Analyzing the Reuse Potential of Migrating Legacy Components to a Service-Oriented Architecture

Grace Lewis, Edwin Morris, Dennis Smith
Software Engineering Institute, Pittsburgh, PA, USA
{glewis, ejm, dbs}@sei.cmu.edu

Abstract
An effective way of leveraging the value of legacy systems is to expose their functionality, or subsets of it, as services. In the business world, this has become a very popular approach because it allows underlying systems to remain largely unchanged, while exposing functionality to a larger number of clients through well-defined service interfaces. The U.S. Department of Defense (DoD) is also adopting this approach by defining service-oriented architectures (SOAs) that include a set of infrastructure common services on which organizations can build additional domain services or applications. When legacy systems or components are to be used as the foundation for domain services, there must be an analysis of how to convert the functionality in existing systems into services. This analysis should consider the specific interactions that will be required by the SOA and any changes that need to be made to the legacy components. We have recently helped an organization evaluate the potential for converting components of an existing system into services that would run in a new and tightly constrained DoD SOA environment. This paper describes the process that was used and outlines several issues that need to be addressed in making similar migrations.

1. Introduction.

With the advent of universal Internet availability, many organizations have leveraged the value of their legacy systems by exposing all or parts of it as services. A service is a coarse-grained, discoverable, and self-contained software entity that interacts with applications and other services through a loosely coupled, often asynchronous, message-based communication model [3]. A collection of services with well-defined interfaces and shared communications model is called a service-oriented architecture (SOA). A system or application is designed and implemented using functionality from these services.

The characteristics of SOAs (e.g., loose coupling, published interfaces, standard communication model) offer the promise of enabling existing legacy systems to expose their functionality as services, presumably without making significant changes to the legacy systems. Case studies are beginning to show that this promise is achievable. Migration to services has been achieved for many domains, including banking applications [9, 10], electronic payment applications [12], and development tools [4].

However, constructing services from existing systems in order to obtain the benefits of an SOA is neither easy nor automatic. In fact, such a migration can represent a complex engineering task, particularly when the services are expected to execute within a tightly constrained environment.

SOA migration tasks can be considered from three different perspectives: 1) the end client or user of the services, 2) the SOA architect, or 3) the service provider. This paper addresses the service provider perspective by describing an approach for making migration decisions. The goal of the paper is to present a case study in migration of legacy components to services, as well as an introduction to the reuse and migration analysis process that was derived from this experience. Section 2 outlines the basic issues in migrating legacy components into services in an SOA. Section 3 discusses the migration case study that solidified our thoughts regarding service migration. Section 4 outlines a process for migration of legacy components to services based on the case study experience. Section 5 provides conclusions and discusses next steps.

2. Creation of Services from Legacy Components.

The most common (but not only) form of SOA is that of Web services, in which all of the following apply: (1) service interfaces are described using Web Services Description Language (WSDL), (2) payload is
transmitted using Simple Object Access Protocol (SOAP) over Hypertext Transfer Protocol (HTTP), and optionally (3) Universal Description, Discovery and Integration (UDDI) is used as the directory service [6].

Enabling a legacy system to interact within a Web services architecture is sometimes relatively straightforward—this is a primary attraction to the approach for many businesses. Web service interfaces are set up to receive SOAP messages, parse their content, invoke legacy code, and optionally wrap the results as a SOAP message to be returned to the sender. Many modern development environments provide tools to help in this process, and commercial organizations are rapidly employing these environments to expose their business processes to the world.

However, characteristics of legacy systems, such as age, language, and architecture, as well as of the target SOA can complicate the task. This is particularly the case when migrating to highly demanding and proprietary SOAs. Such migrations will likely rely less on semi-automated migration, and more on careful analysis of the feasibility and magnitude of the effort involved. This analysis should consider:

1. Requirements from potential service users. It is important to know what applications would use the services and how they would be used. For example, what is the information expected to be exchanged? In what format?
2. Technical characteristics of the target environment. There are many technical underpinnings that need to be understood, especially in proprietary environments, such as bindings, messaging technologies, communication protocols, service description languages, and service discovery mechanisms.
3. The architecture of the legacy system. It is critical to identify architectural elements that could be problematic in the target environment or that could increase the difficulty of the migration effort, such as dependencies on commercial products or specific operating systems, or poor separation of concerns.
4. The effort involved in writing the service interface. Even if it is expected that the legacy system will remain intact, there must be code that receives the request, translates it into calls to the legacy systems, and produces a response. Tools to generate this code may not be available in the particular legacy environment.
5. The effort involved in the translation of data types. Service interfaces usually prescribe a set of data types that can be transmitted in messages. For newer legacy systems and basic data types this can be a small effort, especially if messages are XML documents. But, in the case of complex data types such as audio, video, and graphics, or in legacy programming languages that do not provide capabilities for building XML documents, this effort can be non-trivial.
6. The effort required to describe the services. In an SOA, services advertise their capabilities for other systems to use, and systems find the services they need by using the discovery mechanism prescribed by the target SOA. The more detailed and precise the description of the service, the greater the chances it will be discovered and used appropriately. In critical situations, the description may have to include information about qualities of service, such as performance, reliability, and security; or service level agreements (SLAs) [11].
7. The effort involved in writing service initialization code and operational procedures. Code that is deployed as services will need to initialize itself, announce its availability, and be ready to take requests. This will require the establishment of operational procedures for the deployment of services.
8. Estimates of cost, difficulty, and risk. The information identified in the previous points should provide for more realistic estimates.

3. Case Study.

We were recently involved in an effort to analyze the potential for migrating a set of legacy components from a DoD command and control (C2) system to an SOA. C2 systems are widely used within the DoD and other organizations (e.g., Red Cross, Department of Homeland Security) to direct the movement of personnel and equipment to accomplish a mission.

This section is organized according to the basic high level tasks undertaken by the case study.

3.1 Establishing Stakeholder Context.

We initially met with the government owners of the system and the contractors who had developed the system. At this meeting, an overview of the set of systems, the history of the systems, the migration plans, and the drivers for the migration were presented by these owners and contractors. We were also given a brief orientation to the SOA and were provided with system documentation.

DoD systems have recently focused on the concept of network-centric operations to provide forces with access to integrated information from a variety of previously unconnected sources [1]. This focus requires strong interoperability to ensure that systems work together effectively. To facilitate such interoperability, the DoD has initiated a number of projects that focus on different aspects of the infrastructure for network-centric operations. Several of these projects are developing
SOAs so that C2 applications can be built using functionality from infrastructure services (e.g., communication, discovery, security) and services that are specific to the C2 domain (application domain services). Current and future DoD programs offices have been targeted to contribute application domain services (ADS).

The owners of the C2 system involved in the case study recognized that if a selected set of components from their system are converted to application domain services, they may have applicability for a broad variety of purposes. Our role was to perform a preliminary evaluation of the feasibility of converting a set of these components to application domain services within a specific target SOA. This evaluation assumed that the target SOA would provide basic infrastructure services, such as a security service and a communication service.

3.2 Describe Future Service-Based State.

The system owner had done a preliminary identification of potential services that could be built from components of the legacy system. This analysis was derived from high-level requirements for applications that would be users of services to be provided by the SOA. The system owner had matched legacy functionality to these high-level requirements and provided initial estimates of the services that could be provided to users.

We investigated the target SOA through an analysis of available documentation and through a meeting with the developers. The target SOA is currently under development. It is being built using a variety of commercial products and standards, along with significant custom code. The effort is focused on satisfying a number of specific quality attributes important to the DoD, such as performance, security, and availability. In order to meet these needs, the SOA will impose a number of constraints on potential services. Because the SOA is still under development, the specifications for how to write and deploy services are still unclear.

The target SOA is illustrated in Figure 1. Figure 1 shows that the SOA includes common services (CS) that are to be used by user applications and ADSs. The environment allows for a set of ADSs that derive their requirements from user applications. Groups within DoD are invited to submit proposals for ADSs to meet these requirements, either by building them from scratch or by deriving them from legacy components. An organization considering the migration of legacy systems to provide ADSs must analyze the requirements in detail and match existing functionality to determine what can be used as-is, what has to be modified, and what has to be new development.

Figure 1. Physical view of the target SOA

Even though the full details of compliant services for the DoD SOA have not yet been worked out, the SOA imposes a number of constraints on organizations that are developing ADSs from existing legacy components. These constraints are not common in traditional SOAs, which in general only require conformance to established interfaces and communication mechanisms, and do not place constraints on the environment in which the service is deployed. Some of the constraints/requirements for developers of ADSs that make the migration challenging include:

1. An ADS needs to be self-contained, that is, it should be able to be deployed as a single unit. In this specific target SOA, services must be stand-alone and of small granularity so that they can be deployed as needed on standardized and often limited-resource platforms such as handheld devices. Legacy functionality that has been targeted to provide a service needs to be fully extracted from the system. This includes extracting code that corresponds to shared libraries or that forms the core of a product line.

2. In the target SOA, an ADS must be deployable on a Linux operating system. For Windows-based legacy components this could be a problem, especially if there are dependencies on the operating system through direct system calls or if there is a dependency on commercial products that are only available for Windows systems. Ideally, system calls should be eliminated. If it is not possible, they should be evaluated to see if there are equivalents in the Linux operating system or if this functionality is part of one of the common services.

3. All services in the target SOA will share a common data model and all data will be accessed through a

---

**Figure 1. Physical view of the target SOA**

- **Target Environment**
- **User Application**
- **User Application**
- **User Application**

**ADS 1**... **ADS n**

**Network**

**CS 1**... **CS n**
Data Store common service. The need for a common data model is driven by a desire for information to be shared and understood by all user applications. As a result, services will no longer define internal data. All dependencies on databases and filesystems within the legacy code must be replaced by calls to the data store service. In addition, all data needed by the code must become part of the common data model.

4. An ADS will use the Discovery common service to find and connect to other services. If the ADS will rely on other services, code to discover and connect to these services will have to be written. Once the service is developed it needs to be advertised. This is done by registering the service with the naming service. Once this advertised service has been registered, other applications that wish to use this service will perform a discovery on the available services and choose which service(s) they desire to use.

5. An ADS will use the Communications common service for communicating with other services. The target SOA provides tools for generating data readers and data writers that will take incoming and outgoing data and format it accordingly.

3.3 Describe Existing Capabilities.

The pilot C2 system has two parts: 1) a mission planning system and 2) a mission execution system that adds situational awareness to the planning capability. These two systems were initially developed as part of a product line. Both rely on a set of core components for the data model, data analysis, and visualization.

Logical and development views of the system architecture were not available. This represented a problem for our analysis, and is discussed in more detail in Section 3.4. A physical view of the current system is illustrated in Figure 2.

As shown in Figure 2, in the current environment there is a single instance of the C2 application per machine and an instance of the collaboration server deployed on another machine. Instances of the C2 application interact with the collaboration server to send and receive data updates.

The entire system represented in Figure 2 is under the control of the organization responsible for the legacy system. We used this observation at several points during the study to help the user understand the implications of making the transition from the current environment to an SOA environment.

Given the information about the target SOA, we met with the contractor and representatives of the government to focus on a limited number of legacy components and to select criteria for further screening. We focused on seven potential services that the government team had previously identified as part of its initial analysis of ADS requirements. These seven potential services contained 29 classes.

The current system, written in C++ on a Windows operating system, had a total of about 800,000 lines of code and 2500 classes. In addition, the system was dependent on a commercial database and a second product for visualizing, creating, and managing maps. Both commercial products are available only for the Windows operating system.

In conjunction with the team, we developed characteristics for screening the potential reusable components. These characteristics reflected both the base characteristics provided by the Options Analysis for Reengineering (OAR) [2] technique and our knowledge of the necessary characteristics of services operating within the target SOA. The characteristics included:
• Size
• Complexity
• Level of documentation
• Coupling
• Cohesion
• Number of base classes
• Programming standards compliance
• Black box vs. white box suitability
• Scale of changes required
• Commercial mapping software dependency
• Microsoft dependency
• Support software required

These characteristics formed the basis for the more detailed analysis discussed in Section 3.4.1.

3.4 Analyze Gap.

Given the known and projected constraints of the target SOA, we performed a preliminary analysis of the legacy components to determine their suitability for reuse as services, and the amount of effort and risk that would be involved. This analysis consisted of three separate activities: 1) a gap analysis of the changes to the legacy components that would be necessary for migration to the SOA, 2) an informal evaluation of code quality, 3) an architecture reconstruction to obtain a better understanding of the set of undocumented dependencies. The results of these analyses allowed us to define a service migration strategy based on the risks due to the unknown future state of the target SOA. These analyses are each described in the following subsections.

3.4.1 Analysis of Required Changes. During the gap analysis we analyzed the candidate legacy components in terms of the characteristics that were developed in section 3.3. The characteristics provided input on level of difficulty and risk factors.

We identified dependencies of the selected classes on the commercial mapping software, the commercial database, and Windows. Unfortunately the high level documentation was inadequate. Most of the useful documentation was in the form of code comments and information generated from the DOxygen tool\(^1\). DOxygen can extract after-the-fact data from the C++ code, such as classes, attributes, dependencies, and comments. However, during the analysis we found that the DOxygen tool only picked up first-level dependencies (e.g., those classes that are directly called from the class being analyzed). Any classes called from within the classes considered first-level dependencies are not included. This limitation in identifying lower level dependencies led us to believe that the coupling and the amount of code that was used by each class was higher than could be estimated from the existing documentation. This suspicion was confirmed by legacy programmers.

We found that no consistent programming standard had been enforced during development of the system, leading to idiosyncrasies between the code and documentation produced by different programmers. This increased the difficulty of our analysis, and it would also increase the difficulty of any reuse.

As might be expected from a relatively recent object-oriented system, we found the overall cohesion to be high. The contractor provided estimates for converting the components into services, based on a set of simplifying assumptions on the actual make-up of the target SOA and the final set of user requirements.

A summary of the initial analyses of converting the selected components (classes) to services is illustrated in Figure 3. Base classes are those from which the classes identified as part of the services are inheriting properties in the object-oriented context. “Coupled” classes are those classes that contain code that is used by the classes identified as part of the service. It is important to account for these as they represent code that needs to be migrated. The size data was derived from the DOxygen tool. Programmers who were most familiar with the legacy application, and therefore presumably most familiar with the work required, were asked to provide estimates of effort, level of difficulty and level of risk required for the migration. This subjective data represented only an initial step. The code analysis and architecture reconstruction provided an important “sanity check” on these initial estimates.

<table>
<thead>
<tr>
<th>Services</th>
<th>Number of Services</th>
<th>Number of Classes</th>
<th>Size (LOC)</th>
<th>Effort (MM)</th>
<th>Level of Difficulty</th>
<th>Level of Risk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selected Services</td>
<td>7</td>
<td>16</td>
<td>16,163</td>
<td>12.5</td>
<td>Low to Medium</td>
<td>Low</td>
</tr>
<tr>
<td>Base Classes</td>
<td>4</td>
<td>2,199</td>
<td>4</td>
<td>Low to Medium</td>
<td>Low</td>
<td></td>
</tr>
<tr>
<td>&quot;Coupled&quot; Classes</td>
<td>3</td>
<td>5,388</td>
<td>1</td>
<td>Low</td>
<td>Low</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>23</td>
<td>23,750</td>
<td>17.5</td>
<td>Low</td>
<td>Low</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3. Results of initial analysis

If the migration to services was performed by the existing contractor, the level of difficulty of making these changes would be low to medium, and the risk would be low because of their familiarity with the systems. However, due to the inadequacies in the

---

\(^1\) DOxygen is a documentation system for C, C++ and IDL. It can generate an on-line class browser and/or an off-line reference manual from a set of documented source files (http://www.doxygen.org).
architecture documentation, and the likely underestimation of the amount of code used by the potential services, a number of gaps remained in our understanding of the system. For example, it was mentioned that one of the services made extensive use of the data model. This data model had over 1000 classes and was used by every class included in the potential services. Even though our analysis did not initially focus on the data model, because of its size it now represented the largest potential source of reuse in our study. However, as we will point out in section 3.4.3, the constraints imposed by the target SOA may hinder reuse of the data model.

As a result, it was not possible to accurately know how many other classes are used by a specific service. In addition, we expected that the estimates for rehabilitation of the legacy components were understated. For example, we did not believe that the programmers who provided estimates fully anticipated the effort necessary to locate and remove all calls to the user interface.

To get a better understanding of these issues we performed a code analysis and architecture reconstruction, as detailed in Sections 3.4.2 and 3.4.3.

3.4.2 Evaluation of Code Quality. To address remaining issues, we first analyzed the code through a code analyzer “Understand for C++”. This analysis provided:

- A data dictionary
- Metrics at the project, file, class, and function level
- An invocation tree
- Cross references for include files, functions, classes/types, macros and objects
- A list of unused functions and objects

The code analysis enabled us to validate information from the contractor and to produce input for an architecture reconstruction tool that could be used to identify dependencies.

From the code analysis, we found that the code was better organized and documented at the code level than most code that we have seen. However, as mentioned earlier, there were inconsistencies in the quality and documentation of the code that made the analysis complicated and often required manual work to make sure important information was being extracted:

1. There was not a consistent coding standard. Each individual programmer had a different style.
2. Some parts of the code were much more difficult to navigate, with less cohesion and a more awkward file organization. Naming standards were different for files, classes, attributes, and method names. Code organization styles were different.

3. The organization of files was not standardized. For example, some files that do not perform user interface (UI) functions are located in UI folders. Also, some include files are located with code files, while others are located in a separate folder. Finally, some files contained more than one class, and there are no clear criteria for when this is allowed.

Despite these difficulties, we were able to produce the input for the architecture reconstruction tool.

3.4.3 Architecture Reconstruction. To address the issue of dependencies in more detail, we conducted an architecture reconstruction with a tool called ARMIN [5]. Architecture reconstruction is the process by which the architecture of an implemented system is obtained from the existing system [8].

To begin the architecture reconstruction, we took the output from the code analysis and used ARMIN to aggregate the code into several groups:

- One for each service analyzed
- One for code directly dependent on the commercial mapping software
- One for user interface code
- One for the rest of the code—data model, base classes, utilities, and code that did not belong to any of the above groups

This grouping of code allowed us to identify dependencies between:

- individual services
- services and user interface classes
- services and the commercial mapping software
- services and the rest of the code that mainly represented the data model

We were able to identify a substantial number of undocumented dependencies between classes. Knowledge of these dependencies provided a more realistic understanding of the scope of the migration effort.

There are also dependencies between the primary services that we analyzed and a second forthcoming project that was being planned by the government organization. We cautioned that there will be duplication of work if these are treated as separate projects.

The dependencies with the mapping software and other commercial products are a potential concern in the target environment. The Windows-based mapping software, for example, would need to be verified for use
within the target environment. There may be a different mapping service that would be required by the target environment. There are also dependencies on a commercial database. These will have to be replaced by target-approved data access methods.

The architecture reconstruction also enabled us to document the central role of the data model, and to identify it as a potentially valuable reusable component, even though it had not been identified during the initial analysis. However, this finding was tempered by the fact that in the target SOA environment, potentially all services will have to use a common data model. If this is the case, all elements of the data model will have to be mapped to existing elements of the common data model. Negotiations will have to take place to make sure that data elements that are needed by the services become part of the common data model.

The architecture reconstruction allowed us to confirm that the system is an example of the application of the Model View Controller (MVC) pattern, as indicated by the contractor. The reconstruction also pointed out violations of the MVC architectural pattern that would need to be addressed in any migration effort—specifically calls from the model to the view.

An illustration of how the change from a standard system development effort to an SOA can have unanticipated impacts can be seen from the product line approach that was taken in the initial development of the system. The product line approach was an excellent choice for the legacy application; however, the resulting product line architecture will complicate efforts to isolate services since it led to large dependencies on base code and multiple levels of inheritance. A potential solution to this problem would be to consider each service in itself as part of a product line, but this would require reengineering of the core components.

3.5 Develop Strategy for Service Migration.

In general, we found that the legacy code provides a set of components with significant reuse potential. However, the largest risk in reusing the legacy components concerns the fact that the target SOA has not been fully developed. While its overall structure has been defined, many of the specific mechanisms for interacting with it are still pending. Thus, it is not yet clear what the ultimate requirements will be for a service. Based on these general observations, we recommended deferring the migration effort, and in place focus on the following activities:

1. Require the contractor to update the software architecture documentation and standardize comments in the code.
2. Work with the developers of the target SOA to define what is meant by a compliant service.
3. Work with the team defining the SOA data model to understand its contents and influence it as necessary.
4. Find out if there the vendor has plans for a Linux version of the mapping software or if the target SOA group has plans for a mapping common service to replace the current Windows mapping software.
5. Interact with potential application developers that will be using the services to understand their requirements and develop appropriate service interfaces.
6. Recalculate the cost and effort of migration based on the complete set of code dependencies uncovered during the analysis and on improvements in the organization’s understanding of user requirements and SOA constraints.
7. Investigate the commonality between the current service migration effort and the forthcoming migration project to a different target SOA.

Thus, the recommendations of our analysis can be perceived as “negative” because we recommended that the organization not proceed with its plans. The organization accepted this advice, and while it is difficult to quantify the results of a decision to not proceed, we believe that the deferral will ultimately save the organization several million dollars. This savings will be the result of avoiding an ill-informed and premature migration of the legacy components to services.

4. The Service-Oriented Migration and Reuse Technique (SMART).

Based on the experience illustrated in the case study, we developed an approach for gathering the necessary information and identifying the risks for the migration effort in a more systematic way. This method is the Service-Oriented Migration and Reuse Technique (SMART) [7]. SMART was developed by integrating OAR, architecture reconstruction practices with the ARMINT tool, and a detailed analysis of the requirements of the target SOA.

SMART gathers a wide range of information about legacy components, the target SOA, and potential services to produce a service migration strategy as its primary product. However, SMART also produces products that are useful to the organization whether or not it decides to proceed with migration. These inputs and outputs are depicted in Figure 4.
The activities in SMART were derived from the high level tasks of the case study, as shown in Figure 5:

1. **Establish stakeholder context**: Gather information about goals of the migration effort, expectations, potential service users, legacy system end users and owners, contractors, and legacy components and potential services.

2. **Describe existing capabilities**: Gather information about legacy system characteristics, legacy system architecture, and code characteristics.

3. **Describe the future service-based state**: Gather information about potential services that can be created from the legacy components, and gather sufficient detail about the target SOA to support decisions about what services may be appropriate and how they will interact with the architecture. At this point it is normally possible to either reject legacy components from further consideration, or to identify them – or some parts of them – as likely candidates for service migration.

4. **Analyze the gap between service-based state and existing capabilities**: Identify the gap between the existing state and the future state and determine the level of effort and cost needed to convert the legacy components into services.

5. **Develop strategy for service migration**: Recommend one or more of the identified migration options, select a strategy to achieve the goal, and present the SMART team findings.

Information for the first three activities is gathered using the Service Migration Interview Guide (SMIG), as described in [7]. The SMIG distills the many desired traits of services executing within SOAs into a set of topics and related questions that guide interview sessions. The goal of the SMIG is to assure broad coverage and consistent analysis of difficulty, risk, and cost issues.

Information for the final two activities is gathered in subsequent meetings, and, when possible, in hands-on analysis of the legacy code to answer specific questions and verify assertions.

**5. Conclusions and Next Steps.**

We found that the task of determining how to expose functionality as services, while seemingly straightforward, can have substantial complexity. Our conclusions to the client helped them to avoid a set of mistakes they may have otherwise made. The type of disciplined analysis that we performed appears to have applicability for other organizations that are considering migrations to SOAs.

An early version of the SMART process served to direct and provide discipline to our analysis. SMART has general applicability to corporations and
government organizations that are considering the migration of legacy components to services within an SOA. We are currently updating SMART with the following goals:

- Improve the breadth and consistency of information gathered about the engineering effort necessary to change the legacy artifact into a service. The Service Migration Interview Guide (SMIG) is the first tool intended for this purpose. By incorporating significant technical “know how” into the SMIG, we also further an ultimate goal of transitioning the technique to other users.

- Incorporate decision rules on when it is most useful to include the code analysis and architecture reconstruction steps as part of the process.

- Develop machine support for capturing and analyzing data gathered during the SMART process. This will entail building templates for major artifacts.

- Develop techniques and criteria for determining when a SMART team has captured sufficient information to complete the analysis process.

- Establish a mechanism to capture the net effect of SMART on migration efforts. This information is essential for continued evolution and improvement of SMART.

References.


