The SOA Frontier. Experiences with 3 Migration Approaches

Juan M. Rodriguez Marco Crasso Cristian Mateos
Alejandro Zunino Marcelo Campo Gonzalo Salvatierra

August 3, 2013

Abstract

Service Oriented Architecture (SOA) and Web Services are the current trend to integrate large and distributed systems, which is a common situation in both the business and government worlds. However, within these worlds, systems are commonly written in COBOL because they were developed several decades ago. Therefore, migration of COBOL systems into service-oriented architectures becomes a necessity. Two main approaches are used to migrate COBOL systems to SOA systems: direct and indirect migration. Direct migration implies wrapping the current COBOL routines of a system with a software layer developed under a newer platform that can be used to offer Web Services. In contrast, indirect migration requires re-designing and re-implementing the COBOL routines functionality using a newer platform as well. In this chapter, we propose a novel migration approach, which takes the best of the two previous approaches. To assess the advantages and disadvantages of these approaches, this chapter presents a case study from a government agency COBOL system that has been migrated to a Web Services-based system using the three approaches. As a result of having these migration attempts, we present the trade-off between direct and indirect migration the resulting service interfaces quality and the migration costs. These results also show that this new migration approach offers a good balance to the above trade-off, which makes the approach applicable to similar COBOL migration scenarios.

Keywords: SERVICES-ORIENTED ARCHITECTURE, WEB SERVICES, LEGACY SYSTEM MIGRATION, DIRECT MIGRATION, INDIRECT MIGRATION, SERVICE FRONTIER, SERVICE INTERFACE DESIGN, CODE-FIRST, CONTRACT-FIRST, SEMI-AUTOMATIC COBOL MIGRATION

1 Introduction

Information systems were adopted by enterprises several decades ago, and they have been used ever since. As a result, operationally, most enterprises rely on old, out-of-date systems. These systems are known as legacy systems. Well-known examples of this kind of systems are COBOL systems because, according to Gartner consulting[1],

[1]`http://www.gartner.com`

there are over 200 billion lines of operative COBOL still running worldwide. Furthermore, since enterprises' goals and context vary, in time these systems have suffered modifications to be kept suitable for the enterprises. For example, nowadays it is impossible to conceive a bank that does not offer Home Banking services, yet most bank systems were originally written in COBOL. Therefore, it is common to find a 50 year old technology, such as COBOL, working alongside with the most modern technologies [15], like .Net, AJAX, or JEE, in the same enterprise. As a result of this situation, enterprises have to face high costs for maintaining their systems. This is mainly because these systems usually run on mainframes that must be rented. In addition, there is the necessity of hiring developers specialized in old technologies for updating the system functionality, which is both expensive and rather difficult.

Taking these facts into consideration, many enterprises opt for modernizing their legacy systems using a newer technology. This process is called migration. Currently, a common target paradigm for migrating legacy systems is SOA (Service-Oriented Architecture) [5, 13]. In SOA, systems are built using independent functionalities called *services* that can be invoked remotely. Services are offered as platform-agnostic functionalities that can be used by any system inside or outside their owner organization. To ensure platform independence, Web Services technologies are the commonest way of implementing SOA systems since the former relies on well-known Internet protocols, such as SOAP and HTTP [9]. Therefore, migrating legacy systems to Web Services technologies is a fairly common practice. However, there is no standard recipe to effectively migrate legacy systems to SOA.

Recent literature [17] proposes that a legacy system can be migrated by following two approaches. The first approach, called *direct migration*, consists in wrapping a legacy system with a software layer that exposes the original system programs as Web Services. This approach is known to be cheap and fast, but it has the disadvantage that the legacy system is not replaced by a new one. Instead, the enterprise obtains two systems to maintain: a legacy system, and its newly-built SOA layer. On the other hand, the approach called *indirect migration* is based on re-implementing the legacy system using a modern platform. This approach is expensive and time consuming because not only the system should be reimplemented and re-tested, but also in some cases the business logic embodied in the legacy system should be reverse-engineered. This happens because system documentation could have been lost or not kept up-to-date. The result of an indirect migration is not only an improved version of the system, but also updated documentation for future reference.

From the SOA point of view, an important difference between direct migration and indirect migration is the quality of the SOA frontier that is the set of Web Services exposed to potential consumers as a result of migrating a legacy system. The SOA frontier quality is a very important factor for the success of a SOA system [6, 4, 22] because, as Figure 1 depicts, SOA frontier is used for both service registries and consumer. Service registries use the SOA frontier for indexing services and, then, allowing service consumers to search for them. In contrast, service consumers need the SOA frontier to understand and invoke the services. And a SOA system success can be measure by how many service consumers it has. This is however ignored by most enterprises as direct migration is by nature the least expensive and fastest way of deriving a SOA frontier from a legacy system, but such a SOA frontier commonly is a mere "Internet-ready"

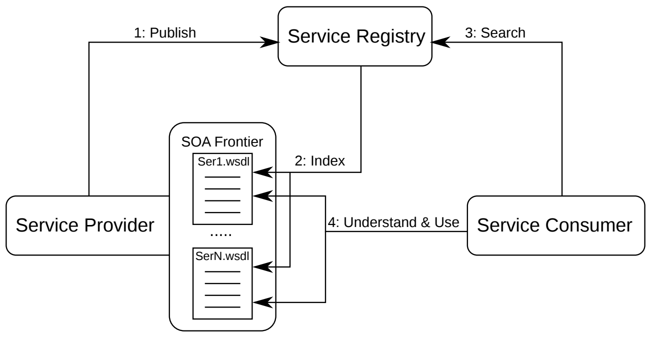

Figure 1: SOA frontier importance.

representation of original system program interfaces, which were not designed with SOA best design practices or even conceived for being used by third-parties as Web Services essentially are. Instead, a SOA frontier design is commonly benefited by the re-implementation of original system programs during indirect migration.

To obtain a better trade-off between cost and SOA frontier quality, we have developed a new migration approach with similar quality than the one obtained using indirect migration, but based on direct migration. Our approach, called *assisted migration*, takes a legacy system migrated using direct migration and performs an analysis of possible refactoring actions to increase the SOA frontier quality. Although this approach does not remove the legacy system, we think that the obtained SOA frontier can be used as a starting point for re-implementing it. Therefore, such a SOA frontier smoothes system replacement because it does not change when the legacy system is replaced by the new one.

For evaluating the viability of assisted migration, we used a case study involving a COBOL system in which direct migration and indirect migration were applied. This system is owned by the biggest Argentinean government agency, and manages data related to almost the entire Argentinean population. The system was firstly migrated using the direct migration approach because of strict deadlines. Since the SOA frontier quality of the first migration was not suitable for new developments, the agency decided to perform a second migration. In this case, the indirect migration approach was used to improve the SOA frontier quality. Having these two migration attempts, we applied the assisted migration approach by feeding it with the direct migration version of the original system. Then, we compared the three obtained SOA frontiers in terms of cost, time, and quality. According to this analysis, our approach produces a SOA frontier nearly as good as that the indirect migration, but at a cost similar to that of direct migration.

The rest of this work is organized as follows. Section 2 discusses related works on migrating legacy system to SOA. This section also presents the case study. Section 3 outlines the traditional migration approaches, namely direct and indirect migration. This section also explains why both methods have been applied to the same system, and discusses the cost difference between the direct migration and indirect migration attempts. Section 3.4 describes the proposed approach to reduce SOA frontier definition costs. Then, Section 3.5 discusses the effects of using each approach in terms of SOA frontier quality. Finally, Section 5 concludes this chapter, and outlines future research opportunities.

2 Background

Migration of mainframe legacy systems to newer platforms has been receiving lots of attention as organizations have to shift to distributed and Web-enabled software solutions. Different approaches have been explored, ranging from wrapping existing systems with Web-enabled software layers, to 1-to-1 automatic conversion approaches for converting programs in COBOL to 4GL. Therefore, current literature presents many related experience reports.

Because migration has been a problem since the first enterprise systems started to become obsolete, there is plenty of work on this topic. For instance, researchers have developed several methodologies for migrating system from their original technology to newer technologies, such as Cold Turkey, Chicken Little [7] and Renaissance [3]. Yet, these methodologies do not explicitly address how to migrate systems to SOA. To solve this issue, researchers have developed Service-Oriented Migration and Reuse Technique (SMART) for selecting migration strategies, but there is a lack of tools for assisting developers to apply it [16]. In addition, SMART does not take into account SOA frontier quality as a main concern in this kind of migration.

Another COBOL to SOA migration methodology is SOAMIG [28] that , in contrast with SMART, provides both a migration methodology and tools. The migration methodology consists of four phases and each of these phases might be carried out in several iterations. The main idea behind SOAMIG is to transform the original system into a SOA oriented one using several translation tools. For instance, it proposes to use a tool that translates COBOL code to Java code which is easier to expose as Web Service. However, this approach might negatively impact on the quality of the SOA frontier because COBOL code was designed using out-of-date design criteria, and this kind of tools do not redesign the system. In addition, COBOL impose some length limitations to routine names and comments that might be probably translated to the SOA frontier, which might also represent a quality issue for the SOA frontier.

In this context, migrating legacy systems to SOA, while achieving high-quality service interfaces instead of just "webizing" the systems is an incipient research topic. Harry Sneed has been simultaneously researching on automatically converting COBOL programs to Web Services [24] and measuring service interfaces quality [25]. In [24] the author presents a tool for identifying COBOL programs that may be *servified*. The tool bases on gathering code metrics from the source to determine program complexity, and then suggests whether programs should be wrapped or re-engineered. As such, programs complexity drives the selection of the migration strategy. As reported in [24], Sneed plans to inspect resulting service interfaces using a metric suite of his own, which comprises 25 quantity, 4 size, 5 complexity and 5 quality metrics.

In [1] the authors present a framework and guidelines for migrating a legacy system to SOA, which aims at defining a SOA frontier having only the "optimal" services and with an appropriate level of granularity. The framework consists of three stages. The first stage is for modeling the legacy system main components and their interactions using UML. At the second stage, service operations and services processes are identified. The third stage is for aggregating identified service elements, according to a predefined taxonomy of service types (e.g. CRUD Services, Infrastructure services, Utility services, and Business services). During the second and third stages, software analysts are assisted via clustering techniques, which automatically group together similar service operations and services of the same type.

This work presents a comparison of applying different migration approaches. Two of them together with their pros and cons are well-known within the community, while the other is a new approach that aims at obtaining the pros of both previous migration without their cons. The following section discusses the case study used for this analysis.

2.1 Case Study

The case under study in this work consists of a data-centric system composed of several subsystems for maintaining data records related to individuals including complete personal information, relationships, work background, received benefits, and so forth. The system is written in COBOL, runs on an IBM mainframe and accesses a DB2 database with around 0.8 PetaBytes. On the other hand, there are some COBOL programs accessing historic data through VSAM (Virtual Storage Access Method), a storage and data access method featured by a number of IBM mainframe operating systems. Moreover, some of the COBOL programs are only accessed through an intra-net via 3270 terminal applications, while other programs are grouped in CICS (Customer Information Control System) transactions and consumed by Web applications. CICS is a transaction manager designed for rapid, high-volume processing, which allows organizing a set of programs as an atomic task. In this case, these programs consist of business logic and database accesses, mostly input validations and queries, respectively.

Table 1: Characteristics of most important system transactions according to the mainframe load in terms of executed calls during January 2010.

Transaction	**SQL**	**COMMAREA(s)**		**Include(s)**	
# of lines	**# of queries**	**# of lines**	**# of files**	**# of lines**	**# of files**
265	2	518	7	683	6
416	2	1114	6	141	5
537	3	10	2	800	29
1088	10	820	4	580	16
543	10	820	4	411	10
705	10	956	6	411	10

For the sake of illustration, a brief overview of only 6 transactions is shown in

Table 1, which indicates the number of non-commented source code lines[2], the number of SQL queries performed, and the number of lines and files associated with a transaction. When a program P_1 calls a program P_2, or imports a Communication Area (COMMAREA) definition C, or includes an SQL definition S, it is said that P_1 is associated with P_2, or C, or S, respectively. On average, each transaction had 18 files, comprised 1803 lines of code, and performed 6 SQL SELECT statements.

3 Employed Migration Approaches

The system described in the previous section has been migrated to a SOA system by employing two different migration approaches. Clearly, this is an uncommon situation in practice, however each migration attempt has reasonable reasons behind them.

The first migration attempt was originated by the need for providing Web access to the legacy system functionality with critic time deadlines and no more resources than the agency's IT department resources present at the moment. In other words, the system needed to be rapidly migrated by permanent staff and using the software development tools which the agency had licenses for use them. One year after, the agency IT members outsourced the second migration attempt to a recognized group of researchers with remarkable antecedents in SOC, Web Services, and Web Services interface design. The reason to do this was that the project managers responsible for building the Web applications found the resulting SOA frontier from the first migration attempt extremely hard to be understood and consequently reused. The next subsections explain each migration attempt in detail. Finally, we have applied a third migration in the context of a research project in which the agency was not involved.

3.1 Direct migration

Methodologically, the IT department members followed a wrapping strategy [2] that comprised 4 steps for each migrated transaction:

1. Automatically creating a COM+ object including a method with the inputs/outputs defined in the associated COMMAREA, which forwards invocations to the underlying transaction. This was done by using a tool called COMTI Builder [14].

2. Automatically wrapping the COM+ object with a C# class having only one method that invokes this object by using Visual Studio.

3. Manually including specific annotations in the C# code to deploy it and use the framework-level services of the .NET platform for generating the WSDL document and handling SOAP requests.

4. Testing the communication between the final Web Service and its associated transaction. This was performed by means of a free tool called soapUI (`http://www.soapui.org`).

[2]SLOC metric for COBOL source code was calculated using the SLOCCount utility available at `http://www.dwheeler.com/sloccount`, which does not count commented lines.

To clarify these 4 steps a word about the employed technologies and tools is needed. A COMMAREA is a fixed region in the RAM of the mainframe that is used to pass data from an application to a transaction. Conceptually, a COMMAREA is similar to a C++ struct with (nested) fields specified by using native COBOL data-types. COMTI (Component Object Model Transaction Integrator) is a technology that allows a transaction to be wrapped with a COM+ (Component Object Model Plus) object. The tool named COMTI Builder receives a COMMAREA as input to automatically derive a type library (TLB), which is accessible from any component of the .NET framework as a COM+ object afterwards. COM+ is an extension to COM that adds a new set of functions to introspect components at run-time. Finally, the tool listed in 4) receives one or more WSDL documents and automatically generates a client for the associated service(s), which allows the generation of test suites.

Basically, each step, but step 4, adds a software layer to the original transactions. In this context, the Wrapper Design Pattern is central to the employed steps, since wrapping consists in implementing a software component interface by reusing existing components, which can be any of a batch program, an on-line transaction, a program, a module, or even just a simple block of code. Wrappers not only implement the interface that newly developed objects use to access the wrapped systems, but are also responsible for passing input/output parameters to the encapsulated components. Then, from the inner to the outer part of a final service, by following the described steps the associated transaction was first wrapped with a COMTI object, which in turn was wrapped by a COM+ object, which finally was wrapped by a C# class that in the end was offered as a Web Service. To do this, implementation classes were deployed as .NET ASP 2.0 Web Service Applications, which used the framework-level services provided by the .NET platform for generating the corresponding WSDL document and handling SOAP requests. As the reader can see, the SOA frontier was automatically derived from the C# code, which means that WSDL documents were not made by human developers but they were automatically generated by the .NET platform. This WSDL document construction method is known as *code-first* [19].

3.2 Indirect migration

Methodologically, the whole indirect migration attempt basically implied five steps:

1. Manually defining potential WSDL documents basing on the knowledge the agency had on the interface and functionality of the original transactions. For each service operation, a brief explanation using WSDL documentation elements was included.

2. Exhaustively revising the legacy source code.

3. Manually refining the WSDL documents defined during step 1 by basing on opportunities to abstract and reuse parameter data-type definitions, group functionally related transactions into one cohesive service, improve textual comments and remove duplicated transactions, which were detected at step 2. For data-type definitions, we followed best practices for naming type elements and constraining their ranges.

4. Supplying the WSDL documents defined at step 3 with implementations using .NET.

5. Testing the migrated services with the help of the agency IT department.

During the step 1, three specialists on Web Services technologies designed preliminary WSDL documents, based on the knowledge the agency had on the functionality of the transactions, to sketch the desired SOA frontier together. This step comprised daily meetings not only between the specialists and the project managers in charge of the original COBOL programs, but also between the specialists and the project managers responsible for developing client applications that would consume the resulting migrated Web Services. Unlike the code-first approach, in which service interfaces are derived from their implementations, the three specialists used *contract-first*, which encourages designers to first derive the technical contract of a service using WSDL, and then supply an implementation for it. Usually, this approach leads to WSDL documents that better reflect the business services of an organization, but it is not commonly used in industry since it requires WSDL specialists [20]. This step might be carried out by defining service interfaces in C# and then using code-first for generating WSDL documents, especially when analysts with little WSDL skills are available.

The step 2 involved revising the transactions code with the help of documents specifying functionality and diagrams illustrating the dependencies between the various transactions to obtain an overview of them. This was done to output a high-level analysis of the involved business logic, since the existing COBOL to some extent conditioned the functionality that could be offered by the resulting services. The target transactions comprised 261688 lines of CICS/COBOL code (600 files). Six software analysts exhaustively revised each transaction and its associated files under the supervision of the specialists during three months. This allowed the external work team to obtain a big picture of the existing transactions.

The step 3 consisted in refining the WSDL documents obtained in the step 5 by basing on the output of the step 2. Broadly, the three specialists abstracted and reused parameter data-type definitions, grouped functionally related transactions into one cohesive service, improved textual comments and names, and removed duplicated transactions. For data-type definitions, the specialists followed best practices for naming type elements and constraining their ranges. From this thorough analysis, potential interfaces were derived for the target services and a preliminary XSD schema document subsuming the entities implicitly conveyed in the original COMMAREA definitions. Conceptually, this represented a meet-in-the-middle approach to service migration that allowed the specialists to iteratively build the final service interfaces based on the desired business services, which impact on the implementation of services, as well as the interfaces derived from the existing CICS/COBOL code, which to some extent condition the functionality that can be exposed by the resulting software services.

The step 4 was re-implementing the services and began once the WSDL documents were defined. Two more people were incorporated in the project for implementing the services using the .NET ASP 2.0 Web Service Application template as required by the agency. Hence, the 3 specialists trained 8 software developers in Visual

Table 2: Required manpower over months of the indirect migration attempt.

Step	People	Role	Time (in months)
1	3	WSDL specialists	1
2	6	Software analysts	3
3	3	WSDL specialists	1
4	8	Software developers	6
5	8	Software developers	2

Studio 2008, C# and a data mapper, called MyBatis[3]. This library frees developers from coding typical conversions between database-specific data-types and programming language-specific ones. MyBatis connects to DB2 mainframe databases using IBM's DB2Connect[4], an infrastructure for connecting Web, Windows, UNIX, Linux and mobile applications to z/OS and mainframe back-end data. It is worth noting that to bind the defined WSDL documents with their .NET implementations, the specialists had to extend the ASP 2.0 "httpModules" support. Concretely, the three specialists developed a custom module that returns a manually specified WSDL document to applications, instead of generating it from source code, which is the default behavior of .NET.

The step 5 was to test the resulting services with assistance of the agency IT department. Basically, each new Web Service was compared to its CICS/COBOL counterpart(s) to verify that with the same input the same output was obtained. If some inconsistency between the new services and the old CICS/COBOL system was detected, the services were revised and re-tested. This step was repeated until the agency IT department had the confidence that the new SOA-based system was as good as the old system from both a functional and non-functional point of view.

3.3 Costs comparison of the direct and indirect migration attempts

As reported by the agency's IT department, it took 1 day to train a developer on the direct migration method and the three tools employed, namely COMTI Builder, Visual Studio and soapUI[5]. Then, trained developers migrated one transaction per hour, mostly because all the steps but one (step 3) were tool-supported and automatic. Since the agency had the respective software licenses for COMTI Builder and Visual Studio tools beforehand, choosing them was a harmless decision from an economical viewpoint.

Regarding the costs of the indirect migration attempt, monetarily, it cost the agency 320000 US dollars. Table 2 details the human resources involved in the second attempt of the project. All in all, it took one year plus one month for 6 software analysts, 2 more

[3]MyBatis,`http://www.mybatis.org/dotnet.html`

[4]IBM's DB2Connect, `http://www-01.ibm.com/software/data/db2/db2connect/`

[5]Professional and paid versions of the two first tools were used, whereas a standard and free version of soapUI was employed.

Table 3: Costs comparison for the migration of 32 transactions during direct and indirect attempts.

Resources	First attempt: direct migration	Second attempt: indirect migration
Developers	1	8
Specialists	0	3
Time	5 days	13 months
Money	u$s 3,000 (*)	u$s 320000

(*) This was the average monthly salary in Argentine for a senior .NET Developer at the time of the bottom-up migration attempt.

developers incorporated at step 4, and 3 specialists to migrate 32 priority transactions. It is worth noting that no commercial tools were needed, apart from the IDE for which the agency already had licenses.

Table 3 presents an illustrative comparison of the resources needed by each migration attempt. The first migration attempt succeeded in delivering Web Services within a short period of time and without expending lots of resources, by employing a direct migration approach with wrapping. As shown in the second column of the Table, with the associated methods and tool-set, it only took 5 days and 1 senior developer to migrate 32 CICS/COBOL transactions. It is worth noting that the first attempt was inexpensive since no software licenses had to be bought, and no developers had to be hired, i.e. a regular member of the IT department performed the first migration attempt. However, this could be not the case for many enterprises and therefore there may be costs associated to buying the necessary tool-set and hiring external manpower when performing a direct migration with wrapping. In contrast, an indirect migration attempt with re-engineering was much more expensive and required more time to be completed. In particular, 8 junior developers (undergraduate UNICEN students), 3 Web Services specialists (UNICEN researchers with a PhD. and several publications in the Web Service area), 13 months and 320000 US dollars for re-engineering the same 32 transactions were required. For this attempt, external specialists and developers were hired, whose salaries have been included in this cost.

3.4 Assisted Migration: Software-Assisted SOA frontier definition

From the previous section it is clear that the indirect migration approach demanded much more resources than its direct counterpart. Indeed, as shown in Table 2, near a half of the time demanded by the indirect attempt was for defining service interfaces. Concretely, the output of steps 1, 2, and 3 of the indirect migration method employed, was the WSDL documents of the SOA frontier, and these steps took 5 months. Regarding needed people, the mentioned three steps required 3 WSDL experts and 6 software developers more than the direct migration attempt. Therefore, we have explored the hypothesis that, to some extent, some of the tasks performed for defining the SOA

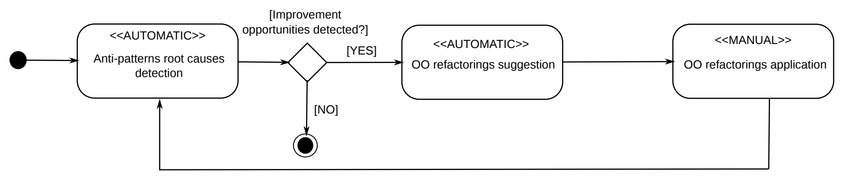
[Improvement opportunities detected?]
<<AUTOMATIC>>
Anti-patterns root causes detection
[YES]
[NO]
<<AUTOMATIC>>
OO refactorings suggestion
<<MANUAL>>
OO refactorings application

Figure 2: Software-assisted SOA frontier definition steps.

frontier could be automated, improving migration efficiency.

Basically, we propose a fast and cheap approach to imitate the tasks performed at some of the steps of the employed indirect migration approach. These tasks include exhaustively analyzing the legacy source code (step 2), and supplying software analysts with guidelines for manually refining the WSDL documents of the SOA frontier (step 3). The step 1, in which interviews were conducted, was left out of the scope of this approach. Thus, the input of this approach was the SOA frontier of the direct migration attempt, instead of those WSDL documents that were sketched together during the meetings of the indirect attempt.

The proposed approach can be iteratively executed for generating a refined SOA frontier at each iteration. The main idea is to iteratively improve the defined service interfaces, by removing those *WSDL anti-patterns* present in them. A WSDL anti-pattern is a recurrent practice that hinders Web Services chances of being discovered and understood by third-parties. In [23] the authors present a catalog of WSDL anti-patterns and describe each of them in a general way by including a description of the underlying problem, its solution, and an illustrative example. Then, the catalog of anti-patterns can be used to compare two WSDL-modeled SOA frontiers. Specifically, one could account anti-pattern occurrences within a given set of service interfaces because the fewer the occurrences are, the better the resulting WSDL documents are in terms of discoverability and understandability.

Figure 2 depicts the proposed steps. Mainly, these steps can be organized in two groups, automatic and manual. First, our approach starts by automatically detecting potential WSDL anti-patterns root causes within the SOA frontier given as input its WSDL documents plus their underlying implementation (“Anti-patterns root causes detection” step). Then, a course of actions to improve the SOA frontier is generated (“OO refactorings suggestion” step). The manual step “OO refactorings application” takes place when the software analysts in charge of migration apply the suggested refactoring actions. Accordingly, software analysts obtain a new SOA frontier and fed it to the anti-patterns root causes detection step. Notice that although this approach uses as input the COBOL code and the SOA frontier generated by the direct migration, the proposed refactorings are intended to be applied on the SOA frontier and the SOA-enabled wrapper, i.e. the COBOL wrapping software layer and the WSDL documents. Hence, the resulting SOA frontier can be deployed over the legacy system without disrupting its normal operation. The next subsections describe both groups of steps.

In the following two sections, we describe how the automatic steps of the assisted migration are performed. In Section 3.4.1, we present how the WSDL anti-patterns root causes are detected. In Section 3.4.2, we outline how OO refactorings are suggested.

Finally, in Section 3.4.3, we describe how much takes to perform these steps, which are automatic. How to apply OO refactorings is not described through this section because it is out of this chapter scope and there is plenty of literature on that topic [11].

3.4.1 Anti-patterns root causes detection

One of the lessons learned from past migration experiences is that manually revising legacy system source code is a cumbersome endeavor. However, such an exhaustive code revision is crucial not only because the legacy system implementation conditions the functionality that can be exposed by the resulting SOA frontier, but also to detect service interfaces improvement opportunities. Thus, the anti-pattern root causes detection step is performed automatically. To do this, by basing on the work published in [22], we have defined and implemented the ten heuristics summarized in Table 4. Broadly, a defined heuristic receives the implementation in CICS/COBOL of a migrated transaction or its associated WSDL document. Then, a heuristic outputs whether specific anti-patterns root causes are present in the given input or not. Actually, in most cases a heuristic output does not provide enough evidence of anti-patterns root causes existences by itself. Therefore, some heuristics have been combined as follows:

- 8 and 9 $\rightarrow$ Remove redundant operations
- 4 $\rightarrow$ Improve error handling definitions
- 3 or 5 or 6 or 7 $\rightarrow$ Improve business object definitions
- 8 $\rightarrow$ Expose shared programs as services
- 1 and 2 $\rightarrow$ Improve names and comments
- 10 $\rightarrow$ Improve service operations cohesion

, where several heuristic ids (see column Id from Table 4) are logically combined within rules antecedents and WSDL document improvement opportunities are rules consequents. It is worth noting that we will refer to WSDL document improvement opportunity, on analogy with removing particular WSDL anti-patterns root causes. For instance, the first rule is for detecting the opportunity to remove redundant operations, which may be the origin of at least two anti-patterns, namely *Redundant Port-types* and *Redundant Data Model* [23]. When processing COBOL code, this rule is fired when two or more programs share the same dependencies (heuristic 8) but also the parameters of one program subsume the parameters of the other program (heuristic 9).

Most heuristics have been adapted from [22], whereas heuristics 6, 7, 8, 9 and 10 were inspired by the migration attempt. Thus, heuristics 6 to 10 will be further explained next. With regard to *"Looking for data-types with inconsistent names and types"* (id 6), the heuristic analyzes names and data-types of service operations parameters to look for known relationships between names and types. Given a parameter, the heuristic splits the parameter name by basing on classic programmers' naming conventions, such as Camel Casing and Hungarian notation. Each name token is compared to a list of keywords with which a data-type is commonly associated. For example,

Table 4: Heuristics for detecting WSDL anti-patterns root causes.

Id	Description	Input	Output
1	Look for comments in WSDL <documentation> elements	WSDL document	**true**: if at least one operation lacks documentation **false**: otherwise
2	Search inappropriate names for services, port-types, operations and arguments	WSDL document	**true**: when the length of a name token is lower than 3 characters (e.g. "c_person", token "c" has 1 character), or when token refers to a technology (e.g. "PeopleDetailsSOAP"), or when an operation name contains two or more verbs or an argument name contains a verb **false**: otherwise
3	Detect operations that receive or return too many parameters	WSDL document	**true**: when at least one operation input/output has more than P parameters **false**: otherwise
4	Look for error information being exchanged as output data	WSDL document	**true**: when an output message part has any of the tokens: "error", ”errors", ”fault", ”faults", ”fail", ”fails", ”exception, ”exceptions,”overflow", ”mistake", ”misplay" **false**:otherwise
5	Look for redundant data-type definitions	WSDL document	**true**: when at least two XSD data-types are syntactically identical **false**: otherwise
6	Look for data-types with inconsistent names and types	WSDL document	**true**: when the name of a parameter denotes a quantity but it is not associated with a numerical data-type (e.g. numberOfChildren:String) **false**: otherwise
7	Detect not used parameters	COBOL source code	**true**: when at least one parameter is not associated with a COBOL MOVE statement **false**: otherwise
8	Look for shared dependencies among two service implementations	COBOL source code	A list of COBOL programs that are copied, or included, or called from two or more service implementations. The list is empty when no shared dependencies are found **true**: when the list is not empty **false**: otherwise
9	Look for data-types that subsumes other data-types	WSDL document	**true**: when an XSD complex data-type contains another complex XSD data-type, or a list of parameters subsumes another list of parameters **false**: otherwise
10	Detect semantically similar services and operations	WSDL document	**true**: when a vectorial representation of the names and associated documentation of two services or operations, are near in a vector space model **false**: otherwise

the token “birthday” is commonly associated with the XSD built-in xsd:date data-type, but the token “number” with xsd:int data-type. Therefore, the heuristic in turn checks whether at least one name token is wrongly associated with the parameter data-type.

The heuristic to *“Detect not used parameters”* (id 7) receives the COBOL source code of a migrated program and checks whether every parameter of the program output COMMAREA is associated with the programming language assignation statement, i.e. the COBOL MOVE reserved word. In other words, given a COBOL program, the heuristic retrieves its output COMMAREA, then gets every parameter from within it, even parameters grouped by COBOL records, and finally looks for MOVE statements having the declared parameter. One limitation of this heuristic is that the search for MOVE statements is only performed in the main COBOL program, whereas copied or included programs are left aside, and those parameters that are assigned by the execution of an SQL statement are ignored by this heuristic.

The heuristic to *“Look for shared dependencies among two service implementations”* (id 8) receives two COBOL programs as input. For each program builds a list of external COBOL programs, copies, and includes, which are called from the main program, and finally checks whether the intersection of both lists is empty or not. In order to determine external programs calls, the heuristic looks for the CALL reserved word.

With regard to “*Look for data-types that subsumes other data-types*” heuristic (id 9), this receives a WSDL document as input and detects the inclusion of one or more parameters of a service operation in the operations of another service. To do this, parameter names and data-types are compared. For comparing names classic text preprocessing techniques are applied, namely split combined words, remove stop-words, and reduce them to stems. For comparing data-types the heuristic employs the algorithm named *Redundant Data Model,* which is presented in [22].

The tenth heuristic, namely *“Detect semantically similar services and operations”*, is based on measuring the similarity among textual information of a pair of services or operations. This heuristic exploits textual information present in WSDL names and documentation elements. Textual similarity is assessed by representing associated textual information as a collection of terms and in turn as a vector in a multi-dimensional space. For each term, there is a dimension in the space, and the respective vector component takes as value the term frequency. Finally, looking for similar operations reduces to looking for near vectors in the space by comparing the cosine of the angle among them [26].

3.4.2 Object-Oriented refactorings suggestion

Until now, this section focused on the first part of the proposed approach, which is intended to reproduce the task of exhaustively looking for SOA frontier improvement opportunities within a legacy source code. Once we have all the evidence gathered by the heuristics, the second part of the proposed approach consists of providing practical guidelines to remove the potential anti-patterns root causes detected. These guidelines consist of a sequence of steps that should be revised and potentially applied by the development team in charge of the migration attempt. It is worth noting that the proposed guidelines are not meant to be automatic, mostly due to the fact that there is not a

Table 5: Association between SOA frontier refactorings and Fowler et al.'s refactorings.

SOA Frontier Refactoring	**Object-Oriented Refactoring**
Remove redundant operations	1: Extract Method $\vee$ Extract Class
Improve error handling definition	1: Replace Error Code With Exception
Improve business objects definition	1: Convert Procedural Design to Object $\wedge$ Replace Conditional with Polymorphism 2: Inline Class 3: Extract Class $\wedge$ Extract Subclass $\wedge$ Extract Superclass $\wedge$ Collapse Hierarchy 4: Remove Control Flag $\wedge$ Remove Parameter 5: Replace Type Code with Class $\wedge$ Replace Type Code with Subclasses
Expose shared programs as services	1: Extract Method $\vee$ Extract Class
Improve names and comments	1: Rename Method $\vee$ Preserve Whole Object $\vee$ Introduce Parameter Object $\vee$ Replace Parameter with Explicit Methods
Improve service operations cohesion	1: Inline Class $\wedge$ Rename Method 2: Move Method $\vee$ Move Class

unique approach to build a SOA frontier and in turn improve or modify it, which makes the automation of these proposed guidelines non-deterministic.

The cornerstone of the proposed guidelines is that classic Object-Oriented (OO) refactorings can be employed to remove anti-patterns root causes from a SOA frontier. The rationale behind this is that services are described as OO interfaces exchanging messages, whereas operation data-types are described using XSD, which provides some operators for expressing encapsulation and inheritance. Then, we have organized a sub-set of Fowler et al.'s catalog of OO refactorings [11], in order to provide a sequence of refactorings that should be performed for removing each anti-pattern root cause. This work bases on Fowler et al.'s catalog since it is well-known by the software community and most IDEs provide automatic support for many of the associated refactorings.

The proposed guidelines associate a SOA frontier improvement opportunity with one or more logical combinations of traditional OO refactorings (see Table 5). The first column of the Table presents SOA frontier improvement opportunities, while the second column describes which OO refactorings from [11] should be applied. As shown in the second column, the OO refactorings are arranged in sequences of refactoring combinations. Combining two refactorings by "$\vee$" means that software developers may choose among them, i.e. they should apply only one refactoring from the set. Instead, using "$\wedge$" means that the corresponding refactorings should be applied in that strict order. Moreover, in the cases of "*Improve business objects definition*" and "*Improve service operations cohesion*", the associated refactorings comprise more than one step. This means that at each individual step developers should analyze and apply the associated refactorings combinations as explained.

Regarding how to apply OO refactorings, it depends on how the WSDL documents of the SOA frontier have been built. Broadly, as mentioned earlier, there are two main approaches to build WSDL documents, namely code-first and contract-first. Code-first refers to automatically extracting service interfaces from their underlying implementation. For instance, let us suppose a Java class named CalculatorService that has one method signature sum(int i0, int i1):int. Then, its code-first WSDL document will have a port-type named CalculatorService with one operation sum related to an input message for exchanging two integers and an output message that conveys another integer. Here, the CalculatorService class represents the outermost component of the service implementation. On the other hand, when following the contract-first approach, developers should first define service interfaces using WSDL and then supplying implementations for them in their preferred programming language.

Then, when the WSDL documents of a SOA frontier have been implemented under code-first, the proposed guidelines should be applied on the outermost components of the services implementation. Instead, when contract-first has been followed, the proposed OO refactorings should be applied on the WSDL documents. For instance, to remove redundant operations from a code-first Web Service, developers should apply the "Extract Method" or the "Extract Class" refactorings on the underlying class that implements the service. In case of a contract-first Web Service, by extracting an operation or a port-type from the WSDL document of the service, developers apply the "Extract Method" or the"Extract Class" refactorings, respectively. When contract-first is used, developers should also update service implementations for each modified WSDL document.

3.4.3 Empirically assessing time demanded for executing heuristics and performing OO refactorings

We hypothesized that automatizing some steps of the indirect migration attempt could reduce the time demanded for executing heuristics and performing OO refactorings. We have empirically assessed the time demanded by each heuristic for analyzing the legacy system under study. The experiments have been run on a notebook with a 2.8 GHz QuadCore Intel Core i7 720QM processor, 6 Gb DDR3 RAM, running Windows 7 on a 64 bits architecture. To mitigate noise introduced by underlying software layers and hardware elements, each heuristic has been executed 20 times and the demanded time was measured per execution. Then, the heuristic executions times have been averaged. Table 6 summarizes the average time required for each automatic operation. Briefly, the average execution time of an heuristic was 9585.78 milliseconds (ms), being 55815.15 ms (less than one minute) the biggest achieved response time, i.e. the "Detect semantically similar services and operations" was the most expensive heuristic in terms of response time.

Furthermore, we have assessed the time demanded for manually applying the OO refactorings proposed by the approach on the SOA frontier resulted from the direct migration attempt. To do this, one software analyst with full knowledge about the system under study was supplied with the list of OO refactorings produced by the approach. It took two full days to apply the proposed OO refactorings. It is worth

Table 6: Average operation time.

Operation	Time
Look for comments in WSDL <documentation> elements	56.55 ms
Search inappropriate names for services, port-types, operations and arguments	39460.25 ms
Detect operations that receive or return too many parameters	58.7 ms
Look for error information being exchanged as output data	57.75 ms
Look for redundant data-type definitions	60.9 ms
Look for data-types with inconsistent names and types	93.45 ms
Detect unused parameters	56.45 ms
Look for shared dependencies among two service implementations	82.95 ms
Look for data-types that subsumes other data-types	115.70 ms
Detect semantically similar services and operations	55815.15 ms

noting that OO refactorings have been applied at the interface level, i.e. underlying implementations have not been accommodated to interface changes. The reason to do that was that we only want to generate a new SOA frontier and then compare it with the ones generated by the previous two migration attempts. Therefore, modifying interfaces implementation, which would require a huge development and testing effort, will not contribute to verifying the aforementioned hypothesis. All in all, to have an approximation of what the total cost of migrating the system under study with this approach would be, let assume that the cost for supplying the refactored SOA frontier with implementations will be near to the cost of step 4 plus step 5 of the indirect approach. Returning to Table 2, the first three rows can be condensed in one row with one software analyst, who must known the system in order to promptly apply OO refactorings suggested, and 2 days.

To sum up, the migration approach described in this section aims at automatically reproducing some of the steps of the indirect migration approach, so that their inherent costs are mitigated and at the same time a SOA frontier with an acceptable quality is obtained. In this sense, the next section provides empirical evidence on the service frontier quality achieved by the three approaches to legacy software migration to SOA described so far, namely direct migration, indirect migration, and our software-assisted migration.

3.5 Service Interface Quality Comparison

After migrating a system, the resulting service interface quality might be affected by several factors related to the migration methodology, and the original system design. While direct migration interfaces heavily depend on the original system design, indirect migration interfaces might be independent of it because indirect migration means re-implementing the old system with new technologies *and* design criteria. Despite having better results in terms of SOA frontier, indirect migration is known to be costly, and time consuming. The assisted migration approach then tries to balance the trade-off between cost and interfaces quality, hence this section presents evidence that assisted migration produces much better service interfaces than direct migration at a fraction of the indirect migration cost.

The evaluation relies not only on a quantitative analysis of lines of code (LOC), lines of comments, and offered operations, but also on a well-established set of quality metrics for WSDL-based interfaces [23]. The advantage of the quantitative analysis is that they are accepted as providers of evidence about system quality in general. Moreover, the advantage over other set of metrics is that they are WSDL document oriented, thereby they are suitable for comparing SOA frontiers quality. These metrics are based on a catalog of common bad practices found in public WSDL documents. These bad practices, which are presented in the well-known anti-pattern form, jeopardize Web Service discoverability, understandability, and legibility. We have used anti-pattern occurrences as a quality indicator because the fewer the occurrences are, the better the WSDL documents are. In addition, we have analyzed business object definitions reuse by counting repeated data-type definitions across WSDL documents, and the use of XSD files to define shared data-types. Notice that other typical non-functional requirements [18], such as performance, reliability or scalability, have intentionally not been considered since we were interested in WSDL document quality after migration. It is worth noting that though the metrics were gathered from the case study presented in Section 2.1, the examples used through this section are general because of our confidentiality agreement with the agency.

The comparison methodology consisted of gathering the aforementioned metrics from the SOA frontiers that resulted from each migration attempt. In this sense, three data-sets of WSDL documents were obtained, namely:

- *Direct Migration*: The WSDL documents that resulted from the first attempt of the project. As such, the associated services were obtained by using direct migration and implementing wrappers to the transactions, and by using the default tool-set provided by Visual Studio 2008 that supports *code-first* generation of service interfaces from C# code.

- *Indirect Migration*: The WSDL documents obtained from the approach followed during the second attempt of the project, i.e. indirect migration and at the same time the contract-first WSDL generation method.

- *Software-Assisted SOA Frontier Definition (or Assisted Migration for short):* The WSDL documents obtained after automatically detecting improvement opportunities on the direct migration data-set, and in turn applying the associated suggested guidelines.

The next subsections present the quantitative metrics comparison results, the qualitative comparison and the data-model reuse analysis, respectively.

3.5.1 Quantitative analysis

Firstly, there was a significant difference in the number of WSDL document generated by each approach. As Table 7 shows, Direct Migration data-set comprised 32 WSDL documents, which means one WSDL document per migrated transaction. In contrast, Indirect Migration and Assisted Migration data-sets had respectively 7 WSDL documents + 1 XSD file, and 16 WSDL documents + 1 XSD file. The first advantage

Table 7: Classical metrics.

Approach	**WSDL documents**	**Offered operations**	**LOC per file**	**LOC per operation**	**Comments per file**	**% Comments**
Direct Migration	32 WSDL documents	39	157.25	129	0.00	0.00%
Indirect Migration	7 WSDL documents + 1 XSD file	45	495.5	88	30.25	6.10%
Assisted Migration	16 WSDL documents + 1 XSD file	41	235.35	97	15.41	6.54%

observed in the Indirect Migration and Assisted Migration data-sets over Direct Migration data-set was the XSD file used for sharing common data-type definitions across the WSDL documents. In addition, having less WSDL documents means that several operations were in the same WSDL document. This happened because the WSDL documents belonging to Indirect Migration and Assisted Migration data-sets were designed to define functional related operations in the same WSDL document, which is a well-known design principle [27].

Secondly, the number of offered operations was: 39, 45, and 41 for Direct Migration, Indirect Migration, and Assisted Migration data-sets. Although originally there were 32 transactions to migrate, Direct Migration resulted in 39 operations because one specific transaction was divided into 8 operations. This transaction used a large registry of possible search parameters plus a control couple to select which parameters upon a particular search represent the desired input, ignoring the rest of them. During the first migration attempt, this transaction was wrapped with 8 operations with more descriptive names, and each of them end up calling the same COBOL routine with a different control couple.

On the other hand, the second and third attempts further divided the CICS/COBOL transactions into more operations. There were two main reasons for this, namely disaggregating functionality and making public common functionality. Disaggregating functionality means that some transactions, which returned almost 100 output parameters, had various purposes, thus they were mapped to several purpose-specific service operations. The second reason was that several transactions internally call the same COBOL routines, which might be useful for potential service consumers. In consequence, what used to be COBOL internal routines now are also part of the SOA frontier offered by the agency.

The resulting average LOC per file, and per operation were also a difference among the data-sets. Although Indirect Migration data-set had more LOC per file than the other two data-sets, it also resulted in less files. However, the number of LOC per operation of the Indirect Migration data-set was the lowest. Interestingly, the Assisted Migration presented a slightly higher number of LOC per operation than the Indirect Migration data-set. In contrast, the number of LOC resulted from applying the first migration attempt was more than twice as much as the LOC generated by other approaches. This means that a service consumer must read more code to understand what an operation does and how to call it. Basically, this makes using the WSDL documents of the Direct Migration data-set harder than using the WSDL documents from both

Indirect Migration and Assisted Migration data-sets.

Table 7 also points out the difference in the number of comment lines of WSDL and XSD code per document. Firstly, WSDL documents belonging to the Direct Migration data-set had no comments because the tools are unable to correctly pass COBOL comments on to COM+ wrappers, and then to WSDL documents. Besides, developers that used these tools did not bother about placing comments manually, which is consistent with the findings reported by previous studies [10, 22]. In contrast, Indirect Migration and Assisted Migration WSDL documents had 30.25 and 16 lines of comments per file, respectively. Despite having more comment lines per file, the percentage of comment lines in Indirect Migration WSDL documents were slightly lower that the percentage of comment lines in Assisted Migration WSDL documents.

3.5.2 Anti-Pattern assessment

Web Service discoverability anti-patterns were inferred from real-life WSDL document data-sets [21]. These anti-patterns encompass bad practices that affect the ability of a service consumer to understand what a service does, and how to use it. Therefore, these anti-patterns' occurrences can be used to evaluate how good a SOA frontier is. Hence, we used the anti-patterns to measure the quality of the WSDL document generated by the different migration approaches. In particular, we found the following anti-patterns in at least one of the WSDL documents in the three data-sets:

- Inappropriate or lacking comments [10]: Some operations within a WSDL have no comments or the comments do not effectively describe their associated elements (messages, operations).

- Ambiguous names [6]: Some WSDL operation or message names do not accurately represent their intended semantics.

- Redundant port-types: A port-type is repeated within the WSDL document, usually in the form of one port-type instance per binding type (e.g. HTTP, HTTPS or SOAP).

- Enclosed data model: The data model in XSD describing input and output data-types is defined within the WSDL document instead of being defined in a separate file, which makes data-type reuse across several Web Services very difficult. The exception of this rule occurs when it is known before-hand that data-types are not going to be reused. In this case, including data-type definitions within WSDL documents allows constructing self-contained contracts, so it is said that the contract does not suffer from the anti-pattern.

- Undercover fault information within standard messages [4]: Error information is returned using output messages rather than Fault messages.

- Redundant data models: A data-type is defined more than once in the same WSDL document.

Table 8: Anti-patterns in the three WSDL data-sets.

Anti-pattern/Data-set	**Direct Migration**	**Indirect Migration**	**Assisted Migration**
Inappropriate or lacking comments	Always	**Never**	When the original transactions use control couples
Ambiguous names	Always	**Never**	When the original transactions use control couples
Redundant port-types	When supporting several protocols	**Never**	**Never**
Enclosed data model	Always	**Never**	**Never**
Undercover fault information within standard messages	Always	**Never**	**Never**
Redundant data models	When two operations use the same data-type	**Never**	**Never**
Low cohesive operations in the same port-type	**Never**	**Never**	When several related transactions use a non related operation, such as formatting routines

- Low cohesive operations in the same port-type: Occurs in Web Services that place operations for checking the availability of the service and operations related to its main functionality into a single port-type. An example of this bad practice is to include operations such as "isAlive", "getVersion" and "ping" in a port-type, though the port-type has been designed for providing operations of a particular problem domain.

Table 8 summarizes the results of the anti-patterns analysis. When an anti-pattern affected a portion of the WSDL documents in a data-set, we analyzed which is the difference between these WSDL documents and the rest of the WSDL documents in the same data-set. Hence, the inner cells present under which circumstances the WSDL documents were affected by a particular anti-pattern. Since there are anti-patterns whose detection is inherently more subjective (e.g. "Inappropriate or lacking comments" and "Ambiguous names") [22], we performed a peer-review methodology after finishing their individual measurements to prevent biases.

Achieved results show that the WSDL documents of the Direct Migration data-set were affected by more anti-patterns than those of the Assisted Migration data-set, while no anti-pattern affected WSDL documents in the Indirect Migration data-set. The first two rows describe anti-patterns that impact on services comments and names [8].

It is reasonable to expect that these anti-patterns affected the WSDL documents of the Direct Migration data-set since all information included in them was derived from code written in CICS/COBOL, which does not offer a standard way to indicate from which portions and scope of a code existing comments can be extracted and reused. At the same time, names in CICS/COBOL have associated length restrictions (e.g. up to 4 characters in some CICS and/or COBOL flavors), names in the resulting WSDL documents were too short and difficult to be read. In contrast, these anti-patterns affected WSDL documents in Assisted Migration data-set only when the original CICS/COBOL is designed using control couples. This is because properly naming and commenting this kind of couples is known to be a complex task [27].

The third row describes an anti-pattern that ties abstract service interfaces to concrete implementations, hindering black-box reuse [8]. We observed that this anti-pattern was caused by the tools employed for generating WSDL documents during the first migration attempt. By default, the employed tool produces redundant port-types. To avoid this anti-pattern, developers should provide C# service implementation with rarely used annotations. Likewise, the fourth row describes an anti-pattern that is generated by many code-first tools, which force data models to be included within the generated WSDL documents, and could not be avoided within the Direct Migration WSDL documents. In contrast, neither the Indirect Migration nor the Assisted Migration data-sets were affected by these anti-patterns.

The anti-pattern described in the fifth row of the table deals with errors being transferred as part of output messages, which for the Direct Migration data-set resulted from the original transactions that used the same COMMAREA for returning both output and error information. In contrast, the WSDL documents of the Indirect Migration data-set and the Assisted Migration data-set had a proper designed error handling mechanism based on standard WSDL *fault* messages.

The anti-pattern described in the sixth row is related to bad data model designs. Redundant data models usually arise from limitations or bad use of the tools employed to generate WSDL documents. Therefore, this anti-pattern only affected Direct Migration WSDL documents. Although there were not repeated data-types at the WSDL document level, the Assisted Migration data-set had repeated data-types at a global level, i.e. when taking into account the data-types in all the documents. For instance, the error type, which consists of a fault code, string (brief description), actor, and description, was repeated in all the Assisted Migration WSDL documents. This is because this data-type was derived several times from the different sub-systems. Finally, this did not happen when using indirect migration because the WSDL document designers had a big picture of the system. We further analyze resulting data-types in the next section.

The last anti-pattern stands for having no semantically related operations within a port-type. This anti-pattern did not affected WSDL documents generated through direct migration or indirect migration. The Direct Migration data-set was not affected because each WSDL document included only one operation, while the Indirect Migration WSDL documents were specifically designed to group related operations. However, the assisted migration approach uses an automatic process to select which operations go to a port-type. In our case study, we found that when several related operations used the same unrelated routines, such as text-formatting routines, the Assisted Migration

Table 9: Data-type definition: Detailed view.

Data Model characteristics/Data-set	**Direct Migration**	**Indirect Migration**	**Assisted Migration**
Defined data-types	182	235	191
Average definitions per data-type	1.29	1.0	1.13
Unique data-types	141 (77%)	235 (100%)	169 (88%)

approach suggested that these routines were also a candidate operation for that service. This resulted in services that had port-types with several related operation, and one or two unrelated operations.

Although the assisted migration has a step to eliminate WSDL document anti-pattern causes, some of the generated WSDL documents were affected by some anti-patterns. This might be for two reasons, the first one is that the assisted migration is an iterative process and we only performed one iteration. The second reason is that the OO refactorings are not enough to remove all the anti-pattern causes. For instance, the Enclosed data model anti-pattern usually results from the tool used for generating WSDL documents, when this tool does not support separating the XSD definitions in another file, regardless how the source code implementing the associated service is refactored. In both cases, further research is needed to fully assess the capabilities of the assisted migration.

3.5.3 Data model analysis

Data model management is crucial in *data-centric* software systems such as the one under study. Therefore, we further analyzed the data-types produced by each migration approach. Table 9 shows metrics that depict the data-types definitions obtained using the different migration approaches. This table has a special focus on which percentage of the data-type are defined more than once, which is undesirable because it hinders data-type definitions reuse. The first clear difference was the number of data-types defined. The Direct Migration data-set contained 182 different data-types and 73% of them were defined only once. Since the associated WSDL documents did not share data-type definitions, many of the types were duplicated across different WSDL documents. In contrast, all the 235 unique data-types were defined for the WSDL documents of the Indirect Migration data-set. Among this set, 104 data-types represented business objects, including 39 defined as simple types (mostly enumerations) and 65 defined as complex types, whereas 131 were elements used for compliance with Web Service Interoperability standards (WS-I)[6]. Finally, 196 data-types were defined in the Assisted Migration data-set. From these 191 data-types, 116 definitions were business objects (34 simple types + 82 complex types), while 75 definitions were elements used for WS-I compliance reasons.

The WS-I defines rules for making Web Services interoperable between different platforms. One of these rules is that message parts always should use XSD *elements*,

[6]Basic Profile Version 1.1: `http://www.ws-i.org/Profiles/BasicProfile-1.1.html`

Listing 1: Complex type definition example.

```
<xsd:complexType name="Cuil">
        <xsd:sequence>
                <!-- Preffix -->
                <xsd:element name="prefijo" type="tns:CuilPrefijo"/>
                <!-- Identity document -->
                <xsd:element ref="tns:Documento"/>
                <!-- Control (validation) digit -->
                <xsd:element name="digitoControl" type="tns:CuilDigito"/>
        </xsd:sequence>
</xsd:complexType>
```

Listing 2: Wrapper element definition example.

```
<xsd:element name="cuil" type="ns:Cuil"/>
```

although according to the Web Service specification message parts might use XSD *elements* or XSD *types*. For example, Listing 1 shows a data-type defined using a complex type, and the element shown in Listing 2 wraps the complex type.

Regarding to data-type repetitions, the Direct Migration data-set included 182 data-types definitions of which 133 where unique. This means that 27% of the definitions were not necessary and could be substituted by other data-type definitions. In contrast, the Indirect Migration data-set comprised 235 data-types –all of them were unique– meaning that the data-types were well defined and correctly shared across the associated WSDL documents. Finally, the Assisted Migration data-set had 191 data-types, and 169 of them were unique. Therefore, Assisted Migration generated WSDL documents almost as good as the the ones generated by the indirect migration. To sum up, the Direct Migration, Indirect Migration and the Assisted Migration data-sets had 1.36, 1, and 1.13 data-type definitions per effective data-type.

The fact that the WSDL documents of the Indirect Migration data-set had fewer data-type definitions for representing business objects (104) than the others (i.e. 182 for the Direct Migration WSDL documents and 116 for the Assisted Migration WSDL documents), indicates a better level of data model reutilization and a proper utilization of the XSDcomplex and element constructors to be WS-I compliant. However, notice that the Assisted Migration and the Indirect Migration data-sets almost included the same number of business objects.

Finally, we studied how the different services belonging to Indirect Migration and Assisted Migration data-sets reused the data-types. We intentionally left out the services generated by the direct migration because they did not share data-types among individual services. Figure 3 depicts a graph in which services and data-types are nodes, and each edge represents a use relationship between a service and a data-type. An evident feature in both graphs is that there was a data-type that is used by most of the services. This data-type is CUIL, which is the main identifier for a person. The main difference between the graphs is that the one belonging to the Indirect Migration is a weakly connected graph without "islands", while the Assisted Migration graph is

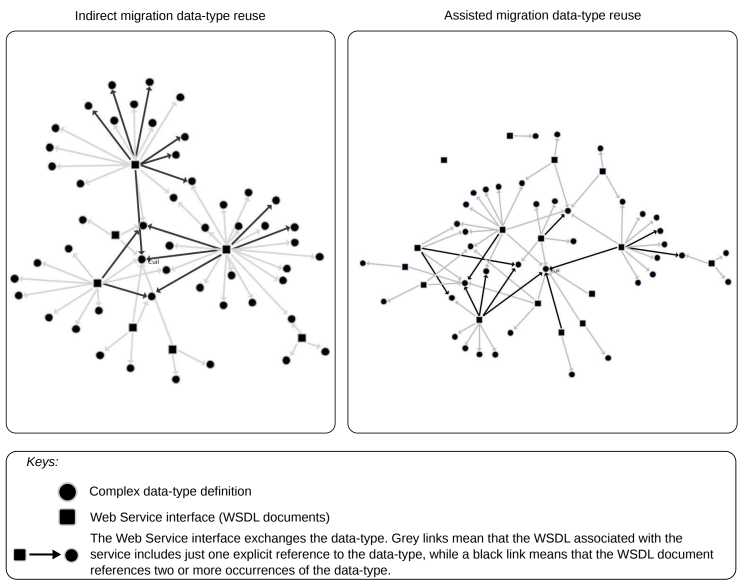
Indirect migration data-type reuse
Assisted migration data-type reuse
Keys:
Complex data-type definition
Web Service interface (WSDL documents)
The Web Service interface exchanges the data-type. Grey links mean that the WSDL associated with the service includes just one explicit reference to the data-type, while a black link means that the WSDL document references two or more occurrences of the data-type.

Figure 3: Data-type reuse comparison.

a disconnected graph. This happened because the assisted migration is not as good as exhaustively detecting candidate reusable data-types by hand. Despite of this, only 2 services were not connected to the biggest graph, which means that the ability of assisted migration to detect data-type reuse is fairly good.

4 Future Research Directions

We are at present extending our work in several directions. First, we are refining the heuristics of our tool so as to make them more accurate. Second, we are planning to enforce our findings by using other legacy systems. As a starting point, we will use an RM-COBOL system that comprises 319 programs and 201828 lines of code. Third, we will investigate whether our ideas hold for legacy systems written in other languages, such as in C or Java. Lastly, even when we found that using direct migration certainly has incidence in service quality at the service frontier level, intuitively many WSDL anti-patterns might be actually introduced by the tools employed to build the corresponding WSDL documents. Indeed, recently, we have shown that there is a statistical correlation between the WSDL anti-patterns present in a SOA frontier depending on the code-first WSDL generation tool being employed [20]. We are therefore planning to extend this analysis to other languages and therefore WSDL construction tools. Our utmost aim is to determine how much WSDL anti-patterns occurrences are explained by the approach to migration itself, and how much of them depend on the WSDL tools used.

Another research direction is taking into account other design service quality metrics (e.g. [12]) for improving the results of applying the assisted migration. These metrics might be used for assisting developers to make decisions when they are performing the manual part of the assisted migration. For instance, these metrics can be used to identify real services from a list of candidate services.

5 Conclusions and Future work

Nowadays, organizations are more and more faced with the need of modernizing their legacy systems to newer platforms. Particularly, current legacy systems are mostly written in COBOL, whereas the target paradigm for migrating these systems is commonly SOA (Service-Oriented Architecture) due to its widely-recognized benefits in terms interoperability and reusability. A question that arises is, however, which is the best way to painlessly moving from a legacy system to exploit the advantages of SOA, since migration is in general an arduous endeavor for any organization.

In this paper, through a real-world case study involving the migration of an old CICS/COBOL system to SOA and .NET, we have shown that the traditional "fast and cheap" approach to migration –i.e. direct migration– produced a not-so-clear SOA frontier. Therefore, the resulting services were hard to reason about and consume by client application developers. Alternatively, through an indirect migration approach, a better SOA frontier in terms of service quality was obtained, at the expense of much higher development costs. The common ground for comparison was basically an established catalog of WSDL anti-patterns [23] that are known to hinder service understandability and discoverability. All in all, an interesting finding from this experience is that there is a relationship between the approach to migration to SOA used and the number of anti-patterns found in the resulting SOA frontiers.

Motivated by the high costs of indirectly migrating that system by hand, we also proposed a semi-automatic tool to help development teams in migrating COBOL systems to SOA. Our tool comprises a number of heuristics that detect bad design and implementation practices in legacy systems, which in turn serve as a mean to propose early code refactorings so that the final SOA frontier is free from as much WSDL anti-patterns occurrences as possible. To evaluate the approach, we used the CICS/COBOL system mentioned above. In the end, results were encouraging, since migration costs were dramatically reduced and service quality was very acceptable and close to that of indirect migration.

References

[1] Saad Alahmari, Ed Zaluska, and David De Roure. A service identification framework for legacy system migration into SOA. In *Proceedings of the IEEE International Conference on Services Computing*, pages 614–617. IEEE Computer Society, 2010.

[2] A. Almonaies, J.R. Cordy, and T.R. Dean. Legacy system evolution towards service-oriented architecture. In *International Workshop on SOA Migration and Evolution (SOME), Madrid, Spain*, pages 53–62. OFFIS, 2010.

[3] Marco Battaglia, Giancarlo Savoia, and John Favaro. Renaissance: A method to migrate from legacy to immortal software systems. In *Proceedings of the 2nd Euromicro Conference on Software Maintenance and Reengineering (CSMR'98)*, CSMR '98, pages 197–, Washington, DC, USA, 1998. IEEE Computer Society.

[4] Jack Beaton, Sae Young Jeong, Yingyu Xie, Jeffrey Jack, and Brad A. Myers. Usability challenges for enterprise service-oriented architecture APIs. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 193–196. IEEE Computer Society, Sept. 2008.

[5] Martin Bichler and Kwei-Jay Lin. Service-Oriented Computing. *Computer*, 39(3):99–101, 2006.

[6] M. Brian Blake and Michael F. Nowlan. Taming Web Services from the wild. *IEEE Internet Computing*, 12(5):62–69, 2008.

[7] Michael L. Brodie and Michael Stonebrake. Darwin: On the incremental migration of legacy information system. Technical report, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, 1993.

[8] Marco Crasso, Juan Manuel Rodriguez, Alejandro Zunino, and Marcelo Campo. Revising WSDL documents: Why and how. *IEEE Internet Computing*, 14(5):30–38, 2010.

[9] John Erickson and Keng Siau. Web Service, Service-Oriented Computing, and Service-Oriented Architecture: Separating hype from reality. *Journal of Database Management*, 19(3):42–54, 2008.

[10] Jianchun Fan and Subbarao Kambhampati. A snapshot of public Web Services. *SIGMOD Rec.*, 34(1):24–32, 2005.

[11] Martin Fowler. Refactorings in Alphabetical Order. `http://www.refactoring.com/catalog/index.html`, 1999.

[12] Michael Gebhart and Sebastian Abeck. Metrics for evaluating service designs based on soaml. *International Journal on Advances in Software*, 4:61–75, 2011.

[13] Anca Daniela Ionita, Alessandra Catapano, Stelian Giuroiu, and Monica Florea. Service oriented system for business cooperation. In *Proceedings of the 2nd international workshop on Systems development in SOA environments*, SDSOA '08, pages 13–18, New York, NY, USA, 2008. ACM.

[14] Richard C. Leinecker. *Com+ Unleashed*. Sams, 2000.

[15] G. Lewis, E. Morris, S. Simanta, and D. Smith. Service Orientation and Systems of Systems. *Software, IEEE*, 28(1):58 –63, jan.-feb. 2011.

[16] Grace Lewis, Edwin Morris, and Dennis Smith. Migration of legacy components to service-oriented architectures. *Journal of Software Technology*, 8:14–23, 2005.

[17] Shing-Han Li, Shi-Ming Huang, David C. Yen, and Cheng-Chun Chang. Migrating legacy information systems to Web Services architecture. *Journal of Database Management*, 18(4):1–25, 2007.

[18] Marin Litoiu. Migrating to Web Services: a performance engineering approach. *Journal of Software Maintenance and Evolution: Research and Practice*, 16(1-2):51–70, 2004.

[19] Cristian Mateos, Marco Crasso, Alejandro Zunino, and Marcelo Campo. Separation of concerns in service-oriented applications based on pervasive design patterns. In *Web Technology Track (WT) - 25th ACM Symposium on Applied Computing (SAC '10), Sierre, Switzerland*, pages 2509–2513. ACM Press, 2010.

[20] José Luis Ordiales Coscia, Cristian Mateos, Marco Crasso, and Alejandro Zunino. Avoiding wsdl bad practices in code-first web services. In *Proceedings of the 12th Argentine Symposium on Software Engineering (ASSE2011) - 40th JAIIO*, pages 1–12, 2011.

[21] Juan M. Rodriguez, M. Crasso, C. Mateos, A. Zunino, and M. Campo. The EasySOC project: A rich catalog of best practices for developing Web Service applications. In *Jornadas Chilenas de Computación (JCC) - INFONOR 2010, Antofagasta, Chile*, pages 33–42. SCC (Sociedad Chilena de la Ciencia de la Computación), 2010.

[22] Juan Manuel Rodriguez, Marco Crasso, Alejandro Zunino, and Marcelo Campo. Automatically detecting opportunities for Web Service descriptions improvement. In *10th IFIP WG 6.11 Conference on e-Business, e-Services, and e-Society (I3E 2010), Ciudad Autónoma de Buenos Aires, Argentina*, volume 431, pages 139–150. Springer Boston, 2010.

[23] Juan Manuel Rodriguez, Marco Crasso, Alejandro Zunino, and Marcelo Campo. Improving Web Service descriptions for effective service discovery. *Science of Computer Programming*, 75(11):1001–1021, 2010.

[24] Harry Sneed. A pilot project for migrating COBOL code to Web Services. *International Journal on Software Tools for Technology Transfer*, 11:441–451, 2009. 10.1007/s10009-009-0128-z.

[25] Harry Sneed. Measuring Web Service interfaces. In *12th IEEE International Symposium on Web Systems Evolution*, pages 111 –115, sept. 2010.

[26] Eleni Stroulia and Yiqiao Wang. Structural and semantic matching for assessing Web Service similarity. *International Journal of Cooperative Information Systems*, 14(4):407–438, 2005.

[27] Edward Yourdon and Larry L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1979.

[28] C. Zillmann, A. Winter, A. Herget, W. Teppe, M. Theurer, A. Fuhr, T. Horn, V. Riediger, U. Erdmenger, U. Kaiser, D. Uhlig, and Y. Zimmermann. The soamig process model in industrial applications. In *Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering*, CSMR '11, pages 339–342, Washington, DC, USA, 2011. IEEE Computer Society.