reused with no significant changes. The code reused from the original repository would also serve as a model for any new coding required to support the several new datasets to be accessed by this evolution. The next step was to identify and recode any platform-related differences in the main AAR component. These differences were primarily related to programming language dialects and compilers, the user interface, and file name structures. Dialect differences were minor and were those commonly found when converting software between compilers of the same programming language. The user interface was designed as a simple "question and answer" session in order to reduce the complexity of the interaction and keep it as platform-independent as possible. File name structures were redesigned to handle the longer path+filename standard that is typical of Windows and Unix operating systems.

The primary business logic, parsing of input files and generation of output files needed no recoding for the new version. Other logic had been previously designed to be independent of the platform. This meant that the bulk of the source code would remain static. This further improved the implementation process and allowed for a more stable result.

Each of the data transformation models (DTMs) were migrated to the new platform with relatively minor changes. These changes were specifically related to the same programming language dialect differences identified while reengineering the AAR component. The DTMs did not directly access any databases, making these components relatively transparent between platforms.

4.3.3 *Testing and Documenting the Solution*

Unit tests were derived from test cases developed during the previous evolution of the interface. This reuse of information saved significant time in developing the new solution. Few problems were found during the unit tests due to the transparency of the business logic between platforms. Problems related to the business logic had been previously identified and addressed in prior evolutions, and no new logic errors had been introduced by the reengineering effort.

Documentation for the solution targeted the casual user and was simple and straightforward. Both technical and day-to-day usage documentation were provided by the software engineers. Technical documentation described topics related to the installation and setup of the software and its subsequent linking to ODBC data sources located on local or remote database servers. User documentation described both input and output data files. The descriptions included specific details of the file formats and field layouts. Output files were generated by the interface that included the reporting of both rejected and reconciled transactions. These reports could be subsequently printed or emailed, along with a log of the program's activities. The log file contained program execution statistics, including the number of orders processed and rejected, and the total amount posted to the vendor's financial datasets. Once the development, testing and documentation phases were complete, the project had consumed only slightly more time than estimated. The resulting solution was considered a significant success, and no problems have been reported by the most recent vendor since its release.

4.3.4 *Evaluating the Results*

This third evolution of the merchant-vendor interface was designed to be as simple as possible for the vendor. The "black box" workings of the application performed many complex decisions and operations needed properly allocate the revenues to appropriate orders. It posted commission costs, returned order credits and other activities that would otherwise require manual processing by human personnel. Various merchants were supported with external data transformation components separate from the main auditing and reconciliation module. Upon completion, the resulting lines of source code (LOC) had increased less than 1%, to just over 14,000 lines. The entirety of this evolution of the interface spanned approximately one hundred and ten (110) hours over thirty-two (32) days, significantly less than previous evolutions. Proportionally, much of the time was consumed during the latter stages of the project. These activities were largely related to testing and documentation phases. Additional time was invested in the preparation ("mocking up") of data for subsequent testing. This effort aided the efficiency of the pending integration testing and deployment phases of the solution, further speeding its time to market.

5. EVALUATING THE APPROACH

An important focus in the development of this product from its inception was the ability to automatically process thousands of records at a time without crashing due to unexpected logical conditions. The development and maintenance of the resulting software needed to be as efficient as possible given available personnel and system domain knowledge. The desired solution was expected to be both highly fault tolerant and maintainable.

5.1 Establishing Fault Tolerance

The main auditing and reconciliation component had been designed to terminate in an error state only if a critical file system event occurred in an unrecoverable context. One example of this type of termination would include the component's inability to establish a connection to a remote database. Another example would be the component encountering full storage media (e.g., hard disk) while attempting to write output files. Recoverable logic errors were trapped and logged to a standard line-sequential text file, as well as to the main console or user display. Any errant records were rejected and written to error files for later manual processing. The application would continue processing with the next pending transaction and in this manner would theoretically automatically process most orders without the need for user intervention. Rejected orders were written to output files in the same standard CSV format as the input file. This enabled the reprocessing of the output data as input at a later time, if necessary. This feedback capability added to the flexibility of the auditing and reconciliation process. Additional fault tolerance was realized from the technique of isolating slowly-evolving business logic from the more dynamic user interface and ever-changing data transformation components of the solution. Future evolutions to reengineer the software for additional platforms and online merchants would have little direct effect on the static business logic related to the vendor enterprise system. This allows for a dramatic reduction in integration efforts of future evolutions.

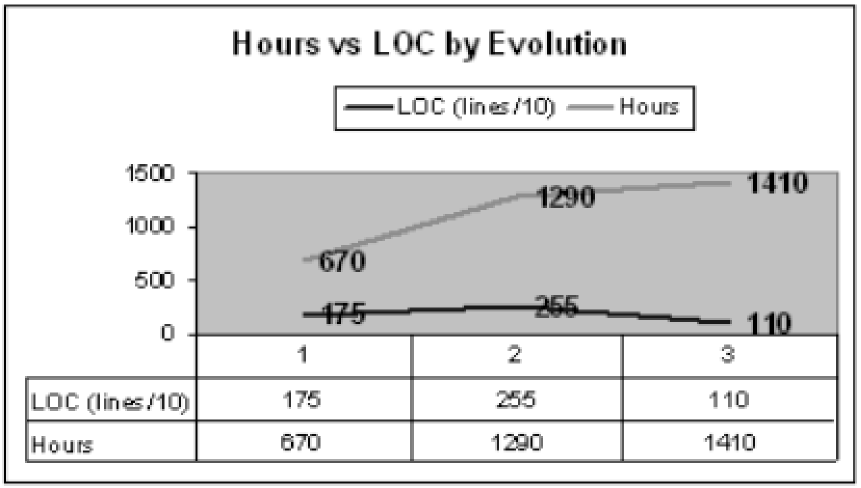
5.2 Improving Each Evolution

The development of the first evolution of the solution lacked significant specification and analysis effort. No plans had been made for future scaling or reengineering of the product. Such foresight would have at least partially affected the initial software design model. This shortcoming was partially mitigated by the efforts of the developers to thoroughly document the engineering process, and partially by the imbedding of a diagnostic trace layer in the source code. The second evolution of the software involved both the reengineering of the product for a new hardware and database architecture and the redesigning of the software for scalability. The latter effort was intended to increase the business value of the solution, allowing for more online merchants to be efficiently added to solution. This effort took the form of multiple "mini evolutions" to add support for each additional merchant. The documentation and logic tracing layer approach introduced in the first evolution were repeated in the second evolution with similar success. Modifications were made to the software design to anticipate further reengineering efforts.

The third and most recent evolution involved strictly the reengineering of the interface to a new platform, with no major changes to the business logic or data format. The lessons learned and leveraged in the second evolution of the solution proved to significantly speed the development and deployment time of the third evolution. Using the concepts and design model established in the first two evolutions continued the cycle of iterative improvement of a steadily evolving software product.

5.3 Analyzing Canonical Results

Comparing the development time of each of the three major evolutions of the software solution initially shows that it is easy to be misled by the apparent increase in development time of the second evolution when compared to the first evolution (Figure 6). In this case, it is important to consider the context of the spike in resources consumed. Figure 6 shows that the total lines of source code (LOC) of the solution increases with each new evolution, but by a markedly lower rate between the second and third evolutions.



	1	2	3
LOC (lines/10)	175	255	110
Hours	670	1290	1410

Figure 6: Comparison of Evolutions

The figure also shows that significantly fewer hours were required to develop the third evolution compared to the first two. Combined, these are indicators of the successful engineering of a highly evolvable software product. In this context, “highly evolvable” translates into the quantifiable reduction of effort spent developing and deploying new iterations. It is anticipated that the trend will continue with subsequent evolutions of the interface.

The second evolution is more accurately characterized as having three separate development stages (Figure 7). The first of the three stages was largely related to the reengineering of the software to the new hardware/database architecture, while the next two were intended to scale the software, adding support for additional merchants. These stages included modification to the software design itself, rather than simple platform-to-platform reengineering activities.

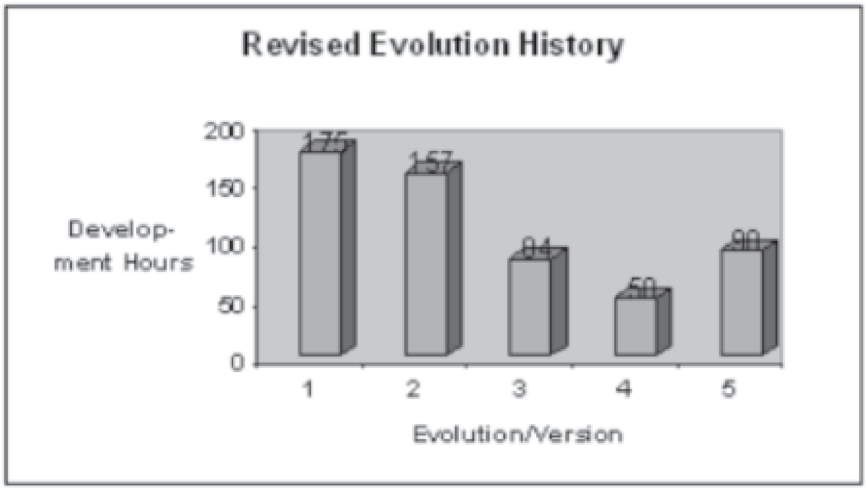


Figure 7: Decomposition of the Second Evolution

The figure shows more accurately the increased efficiency of each successive evolution of the software, illustrating the increasing efficiency of each stage of the second evolution where merchant scalability was introduced. While the resources expended during the third evolution exceeded those expended by the third stage of the second evolution, it should be noted that such a comparison would also be misleading. The third stage of the second evolution involved only new coding to support an additional merchant, a task far less complicated than the migration to a new platform. In the final evolution, there were no changes to the overall architecture of the solution. Changes were strictly limited to the reengineering of the product for a new hardware and database combination.

5.4 Identifying the Successes and Failures

Several aspects of the development process contributed to the success of this project. Arguably the most significant was the reuse of source code from the software repository of the legacy enterprise system hosted by the vendor. Reuse of existing source code contributes to both a reduction in development time and an increase in system stability. This is especially true of source code related to established database-related processes. Introduction of new logic errors is less frequent, and this subsequently reduces the time expended unit-testing the resulting application.

One of the most important aspects of the successful implementation and subsequent evolution of the software in this project was the reuse of existing source code and software design. However, there were other traits that contributed to the success of this project. They include the flexible, loosely coupled design model, the use of standard data formats, the process and logic tracing documentation, and the deliberate coding of the interface with future evolution in mind. Documentation efforts were another aspect of the development process that contributed to the success of the project. Documenting the process of each evolution at an appropriately granular level by the software engineers was useful in the iterative improvement of the software design model. This resulted in the reduction of development time for each successive version. Imbedding a logic tracing layer into the software components further aided in the documentation of each component and of the interface solution overall. It proved to further aid in other development activities, from the establishing and verification of unit tests to the efficient identification and repair of software errors prior to deployment.

The use of popular data formats such as XML and CSV helped decouple the software interface from the provider of the reconciliation data. This allowed the architecture to be more flexible while having no direct effect on the business logic itself. A reduction in unit and regression testing efforts was a noticeable benefit of these techniques, contributing to the overall success of the project. Finally, the “loosely coupled” nature of the solution contributed to the success of each evolution of the software. While tightly coupled to the vendor’s legacy enterprise system and to the architecture upon which it was deployed, the merchant-vendor interface was designed to be loosely coupled with providers of the incoming financial data. This design allowed the inbound data to be transformed into a generic internal format prior to being processed by the main auditing and reconciliation component. The advantage of this type of decoupling is further realized when contemplating future changes made to the data stream by the individual merchants or providers. Merchants may add or remove noncritical fields to their data format, or completely change how the data is delivered (i.e., changing from an XML to CSV format or vice versa). These changes would theoretically affect only the particular data transformation module tasked with processing that merchant’s data. No modifications would be necessary to the main auditing and reconciliation component. This decoupling eliminates the introduction of new errors into an otherwise static and stable process. It also eliminates the need to retest all supported merchants when implementing changes related to the format of only one merchant. While it is important to identify and leverage the successes of a particular software development model, it is equally crucial to identify and learn from the failures or shortcomings. The project described by this paper helped highlight a particular shortcoming that affected the early evolutions of the project. The amount of time allocated to the specification and analysis phases of each evolution of the project were minimal. Early in the project, this hampered the initial deployment of the solution due to a significant number of change requests. It also required later modification of the source code to allow for additional scalability. Evaluating the approach from inception through the third and most recent deployment, one can conclude that while reducing or eliminating time spent planning and specifying a software solution may at times seem tempting, reasonable or cost-effective, the reality is that a remarkable additional effort is often required later. In the project studied by this paper, the costs came in the form of a dozen post-release patches applied to the first evolution of the software, and in the extensive recoding that was performed to plan for future scalability and reengineering requirements. At the onset of the project, future reengineering efforts were neither expected nor planned for. This resulted in a significant post-release period of change requests through which the business rules and data details were refined. This proved to be more costly in resources than the original development effort, perhaps illustrating that skipping crucial software development life cycle steps at the onset of a new software integration projects ultimately consumes more resources than are saved.

6. SUMMARY AND FUTURE WORK

This paper is a case study of a particular real-world project that involved the integration of a legacy multi-channel commerce enterprise system with multiple established and emergent online merchants and payment providers. The purpose of the integration solution was to automate the auditing and reconciliation of the financial details related to online purchases of vendor products via third-party merchants. Each evolution of the solution was described, from the initial engineering of a "one off" custom product to a highly stable, scalable and maintainable solution capable of efficient future adaptations. The lessons learned and the techniques employed during each evolution were documented and analyzed by the developers in order to improve the process as the software evolved. The design model was refined during each subsequent evolution, further increasing both the current and future business value of the solution.

6.1 Techniques for Successful Integration

Overall a highly successful project, this case study addressed three major areas that represent some of the major goals in software engineering and reengineering: The project took advantage of legacy assets. This was achieved at both the software level in the form of reusing source code from an existing repository, and at the environment level in the form of the use of the same language implementation and compiler licensing of the vendor's existing legacy software. It also took advantage of user and developer familiarity with the legacy system by keeping the complexity low for both the user interface and the component output. This resulted in a product that had the look, feel and behavior of software and data formats with which the end-users were already accustomed.

Second, the project eventually focused upon planning for future maintenance and evolution. Anticipating that other vendors would likely request identical or similar merchant-vendor integration solutions, the software solution was ultimately designed to be scaled to support future vendors and merchants, as well as other hardware and database architectures. By engineering source code that was self-documenting and well-organized, highly cohesive at the component level, yet loosely and flexibly coupled at the user and data interaction level, future implementations of the solution would be an increasingly measurable and predictable task.

Finally, by planning and designing for fault tolerance, future maintenance of the solution would require less extensive regression testing. Additional maintenance activities related to scalability would introduce fewer new software bugs, and each resulting evolution would be extremely reliable, an attribute crucial to automated financial processes.

While the solution itself - already three years old as of this writing - will soon take its own place among software described as "old" or "legacy", it does not follow the stereotypical trend of legacy software envisioned and described by Lehman [16]. That is, each future evolution is instead expected to take at worst a similar or predictable effort, and at best a significantly more efficient effort. Rather than exhibiting an increasingly complex and degraded quality in both program understanding and structural design, this solution was designed to make future efforts as straightforward as possible.

6.2 Future Work

The reengineering of software that integrates legacy enterprise systems and their more modern web-based partners can be even more complex and challenging process than the reengineering and maintenance of the legacy software systems themselves. Indeed, much of the added complexity is realizing a solution that satisfies both the requirements of a slowly evolving legacy enterprise system with the requirements of the constantly evolving and emergent online merchant systems. There is no evidence indicating that there will be a decrease in the proliferation of e-commerce and other new sales channels by well-established vendors. Therefore, learning from the lessons derived from real-world implementations of this type of integration is crucial to the efficient success of the future attempts by others. More scrutiny and measurement, both quantitative and qualitative, should be applied to the trend of integrating systems that extend the use and visibility of legacy multichannel commerce systems.

7. CONCLUSION

Software inevitably requires maintenance [1][17]. Should the software engineered for this particular solution require future maintenance to support additional merchants, or reengineering to support new architectures, much of the process has now been identified, documented and anticipated. The design of the solution specifically lends itself to efficient change, without the characteristics of more poorly designed systems that grow increasingly more complex and difficult to maintain over time. While software reengineering projects are perhaps the most complex of all evolution strategies, the project described by this paper involved deliberate measures to mitigate this challenge. The reuse of source code from legacy repositories, the encapsulation of platform-specific logic, the decoupling of business logic from the scalable aspects of the system and the documentation and analysis of the development process resulted in a highly evolvable product designed to anticipate the inevitable future of changing business requirements and architectures.

References

1. Basili, V.R.; Briand, L.; Condon, S.; Yong-Mi Kim; and Melo, W.L.; Valen, J.D. "Understanding and predicting the process of software maintenance release." *Proceedings of the 18th International Conference on Software Engineering* (ICSE 1996: March 25-29, 1996; Berlin, Germany). Los Alamitos, CA; IEEE Computer Society Press: 1996.

2. Bennett, K. H. and Rajlich, V. T. "Software maintenance and evolution: a roadmap." *Proceedings of the Conference on the Future of Software Engineering* (June 4-11, 2000; Limerick, Ireland,). ACM: New York, NY, pp. 73-87.

3. Bennett, K.H. "Software services and software maintenance." *Proceedings of the 7th European Conference on Software Maintenance and Reengineering*, pp. 3-12, March 2003.

4. Bianchi, A.; Caivano, D.; Marengo, V.; Visaggio, G. "Iterative Reengineering of Legacy Systems." *IEEE Transactions of Software Engineering*, vol. 29, no. 3, pp. 225-241, March 2003.

5. Bisbal, J.; Lawless, D.; Bing Wu; and Grimson, J. "Legacy information systems: issues and directions," *IEEE Software*, vol. 16, no. 5, pp. 103-111, September/October 1999.

6. Blackburn, J.D.; Scudder, G.D.; and Van Wassenhove, L.N. "Improving speed and productivity of software development: a global survey of software developers." *IEEE Transactions on Software Engineering*, vol. 22, no. 12, pp. 875-885, December 1996.

7. Canfora, G.; Fasolino, A. R;, and Tortorella, M. "Towards reengineering in reuse reengineering processes." *Proceedings of the International Conference on Software Maintenance* (ICSM: 1995: October 17-20, 1995; Opio, France). Washington, DC: IEEE Computer Society Press:1995.

8. Chikofsky, E. "Sustain, Enhance, or Replace: Making Decisions on Systems." *22nd IEEE International Conference on Software Maintenance* (ICSM 2006). p. 134, September 2006.

9. Chikofsky, E.J.; and Cross, J.H. “Reverse engineering and design recovery: a taxonomy.” *IEEE Software*, vol.7, no.1, pp. 13-17, January 1990.

10. De Lucia, A.; Fasano, F.; Oliveto, R.; Tortora, Genoveffa **“**Recovering traceability links in software artifact management systems using information retrieval methods**”**, *ACM Transactions on Software Engineering and Methodology,* vol. 16, no. 13, September 2007.

11. Fayad, M. E.; Laitinen, M.; and Ward, R. P. “Thinking objectively: software engineering in the small.” *Communications of the ACM*, vol. 43, no. 3, pp. 115-118, March 2000.

12. Frakes, W. B.; and Terry, C. “Software reuse: metrics and Models.” *ACM Computing Surveys*, vol. 28, no. 2, pp. 415-435, June 1996.

13. Hasselbring, W. “Information system integration.” *Communications of the ACM*, vol. 43, no. 6, pp. 32-38, Jun. 2000.

14. Huang, S.; Tilley, S.; VanHilst, M.; Distante, D.; “Adoption-Centric Software Maintenance Process Improvement via Information Integration.” *Proceedings of the 13th IEEE International Conference on Software Technology and Engineering Practice* (STEP 2005: September 24-25, 2005; Budapest, Hungary). Los Alamitos, CA: IEEE Computer Society Press: 2006.

15. Jelber S. S.; Lethbridge, T.C.; and Matwin, S. “Mining the maintenance history of a legacy software system.” *Proceedings of the International Conference on Software Maintenance* (ICSM: September 22-26, 2003; Amsterdam, The Netherlands). pp. 95-104, September 2003.

16. Lehman, M. M. “Programs, life cycles, and laws of software evolution.” *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060-1076, September 1980.

17. Lehman, M. M.; and Ramil, J. F. “Software Evolution and Software Evolution Processes.” *Annals of Software Engineering*, vol. 14, no. 1-4, pp. 275-309, December 2002.

18. Lung, C. “Software Architecture recovery and restructuring through clustering techniques” *Proceedings of the third international workshop on Software Architecture* (November 1-5, 1998; Orlando, Florida). pp. 101-104, ACM: New York, NY, 1998.

19. Mili, H.; Mili, F.; and Mili, A. “Reusing software: issues and research directions.” *IEEE Transactions on Software Engineering*, vol.21, no.6, pp.528-562, June 1995.

20. Ming-Ling, C.; and Shaw, W.H. “Distinguishing the critical success factors between ecommerce, enterprise resource planning, and supply chain management.” *Proceedings of the 2000 IEEE*, pp.596-601, Engineering Management Society: 2000.

21. Muller, H.; Wong, K.; and Tilley, S. “Understanding software systems using reverse engineering technology.” *In The 62nd Congress of L’Association Canadienne Francaise pour l’Avancement des Sciences Proceedings* (ACFAS), 1994.

22. O’Hanlon, C. “A conversation with Werner Vogels.” *Queue*, vol. 4, no. 4, pp. 14-22, May 2006.

23. Petrecca, L. “Google wants to handle your online checkout.” Jun. 2006; Last accessed on February 26th, 2009: http://www.usatoday.com/tech/products/services/2006-06-29- googlecheckout_x.html.

24. Robertson, P. “Integrating legacy systems with modern corporate applications.” *Communications of the ACM*, vol. 40, no. 5, pp. 39-46, May 1997.

25. Seacord, R. C.; Plakosh, D.; and Lewis G. A. "Modernizing Legacy Systems." *SEI Series in Software Engineering*, Addison Wesley: 2003.

26. Sneed, H. M. "Encapsulation of legacy software: A technique for reusing legacy software components." *Annals of Software Engineering,* vol. 9, no. 1-4, pp. 293-313, January 2000.

27. Sneed, H. M. "Using XML to Integrate Existing Software Systems into the Web." *Proceedings of the 26th International Computer Software and Applications Conference* (COMPSAC 2002). pp. 167-172. IEEE Computer Society: Washington, DC, 2002.

28. Sneed, H. M. "Wrapping Legacy COBOL Programs behind an XML-Interface." *Proceedings of the Eighth Working Conference on Reverse Engineering* (WCRE: October 2-5, 2001, Stuttgart, Germany). p.189. IEEE Computer Society: Washington, DC, 2001.

29. Sneed, H.M. "Integrating legacy software into a service oriented architecture." *Proceedings of the 10th European Conference on Software Maintenance and Reengineering* (CSMR: March 22-24, 2006; Bari, Italy). pp. 11-14, March 2006.

30. Sneed, H.M. "Planning the Reengineering of Legacy Systems." *IEEE Software*, vol. 12, no. 1, pp. 24-34, January 1995.

31. Sneed, H.M. "Risks Involved in Reengineering Projects." *Proceedings of the IEEE Sixth Working Conference on Reverse Engineering* (WCRE: October 6-8, 1999; Atlanta, Georgia). pp 204-211. IEEE Computer Society Press: 1999.

32. Storey, M. D.; Muller, H.A.; and Wong, K. "Manipulating and documenting software structures." *Object-Oriented Technology for Database and Software Systems*, pp. 240-252, World Scientific: 1995.

33. Umar, A. "The emerging role of the Web for enterprise applications and ASPs." *Proceedings of the IEEE*, vol. 92, no. 9, pp. 1420-1438, IEEE Computer Society Press: 2004.

34. Yang, J., and Papazoglou, M. P. 2000. "Interoperation support for electronic business." *Communications of the ACM*, vol. 43, no. 6, pp. 39-47, June 2000.