

Can Legacy Systems Beget Product Lines?

Nelson Weiderman, John Bergey, Dennis Smith, Scott Tilley
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA, USA
{nhw,jkb,dbs,stilley}@sei.cmu.edu

Abstract. In many respects it is easier to formulate an architecture for a family of products if one assumes that the systems are being developed from scratch. But the vast majority of systems development efforts today start from a cornucopia of legacy systems. Significant progress in component-based architecture, system understanding, object-technology, and net-centric computing now makes it possible to evolve these legacy systems to a state in which they exhibit many of the characteristics of product lines. Systems in well-established domains are migrating to distributed object systems that exhibit large-scale reuse from a core set of assets while keeping the legacy systems largely intact. Many of these systems have evolved without overtly using product line terminology or practices and have been off the radar screen of the product line community as a starting point for product families. The advocates of product lines need to recognize this “distributed legacy evolution model” as an integral part of their practices for developing information systems.

1. Introduction

Legacy systems have usually been thought of as an albatross rather than an asset when it comes to evolving to more modern, productive, and useful systems. Software developers and software researchers have tended to focus their attention on “better, cheaper, faster” ways to build new systems rather than on evolution paths for heritage applications. They were justified in this focus since the stovepipe systems built in the past were virtually intractable in the face of available program understanding and integration technology. The emphasis on component-based architecture and product lines has clearly raised the state of the practice for developing new systems from scratch, but has contributed far less to the practice of evolving legacy systems.

The bad news is that there are few developments today that have the luxury of starting from scratch because of the huge investments in, and reliance on, legacy systems. The good news is that there are new technologies and practices that make the migration path more tractable.

One of the reasons that the situation is changing so rapidly is the emergence of integrating infrastructures. With improved integration we have seen the the World Wide Web (the Web) and electronic commerce flourish. Where once information systems were isolated and difficult to access, they can now be accessed using the Web and its interfacing software.

The Internet is being used in a number of innovative ways to connect users and their stovepipe systems both inside organizations and between organizations. Within organizations, the Web is not only being used to connect departments such as marketing, sales, and engineering, but also to connect teams of software developers around the world working around-the-clock on the same project. Between organizations, the Web is being used to connect businesses with their suppliers and their customers. It is becoming a medium for placing orders, receiving delivery, and checking status.

There are many ways of evolving to product lines. Regardless of the starting point, the goal is to develop higher quality systems, faster, with higher productivity and improved efficiency. Product lines accomplish this goal by facilitating the systematic reuse of software assets. The emphasis is on strategic, coarse-grained reuse that leverages models, architectures, designs, documentation, testing artifacts, people, processes, and implementations.

The leverage of product lines is that software assets can be reused in different contexts. The cost of producing consecutive systems with the same asset base decreases over time. The domain model is reused from one application to the next and the productivity of the software development staff increases proportionately and hence the assets have a greater return on investment over time.

On the other hand, unintegrated (stovepipe) software assets that are not used for continuous production of additional assets become stale and require more and more resources to maintain them. Hence their value may decrease over time and eventually there may be more cost associated with their continued maintenance than benefit from their continued use. At that point the software becomes a liability with no leverage. Until recently this was the end of the story.

Now, it is becoming possible to leveraging software assets either at conception of the development of a product line or after the fact by extracting the necessary components from existing legacy assets. New product starts should plan for this reuse of software assets in advance. In fact, some plans for product lines arise out of expediency when it becomes apparent that the resources are not available to construct two similar systems without exploiting commonality [[3]]. *A priori* development of systems using a product line practice approach is an active research area and deserves continuing attention. For example see [[1]]. But the substantial contributions to migrating stovepipe legacy systems to product lines have gone largely unrecognized by the product line community and needs further attention.

We take the position that starting a product line effort with legacy assets is not only possible, but is in most cases preferable to starting from scratch. We posit that this leveraging of legacy assets is enabled through the convergence of a set of practices that we try to illuminate at a high level. We make our case by elaborating on a distributed legacy evolution model [[10]] with the following components:

- an enterprise approach for guiding decision making for system evolution,
- developing a technical understanding of systems at a high level of abstraction,
- using distributed object technology and wrapping for system evolution, and

- using network-centric computing for system evolution.

We also present credible evidence of progress and experience that supports this approach. While the range of application for these ideas may not include all classes of systems (real-time embedded applications may be one such exception), we believe that applicable scope is quite broad.

2. Using an Enterprise Approach for Decision Making

System evolution and technology insertion do not take place in a vacuum. Many attempts at evolution and migration fail because they concentrate on a narrow set of software issues without considering the broader set of management and technical issues. Evolution takes place in the context of an organizational setting that varies considerably in terms of the culture and the readiness to incorporate change. While there may be many complex technical problems that are largely unprecedented, a focus on the technical problems to the exclusion of the enterprise problems is a recipe for disaster. Hence it is crucial to plan for change in the context of the enterprise.

The Software Engineering Institute has developed an “Enterprise Framework for the Disciplined Evolution of Legacy Systems [[2]] as a guide for organizations planning software evolution efforts, such as migrating legacy systems to more distributed open environments. This framework draws out the important global issues early in the planning cycle and provides insight and guidance for a disciplined evolution approach.

In addition to the software engineering and technology considerations, the enterprise approach addresses the needs of the customer, the organization’s strategic goals and objectives, the operational context of the enterprise, as well as the current legacy systems and their operational environment. It recognizes the central importance of both software engineering and systems engineering (and their interplay) to the system evolution initiative. The seven elements of the framework are the organization, the legacy system, the target system, the project, systems engineering, software engineering, and technologies.

These elements are applicable to a wide class of system evolution initiatives. In practice, the specific composition of the framework and their interrelationships are a function of the enterprise, its culture, and its management and technical practices (lifecycle activities, processes, and work products that are used to carry out the tasks described in the project plan and migration strategy).

3. Developing High-Level System Understanding

Program understanding is the (ill-defined) deductive process of acquiring knowledge about a software artifact through analysis, abstraction, and generalization [[5]]. Clearly, program understanding is a prerequisite for software evolution. However, we assert that the nature of program understanding should change from an understanding of the internals of software modules (white-box reengineering) to an understanding of the interfaces between software modules (black-box reengineering). A more detailed explanation of this approach can be found in [[9]].

Understanding is critical to our ability to evolve unproductive legacy assets (e.g., obsolete, overly-constrained, or stagnating components) into reusable assets that can contribute to a product line approach. Legacy assets may be aging software systems that are constructed to run on various obsolescent hardware types, programmed in obsolete languages, and suffer from the fragility and brittleness that results from prolonged maintenance. As stovepipe software ages, the task of maintaining it becomes more complex and expensive and the asset becomes more of a liability than an asset. While bottom-up program understanding has its place, it is often the case that software and system engineers spend inordinate amounts of time trying to reproduce the system's high-level architecture from low-level source code.

Legacy code can be difficult to understand for many reasons. It may have been created using *ad hoc* methods and unstructured programming. It may have been maintained in crisis mode with no updates to the higher-level documentation. There may be little or no conceptual integrity of its architecture and design. But every system has an architecture even if it is not written down. It is this architecture and high level understanding of the structure of the legacy system that must be the focus of a program understanding effort.

Program understanding is a relatively immature field of research in which the terminology and focus are still evolving. Tilley and Smith [[5]] describe three promising lines of research: investigating cognitive aspects, developing support mechanisms, and maturing the practice. Each of these lines should be tailored to a high-level, white-box form of program understanding necessary for more rapid and cost-effective migration. Evidence that the high-level understanding approach bears fruit is given by the examples cited in Section 6.

4. Distributed Object Technology and Wrapping

The approaches for software evolution of legacy systems are being dramatically changed by distributed object technology and wrapping [[8]]. Traditionally, the approach taken to legacy systems reengineering has been to understand the system's structure and to extract its essential functionality so the whole system or a series of pieces of the system could be transformed into a more evolvable system over the long term. But distributed object technology is changing the nature and economics of legacy system reengineering.

Traditional reengineering is based on "deep" program understanding and reverse engineering. The cost/benefit ratio of this approach is staying the same in the face of new technologies such as CORBA, Java, and the Web because deep understanding is linear relative to the size of the program and the new technology provides no additional leverage. However, the benefits of "shallow" interface understanding and component wrapping using these distributed object technologies is rising rapidly relative to the replacement cost. As a result, the economic balance is changing from traditional transformation-based reengineering to wrapper-based reengineering [[9]]. These economic factors are having a significant impact on many organizations struggling with modernizing their systems.

Component-based architecture and product lines are concerned with design abstractions for system-level structure. By "system-level" we mean something larger than a single computer program. The significance of making distribution extensions to OT is that software designers and maintainers have at their disposal

the means of expressing abstract system designs and, more importantly, have tools for quickly fabricating working versions of these designs. That is, there is a more direct path now than ever before from abstract architectural concepts to concrete implementation of these concepts using DOC.

5. Network-Centric Computing

As is evident from the examples cited in the next section, the Net¹ is causing a “sea change” in both the nature of enterprise applications and development methods used to create them. There are clear and well-documented major trends toward electronic commerce and network-based development and collaboration. The use of the network has expanded far beyond e-mail and access to vast information sources. It has become a universal medium for information exchange. The value of the Internet to software engineering and systems development must be recognized and exploited. The promise of product line development for distributed legacy evolution will not be achieved until this medium is used more effectively.

The influence of network-centric computing (NCC) on software evolution can be summarized in three words — universality, ubiquity, and access. Universality is provided by portable executable content, such as Java applets, which runs on multiple platforms and operating systems. Making established user interfaces, such as web browsers, available on almost any client provides ubiquity. Making vast quantities of corporate data, which previously were inaccessible in mainframe-based databases available on the Net to the ubiquitous client software, provides accessibility.

One of the primary drivers of NCC is economics. Because applications and data are downloaded from servers on demand, there is a potential reduction in the cost and complexity for system administrators in managing a corporate network [[6]]. Maintenance can be done at one central location rather than at thousands of sites in the organization, thereby reducing total cost of ownership (TCO). The tradeoff is that the end-users lose control and customizability of the local machine. However, they may gain significantly by increasing their productivity in their primary tasks by not being responsible for application installation, system administration, and troubleshooting tasks. Thus, NCC leverages system administration resources.

NCC can also leverage software assets in a number of ways by making them available over the Net. In its simplest form, just the user interface might be changed. Instead of accessing the enterprise database through an idiosyncratic user interface, the database can be accessed through a network browser. This transition has been accomplished many times by many organizations and, by now, should no longer be considered a high-risk, unprecedented form of software evolution. Many enterprise-wide intranets have taken this approach without significantly changing the underlying software base.

The next level of complexity involves partitioning the application into separate components so that the new version of the system can operate in a client/server manner. Once this step has been taken, the software is more free to evolve the

¹ The term *Net* as used here includes the Internet (the global computer network), intranets (local networks that are usually isolated from the Internet by a firewall), and extranets (extensions of an intranet into the Internet in a secure manner).

individual parts into a reusable set of components that can provide the basis for a product line. In the first case (changing the interface) the business benefits from the universality, ubiquity, and access, but does not reap the rewards of Business Process Reengineering (BPR). In the second case (restructuring), the leveraging of assets and return on investment becomes paramount.

6. Examples of Successful Legacy System Evolution

One example of a successful legacy evolution is Wells Fargo Bank's online electronic banking system. Wells Fargo started offering real-time access to account balances via the Web starting in May 1995 and has expanded those services since then to include transferring funds, seeing cleared checks, examining credit card charges and payments, downloading transaction files, requesting service transactions, and paying bills [[11]]. The system has 100,000 enrolled customers and was handling 200,000 business object invocations per day as of early 1997 [[7]].

Wells Fargo has accomplished this by leaving their legacy systems largely untouched while adding the CORBA middleware to create a three-tiered client server system. The "customer" object and the "account" object allow the definition of a customer relationship whereby the client can first get all information about the customer's relationship with the bank and then, for each account owned by the customer, get the relevant summary information. Wells Fargo found that the key to enabling reuse of legacy systems was in having, maintaining, and sharing a well-architected enterprise object model.

The centerpiece of each monthly issue of Distributed Object Computing (DOC) magazine is a deployed case study such as Wells Fargo. They describe the development, the business case for building and deploying the application, the laundry list of technology used on the project and, when available, the staffing and deployment information. In their first eight issues, they have featured a web-based banking system, an airline reservation system, a criminal justice suspect index system, a newsmedia system to provide personalized news and digital content, a 911 emergency response system, a management information system for monitoring and directing large projects, an electric power exchange system for the electric power industry, and an information system for a utility company. In most cases these development efforts made heavy use of legacy software assets.

As an example of the specific leverage provided by technologies such as CORBA and Java, Allied Signal Engines, a business unit of Allied Aerospace, has reported a cost savings of \$750,000 per new application [[4]]. This was accomplished by moving to a component-based software architecture and by outsourcing a major portion of the actual coding effort to an offshore development company in India. They found that they could "raise the starting point for every application, reducing the cycle time and thereby reducing costs." Their wide-ranging development efforts were made possible, at least in part, by the network-centric computing models that were described in the previous section.

The common themes running through all these examples are rapid development (months rather than years), the use of DOC and the Web, component-based architecture, and the use of the previously existing legacy infrastructure. Domain analysis is not the issue it is for new development because the domain is well-

established, but significant effort goes into defining new business objects and redefining the business goals. Clearly, these “makeovers” are the basis of product families in the sense that they result in a set of core assets (distributed objects) that form the basis of future product development.

7. Conclusions

The approaches to software evolution are changing rapidly along with the changing technology. The changing technology is pushing the evolution of systems in several ways. Two approaches to software evolution appear to be on the decline. First, it is rarely possible, because of huge investments in legacy systems that have evolved over many years, to replace those systems and start from scratch. So the “big bang” approach to software migration is not often feasible. Second, it is increasingly less attractive to continue maintaining traditional (functional) legacy systems at the lowest level of abstraction expecting them to evolve into maintainable assets. So the fine-grained maintenance approach is also undesirable because it neither adds value to the asset nor provides for future leverage.

The recommended approach for systems evolution can be summarized briefly as follows:

- Understand the goals and resources of the enterprise with respect to a system evolution project. Use a software evolution framework to plan a disciplined system evolution.
- Understand the legacy system at a high level of abstraction using system understanding technology paying particular attention to interfaces and abstractions. Find the encapsulatable components of the legacy system upon which to build.
- Consider middleware and wrapping technologies for encapsulating subsystems and creating distributed objects that form the basis for product line systems. Apply those technologies in accordance with the framework.
- Consider using the World Wide Web for expanding the scope of the legacy system and as a development tool. Capitalize on the universality, ubiquity, and access that the Web provides.

As is so often the case in software engineering, this approach to software evolution raises the level of abstraction so that our resources are being used more effectively. Economic realities are pushing us from low-level maintenance activities to high-level transformations. A focus on architecture and product lines is facilitating large-scale reuse in construction where before we were satisfied with small-scale reuse.

The use of these new approaches is still somewhat risky and advanced, but by no means unprecedented. They have been employed in prototypes, tested in small systems, and used to transform large systems. Useful and production-quality tools are now available. New developments are occurring at “Internet speed”. Product line advocates must diligently follow the developments in integration and web technologies as well as legacy system migration techniques. Failure to do so will make their work much less relevant to the practitioners who have no other choice but to start from their legacy systems.

8. References

- [1] Bass, Leonard; Clements, Paul; Cohen, Sholom; Northrop, Linda; & Withey, James. *Product Line Practice Workshop Report*. (CMU/SEI-97-TR-003). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1997.
- [2] Bergey, John; Northrop, Linda; & Smith, Dennis. *Enterprise Framework for the Disciplined Evolution of Legacy Systems*. (CMU/SEI-97-TR-007). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1997.
- [3] Brownsword, Lisa & Clements, Paul. *A Case Study in Successful Product Line Development*. (CMU/SEI-96-TR-016). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1996.
- [4] Gill, Philip J. "CORBA Proves Its Value." *Object Magazine* 7, 8 (October 1997): 10-11.
- [5] Tilley, Scott & Smith, Dennis. *Coming Attractions in Program Understanding* (CMU/SEI-96-TR-019). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1996.
- [6] Tilley, Scott & Smith, Dennis. *On Using the Web as Infrastructure for Reengineering*. Proceedings of the 5th Workshop on Program Comprehension, May 28-30, 1997, Dearborn, Michigan. IEEE Computer Society Press, pp 170-173, 1997.
- [7] Townsend, Erik S. "Wells Fargo's 'Object Express'." *Distributed Object Computing* 1, 1 (February 1997): 18-27.
- [8] Wallnau, Kurt; Weiderman, Nelson; Northrop, Linda. *Distributed Object Computing with CORBA and Java: Key Concepts and Implications*. (CMU/SEI-97-TR-004). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1997.
- [9] Weiderman, Nelson; Northrop, Linda; Smith, Dennis; Tilley, Scott; & Wallnau, Kurt. *Implications of Distributed Object Computing for Reengineering*. (CMU/SEI-97-TR-005). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1997.
- [10] Weiderman, Nelson; Bergey, John; Smith, Dennis; Tilley, Scott; & Wallnau, Kurt. *Approaches for Legacy System Evolution*. (CMU/SEI-97-TR-014). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1997.
- [11] Wells Fargo Bank. Wells Fargo's WWW Homepage [on-line]. Available

WWW: <URL: <http://www.wellsfargo.com>> (1997).