Implications of Distributed Object Technology for Reengineering

Nelson Weiderman
Linda Northrop
Dennis Smith
Scott Tilley
Kurt Wallnau
June 1997

Tehnical Report
CMU/SEI-97-TR-005
ESC-TR-97-005

Technical Report
CMU/SEI-97-TR-005
ESC-TR-97-005
June 1997

Implications of Distributed Object Technology for Reengineering

Nelson Weiderman
Linda Northrop
Dennis Smith
Scott Tilley
Kurt Wallnau

Reengineering Center
Product Line Systems
Dynamic Systems

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This report was prepared for the

SEI Joint Program Office
HQ ESC/AXS
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

(signature on file)

Thomas R. Miller, Lt Col, USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

This work was created in the performance of Federal Government Contract Number F19628-95-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 52.227-7013.

This document is available through SAIC/ASSET: 1350 Earl L. Core Road; PO Box 3305; Morgantown, West Virginia 26505 / Phone: (304) 284-9000 / FAX (304) 284-9001 / World Wide Web: http://www.saic.com/contact.html / e-mail: webmaster@cqpm.saic.com

Copies of this document are available through the National Technical Information Service (NTIS). For information on ordering, please contact NTIS directly: National Technical Information Service, U.S. Department of Commerce, Springfield, VA 22161. Phone: (703) 487-4600.

This document is also available through the Defense Technical Information Center (DTIC). DTIC provides access to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential contractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center / Attn: BRR / 8725 John J. Kingman Road / Suite 0944 / Ft. Belvoir, VA 22060-6218. Phone: (703) 767-8274 or toll-free in the U.S. — 1-800 225-3842).

Table of Contents

Implications of Distributed Object Technology for Reengineering

Abstract: Distributed object technology is profoundly changing the ways in which software systems evolve over time. To a large extent, the focus of reengineering has been to understand legacy systems and to extract their essential functionality so that they can be rewritten as more robust and more maintainable systems over the long term. However, object technology, wrapping strategies, and the Web may be changing the focus and economics of reengineering. The question posed by this paper is the extent to which reengineering strategies ought to continue to use program understanding technology. The cost/benefit ratio of certain forms of program understanding appears to be staying roughly the same over time, while the cost/benefit ratio of wrapping legacy systems or their subsystems is dropping rapidly. As a result, new reengineering strategies that place less emphasis on deep program understanding, and more emphasis on distributed object technologies, should now be considered.

1 Introduction

Software systems have become larger, more complex, and more long lived over the years. Early models of the software life cycle had systems being maintained for a number of years until they were retired and replaced. Current models of the life cycle tend to view systems (when properly constructed) as capable of continuous evolution over time [Tilley 95]. One persistent dilemma of system and software engineering is how to move a large body of legacy code from its current state to a state in which it can evolve in disciplined ways.

Reengineering activities improve one's understanding of software and improve the software itself. The choice of an evolution strategy, and the choice of what parts of a legacy system to evolve, depend on both technical and business choices. The technical choices depend on the state of the legacy system and the tools that are available to reengineer it. The business choices are based on two related cost/benefit tradeoffs. First, there is the tradeoff between doing business the old way versus adopting a new enterprise strategy. Second, there is the tradeoff between maintaining an existing system versus replacing the existing system with one having lower operating and maintenance costs.

Legacy systems that are undocumented, brittle, and no longer serve their intended purpose suggest significant structural changes. Legacy systems that are well documented, robust, and meet most enterprise needs suggest less drastic measures. Between these two extremes lie the interesting cases where the relative costs and benefits of various strategies must be analyzed in detail. One such intermediate case is the situation in which software is required for the enterprise mission, but becomes unmaintainable due to deficiencies in documentation, structure, or performance. Here the choice of whether and how to evolve the software becomes more fuzzy. Unfortunately, there is little quantitative data to justify the decisions that must be made, but that situation is slowly improving.

The question posed by this paper is the extent to which reengineering strategies ought to use program understanding (i.e., deep understanding of program structure) versus wrapping what exists with less complete understanding as a basis for system evolution. To what extent should we apply program understanding tools versus wrapping tools as a solid foundation for moving forward? The answers depends on the costs of applying the applicable tools relative to the benefits of the evolved system. In the technical dimension, one looks at the technical benefits of the improved software system relative to the technical costs of employing the tools. In the business dimension, one looks at the business benefits including time to market and new business opportunities, relative to both the short-term development costs and the long-term maintenance costs.

Our hypothesis is that the fine-grained, exhaustive approaches using program understanding for system reengineering will be supplanted by higher level approaches that will make software evolution cheaper, faster and smoother. This hypothesis is based on current dramatic shifts taking place in object-oriented programming and distributed object technology (DOT) on the Web and the associated powerful tools for software evolution. In particular, the tools provide software integration technology that has heretofore prevented large-grained use of subsystems as well as distribution technology that makes applications ubiquitous and easy to use. As these trends continue, it will often make more sense to wrap legacy applications than it will to understand them in depth and then to transform them into modified systems. Distributed object technology is not completely mature and robust in all cases, but it is driving down the cost of wrapping much faster than reverse engineering technology is bringing down the cost of deep program understanding.

In the remainder of the report we will first provide some definitions of various software evolution concepts and a taxonomy of activities. Then we will provide an overview of distributed object technology with examples of tools from that technology. Then we will give examples of the use of the technology and its implications for software maintenance and evolution. Finally, we will discuss the implications for the future of software maintenance and reengineering. For a more complete treatment of distributed object technology itself with more fully elaborated examples, refer to the companion paper *Distributed Object Technology with CORBA and Java: Key Concepts and Implications* [Wallnau 97].

2 Assessing the Health of Legacy Systems

Legacy systems are like living organisms. They exist in the context of an environment that influences their state of health. They can become more healthy or less healthy depending on changes in the environment and the treatment they receive. We define three broad categories of legacy system health: healthy, ill, and terminally ill. In the healthy state, legacy systems satisfy the current enterprise needs and are kept healthy by routine maintenance. In the ill state, the legacy system's health has deteriorated to the point that some kind of non-routine intervention is required. In the terminally ill state, the life of the system can be prolonged by extraordinary life support, but heroic measures are required and are often not economically justified. We will elaborate further on each of these states and the implication for possible treatments in each state.

When a system is healthy, the enterprise strategy and the legacy system are in harmony. Either the legacy system is satisfactorily handling current enterprise needs or the needs are changing in relatively minor ways such that the legacy system can be updated and maintained in a timely and economical fashion. There are three types of maintenance: corrective, perfective, and adaptive. Corrective maintenance refers to the correction of errors in the legacy system. Perfective maintenance includes enhancements to both the functional and non-functional attributes of the legacy system (for example, adding new capabilities or improving the resource usage or documentation). Adaptive maintenance is concerned with adaptations to meet changing requirements of the operating environment (for example, a new communication protocol or a new version of the operating system).

When a system is ill, it may be that way for one of three reasons. Either it was created that way, its internal systems deteriorated over time due to advancing age, or some external environmental influence caused the illness. So it is with legacy systems. Some are poorly designed and constructed to start with. This situation is frequently observed, but it is understandable as we continue to learn better ways to build systems. Some legacy systems start out in a good state of health, but decline when less-than-adequate routine maintenance is performed due to time or budget constraints. The conceptual integrity of a system is not always maintained after a key person leaves the team and when patch after patch is made to the system without a disciplined process. Finally, a legacy system's health may deteriorate due to external influences. These external influences may include new software standards or new demands for connectivity that emerge over time. If modifications to accommodate these influences include additional layers of software, they can often have a deleterious effect on system health.

When a system is terminally ill, its return to a healthy state is not expected. Either the enterprise need has progressed far beyond the capabilities of the software, or the software is in such a case of disrepair that it cannot be economically restored to good health. It is important to recognize this state, since without this recognition, this state can become the software equivalent of "golden handcuffs." Managers are understandably reluctant to part with software in which they have invested heavily. When time and money and politics are involved, it is hard

to let go, even when it is obvious to a dispassionate observer that the time of the legacy system has passed.

Assessment, diagnosis, and prognosis is as difficult in legacy system health as it is in human health. Careful analysis is required, but even after careful analysis experts may disagree about the nature and severity of the ailment, the nature of the treatment, and the prospects for recovery. Just as in human health, the technology for treating legacy system ill-health marches on. The next section will present a taxonomy of operational activities. Following that we will describe promising technologies in more detail.

3 A Taxonomy of Operational Activities

In this section we elaborate on some of the activities that are appropriate for both healthy and unhealthy legacy systems. The activities are assessment, maintenance, transformation, replacement, and combination strategies. There are many overlapping subsidiary activities in these five major areas. The subactivities mentioned under each major category are those most critical to that particular area, rather than the only activities that might be undertaken.

3.1 Assessment

The most important point about assessment is that it should be carried out in stages of increasing detail, with the possibility of terminating the analysis whenever it is possible to reach a valid conclusion. In other words, this is a breadth-first, rather than a depth-first, search of the information space that represents the legacy system. Some systems are so brittle and in such a state of disrepair that it is immediately obvious that a program understanding effort would be of little use. If there is nobody left that knows anything about the system and there is no source code available for the system, it is a sure sign that further analysis will be futile and that the system is terminally ill. On the other hand, if there are documents to view and people to talk to, it is worth looking further with scans of the tables of contents of the documents and short interviews with principals. If these activities are promising, then more detailed reading and interviewing takes place. Cost analysis should always be a factor in the assessment of the health of a legacy system. When the cost of understanding a legacy system and diagnosing its problems is the same order of magnitude as the cost of replacing it, then management should start to question the wisdom of continuing the diagnostic procedures.

Some of the activities associated with assessment are the following:

- analyzing enterprise goals
- preparing a strategy and budget for diagnostic procedures
- analyzing system usage patterns
- analyzing maintenance history
- analyzing architectural structure
- analyzing the system knowledge base
- selecting and applying diagnostic tools
- assessing the current system's state of health

3.2 Maintenance

Until relatively recently, maintenance was the only operational activity associated with system change and evolution. Legacy system were built and then maintained until they were replaced. Systems were not built anticipating major structural changes. They were changed with small localized changes. Bugs were corrected, systems were perfected by changing software artifacts or software characteristics, and functional enhancements were made to respond to new

requirements or to new operating environments. Often, the compound impact of many small changes is significantly greater than the sum of the individual changes due to the erosion of the system's conceptual integrity.

Just as there are standard processes for building systems, there are standard processes for maintaining systems. Many of the characteristics of these processes overlap with the more aggressive forms of system evolution activities described below. The subactivities most closely associated with the maintenance activity include the following [Choi 92]:

- assessing and using maintenance workbenches and static and dynamic analysis tools
- understanding the existing code (program comprehension)
- determining where to make changes
- assessing the impact of the changes (impact analysis)
- rebuilding code after changes
- regression testing to validate changes

3.3 Transformation

Transformations are a form of restructuring (more extensive than maintenance described above) that are appropriate for systems that are either healthy or ill. The assessment activity may have determined that large parts of the system can be reused as it evolves to greater functionality or to a healthier state. We distinguish between two types of transformations depending on whether an understanding of the internals of a legacy system is required (white-box transformation) or just the external interfaces of the respective subsystems is required (black-box transformation). White-box transformation encompasses a form of reverse engineering that emphasizes deep understanding of individual modules and internal restructuring activities. Black-box transformation encompasses a form of reverse engineering that emphasizes shallow understanding of module interfaces and wrapping activities. Transformations of both kinds could be performed on different subsystems within the same system. White-box transformations can be considered the more technically challenging of the two activities and require that the legacy system be in somewhat better health. Another way of thinking about this distinction is to consider white-box transformations as a form of low-level reengineering (analogous to programming-in-the-small) and black-box transformation as a form of high-level reengineering (analogous to programming-in-the-large).

In deciding which of the two transformation paths to follow, the following activities are undertaken:

- assessing the health of the system specifically as it relates to encapsulation, modularity, coupling, and cohesion
- assessing the skills of the personnel available
- assessing the capabilities of the technology accessible within the business unit
- modeling, as necessary, of the application domain, domain-specific architectures, architectural principles, and system components

- selecting and using of decision analysis tools
- assessing the costs, schedules, and risks of various courses of action
- planning the transition strategy

White-box transformation

Program understanding is a white-box form of reverse engineering that consists of activities aimed at recovering lost structure and documentation, so that system enhancements can start from a solid foundation. Reverse engineering has been defined as “the process of analyzing a subject system to identify the system’s components and their interrelationships, and create representations of the system in another form or at a higher level of abstraction [Chikofsky 90].” The program understanding process is one of modeling the domain, extracting information from the legacy system using appropriate extraction mechanisms, and creating abstractions that help in the understanding of the resulting structures [Tilley 95]. Outcomes include redocumentation of the architecture and the program structures and recovery of the design.

The activities that are specific to white-box transformation include the following:

- interviewing of design, development and maintenance teams
- constructing domain-specific models of the application using conceptual modeling techniques
- static and dynamic analysis of source code using appropriate extraction mechanisms
- creating abstractions that facilitate program understanding and permit navigation, analysis, and presentation of the resultant information structures

Black-box transformation

Wrapping or encapsulation is a black-box form of reverse engineering that tries to avoid the problems of understanding the internal structure of the legacy system. Rather, it builds a cocoon around units of software that are working well, serve a well-defined need, and have clearly defined external interfaces. Encapsulation can take place at various levels: the job level, the transaction level, the program level, the module level, and the procedure level [Mattison 94]. Wrapped subsystems are often called “servers” since they provide a well-defined service function. Examples include database servers, map servers, simulation servers, or communication servers. Wrapping is concerned with packaging and is consistent with the precepts of object technology (OT). The very notion of objects is tied to encapsulating “state” with a set of “methods” to manipulate that state. Wrapping activities are less demanding than deep program understanding since only the external interfaces must be understood. However, the limitations of wrapping are inherent in the limitations of the existing functionality. The existing subsystem must provide a service for an existing need that will be static over time. Furthermore, both the good and bad software attributes of the original subsystem remain.

The activities that are specific to wrapping include the following:

- identifying encapsulated data and databases
- identifying encapsulated functions

- evaluating middleware solutions
- wrapping of data and functions
- writing glue code

3.4 Replacement

Replacement is appropriate for legacy systems that are terminally ill, such as when the software can no longer be maintained cost effectively or when the hardware platform is no longer supported. In many cases, new enterprise strategies require substantial changes in software capabilities. For example, if an enterprise finds a corporate intranet that allows for advantages such as replacement of paper expense reports and purchase requests with online forms, the old transaction-oriented processing systems may have to be replaced with Web pages with net-accessible forms. The reengineering of the enterprise, in this case, may require the replacement of all or most of the batch-oriented systems. A detailed understanding of the internal structure of the legacy systems does not make economic sense in order to move forward.

Replacement calls for the following activities:

- reassessing the enterprise goals and objectives as well as customer needs
- formulating system requirements
- evaluating legacy system assets
- analyzing cost tradeoffs
- planning, design, testing, and evaluating the new system
- formulating a transition strategy
- implementing the transition plan, including
  - organizational infrastructure readiness
  - site preparation
  - system installation
  - trial deployment
  - system evaluation and acceptance
  - user training and support

3.5 Combination Strategies

Sometimes a combination of different activities is called for. In fact, a combination strategy is often the soundest and most cost-effective approach. Parts of the legacy system can be fruitfully maintained, others can be transformed by white-box techniques, others can be transformed by black-box techniques, and still others can be replaced. This is a holistic approach to a disciplined evolution of systems, which addresses each part of the whole system in the appropriate way. Careful analysis is required on several fronts. How well is the system satisfying its needs? How is its internal health? What pieces of the whole system require the most treatment? What technologies will provide the most benefit to the system for the least cost?

Only after these questions are answered can a plan be developed for the maintenance, restoration, or creation of the good health for a legacy system.

Among the activities unique to a combination strategy are the following:

- assessing the encapsulation/modularity properties of the system
- assessing separate components with respect to their enterprise goals and technical soundness
- identifying the system components to be maintained, transformed, or replaced

Since the conjecture of this paper is that DOT has a profound impact on reengineering, we next describe that technology and how it is being used. We then give some specific examples of that technology and provide case studies of how the technology is being used in practice. We make the case that the technologies and most of the examples given fall clearly in the transformation taxonomy category above and specifically in the “black-box” subcategory. The examples have been chosen to provide evidence for the conjecture that the relative cost and practicality of wrapping favors its use compared to deep understanding.

4 Distributed Object Technology and the Web

DOT is evolving from desktop client/server computing just as desktop computing evolved from mainframe computing (see, for example, [Hamilton 96]). It is driven by two key enabling technology areas: Web computing and middleware. Web browsers such as Netscape and Internet Explorer, together with universal resource locators (URLs), hypertext markup language (HTML), and Web-oriented general purpose languages (such as Java), serve to merge networks within an organization (intranets) or universally (Internet). With the click of a mouse, client machines can access servers anywhere in the corporation or anywhere in the world. Web browsers provide the access to global information.

Java computing (a prominent example of Web-enabled distributed object technology) is composed of an object-oriented programming language, its associated class libraries, and the definition of a Java virtual machine. The Java virtual machine is delivered with the Web browser, which provides "mobile objects" or "executable content" within Web pages. Java "applets" generate even more interest in Web technology because they allow Web page designers to add interaction and animation to previously static pages. An applet is invoked by an "applet tag" within an HTML page in a browser. The browser recognizes the applet tag, retrieves the Java applet from the server, and then executes the applet using the Java interpreter. Thus Java computing brings mobile objects to the Web on a universal machine.

Middleware is the technology that facilitates integration of components in a distributed system. It is software that allows elements of applications to interoperate across network links, despite differences in underlying communications protocols, system architectures, operating systems, databases, and other application services [Rymer 96]. While the details of these technologies vary considerably, middleware products usually provide a run-time infrastructure of services for use by components to interact with each other. Middleware makes it possible to develop architectural patterns that represent innovative design solutions for specific system design problems. This technology provides the object-oriented link between the Java applets and the new or legacy information systems. It provides the integration technology that enables distributed objects to interact over the Web. One example of this technology is the Common Object Request Broker Architecture (CORBA) [OMG 95], an interfacing standard promulgated by the Object Management Group (OMG), a consortium of over 600 member organizations.

4.1 CORBA and Java

This section describes how two prominent DOT tools, namely CORBA for middleware and Java applet technology for Web computing, are having a profound impact on reengineering legacy systems. We are using these examples to make the discussion more concrete and because they are the most visible and popular representatives of their respective classes. While this may color our discussion of DOT with the peculiarities of CORBA and Java, our intent is to address the broader aspects of reengineering with such tools.

To provide the necessary background and perspective, we give a brief explanation of the genesis of these tools and their current state. They descend from two broad classes of DOT progenitors. Operating systems and distributed systems infrastructures influenced CORBA, and programming languages and the Web gave way to Java.

4.2 Operating Systems/Distributed Systems Influence and CORBA

The key concept for DOT stemming from operating systems is interconnection technology. Many levels of abstraction can be used to describe the connection between machines on networks. As early as 1980, an Open Systems Interconnection (OSI) architecture [Zimmermann 80] defined a hierarchy of seven layers, each representing a collection of communication functions from bits and bytes at the lowest level to application protocols at the highest level. "Sockets" between processes on the same or different machines became the primary mechanism provided by operating systems to connect distributed systems in networks. Remote procedure calls (RPCs) became a higher level (more abstract) mechanism for using the sockets.

As object technology became more popular through the 1980s, there was more interest in bundling the concept of objects with the concept of transparent distributed computing. Objects, with their inherent combination of data and behavior and their strict separation of interface from implementation, offer an ideal package for distributing data and processes to end-user applications. Objects became an enabling technology for distributed processing. In the early 1990s, the OMG defined a standard for the distribution of objects [OMG 96]. The standard defined CORBA, which provided a standard by which object technology could be used in distributed computing environments. The latest version of this standard, CORBA 2.0, addresses issues related to interface, registration, databases, communication, and error handling. When combined with other object services defined by OMG's Object Management Architecture (OMA), CORBA becomes a middleware standard that facilitates full exploitation of object technology in a distributed system. However, if we were to characterize CORBA technology in the simplest possible language, it would be to say it is an object-oriented RPC.

CORBA is concerned with interfaces and does not specify implementation. It is a standard for which there are many (at least a dozen at present) current products referred to as Object Request Brokers (ORBs). Among them are ORBIX by IONA Technology, NEO by SunSoft, ObjectBroker by Digital, VisiBroker by VisiGenic, PowerBroker by Expersoft, SmallTalkBroker by DNS Technologies, Object Director by Fujitsu, DSOM by IBM, DAIS by ICL, SORBET by Siemens Nixdorf, and NonStop DOM by Tandem. The OMG sponsors a permanent showcase on the Web to demonstrate the interoperability between ORBs from various vendors according to the CORBA 2.0 specification. The major product entry that is not CORBA-compliant is Microsoft's Distributed Component Object Model (DCOM).

4.3 Programming Language/Web Influence and Java

The key concept coming from programming languages together with Web technology is that of “mobile objects” or “executable content,” which is possible with Java. Java is “a blue collar language” that includes features from C, C++, Smalltalk, Simula, Mesa, and Modula [Gosling 96]. As a language, Java is appealing but not revolutionary. It is strongly typed and provides garbage collection, dynamic linking, interfaces, packages, exception handling, and built-in support for threads at the language level. It is much more minimalist than Ada 95 and C++, and is not a hybrid language as are both of these and many of the other object-oriented languages (CLOS, Visual Basic, Object Pascal, Objective C). It is more traditional in its syntax and semantics than Smalltalk. Because it borrows from so many other languages, Java feels familiar to most programmers. Rather than producing machine-specific instructions, Java is translated into vendor-neutral bytecode. The Java virtual machine (JVM), which lives in an operating system or a Web browser, translates the bytecode into the machine-specific instructions. This gives Java its platform independence, making Java applications completely portable.

The real promise of Java is how it is being used and delivered in the context of the Web. With the release of Netscape 3.0 in August 1996, “Java applets” became executable on all major platforms that Netscape supports. Namely, it became possible to execute Java programs on UNIX boxes, Windows/Intel machines, and Macintoshes by using a Web browser and by accessing pages on the Web that have executable content (the applets). The Web thus becomes the delivery vehicle for programs that can execute on a client machine and that can reach back to the server for customized information retrieval.

This power is derived not from the language per se, but from the architecture-neutral approach used by Java. The JVM, including the Java interpreter, is part of the version of Netscape Navigator delivered for each platform. First, an applet written in Java is transported to the browser, then the applet is executed using the browser’s interpreter. While the execution of applets is somewhat slow, there are “just-in-time” (JIT) Java compilers that will compile the bytecode on the client machine before executing them, resulting in improved performance.

Another key component of the Java technology is the application programming interfaces (APIs) that are standard, but separate from the language. These include the Abstract Window Toolkit (AWT) that provides a complete set of classes for writing graphical user interface (GUI) programs. In addition there are classes for applets, input/output, networking, and debugging. With these additional classes it can be said that Java is decidedly more “net aware” and “Web aware” than other object-oriented programming languages.

CORBA and Java each offer substantial support for the construction of distributed systems. Together, they present application developers with unprecedented capability to build new systems and to reengineer legacy systems. DOT provides architectural choices not previously available and suggests new development and engineering processes. In the next sections we explore these new capabilities and processes.

5 The Impact of CORBA on Reengineering

Software architecture is a topic of considerable interest in the 1990s. While many different perspectives exist, there is general agreement that software architecture deals with design abstractions for system-level structural concerns. By “system-level” we mean something larger than a single computer program. As noted by Garlan and Shaw [Garlan 93], such concerns include “gross organization and global control structure; protocols for communication, synchronization, and data access; physical distribution;” etc. It is at this higher abstract level, the architectural level, that reengineering must distinguish itself from software maintenance. It is also at the architectural level that the costs and benefits of wrapping versus deep program understanding are most highly leveraged.

DOT provides mechanisms that address many of these concerns in a way that builds upon already proven abstraction mechanisms (objects, messages, inheritance) available in object technology. The significance of these distribution extensions to object technology is that software designers now have at their disposal the means of expressing abstract system designs and, more importantly, now have tools for quickly fabricating working versions of these designs. That is, there is a more direct path now from abstract architectural concepts to concrete implementation of these concepts. These concrete implementations frequently are constructed from large-grained legacy components.

It is useful at this point to draw a comparison between the use of DOT for architectural patterns and the use of middleware technologies. Recall that middleware refers to technologies that facilitate integration of components in a distributed system. In software architecture terms, middleware products provide services corresponding to well-known architectural idioms (e.g., blackboard and implicit invocation).[1]

DOT provides more general, but lower level building blocks upon which a variety of middleware services can be implemented. Thus, one would use a DOT such as CORBA to define a collection of related object types that implement a message-based implicit invocation system. Further, the object-oriented heritage of DOT makes it feasible (and possibly desirable) to define these middleware object types more abstractly in the form of a design pattern that can be tailored to specific design problems [Gamma 95]. It is also possible to use DOT to develop architectural patterns that do not correspond to middleware, but instead represent innovative design solutions for specific system design problems.

5.1 Examples of CORBA-Based Wrapping of Legacy Systems

To illustrate the use of DOT to express architectural abstractions and to demonstrate the potential impact of CORBA on reengineering, we briefly present three examples that use com-

1. We use the term *pattern* to refer to an object-oriented style of describing architectural idioms and their implementations.

mercially available implementations of CORBA. The first two are relatively small prototypes meant to demonstrate proof-of-concept. The third is a large system in full-scale production. Later, we shift our attention to the impact of Java on software system design and reengineering. Although experiences with architectural uses of Java are more limited than is the case with CORBA, they emphasize different aspects of distribution than CORBA, and even limited experiences with Java reveal the potential impact on reengineering concepts. These examples were selected to illustrate the changing economics of evolving legacy systems with wrapping techniques.

5.1.1 Middleware for Distributed, Real-Time Simulation

The Defense Modeling and Simulation Organization (DMSO) is leading the development of the High-Level Architecture (HLA), an architecture for modeling and simulation within the US Department of Defense (DoD). One component of the HLA is the run-time infrastructure (RTI), a collection of services supporting (among other things) real-time interoperation among legacy simulations (as, for example, in joint simulation exercises [Calvin 96]). A prototype RTI was implemented by Mitre and Lincoln Labs using Iona Technology's ORBIX, a CORBA-compliant ORB.

Using this RTI, simulations are viewed as "black boxes" that execute remotely from the central middleware services. There is a subtle point worth noting about the nature of the RTI services: their focus on coordination rather than functionality. The underlying idea is that the difficult design issues in distributed systems are related to coordinating the execution of existing functionality (e.g., synchronization, currency, consistency) and that design abstractions should therefore focus on these issues, rather than on questions of what functionality is provided by specific components. A discussion of these ideas can be found in [Abowd 93]. However, the key point is that DOT provided the means for developing a novel and innovative design solution without modifying the legacy simulations.

5.1.2 Structural Patterns for Distributed Component Integration

The second example is a prototype toolkit to aid in manufacturing engineering design. The toolkit was developed by the Software Engineering Institute, the Manufacturing Engineering Laboratory of the National Institute of Standards and Technology, and Sandia National Laboratory. Details of this case study are described in [Wallnau 96]. The essential design problem was to use CORBA to update a system of (legacy) manufacturing engineering design tools developed over many years by Sandia, called the SEACAS toolkit. The updated toolkit would make these non-distributed, platform-specific tools remotely accessible (including access from a wide-area network) to end users on different platforms. Further, the tools could be integrated at the level of remote objects.

Unlike the DMSO RTI example, the design approach taken in the SEACAS project illustrates the use of DOT to develop design patterns (albeit simple ones). The primary design pattern, and its implementation in CORBA, illustrate several aspects of how DOT supports innovative

design. This “Consistency Manager” pattern concerned the integration of off-the-shelf components; namely it answered the question of how to “wrap” components in order to make them work together. To date, component wrapping has been an undisciplined and ad hoc procedure (although there have been attempts to make this a more rational process). The pattern, however, provided a uniform set of wrapping requirements. Rather than having to view each wrapping problem in terms of unique tool functionality, each tool instead had to be wrapped in such a way as to interact with a very limited repertoire of coordination primitives already described by the patterns. This both simplified the wrapping process and provided insight into pattern-based techniques for “qualifying” components for fitness for use. Furthermore, this pattern addressed the coordination aspects of the design problem, i.e., how the SEACAS tools would execute and interact in the face of long design sessions and potentially unreliable (wide-area) network connections.

5.1.3 CORBA-Based Production Application System for Online Banking

CORBA is often viewed as a risky new technology, but it has been used successfully in large commercial applications. One example 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 [Wells Fargo 97]. The system has 100,000 enrolled customers and was handling 200,000 business object invocations per day as of early 1997 [Townsend 97].

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. In an independent audit in late 1996, after evaluating the risks and benefits, the accounting and consulting firm KPMG recommended the continued use of CORBA technology and confirmed Digital’s ObjectBroker as an appropriate product for their application. Wells Fargo found that the key to enabling reuse of legacy systems was in having, maintaining, and sharing a well-architected enterprise object model.

5.2 Summary

Note that in each of these three examples, an existing legacy system has been transformed into a system with enhanced functionality without requiring a deep understanding of the internal structure of that system. Rather, what is required is an understanding of that system’s external interfaces in terms of the enterprise objects. In each case the legacy system has undergone a black box transformation to satisfy the enterprise’s needs. The greater the degree to which the object model has been already been accepted in the legacy system (as in

the final example), the easier will be the transformation. But starting with a non-object based system does not preclude wrapping and creating an object infrastructure for coordination or for developing design patterns (as in the first two examples).

6 The Impact of Java on Reengineering

Java is a much more recent technology than CORBA, but it is clear that it has already had, and will continue to have, an enormous impact on reengineering in the DOT marketplace. The reasons for this are simple: the affinity of Java with the Web, the fact that the Web and browsers are now part of the standard computing vocabulary, and the integration of Java with Web browsers that run on all popular computing platforms. As already noted, Java as a language is cleaner and leaner than C++. Moreover, the integration of Java with the Web adds a dimension to object technology not available in languages such as C++, namely mobile objects. That is, with Java it is possible to write object implementations that are delivered to the end user's execution environment, along with any supporting class libraries that are needed to implement the object. These mobile objects are the essence of the powerful Java applets.

One way of differentiating Java from CORBA is that Java provides mobile objects within a homogeneous computing environment (i.e., all computation occurs within a Java virtual machine), while CORBA provides remote objects assuming a heterogeneous computing environment. In the Java case, there is the advantage of uniformity and portability across many physical machine architectures. The data types on the virtual machine are coerced to be the same on all platforms. While this certainly provides a cleaner global implementation model, there are also some costs. The most obvious is the architectural mismatch between the virtual machine and the actual machine that must be resolved by software. This mismatch is resolved by a Java interpreter or JIT compiler on each physical architecture. It is important to remember that the Java interpreter and these JIT compilers are part of the JVM included in the Web browser.

Recent developments in both CORBA and Java, however, have begun to blur this distinction. The newest version of Java, Version 1.1, supports a feature called remote method invocation (RMI). RMI provides a naming service for Java programs to look up object references for objects that are executing on remote Java virtual machines, and to then invoke methods on these remote objects. In effect, RMI provides ORB functionality fully integrated with the Java language and runtime environment. Unlike CORBA, however, the RMI ORB is fully integrated with the Java language and runtime environment. Thus, while CORBA interfaces are described using an architecture-neutral interface description language (IDL), interfaces of remote Java objects are described using the Java interface construct.

Just as Java has been adding CORBA-like capabilities, CORBA has been evolving to accommodate the impact of Java. Most notably, ORB vendors are now supporting the development of Java clients for CORBA objects. Java applets that access remote CORBA objects (such applets are called "orblets") have the effect of pulling CORBA into the Web, and therefore make it feasible to consider the use of Web browsers as delivery vehicles for CORBA-based object services.

So in terms of legacy system reengineering, the influence of Java is not the capability of providing animated applets, or providing the "thin client" interface to existing databases. Rather,

its influence is in general distributed business applications, with the initial domain being electronic commerce [Chung 97, Hamilton 97]. The impact of Java is the ability to integrate the distributed object environment that is provided by middleware products such as CORBA. Java does this better than other languages because it is platform independent (because of the virtual machine definition) and because it has the market momentum that is driving powerful new class libraries and APIs specifically designed for integration and reuse of legacy systems. More than other languages, Java provides the infrastructure for wrapping and integrating the enterprise objects that form the basis of a distributed object system.

The Java Enterprise API supports connectivity to enterprise databases and legacy applications and it consists of four separate areas.

1. Java Database Connectivity (JDBC) is a standard Structured Query Language (SQL) database access interface that provides uniform access to a wide range of relational databases.
2. Java RMI is a remote method invocation between peers, or between client and server when applications at both ends of the invocation are written in Java.
3. Java IDL provides seamless interoperability and connectivity with CORBA. It also includes the Java/IDL Language Mapping Specification, an IDL-to-Java compiler, and a portable Java ORB core that supports the Internet InterORB Protocol (IIOP), which allows developers to build Java applications that are integrated with heterogeneous business information assets.
4. The Java Naming and Directory Interface (JNDI) provides a Java application with a unified interface to multiple naming and directory services in the enterprise.

The first three components of the Enterprise API are available now, but the fourth and many other APIs, many of which support reengineering goals, are in various stages of development. Sun Microsystems is working with industry leaders on specification and implementation issues. The current list and schedule can be found at [Javasoft 97a].

Today, there are fewer examples of Java or CORBA/Java applications than there are CORBA applications, but there has been a good deal of testing and prototyping (e.g., see [Harkley 97]). The growing sentiment is that CORBA/Java applications can be written today and that they provide the best platform for creating Web-based client applications. In the area of electronic commerce, CommerceNet (a consortium of 250 member companies launched in 1994, which seeks solutions to technology issues, sponsors industry pilots, and fosters market and business development) has chosen Java as its implementation language [Tenenbaum 97]. The Java API to support this set of purchasing, banking, and finance applications is the Java Electronic Commerce Framework (JECF). It provides a user interface for online purchasing and other financial transactions, a secure encrypted wallet database, access to strong cryptography, applets, and a purchasing infrastructure. The status of this API can be found at [Javasoft 97b].

7 The Economics of DOT and Reengineering

Examples of the use of Web technology to evolve legacy systems are abundant. Laddaga and Veitch [Laddaga 97] point out that DOT "is specifically designed to support rapidly changing software with cost proportional to that of the change, rather than the size of the entire application." In a series of studies, International Data Corporation has conducted four in-depth economic analyses at major corporations. These well-documented studies show a return on investment averaging well over 1000 percent [Campbell 96]. They emphasized three themes: rapid deployment on heterogeneous platforms, widespread acceptance and use due to ease of use of the browser technology, the realization of the promise of openness, and the ability to replace components at will. In all cases, the use of intranets enabled the companies to take advantages of new enterprise strategies. The studies were conducted at Cadence Design Systems, Inc.; Booz, Allen & Hamilton; Silicon Graphics, Inc.; and Amdahl Corporation.

The projects ranged in cost from $1.4 million to $4.2 million and are described briefly in the paragraphs that follow. Whereas previous examples stressed the technology aspects of moving to middleware and network computing, these examples stress the enterprise and economic aspects. They also stress the wrapping of data rather than the wrapping of functionality. In the following discussion, we define return on investment (ROI) as the effective interest rate that an enterprise entity receives for an investment (i.e., how much more than a dollar is returned for every dollar spent to implement and operate the system). Payback period is the amount of time it takes to recoup the costs of implementing the system.

Cadence Design Systems is a leading supplier of electronic design automation (EDA) software tools and professional services. The company employs over 3,000 people and has a product line of over 1,000 products and services. Their legacy system supported the sales force with point-to-point communication and relied on personal relationships. It was replaced with a system that allowed the use of a wide range of hardware to access reference documentation, forms to facilitate internal processes, and templates and materials used to persuade customers. The total 3-year cost of the system was $1.4 million with a projected 3-year savings of $7.6 million. Much of the savings were in reduced training time for new personnel. The 3-year ROI was 1766% with a payback period of 0.15 years.

Booz, Allen & Hamilton is an international management and technology consulting firm with over 6,500 employees. Their challenge was to address information-sharing needs in a way that would be time- and cost-effective for employees around the world, but also be helpful to a collaborative and team-supporting structure. They needed to support a consultants' ability to create and share knowledge, and they wanted to keep the legacy corporate infrastructure rather than discard it. They chose an international intranet for its ease-of-use, large number of vendors, and open standards. Over 18 months, 6 developers developed Knowledge On-Line (KOL), a Netscape-based intranet with links to internal knowledge systems and external stores of information. This project showed a ROI of 1389% over 3 years with a payback period of 0.19 years.

Silicon Graphics pioneered the development of visual computing in developing graphics workstations, multiprocess servers, advanced computing platforms, and applications software. They have 10,000 employees in 100 offices worldwide. Their application was the automation of the process of purchasing standard items for employees, including the selection of items and routing of the purchase requisition for approval. It was necessary to predefine a catalog and streamline the communication to suppliers. Using a Web-based electronic interface, the employees interact with relational databases that store information about the employee, the catalog of standard items, and routing information. During an approval process, each person in the process receives e-mail with an attachment that lets the persons click to approve, modify, or reject the order. Orders are then automatically sent out electronically or by fax and dropshipped to the employee's desk. This project showed a 3-year ROI of 1427%. The payback period was 0.18 years.

Amdahl began as a manufacturer of plug-compatible mainframe technology, but is now a supplier of complete enterprise computing solutions. As Amdahl began to acquire new companies and gain leverage from the collective knowledge of its employees, the company needed to find a far-reaching solution to information sharing. Intranet Web technology offered a medium that could support the sharing of information and collaboration, namely an electronic library of information. The use of the corporate intranet has mushroomed to include the following: information from research firms, library information, frequently asked questions, policy and procedures manuals, job openings, tools to view and evaluate profit and loss statements, competitive analysis notebooks, and newsletters. The ROI was computed to be 2063% over 3 years, with a payback period of 0.13 years.

As another example, one of the authors participated in a distributed object technology project called the Meteorological Anchor Desk (METOC) that used some of these technologies [Veltre 97]. A system was built from scratch that provided audio-visual collaboration, a net site that delivered weather products from diverse producers, and a weather browser. It was produced as an evolutionary development delivered in increments over the course of two years with a small team and with considerable user involvement. The weather browser is particularly relevant to this discussion. The weather browser overlays environmental conditions (e.g., temperature, pressure, cloud cover, rain over some period of time, sea conditions, ice thickness) on a map and can, under user-control, show how those weather conditions affect military operations and platforms (e.g., ships, planes, tanks, missiles).

A "weather specialist" was implemented that plugs into a map server that displays the maps. The specialist gets real-time data by calling a data server that provides gridded data products from a database that is updated periodically. CORBA allowed the weather specialist object to register the weather specialist with the ORB. The specialist then waits for client requests. When the map server gets a request to show weather, it asks the ORB for a weather specialist object. Then it asks for the needed information from the object and displays it. The important point of this example is that we provided weather functionality without knowing anything (other than coordinates) about the rendering of the map or other map details such as projections. Similarly, the data server hid the details of how gridded data are produced.

In general, the METOC Anchor Desk was a new application that was required to overlay an existing organizational structure with a legacy of databases and operational procedures. It was found that the evolutionary approach and the DOT tools enabled a rapid development of significant proportion with few resources. Since its initial delivery, the METOC Anchor Desk has continued to evolve in response to requests from the users. This evolution has been smooth and seamless, at least in part due to the Web and middleware technologies that were used in its construction.

8 Conjectures About DOT and Program Understanding

The speed at which DOT is being developed and adopted is perhaps unprecedented. Authors (e.g., [Bruzas 97]) speak of “Internet time” in which events (both technological advances as well as application time-to-market) happen in weeks and months rather than months and years. This is especially true of Java. Predicting future trends in DOT is therefore fraught with uncertainty, and can scarcely be considered as having any scientific basis. Recently (February 1997) the journal *Distributed Object Computing* was launched exclusively to discuss DOT issues. The *Communications of the ACM* had a special section on DOT in the May 1997 issue [Laddaga 97].

The vast number of existing legacy systems represent both a substantial investment and core business functionality. In most cases they are less than healthy. Organizations can neither afford to redevelop these systems, nor can they afford to continue to keep them isolated from current technology improvements. DOT is now sufficiently mature to support legacy system migration using black-box transformations. “Object wrapping” can provide an object interface to the legacy systems. Clients can view the legacy system through a simple CORBA API presented by the wrapper. The wrapping layer can communicate with the legacy system via sockets, RPC, or an API. Once wrapped, the legacy system (or its subsystems) becomes highly reusable software components. The Web, as already noted, can be considered simply another client that needs to communicate with the wrappers. Consequently, the legacy applications will be accessible from the Web. “Together, Java and CORBA-based object wrappers can be used as the basis for a sophisticated and dynamic set of applications to other technologies [Klinker 96].”

Our starting hypothesis was that DOT changes the economics of reengineering because it dramatically lowers the cost of wrapping legacy systems, but only marginally lowers the cost of transformations involving deep program understanding. The reason for this is that white-box transformation is driven by the legacy system artifacts rather than by the available transformation tools. The wrapping problem is driven by the system wrapping tools. If a legacy system is unstructured and has little documentation, then even significant improvements in reverse engineering technology will not make the system a candidate for this form of transformation. However, with significant improvements in wrapping technology, that same system can be evolved at significantly lower cost.

There is ample evidence that reverse engineering is a difficult problem, and although improvements are being made, those improvements are coming slowly. Tilley and Smith [Tilley 95] note that software undergoing maintenance does not fit the mold supported by common maintenance tools, nor do the tools offer significant help for the hardest problems. Many legacy systems are built using multiple languages, many of which may be considered obsolete. Program understanding, redocumentation, and program translation are all areas for which there are tools, but they all depend on a starting legacy system that is tractable.

On the other hand, there is ample evidence that wrapping technology is making rapid strides. Wrapping or encapsulation has been found to be a more economical alternative to either redevelopment or reverse engineering [Sneed 96a]. Even previously intractable COBOL libraries have been successfully recycled with these techniques [Sneed 96b]. As pointed out by Winsberg [Winsberg 95] wrappers hide the “legacy systems mess,” but with the anticipation that in time the legacy system will wither and die. With true interoperability and encapsulation, this seems to be a reasonable expectation because subsystems can be changed without having an impact on other subsystems.

In the case of new developments, time to market and widespread distribution have become two of the key architectural drivers (e.g., see [Campbell 96, Veltre 97, Sun 96]). Only by large-grained use of legacy system components can developers meet these stringent requirements. Even in new developments, component parts will rarely be developed from scratch if the technology exists to integrate them with a new user interface and new coordination components (see Section 5.1.1). DOT and its middleware are now providing the necessary integration technology. Browsers and the Web are providing the distribution technology. Large new developments will rarely, if ever, start from scratch.

9 Summary

There is mounting evidence that distributed object technology will have a profound impact on how legacy systems are going to evolve in the future. One major impediment to large-grained reuse and integration of separately developed systems has been inadequate integration technology or the misapplication of the existing technology (see, for example, [Garlan 95]). Now there are promising signs and examples of large-grained reuse. This is enabling large systems to be built and enabling legacy systems to evolve for much lower cost. It suggests that the white-box transformation strategies should receive less attention than black-box wrapping strategies in the disciplined evolution of legacy systems.

This report has made use of anecdotal evidence to show that reengineering of legacy systems using DOT has become viable and economical. Much progress has been made in this area and the developments are being made at an astonishing pace. The report has *not* dwelt on (traditional) reverse engineering and its viability and cost. But it seems clear that unless there are astonishing new developments in the field of deep program understanding, the economic balance will inevitably shift to shallow interface understanding.

Over then next several years, there will be more and more examples of the use of DOT for the evolution of legacy systems. To be fair, a thorough review of reverse engineering techniques and benefits should be undertaken and the economic benefits compared. The practical question is the extent to which the legacy systems need to be "understood" and the extent to which they can be "wrapped." It should be determined if the class of applications described in this paper are typical, and if not, what classes of applications lend themselves to the new technologies and what classes do not. The design patterns most appropriate for DOT should be investigated and refined. These research outcomes should reorient the goals of reverse engineering to those activities that can be justified on a cost/benefit basis relative to forward engineering.

In short, we have tried to present some of the mounting evidence that reengineering, especially for healthy systems, can proceed by moving directly forward rather than first taking two steps backward. Because of technological innovations, deep understanding of what has been wrought by systems and software engineers in the past now seems to be much less important than understanding how to wrap what we currently have in order to carry it into the future.

References

[Abowd 93] Abowd, G.; Bass, L.; Howard, L.; and Northrop, L. *Structural Modeling: An Application Framework and Development Process for Flight Simulators.* (CMU/SEI-93-TR-14, ADA 271348). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1993.

[Bruzas 97] Bruzas, John and Marchetti, Roseanna K. “Smalltalk and Java.” *Distributed Object Computing* 1, 2 (March 1997): 32-34.

[Calvin 96] Calvin, J. and Weatherly, R. “An Introduction to the High Level Architecture (HLA) Runtime Infrastructure (RTI)," 705-715. *Proceedings of the 14th DIS Workshop on Standards for the Interoperability of Defense Simulations.* Orlando, FL, March 1996. Orlando, FL: University of Central Florida, 1996.

[Campbell 96] Campbell, Ian. *The Intranet: Slashing the Cost of Business*. Framingham, MA: International Data Corp., 1996.

[Chikofsky 90] Chikofsky, Elliot and Cross, James. “Reverse Engineering and Design Recovery: A Taxonomy.” *IEEE Software* 7, 1 (January 1990): 13-17.

[Choi 92] Choi, W.M. and von Mayrhauser, A. “Assessment of Support for Program Understanding.” *Proceedings of the Second Symposium on Assessment of Quality Software Development Tools*. New Orleans, LA, May 27-29, 1992. Los Alamitos, CA: IEEE Computer Society Press, 1992.

[Chung 97] Chung, Winston. “The Next Wave of Java.” *Distributed Object Computing* 1, 2 (March 1997): 28-31.

[Gamma 95] Gamma, E.; Helm, R.; Johnson, R.; and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[Garlan 93] Garlan, David and Shaw, Mary. “An Introduction to Software Architecture.” *Advances in Software Engineering and Knowledge Engineering*. Vol I. River Edge, NJ: World Scientific Publishing Company, 1993.

[Garlan 95] Garlan, D.; Allen, R.; and Ockerbloom, J. “Architectural Mismatch — Why It’s Hard to Build Systems Out of Existing Parts.” *Proceedings of the 17th International Conference of Software Engineering*. Los Alamitos, CA: IEEE Computer Society Press, 1995.

[Gosling 96] Gosling, James. “The Feel of Java.” *Proceedings of the 1996 Conference on Object Oriented Programming, Systems, Languages, and Applications*. San Jose, CA, October 6-10, 1996. New York, NY: Association for Computing Machinery, 1996.

[Hamilton 96] Hamilton, Marc A. “Java and the Shift to Net-Centric Computing.” *Computer* 29, 8 (August 1996): 31-39.

[Hamilton 97] Hamilton, Scott. “E-Commerce for the 21st Century.” *Computer* 30, 5 (May 1997): 44-47.

[Harkey 97] Harkey, Dan and Orfali, Robert. “And CORBA Meets Java.” *Distributed Object Computing* 1, 2 (March 1997): 48,60.

[Javasoft 97a] Javasoft. *Java API Overview and Schedule* [online]. Available WWW: <URL: http://java.sun.com:80/products/api-overview/index.html> (1997).

[Javasoft 97b] Javasoft. *Java Commerce Home Page* [online]. Available WWW: <URL: http://java.sun.com/products/commerce/> (1997).

[Klinker 96] Klinker, P.; Marsh, J.; Tisaranni, J.; and Zahavi, R. “How to Avoid Getting Stuck on the Migration Highway.” *Object Magazine* (October 1996): 47.

[Laddaga 97] Laddaga, Robert and Veitch, James. “Dynamic Object Technology — Introduction.” *Communications of the ACM* 40, 5 (May 1997): 38-40.

[Mattison 94] Mattison, R. *The Object-Oriented Enterprise*. New York, NY: McGraw-Hill, 1994.

[OMG 95] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, Rev. 2.0 (OMG Document 96-03-04). 1995.

[OMG 96] Object Management Group. *Welcome to OMG’s Home Page* [online]. Available WWW: <URL: http://www.omg.org> (1997).

[Rymer 96] Rymer, J. “The Muddle in the Middle.” *Byte*, April 1996.

[Sneed 96a] Sneed, Harry M. “Encapsulating Legacy Software for Use in Client/Server Systems,” 104-119. *Proceedings of Third Working Conference on Reverse Engineering,* Monterey, CA, November 1996. Los Alimotos, CA: IEEE Computer Society Press, 1996.

[Sneed 96b] Sneed, Harry M. “Object-Oriented COBOL Recycling,” 169-178. *Proceedings of Third Working Conference on Reverse Engineering,* Monterey, CA, November 1996. Los Alimotos, CA: IEEE Computer Society Press, 1996.

[Sun 96] Sun Microsystems. *Computing in the Enterprise* [online]. Available WWW: <URL: http://www.sun.com/javacomputing> (1996).

[Tenenbaum 97] Tenenbaum, J.M.; Chowdhry, T.S.; and Hughes, K. “Eco System: An Internet Commerce Architecture.” *Computer* 30, 5 (May 1997): 48-55.

[Tilley 95] Tilley, Scott R. and Smith, Dennis B. *Perspectives on Legacy Systems Reengineering (Draft)* [online]. Reengineering Center, Software Engineering Institute, Carnegie Mellon University. Available WWW: <URL: http://www.sei.cmu.edu/~reengineering/pubs/lsysree/> (1995).

[Townsend 97] Townsend, Erik S. "Wells Fargo's 'Object Express'." *Distributed Object Computing* 1, 1 (February 1997): 18-27.

[Veltre 97] Veltre, Robert and Weiderman, Nelson. "The Meteorological Anchor Desk System: A Case Study in Building a Web-Based System from Off-the-Shelf Components." In *Software Architecture in Practice*. Bass, L.; Clements, P.; and Kazman, R. Reading, MA: Addison-Wesley, 1997.

[Wallnau 96] Wallnau, K. and Wallace, E. "A Situated Evaluation of the Object Management Group's (OMG) Object Management Architecture (OMA)." *Proceedings of the 1996 Conference on Object Oriented Programming, Systems, Languages, and Applications*. San Jose, CA, October 6-10, 1996. New York, NY: Association for Computing Machinery, 1996.

[Wallnau 97] Wallnau, K.; Weiderman, N.; and Northrop, L. *Distributed Object Technology with CORBA and Java: Key Concepts and Implications*. (CMU/SEI-97-TR-004). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1997.

[Wells Fargo 97] Wells Fargo Bank. *Wells Fargo's WWW Homepage* [online]. Available WWW: <URL: http://www.wellsfargo.com> (1997).

[Winsberg 95] Winsberg, Paul. "Legacy Code: Don't Bag It, Wrap It." *Datamation* 41, 9 (May 15, 1995): 36-41.

[Zimmermann 80] Zimmermann, H. "OSI Reference Model -- The ISO Model of Architecture for Open Systems Interconnection." *IEEE Transactions on Communications* 28, 4 (Apr. 1980).

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. **AGENCY USE ONLY** (leave blank)	2. **REPORT DATE** June 1997	3. **REPORT TYPE AND DATES COVERED** Final
4. **TITLE AND SUBTITLE** Implications of Distributed Object Technology for Reengineering		5. **FUNDING NUMBERS** C — F19628-95-C-0003
6. **AUTHOR(S)** Nelson Weiderman, Linda Northrop, Dennis Smith, Scott Tilley, Kurt Wallnau		
7. **PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)** Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213		8. **PERFORMING ORGANIZATION REPORT NUMBER** CMU/SEI-97-TR-005
9. **SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)** HQ ESC/AXS 5 Eglin Street Hanscom AFB, MA 01731-2116		10. **SPONSORING/MONITORING AGENCY REPORT NUMBER** ESC-TR-97-005
11. **SUPPLEMENTARY NOTES**		
12.a **DISTRIBUTION/AVAILABILITY STATEMENT** Unclassified/Unlimited, DTIC, NTIS		12.b **DISTRIBUTION CODE**

13. **ABSTRACT** (maximum 200 words)

Distributed object technology is profoundly changing the ways in which software systems evolve over time. To a large extent, the focus of reengineering has been to understand legacy systems and to extract their essential functionality so that they can be rewritten as more robust and more maintainable systems over the long term. However, object technology, wrapping strategies, and the Web may be changing the focus and economics of reengineering. The question posed by this paper is the extent to which reengineering strategies ought to continue to use program understanding technology. The cost/benefit ratio of certain forms of program understanding appears to be staying roughly the same over time, while the cost/benefit ratio of wrapping legacy systems or their subsystems is dropping rapidly. As a result, new reengineering strategies that place less emphasis on deep program understanding, and more emphasis on distributed object technologies, should now be considered.

14. **SUBJECT TERMS** CORBA, distributed object technology (DOT), Java, legacy systems, reengineering, World Wide Web, wrapping strategies			15. **NUMBER OF PAGES** 33 16. **PRICE CODE**
17. **SECURITY CLASSIFICATION OF REPORT** UNCLASSIFIED	18. **SECURITY CLASSIFICATION OF THIS PAGE** UNCLASSIFIED	19. **SECURITY CLASSIFICATION OF ABSTRACT** UNCLASSIFIED	20. **LIMITATION OF ABSTRACT** UL