

Reducing Efforts in Web Services Refactoring

Guillermo Rodriguez[1(✉)], Leonardo Fernández Esteberena[2], Cristian Mateos[1], and Sanjay Misra[3]

[1] ISISTAN-UNICEN-CONICET, Tandil, Argentina
{guillermo.rodriguez,cristian.mateos}@isistan.unicen.edu.ar
[2] Globant, Tandil, Argentina
[3] Covenant University, Ota, Nigeria
ssopam@gmail.com

Abstract. In Service-Oriented Computing, systems provide a service interface described by a computer-readable language called Web Services Description Language (WSDL), but document descriptions often exhibit design problems as systems expand. Moreover, a major problem in this type of applications is its growth; as size and complexity of applications increase, the probability of duplicity of code also increases. This issue could have a negative impact on quality attributes, such as performance, maintainability and evolution, among others, providing developers with some clues to detect refactoring opportunities. Conducting a detection process of these opportunities could be a daunting task; however, this work proposes a methodology to conduct manual refactoring of service descriptions of legacy systems. The methodology allows software developers to collect metrics of time effort and space reduction. The metrics indicate a 75% reduction in services and 63% of the lines of code on average. Additionally, a statistical analysis carried out with the obtained metrics yielded as result that the number of services refactored according to the size of its refactoring group is adjusted to a logarithmic function. We also analyzed how refactoring time increases with the size of services. The results indicate that our methodology could be applied in similar case-studies based on legacy systems.

Keywords: Software refactoring · Web Services · Web service description language · Statistical analysis

1 Introduction

In the context of Service Oriented Computing (SOC), a widespread implementation of services is the use of Web Services [1]. Web services enable interactivity among software components. The communication transmits distributed data and information between those components via standard Web protocols [2]. A Web Service is a piece of software designed to enable interoperability in machine-to-machine interactions, making use of different technologies and protocols such as

© Springer Nature Switzerland AG 2019
S. Misra et al. (Eds.): ICCSA 2019, LNCS 11622, pp. 544–559, 2019.
https://doi.org/10.1007/978-3-030-24305-0_41

SOAP, HTTP and XML along with other standards. To exhibit their functionality, Web Services provide an interface described by the Web Services Description Language (WSDL) [3].

However, from a technical perspective, software developers need to dedicate significant effort to discovering sets of suitable services, interpreting them, developing software that overcomes their inherent data and process mismatches, and finally composing them into a complex process [4]. Furthermore, a common difficulty that arises with the use of WSDL documents is that they are prone to develop design problems as the system expands, forcing them to increase development effort for adding new functionalities [5]. Another challenge of WSDL documents is how to integrate service based tools and contents, mobile applications and cloud computing services into the legacy systems [6].

To deal with those issues, refactoring seems to be the most suitable strategy to apply. By refactoring the code, the internal structure of the system can be improved but the system behavior is not altered. Ideally, developers should allocate time in their development iterations to fix refactoring issues, particularly those that are critical for maintenance or harm the system architecture [7].

In this work we propose a novel methodology to conduct a refactoring process of legacy service-oriented applications wrapped by means of WSDL documents. We have previously developed a tool to conduct a semi-automated detection of refactoring opportunities [8]. Our aim is to provide metrics that quantify in general terms of WSDL documents before and after the refactoring, and the effort required to achieve this process manually. Additionally, a statistical analysis carried out with the obtained metrics yielded as result that the number of services refactored according to the size of its refactoring group is adjusted to a logarithmic function. We also analyzed how refactoring time increases with the functional size of services.

The remaining of this paper is organized as follows. Section 2 describes the background. Section 3 presents the related work. Section 4 presents the proposed approach to conduct the refactoring process. The empirical evaluation of the approach is explained and summarized in Sect. 5. Finally, conclusions and future lines of research are stated in Sect. 6.

2 Background

Service-Oriented Computing paradigm promotes the efficient development and deployment of an application in terms of cost and time. To flexibly enable the composition of applications Web Services are utilized as building blocks [9]. In this context, web services are usually described by means of WSDL documents. This language is an XML format that allows software developers to describe service functionality and communication protocols to be accessed through the Internet [10]. The main conceptual elements in WSDL documents are Types, Messages, PortTypes, Bindings and Services, among others. Types contain the definitions of the types of data that will be used in/out the service. To specify types, an XML schema is utilized. Each data type defines an element with a complexType inside it, which lists a series of elements of basic types (built-in).

Messages define the contents of a message abstractly by using data types defined in *wsdl: types*. Each message consists of one or more parts that will make use of an element or complexType, referencing them with the element and type message attributes respectively. There will be as many *wsdl:message* tags as necessary.

A PortType is a set of abstract operations that indicate which abstract messages are received and sent. WSDL supports several operational patters including one-way (receives a message), request-response (receives a message, then sends one in response), request-response (sends a message, then receives one in response) and notification.

To define operations, their messages are detailed by means of the *wsdl:operation tag*. The fact that they are request-response operations are specified implicitly by firstly placing the *wsdl:input* tag and then *wsdl:output*. Bindings provide the format and protocol details for the messages and operations of a PortType. For example, the operations might be accessed by means of the protocol SOAP, and then a specific SOAP binding is declared in the document [11]. Finally, Services are a set of related ports under the same name. Each port defines a particular and concrete endpoint of the communication, linking a binding with a location address.

3 Related Work

Software evolvability is an important quality attribute, however, sometimes it is hard to achieve specially in long-living systems that are in continuous change. Moreover, some authors have concentrated on supporting evolvability in SOA systems by improving web services descriptions [12]. Mateos et al. have introduced an Eclipse plug-in to assist software developers in obtaining WSDL documents from the code, minimizing the number of anti-patterns [13]. Webster et al. have presented an approach to try to resolve the conflict which is specified when the service provider needs evolve its interface to respond to new requirements, without affecting consumers of that service [14]. In [10], the authors have presented a novel structural-semantic approach to help developers in retrieving and selecting services. The approach is based on a comprehensive structural scheme for service Interface Compatibility analysis, and WordNet as the semantic support to assess identifiers of operations and parameters.

The authors in [15] have revealed a lack of systematic guidance on the refactoring process for existing monolithic applications. However, there is a wide range of semi-automated techniques to support the refactoring process. In [16], the authors propose a semi-automatic approach for migrating applications to a PaaS environment, avoiding cloud restrictions through user-defined refactorings. In [17], the authors have generated insights into service-oriented evolution qualities, and provided a modifiability comparison of the two popular service-based architectural styles by qualitatively mapping principles and patterns of Service-Oriented Architecture (SOA) and Microservices onto tactics.

Although there is a considerable number of research works that have addressed the refactoring process of service-oriented applications, our research

work proposes a novel methodology beyond a systematic guidance that also provides an statistical analysis of refactoring time as a function of the size of services.

4 Our Approach

The general steps of the proposed methodology are explained by means of an activity diagram (Fig. 1). The methodology begins with the initial classification and then a iterative procedure takes control until all original services are grouped into a refactored service. Each iteration is responsible for refactoring a service and consists of the following steps: Detection, Design and Specification. At the end of each iteration a question is made: Are there original services not included in the refactored services? If so, another iteration must be started from the Detection step otherwise the refactoring is done.

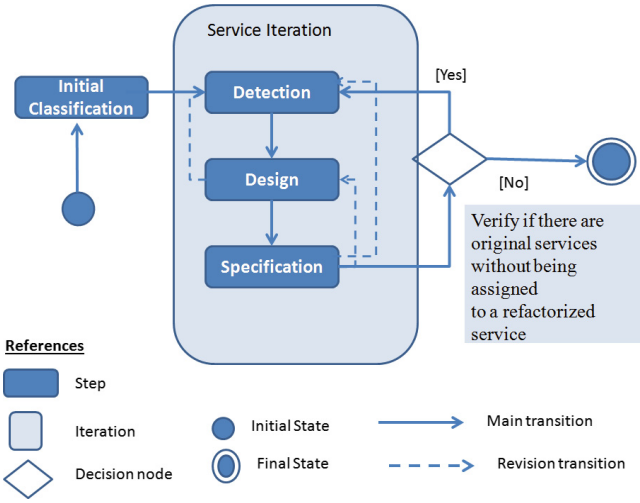

Fig. 1. Our proposed methodology.

Note that the methodology shows which step should be passed at the end of the previous step using the main transitions, but also Fig. 1 indicates revision transitions that are taken during the development of a step to go back when any of the previous decisions should be improved. The following sub-sections shed some light on each of the steps of the proposed methodology.

4.1 Initial Classification

Initially, WSDL documents are received with no defined organization. Starting to generate refactoring groups in this situation would involve browsing all the

documents and internally exploring a considerable number of operations, with no notion of its functionalities. To deal with this issue, a first initial classification is done. This allows for:

- Having a first contact with the system, familiarizing in general with the types of services provided and sometimes suggesting how they relate each other.
- Creating an organizational structure that divides services into areas of possible clustering. In many cases an internal search in a classification area is enough to select a refactoring group.
- Even in several cases where the formation of a group requires the detailed exploration of more than one classification area (in taxonomic terms) it is possible that the structure act as an index, speeding up any search process.

The classification is not intended to accurately delineate groups for refactoring; this task demands effort and will be executed during service iterations. On the contrary, it must be general and brief. Since each WSDL document contains one service tag, the document name is a suitable indicator of the functionalities the service provides and sufficient for this categorization.

To carry out the classification, a tree structure is adopted, where two nodes are created: one with all the original documents and the other for the classified documents, which will then contain a subnode for each category. The original services are sorted alphabetically, taken one by one or from nodes with similar names. Then, they are copied in the (sub)node of one of the areas according to the semantic similarity with respect to the already classified services of each area. If the area does not exist yet (as occurs at the beginning) a new subnode is created within the root node of classified services (that is, without creating more depth in the tree structure) that should be named with a business area or process, a system concept or ultimately any noun that is representative of the functionality of the services. The tree structure is mapped by our supporting tools to a directory structure in persistent storage.

The granularity of the classification should be enough to separate most of the services with similar names to each other but not so much as to create subnodes that allocate only two or three documents. Besides, it is recommendable to avoid incurring details that could only be resolved by exploring the content of the WSDL documents by the developer. For these two reasons, it is suggested to create a category named "Other" for the services that are particular to enter into other classifications or that cannot be classified without a WSDL document perusal.

After classifying all the documents, each subnode is browsed and its content is reviewed, making local changes to services that are considered poorly classified and being aware of the features provided, without accessing the content of the documents. This first task should not take more than one hour (of useful working time) and the resulting artifact is a set of nodes, or directories, representing functional areas of the system, which contain all the original WSDL documents.

4.2 Service Iteration

After the initial classification, a series of iterations for refactoring the services are conducted, each one creates a service in the new system that supplies the functionalities of a group of original services. The iteration is divided into three steps: detection, design and specification, which must be performed preferably without major interruptions, so as not to hinder the process and avoid revisions that would increase the total necessary time. Although the steps are commonly performed in order, it is not a completely linear process: in several circumstances it becomes necessary to return temporarily to a previous step to make some revisions and changes.

Detection. The first step in our methodology is to find a group of services that can be unified. The discovery of refactored services is not necessarily achieved at the end of this step. In several cases, for example, decisions as to whether the original services will comprise a single refactored service or several services, whether any of them should be left out or in, whether it is better to unify them or not, they are just made in the step of design and occasionally modified even during the specification step. The aim of this first step is to find a set of potentially candidate groupable services.

Firstly, the category directories of the initial classification are used as input (they will have been modified if one or more refactoring iterations have passed) and one directory is chosen. The choice may be at random, however, in case there have been previous iterations, it is useful to prioritize areas related to the recent refactorings to take advantage of the developer's familiarity and his recall capabilities.

Within the chosen category, WSDL documents are opened and candidates for unification are sought for evaluating the content of the documents. Once a group of well-defined candidates is selected, possible members in the category "Other" (if it exists) and in other categories considered to be closely related are revised, looking for names that suggest a relationship and opening more candidate documents for evaluation. The detection ends when a group of unified services is considered done and there is no other service to add to this group. The evaluation criteria for the selection of candidate services are:

- The services provide the same functionality: sometimes, functionality provided are quite similar, but sometimes it is necessary to elucidate if the functionality is at least partially overlapping. One way to determine this is to look at the names of the services and operations that define and ask: could the functionality of candidates be supplied by combining or intelligently using operations of other candidates, at least partially?
- The domain of the services is the same: in most cases the services which must be grouped, work on the same data set. The natural place to find out is in dataTypes defined by types. It is highly likely that candidate services will have definitions of dataTypes with equivalent names, but more important is that they have within their definitions various elements with common names

and types. Another possibility is that there are many elements with the same names in more than one service, but distributed between definitions of different data types. This is also another reason to think that services work on the same domain and that the design of dataTypes could be generalized.
- The ultimate goal of service provision is the same: even when services have disjoint functionalities, these services must be candidates if the fundamental reason of their existence for the user is the same. Finding clues about this requires looking beyond the immediate meaning of the names of operations and visualizes in which context the user will make use of the service. A useful question is: at the time of use, would the user naturally find all the operations of the services if they were included in a service with a more general name? If the answer is yes and the granularity is fine enough, unification must be considered.
- The services belong to the same business area or domain: there are few cases in which a service refactored is not defined by original services belonging to the same category. This issue is the main reason why the initial classification accelerates the Detection step, but at this stage all the content of the documents can be used (operations names, dataTypes, message content, etc.) to find new relations between services. DataTypes are especially useful in opportunities where the service name obfuscates its functionality but the names of the elements in the dataTypes reveal a relationship at concept level.

Not all criteria must be met to establish services as candidates, nor is it strictly necessary to check the criteria one by one separately. Instead, WSDL documents are browsed and a consideration is made of whether the services comply with them to a greater or lesser extent before making the decision that a group for refactoring has actually been detected. In case of doubt whether a service should be part of the group or not, it is better to include it: the original services that will integrate the refactored service can be decided with design considerations in the subsequent steps. On the other hand, if it is decided that the selected group is wrong, it should be discarded and start the detection step again.

Finally, a service can be labeled with a different category to another during the Detection step and if it is necessary a new category can be created. Detection requires more research and time effort than the initial classification, but it is basically the continuation of the same, carried out at the same time as gaining detailed knowledge of the system through the other stages of the iterations. The result of the Detection step is the set of services to be unified.

Design. Once a group of candidate services is defined, it is necessary to determine how the service will be actually refactored. The Design step aims to solve the main problems of integration of the original services and to give shape to the refactored service. In particular, this step consists of the decisions about how operations and dataTypes will be generalized. The two main activities and the criteria for each are:

- The generalization of operations, so that a minimum set of operations covers all the original features. It is not vital that the functions given are exactly same as the refactored service allows for performing all the previous actions with such ease. Also, it is not a problem that the refactor system provides a bit more functionality than before, as long as a new definition of an operation does not necessarily imply a substantial drift in the quality of service.
- The generalization of dataTypes, with the main objective of minimizing the size of description documents. In this case, the size refers to the number of definitions of dataTypes, the number of messages and the number of lines of code. Additionally, this activity takes into account to a lesser extent the modifiability of messages content. This contributes to the centralization and to locate changes in the parameters of operations, in addition to improve readability for subsequent changes and for future refactorings. In any case, the priority is always to give the ability to reuse dataTypes with the main aim to reduce the number of lines of code.

The approach starts with the activity that is considered to be less complex for the candidates and then follow with the others, adapting it to the first perspective of the solution.

In practice, the recommendation is to start with dataTypes, especially for large groups, because these groups have a multitude of operations whose delimitation is not clear: the dataTypes give a real look at the information that the user will need to provide in order to use the operation. This issue makes a guide to group operations with data requirements to decide whether the operations need to be defined separately. For many services centered on queries to a database this approach arises naturally; the general design will rotate around the information that must be received and sent. In addition it is common to use dataTypes with the same name as the operations in the case of the SOAP protocol, so rethink the dataTypes often provide enough information to reorganize operations later.

Alternatively, when the services have similar benefits or the same subsequent goals, it is possible to start off by setting the final group of operations and then define dataTypes for the objects involved in each new operation, ignoring previous dataTypes except to collect the elements contained. This approach was used less frequently in practice, but it is almost mandatory for cases where the data domain of candidates is not the same.

The goal of the Design step is to create a general concept for service as briefly as possible, to be expanded and refined during the specification. The design experience of each developer will propose different alternatives on how to achieve this, some may be more suitable than others in terms of conceptual integrity but, beyond that, any solution is acceptable as long as it meets the aforementioned criteria in the two activities. However, to begin with, the following approach to a solution is suggested: to browse all candidate documents, to identify repeated entities (referring primarily to operations and types) and answer the following questions:

- Is it one entity included within another? For dataTypes this means that the elements of a definition are a subset of the elements of another definition.

Operations should be analyzed semantically if one of them can be used to obtain the same (or the same ones and more) end effects than the other. If so, the entity that solves only the particular case must be eliminated.
- Could two entities be daughters of another entity in an inheritance tree? If true, both entities must be eliminated and the general case must be defined.

Moreover, in the case of dataTypes, it is also important to consider whether they define all abstract objects that are individually required. Sometimes, elements of several abstract concepts are merged into the same dataType. It is desirable as long as the objects have no more than one dependency from different origins, because that would force them to repeat their elements. For example, the data of an Order can be next to those of the Invoice within a dataType used by an operation, but on the other hand, the data of the Invoice are repeated next to those of the Delivery inside another dataType used by a second operation. This situation is undesirable from the viewpoint of modifiability, minimization of lines of code and intelligibility; a possible solution is to define the complexType of the invoice to be used by the other two elements. Situations may involve more complex relationships, but the solution can always be achieved by identifying involved objects that need to be reused or used individually and create a definition for each object.

The proposed solution must include all services that will be unified and therefore some services candidates can be disregarded if they fail to fit the design or are considered to work better for outside of this refactored service. Additionally, it is possible to deal with considerable difficulty to find a general design that show the necessity of dividing the group of candidate services into more than one service refactored. In that case, the group more studied and defined is taken to continue with the current step. If, on the other hand, the solution allows spectrum of services, it is possible to go back to the Detection step to search for more potentially compatible candidates before proceeding. Once a viable solution has been found and the documents that will be part of the refactored service has been selected, they are taken to a separate node, which later will be named with the new service name.

The result of the stage is the set of original services that will define the refactored service on the one hand, and the design concept on the other. At this stage, this set (e.g. mapped to a WSDL document folder) goes to the next step: Specification.

Specification. The last step is to specify the service, this means defining the refactored service in written format, with enough detail so that the materialization can be derived easily. For that, software developers should declare all dataTypes, messages, portTypes and operations that the refactored service will have and all its operations must be explained concisely. This level of detail forces the details and occasionally highlights unforeseen design conflicts, due to particularities in the relationships between operations, messages and dataTypes of the services. The sub-steps are listed as follows:

- To declare the service. As a consideration to contribute to the readability and understanding of the Web Services of the system, it is preferred using names related to processes, concepts or business nouns (as specified in the Oracle standard [10]).
- To declare operations based on the concept proposed at the design step. For the name of the operations, the use of verbs is highly recommended.
- To declare the dataTypes taking into account the data that is necessary for the messages of each operation: first declaring the dataTypes raised in the solution of the previous step and then ensuring to have the dataTypes that will be needed to send and receive the information associated with each operation. At this point, protocols and patterns will be used, respecting the style of the original system but making sure to simplify it.
- To declare the messages, make sure to cover the supported protocols but minimizing the number declared. For example, the use of SOAP protocols is allowed, SOAP 1.2, HTTP-GET and HTTP-POST, but this requires only two variations in the messages: SOAP (containing a single dataType that bears the name of the operation) and HTTP. Additionally, it is important to reuse the messages when the same dataType is required for more than one operation.
- To declare the portTypes, which in practice are always one per supported protocol.
- To define the operation of operations in natural language. It is preferable that explanations are short and preferably rigorous.

The final result after applying our methodology is a document that specifies the refactored service, with all the information necessary for its materialization from the group of original services.

5 Experimental Results

Our proposed methodology was applied on a case-study that contains a total of 252 operations in 211 services, each defined by a WSDL document. By applying our methodology, that number was reduced to 140 defined operations in 52 services. It is assumed that any changes produced by this refactoring should also impact accordingly on the components that interact with the web services, both on the server or on the client sides; however, this analysis is out of the scope of this work. Web services refactorings maintain the original style and support the initial set of functionalities.

In order to quantify the results achieved and the effort required by and for manual refactoring, a set of metrics related to both working time and features of WSDL documents before and after the refactoring were taken into account.

5.1 Analysis of Metrics

To evaluate the impact of applying our methodology, Table 1 shows the results of the metrics before and after the refactoring process. The last column calculates the percentage of gain achieved.

Table 1. Obtained reductions after refactoring.

	Original system	Refactored system	Reduction
Services	211	52	75.36%
Operations	252	140	44.44%
DataTypes	1144	616	46.15%
Messages	1512	458	69.71%
PortTypes	633	104	83.57%
Lines of Code	38912	14201	63.50%

Figure 2 shows that refactoring significantly reduced the size of the elements of WSDL documents, such as services, operations, dataTypes, messages, portTypes and lines of code. For example, the number of services and operations fell by 75% and 44% respectively. Figure 3 shows the mean and standard deviation of reduction of each WSDL element analyzed.

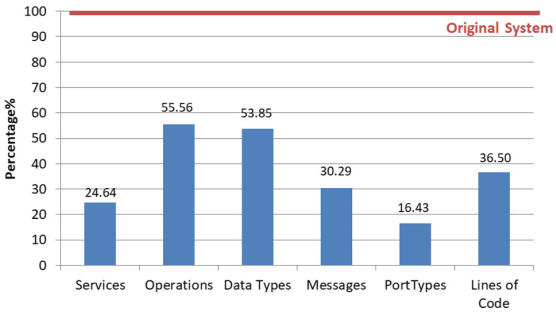

Fig. 2. Refactored system as percentages of original system.

Figure 4 shows a histogram with the absolute number of refactored services (axis X) that are composed of different number of original services (axis Y) and the best adjustment function found using the Origin software[1]. The number of services refactored according to the size of its refactoring group is adjusted to a logarithmic function:

$$y = a - (b * ln(x + c))$$

In this case, $a = 6.4$; $b = 2.3$ and $c = -0.995$ with a determination coefficient of adjusted of $R^2 = 0.93$.

The dataset size is insufficient to perform a generalization and the coefficient of determination may be due to over-adjustment. However, the residual distances of data to the logarithmic function seem to have a random distribution,

[1] http://www.originlab.com/.

so the high coefficient R^2 shows the curve is representative for this particular refactoring. The analysis also goes according to the intuitive perception that as refactoring progresses, it is increasingly difficult to recognize large refactoring groups.

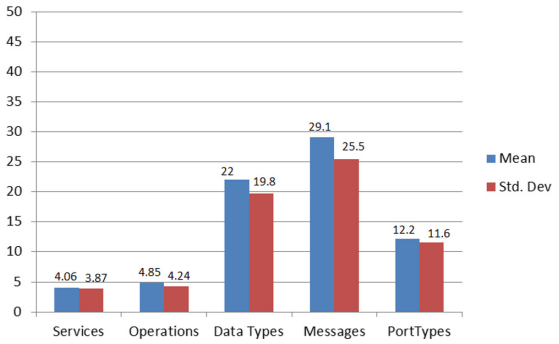

Fig. 3. Mean and standard deviation of refactoring groups.

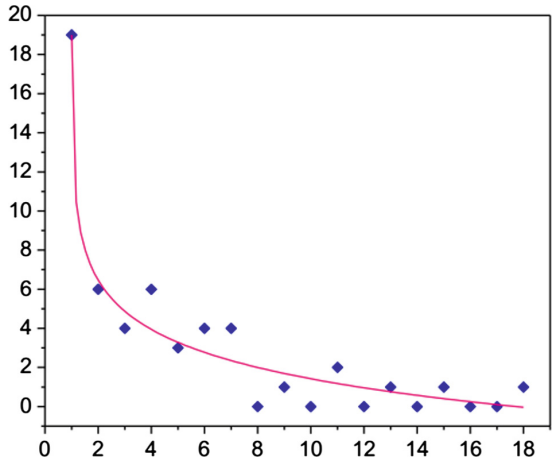

Fig. 4. Histogram of distribution of refactored services by number of members.

Figure 5 depicts the percentage of time to perform each task of our methodology. Approximately, 50% of the time was spent making the specification itself and 5% of the total time was used in revisions on how the specification was performed, this means that 57% of the total time is directly invested in the specification of services. Searching for candidates services for refactoring,

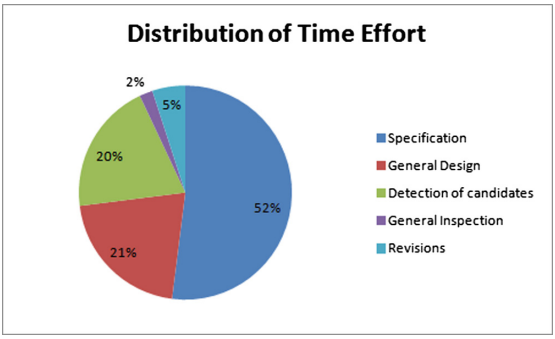

Fig. 5. Distribution of time invested in each refactoring step.

posing general solutions and inspecting the final system accounted for 43% of the time, this is the equivalent to 75% of the time required to perform the specification.

Table 2 shows the total time required by each step of the methodology when conducting the case-study: Detection step 20%, Design step 21%, Specification step 52%, Revision step 5%, General Inspection 2%. Moreover, Table 3 shows on average the time required by each of the aforementioned steps at service level.

Table 2. Refactoring times to refactor the whole system.

Task	Time elapsed
Detection step	6 h 25 m
Design step	6 h 29 m
Specification step	16 h 27m
Revision step	1 h 30 m
General inspection	0 h 45 m
Total time	*31* h *38* m

5.2 Discussion

It is also worth noting how the refactoring times increase with the size of the services. Figure 6 (right) shows the time required to refactor a service seems to have a weak correlation with the number of lines of code in the initial refactoring group ($R^2 = 0.31$), but Fig. 6 (left) implies an expected greater relationship with the final code lines ($R^2 = 0.54$).

This means that refactoring groups, even if they are large, can take a short time if they can be reduced to a few lines of code. Figure 5 also shows that, in general, refactoring times for services with few lines of code are consistently low: less than 30 or 40 min for almost all groups with 1000 lines of code or less, but it

Table 3. Refactoring times at service level.

Task	Mean	Std. Dev
Detection step	5 m 52 s	5 m 22 s
Design step	7 m 22 s	11 m 27 s
Specification step	19 m 0 s	20 m 11 s
Total time	*32* m *20* s	*33* m *01* s

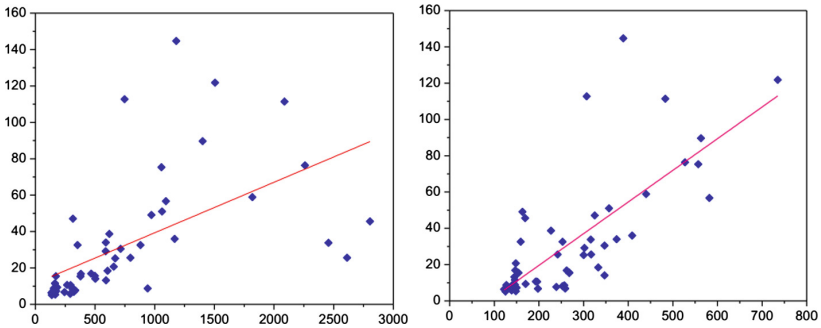

Fig. 6. Left: Refactoring time (Y) according to Line of Code of the original services (X) – Right: Refactoring time (Y) according to Lines of code of the final service (X).

is much more difficult predict how long it will take to refactor a group with much more lines of code. Some of them are especially problematic and may require a much larger redesign to accommodate all of their original services.

To sum up, by using our methodology we simplified the 211 original services with 254 operations with 38,912 lines of code to a specification that defines 52 refactored services that contain a total of 140 operations with 14,201 lines of code; in total, it took 31 hours and 38 min of useful work for the implementation of the refactoring methodology. In other words, the results shows a reduction of 75.35% in the number of services, 44.88% in the number of operations, and in 63.5% in the lines of code.

6 Conclusions

In this work, we proposed a novel methodology to tackle the refactoring of the description of web services belonging to real legacy systems. To evaluate the methodology and the required effort were defined metrics that were monitored throughout the methodology and which may be relevant when comparing this refactoring with other systems. Furthermore, these metrics are intended to act as reference when comparing the manually refactoring of the same system with the assistance of automation tools.

The relationship between the decrease in services, operations, data types, messages, portTypes and lines of code was consistent with what would be expected of the relationships between the elements of WSDL and the patterns of our case-study. Clusters of original services in services refactorings show a logarithmic distribution described as the number of services refactored according to the size of the groups (the larger size, the fewer occurrences). This finding confirms our hypothesis that as refactoring progresses, it is increasingly difficult to recognize large refactoring groups. The time required to refactor each service is consistently less than 30 or 40 min for groups with 1,000 lines of code or less, but as the size increases the unpredictability of the effort required also increases. As future work, we plan to compare manual refactoring with a tool-support refactoring. Our goal is to compare not only refactoring effort and space reduction but also performance of the tool-support approach in terms of errors made.

We are also planning to validate the approach with more datasets coming from SOA contexts. Particularly, we will use an additional dataset from another real migration from COBOL to SOA [18]. Further increasing process automation will be also in our future agenda.

References

1. Schekkerman, J.: What you all need to know about services orientation! structuring the enterprise around services. The Differences between Hype, Hope and Reality. Institute for Enterprise Architecture Developments (2006). Accessed 3 July 2017
2. Lojka, T., Bundzel, M., Zolotová, I.: Service-oriented Architecture and Cloud Manufacturing. Acta Polytech. Hung. **13**(6) (2016)
3. Fowler, M.: Refactoring: Improving the Design of Existing Code. Pearson Education India (1999)
4. Pedrinaci, C., Domingue, J.: Toward the next wave of services: linked services for the web of data. J. UCS **16**(13), 1694–1719 (2010)
5. Rodríguez, G., Soria, Á., Teyseyre, A., Berdun, L., Campo, M.: Unsupervised learning for detecting refactoring opportunities in service-oriented applications. In: Hartmann, S., Ma, H. (eds.) DEXA 2016. LNCS, vol. 9828, pp. 335–342. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-44406-2_27
6. Garcia-Penalvo, F.J., Alier, M., Lytras, M.D.: Some reflections about service oriented architectures, cloud computing applications, services and interoperability. J. Univ. Comput. Sci. **18**(11), 1405–1409 (2012)
7. Erickson, J., Siau, K.: Web services, service-oriented computing, and service-oriented architecture: separating hype from reality. In: Principle Advancements in Database Management Technologies: New Applications and Frameworks, p. 176 (2009)
8. Rodriguez, G., Teyseyre, A., Soria, A., Berdun, L.: A visualization tool to detect refactoring opportunities in SOA applications. In: XLIII Latin American Conference on Informatics. IEEE (2017)
9. Ezenwoke, A., Misra, S., Adigun, M.O.: An approach for e-commerce on-demand service-oriented product line development. Acta Polytech. Hung. **10**(2), 69–87 (2013)
10. Garriga, M., et al.: A structural-semantic web service selection approach to improve retrievability of web services. Inf. Syst. Front. **20**(6), 1319–1344 (2018)

11. Which style of WSDL should I use? 24 May 2005. https://www.ibm.com/developerworks/library/ws-whichwsdl/. Accessed 3 July 2017
12. Rodriguez, J.M., Crasso, M., Mateos, C., Zunino, A.: Best practices for describing, consuming, and discovering web services: a comprehensive toolset. Softw. Pract. Exp. **43**(6), 613–639 (2013)
13. Mateos, C., Rodriguez, J., Zunino, A.: A tool to improve code-first web services discoverability through text mining techniques. Softw. Pract. Exp. **45**(7), 925–948 (2015)
14. Webster, D., Townend, P., Xu, J.: Interface refactoring in performance-constrained web services. In: 2012 IEEE 15th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), pp. 111–118. IEEE (2012)
15. Fritzsch, J., Bogner, J., Zimmermann, A., Wagner, S.: From monolith to microservices: a classification of refactoring approaches. In: Bruel, J.-M., Mazzara, M., Meyer, B. (eds.) DEVOPS 2018. LNCS, vol. 11350, pp. 128–141. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-06019-0_10
16. Borges, M., Barros, E., Maia, P.H.: Cloud restriction solver: a refactoring-based approach to migrate applications to the cloud. Inf. Softw. Technol. **95**, 346–365 (2018)
17. Bogner, J., Wagner, S., Zimmermann, A.: Using architectural modifiability tactics to examine evolution qualities of Service-and Microservice-Based Systems. SICS Softw.-Intensive Cyber-Phys. Syst., 1–9 (2019)
18. Mateos, C., Zunino, A., Flores, A., Misra, S.: COBOL systems migration to SOA: assessing antipatterns and complexity. Inf. Technol. Control (2019, in press)