Automated Software Engineering (2019) 26:275–312
https://doi.org/10.1007/s10515-019-00256-4

Improving web service interfaces modularity using multi-objective optimization

Sabrine Boukharata[1] · Ali Ouni[1] · Marouane Kessentini[2] · Salah Bouktif[3] · Hanzhang Wang[4]

Received: 5 April 2018 / Accepted: 25 April 2019 / Published online: 15 May 2019

Abstract
Service interface is a critical component in a service-oriented architecture (SOA). As first-class design artifact, a service interface should be properly designed to provide best practice of third-party reuse. However, a very common bad service design practice in existing SOAs is to place semantically unrelated operations implementing several abstractions in a single interface. Poorly designed service interfaces can have a negative effect on all client applications that use these services. Indeed, services with such poor interface structure tend to be difficult to comprehend, maintain and reuse in business processes, leading to unsuccessful services. Necessarily, then, service designers should "refactor", i.e., restructure, their service interface into smaller, more cohesive interfaces, each representing a specific abstraction. To address this problem, we introduce a novel approach, namely *WSIRem*, to support service's developers in improving the modularization of their service interfaces. *WSIRem* is based on a multi-objective search-based optimization approach to find the appropriate modularization of a service interface into smaller, more cohesive and loosely coupled interfaces, each implementing a distinct abstraction. *WSIRem* has been empirically evaluated on a benchmark of 22 real-world Web services provided by Amazon and Yahoo. Results show that the automatically identified interfaces improved the services interface structure. Qualitative evaluation of *WSIRem* with developers showed the performance of *WSIRem* in terms of understandability, where the new *WSIRem* interfaces were recognized as 'relevant' from developers point of view with more than 73% of precision and 77% of recall. Overall, the obtained results show that *WSIRem* outperforms state-of-the-art approaches relying on traditional partitioning techniques.

Keywords Web service · Web service interface · Service design · Modularity · SOA · Searchbased software engineering

✉ Ali Ouni
ali.ouni@etsmtl.ca

Extended author information available on the last page of the article

1 Introduction

The service-oriented computing (SOC) is becoming the leading edge of modern software engineering and it is increasingly adopted in the software industry. Services are, in general, provided by third-parties who only expose services interfaces to the outer world. These services are commonly treated as "black-boxes" with abstract interfaces constituting the only visible part of the system. Service interfaces are the main source of interactions with the user to adopt the services in real-world applications. It is widely recognized that well-designed service interface can accelerate project schedules and make the service oriented architecture (SOA) solution more responsive to business needs. Indeed, service interfaces with well-defined abstractions and cohesive operations are easy to comprehend and reuse in business processes (Perepletchikov et al. 2007a).

In the context of implementing SOA solutions, the structure of a service interface is critical. Like any other software system, Web services need to be changed and updated frequently to add new functionalities in response to client needs (Romano and Pinzger 2012). For example, a hotel management Web service must, over time, offer new features, become more reliable and respond faster. However, these continuous changes may increase the complexity of the service interfaces and even taking them away from their original design (Romano and Pinzger 2012). This may in turn introduce side effects known as *antipatterns*—symptoms of bad design and implementation practices—that often lead to different usability, understandability and maintainability problems as well as runtime errors (Ouni et al. 2017b, a). Common modularity-related antipatterns include the multi-service, nano-service, chatty and data service interfaces (Ouni et al. 2017b; Palma et al. 2014). The presence these two particular antipatterns would imply often the co-occurrence of other related antipatterns such as the data service and the chatty service interface. Thus, poorly designed service interfaces may have a negative effect on all applications that use the services.

Service providers continuously try to improve the quality of their service interface descriptions to ensure best practice of third-party reuse (Athanasopoulos et al. 2015; Ouni et al. 2016). Although this observation might sound obvious, developers tend to take little care of their service WSDL descriptions as several researchers have pointed out (Athanasopoulos et al. 2015; Crasso et al. 2010; Ouni et al. 2017b; Palma et al. 2014). Most of service descriptions are designed in only one single interface regrouping all the operations together. Indeed, a common bad design practice, i.e., antipattern, that often appear in real-world Web services is to group together a large number of semantically unrelated operations in a single interface (Athanasopoulos et al. 2015; Crasso et al. 2010; Ouni et al. 2016, 2018). Such interfaces tend to cover several distinct core abstractions and processes, leading to many operations associated with each abstraction. This inappropriate service interface modularity will result in poorly designed applications that tend to be hard to use, discover, implement, maintain and evolve (Haesen et al. 2008; Perepletchikov et al. 2010; Romano and Pinzger 2014; Ouni et al. 2018).

Best practice for service design suggests that services should expose their operations in a modular way, where each module, i.e., interface, defines operations that handle one abstraction at a time (Perepletchikov et al. 2008, 2010). Service interfaces will

consequently exhibit low coupling and high cohesion (Dudney et al. 2003). Low coupling means that a service interface, i.e., port-type, is independent to other interfaces, allowing an effective reuse. Cohesion refers to how strongly related the operations themselves are. High cohesion means that the service operations are related as they operate on the same, underlying core abstraction.

Web service bad design practices and antipatterns have been recently studied, and different approaches have been proposed to identify Web service interfaces suffering from bad design practices (Král and Zemlicka 2009; Ouni et al. 2015a, 2016, 2017a, b; Palma et al. 2014). However, fixing these antipatterns is still unexplored and it is a manual, complex, time-consuming and error-prone task. Indeed, designing a service interface with the right number of interfaces, i.e., port types, and an appropriate assignment of operations to port types is a non-trivial task. This can due to several reasons, especially, when developers are under pressure and stress to meet several release deadlines (Athanasopoulos et al. 2015; Masoud and Jalili 2014; Mkaouer et al. 2015), or also due to poor migration processes when moving from legacy code to Web service architectures (Mateos et al. 2015a). In fact, the number of operation combinations to explore is exponentially high, leading to a large and complex search space.

To address this problem, we introduce a novel approach for Web service interface remodularization, namely *WSIRem* in order to support service's developers in improving the design structure of their service interfaces. *WSIRem* seeks at finding the suitable service interface modularization by partitioning large interfaces exposing a large number of semantically unrelated operations into smaller, loosely coupled and more cohesive interfaces. The goal is to support the service's developers by providing a wide range of possible remodularization solutions and making the Web services interface easier to use.

WSIRem formulates service interface remodularization problem as a multi-objective optimization problem to find the optimal modularization of operations. To this end, *WSIRem* adopts the non-dominated sorting genetic algorithm (NSGA-II) (Deb et al. 2002), to find the best trade-off between the following objectives (*i*) minimizing coupling, (*ii*) maximizing cohesion, (*iii*) minimize the interfaces modifications. Coupling refers to the number of dependencies among all service interfaces, whereas cohesion refers to how strongly related are the operations within each interface. Typically, low coupling is preferred as this indicates that a group of operations covers a single abstraction of a Web service. In contradiction to coupling, the cohesion within an interface should be maximized to ensure that it does not contain operations that are not part of its underlying abstraction. At the same time, the number of required modifications to the original interface should be minimized to reduce the service remodularization effort while avoiding potential design disruption.

To evaluate the effectiveness of *WSIRem*, we conducted two empirical studies on a benchmark of 22 real-world services, provided by Amazon and Yahoo. The first empirical study aims to evaluate how well *WSIRem* can improve the design quality of service interfaces comparing to recent state-of-the-art approaches (Athanasopoulos et al. 2015) and (Ouni et al. 2016) which use traditional clustering techniques to improve the modularization of service interfaces. The second empirical study aimed at investigating whether the provided remodularization solutions by *WSIRem* are 'useful' and relevant from a developer's point of view. We define a useful solution as a solution

that a developer would consider as a meaningful and acceptable and is willing to apply it as it is to the considered interface. This is one of the main contributions of this paper, as we involved 14 expert developers to assess the usefulness of automated solutions for service interface remodularization.

The results show that the interface remodularization solutions proposed by *WSIRem* are able to: (*i*) significantly increase the cohesion of the refactored interfaces while maintaining an acceptable coupling; (*ii*) achieve results similar to manually performed remodularizations by developers at 73% of precision and 77% of recall, on average; and (*iii*) recommend useful remodularization solutions for developers to improve service understandability and reuse.

The rest of this paper is organized as follows. Section 2 provides the necessary background along with a motivating example. Section 3 describes the formulation of the service interface modularization problem. Our approach, *WSIRem*, is discussed in Sect. 4. Our empirical study and its results are presented in Sect. 5. Threats to validity are analyzed in Sect. 6, while the related work is discussed in Sect. 7. Finally, our conclusions and future work are stated in Sect. 8.

2 Background and motivation

2.1 Background

A Web Service is defined according to the W3C[1] (World Wide Web Consortium), as "a software application identified by a URI, whose interfaces and bindings are capable of being defined, described, and discovered as XML artefacts. Its interface is described as a WSDL (Web service Description Language) document that contains structured information about the Web service's location, its offered operations, the input/output parameters, etc."

A Web service interface corresponds to a WSDL port type, which is the most important WSDL element. A Web service has at least one interface. This WSDL element describes a Web service, the operations that can be performed, and the messages that are involved. It can be compared to a function library (or a module or a class) in a traditional programming language.

Modularity. Service interface *modularity* can be defined as the degree to which the operations of a service are related to the same abstraction and well partitioned into cohesive interfaces. A good *modularization* of a design leads to a service which is easier to use, design, develop, test, maintain, and evolve. The importance of design modularity was best articulated by Card and Glass (1990): "*perhaps the most widely accepted quality objective for design is modularity*". Although modularity tends to be a subjective concept, measuring the degree of modularization of a software design can be achieved through two quality measures: cohesion and coupling (Budgen 1999).

Service abstraction. Service abstraction is a design principle that is applied within the service-orientation design paradigm so that the information published in a service contract is limited to what is required to effectively utilize the service. We refer to

[1] http://www.w3.org/TR/wsdl20/.

abstractions as reflected in application programming interfaces (APIs), through which the state of a business transaction can be captured and flexibly manipulated. For example, method APIs such as createOrder, checkInventory, and makePayment in the same service can support the following distinct abstractions Order, Payment, and Inventory, respectively (Dudney et al. 2003; Newcomer 2002).

Refactoring. Software refactoring is defined by Fowler (1999) as "*the process of changing the internal structure of a software to improve its quality without altering the external behavior*". Refactoring is recognized as an essential practice to improve software quality. Dudney et al. (2003) have defined an initial catalog of refactoring operations for Web services including the following operations:

- *Interface Partitioning*: focuses on deciding what the core abstraction of the service should be, and moving the methods that don't fit that abstraction to some other service(s) (Dudney et al. 2003).
- *Interface Consolidation*: focuses on consolidating the separate but related services into single, more cohesive services (Dudney et al. 2003).
- *Bridging Schemas or Transforms*: this refactoring introduces an adaptor component within the handling of electronic document exchanges with multiple parties, to adapt differences between business document formats (Dudney et al. 2003).
- *Web Service Business Delegate*: this refactoring adds an intervening Java component between a Web service endpoint receiver (usually a servlet) and an underlying business logic component, such as a Session Bean, to remove Web service-specific handling code from the business logic component and/ or to remove Web service-specific data types and coupling from the component to facilitate easier reuse (Dudney et al. 2003).

This paper focuses on automating the *Interface Partitioning* refactoring to improve service modularization.

2.2 Motivating example

Modularity is considered to be one of the most important software design quality aspects (Card and Glass 1990). To illustrate some of the problems related to service interface modularity, let us consider a real-world case, the Amazon Elastic Compute Cloud service (EC2)[2] provided by Amazon. The major interface of EC2 exposes a quite large number of operations (87 operations) offering a variety of business abstractions including managing images, volumes, security, instances, and snapshots, grouped in a single interface, `AmazonEC2PortType`.

Consequently, for a client who wants to manage images through EC2, s/he should study the specification of the AmazonEC2PortType interface (which consists of 4,261 lines of WSDL and XML schema definitions) and a 812-pages API documentation guide.[3] However, only few operations can be useful for managing images (*CreateImage()*, *RegisterImage()*, *DeregisterImage()*, *DescribeImages()*, *ModifyImageAttribute()*, *ResetImageAttribute()*, and *DescribeImageAttribute()*).

[2] http://s3.amazonaws.com/ec2-downloads/2009-10-31.ec2.wsdl. Accessed on February 15th, 2019.

[3] https://aws.amazon.com/documentation/ec2/.

Such a poor interface madularization might lead to a service which is not easy to reuse in business processes, hard to maintain and extend because of its large number of non-cohesive operations. Above that, such services are at the risk of suffering from a high response time or even unavailable to end users because they are overloaded (Dudney et al. 2003). On the other hand, service interfaces inevitably change over time. Thus, when a single interface implements a lot of operations, many clients will be affected by the change. In some cases, this means that client invocation code has to change to accommodate the new interface changes (Dudney et al. 2003). From a provider side, well-designed interfaces are easy to maintain and evolve by developers. Furthermore, when several developers work on one single large interface implementation, there could result into multiple issues of developer contention for changes to server-side implementation artifacts (Dudney et al. 2003). Thus, good service design might have multiple benefits on the development process, including an improving the development organization, improving developers coding practices and productivity.

Consequently, the `AmazonEC2PortType` interface design can be significantly improved with a more cohesive modularization to provide better reusability, understandability and maintainability. The provided operations can be exposed within separate interfaces, for instance, an interface for managing images, another for volumes, another for security, etc. This improved moudularization would simplify the comprehension of the functionalities that the client actually needs.

As a consequence, developers are encouraged to refactor their service interfaces to provide best practice of third party reuse. Motivated by the above mentioned issues, this work aims at providing an automated approach to support developers in improving their services interface modularity.

3 Problem formulation

3.1 Formulating service interface modularization

We define a Web service interface Modularization $\mathcal{M}$ as a decomposition of the set of operations O into a set of service interfaces *si*, where each interface represents a WSDL port type *PT*, i.e., a container of operations. We define the interface size, *size(si)*, by the number of its operations.

Consider a Web service with set of operations $O = \{op_1, op_2, \ldots, op_n\}$ where n is the number of operations in the service. The set of possible modules, i.e., interfaces, is represented by $\mathcal{M} = \{si_1, si_2, \ldots, si_m\}$ where m is the number of service interfaces, and each interface has its unique number $1, 2, \ldots, m$. A possible modularization solution for this problem is defined by the decision variables $X = \{x_1, x_2, \ldots, x_n\}$, where $x_i = si$ indicates that op_i belongs to interface si.

Figure 1 shows a service interface modularization fragment example extracted from the Amazon EC2 Web service. A simple solution of the following operations `RunInstances()`, `RegisterImage()`, `DescribeImages()`, `DescribeVpcs()`, `StartInstances()`, `CreateImage()`, `CreateVpc()`, `ModifyImageAttribute()`, `StopInstances()`, `DeleteVpc()`, `TerminateInstances()` is encoded by $X = \{2, 1, 1, 3, 2, 1, 3, 1, 2, 3, 1\}$, which denotes

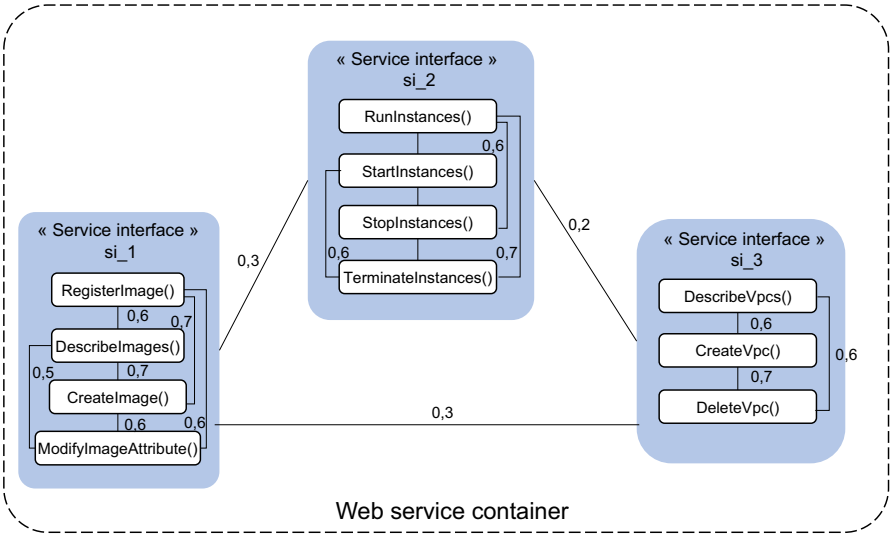


Fig. 1 An example of Web service interface modularization from the Amazon EC2 Web service (fragment)

a modularization of the eleven Amazon EC2 Web service operations into three service interfaces. The operations `RegisterImage()`, `DescribeImages()`, `CreateImage()` and `ModifyImageAttribute()` are in service interface si_1, `RunInstances()`, `StartInstances()`, `StopInstances()` and *TerminateInstances()* are in si_2, and finally `DescribeVpcs()`, `CreateVpc()` and `DeleteVpc()` are in si_3. Moreover, different service operation dependencies exist in order to implement the required functionalities by the service. An appropriate modularization should maximize the cohesion within an interface while minimize coupling between interfaces.

Problem complexity Finding the best partitioning of operations into cohesive service interfaces is not an obvious task for developers as the number of possible partitions can be very large causing a combinatorial explosion. The search space tends to be enormous as the number of possible partitions is given by:

$$B_n = \sum_{k=0}^{n} \binom{n}{k} B_k \tag{1}$$

where B_n, counts the number of different possibilities of how a given set of n operations can be divided into interfaces. The order of the partitions, i.e., interfaces, as well as the order of the operations within an interface do not need to be considered. For instance, consider the `AmazonEC2PortType` Web service which exposes 87 operations in the version 2010 (Romano and Pinzger 2012). To find the right interface partitioning for `AmazonEC2PortType`, the number of combinations of its 87 operations, a developer need to explore $B_{87} \approx 6.39 \times 10^{98}$ possible ways to create interfaces. Due to this huge search space, an exhaustive search is unsuitable. Instead, a heuristic search

maybe efficient for this kind of combinatorial problems (Mkaouer et al. 2015; Harman et al. 2012).

3.2 Interface cohesion

Cohesion is a widely used metric in SOA to measure how strongly related are the operations of a service interface (Ouni et al. 2016; Athanasopoulos et al. 2015; Perepletchikov et al. 2010; Athanasopoulos and Zarras 2011). *WSIRem* employs three commonly used interface cohesion metrics that will drive the remodularization search process: sequential, communicational, and conceptual cohesion. Our cohesion metrics focus on interface-level relations, as service implementation is typically not provided by the service providers. Similarly, we do not consider information concerning the usage of operations by clients (i.e., operation invocations by external clients), as this information is mostly influenced by the specific scenario where the service is used, and is not always available especially for new Web services.

Lack of sequential cohesion (LoC_{seq}) The sequential similarity S_{seq} between two operations quantifies the sequential category of cohesion (Perepletchikov et al. 2010).

Definition 1 (*Sequential similarity*) Two operations are deemed to be connected by a sequential control flow if the output from an operation is the input for the second operation, or vice versa. Formally, let $op_1, op_2 \in si$, two operations belonging to an interface si, then S_{seq} is defined as follows:

$$S_{seq}(op_1, op_2) = \frac{MS(I(op_1), O(op_2)) + MS(O(op_1), I(op_2))}{2} \tag{2}$$

where $I(op)$ and $O(op)$ refer to the input and output messages of the operation op, respectively; and $MS(I(op_1), O(op_2))$ is the function that returns the message similarity between two messages $I(op_1)$ and $O(op_2)$.

Definition 2 (*Message similarity (MS)*) Message similarity is the degree at which two messages share common elements. Two messages are similar if they have common parameters, or similar types of parameters. The MS of two messages m_1 and m_2 corresponds to the average of:

- *The number of common subtrees:* it corresponds to the sum of the orders of common bottom-up subtrees of m_1 and m_2, divided by the order of the message that results from the union of m_1 and m_2, as defined in Athanasopoulos and Zarras (2011).
- *The number of common primitive types:* it corresponds to the Jaccard similarity between m_1 and m_2, i.e., the ratio of common primitive types in m_1 and m_2, divided by the union of primitive types of m_1 and m_2.

By combining these two measures taking their average values, MS aims at capturing message similarity. The more two messages share common subtrees and primitive types, the more they are likely to be related.

Definition 3 (*Lack of sequential cohesion* (LoC_{seq})) The lack of sequential cohesion LoC_{seq} of an interface si is defined as the complement of the average S_{seq} of all pairs of operations belonging to the interface si. Formally, LoC_{seq} is defined as follows:

$$LoC_{seq}(si) = 1 - \frac{\sum_{\substack{\forall(op_i, op_j) \in si \\ op_i \neq op_j}} S_{seq}(op_i, op_j)}{\frac{|si| \times (|si|-1)}{2}} \tag{3}$$

Lack of communicational cohesion (LoC_{com}) The Communicational Similarity S_{com} between two operations quantifies the communicational category of cohesion (Perepletchikov et al. 2010).

Definition 4 (*Communicational similarity*) Two service operations are deemed to be connected by a communication similarity, if they share (or use) common parameter and return types, i.e., both operations are related by a reference to the same set of input and/or output data. Formally, let m_1 and m_2, two operations, then S_{com} is defined as follows:

$$S_{com}(op_1, op_2) = \frac{MS(I(op_1), I(op_2)) + MS(O(op_1), O(op_2))}{2} \tag{4}$$

where $I(op)$ and $O(op)$ refer to the input and output messages of the operation op, respectively; and $MS(I(op_1), I(op_2))$ is the function that returns the message similarity between two messages $I(op_1)$ and $I(op_2)$.

Definition 5 (*Lack of communicational cohesion* (LoC_{com})) The Lack of communicational cohesion of an interface si is defined as the complement of the average S_{com} of all pairs of operations belonging to the interface si (Athanasopoulos and Zarras 2011). Formally, LoC_{com} is defined as follows:

$$LoC_{com}(si) = 1 - \frac{\sum_{\substack{\forall(op_i, op_j) \in si \\ op_i \neq op_j}} S_{com}(op_i, op_j)}{\frac{|si| \times (|si|-1)}{2}} \tag{5}$$

Lack of semantic cohesion (LoC_{sem}) The Semantic Similarity LoC_{sem} between two operations quantifies the conceptual category of cohesion. We introduce a concrete refinement of the conceptual cohesion, as it is regarded as the strongest cohesion metric (Perepletchikov et al. 2008).

Definition 6 (*Semantic similarity*) The semantic similarity S_{sem} is defined as the meaningful semantic relationships between two operations, in terms of some identifiable domain level *concept*. We expand the existing definition to get more meaningful sense of the semantic meanings embodied in the operation names.

Our semantic similarity measure is based on a lexical analysis on operation signature. First, *WSIRem* parses the WSDL files and extracts all operation names, used data types, and comments. Then, the collected terms are tokenized using a camel case splitter where each name is broken down into tokens/terms based on commonly used coding standards. Thereafter, *WSIRem* removes stop words and filters out all common English words[4] (common words such as "the", "is", "at", "which", "on", …), and

[4] http://www.textfixer.com/resources/common-english-words.txt.

applies stemming on all descriptions employing the power of the Stanford's toolkit CoreNLP.[5] Stemming is a common natural language processing (NLP) technique to identify the words root, and it is essential to make words such as "playing", "player", and "play" all match to the single common root "plai". Stemming can improve the results of later NLP processes, since it reduces the number of words. *WSIRem* also removes non-text items such as numerals.

Thereafter, to capture semantics or textual similarity between two bags of words A and B extracted from two operations op_1 and op_2 respectively, we use the cosine of the angle between both vectors representing A and B in a vector space using *tf-idf* (term frequency-inverse document frequency) model. We interpret term sets as vectors in the n-dimensional vector space, where each dimension corresponds to the weight of the term (tf-idf) and thus n is the overall number of terms. Formally, the S_{sem} between op_1 and op_2 corresponds to the cosine similarity of their two weighted vectors $\mathbf{A}$ and $\mathbf{B}$ and defined as follows:

$$S_{sem}(op_1, op_2) = cosine(\mathbf{A}, \mathbf{B}) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \times \|\mathbf{B}\|} \tag{6}$$

Definition 7 (*Lack of semantic cohesion* (LoC_{sem})) The Lack of semantic cohesion LoC_{sem} of an interface si is defined as the complement of the average S_{sem} of all pairs of operations belonging to the interface si. Formally, LoC_{sem} is defined as follows:

$$LoC_{sem}(si) = 1 - \frac{\sum_{\substack{\forall(op_i, op_j) \in si \\ op_i \neq op_j}} S_{sem}(op_i, op_j)}{\frac{|si| \times (|si|-1)}{2}} \tag{7}$$

Lack of cohesion (LoC)

Definition 8 (*Lack of Cohesion* (LoC)) The Lack of Cohesion LoC metric covers all possible aspects of service interface cohesion as captured by the previously defined metrics LoC_{seq}, LoC_{com} and LoC_{sem}. Thus, it quantifies the total (overall) cohesion of a service interface. LoC of an interface si is defined as follows:

$$LoC(si) = w_{seq} * LoC_{seq}(si) + w_{com} * LoC_{com}(si) + w_{sem} * LoC_{sem}(si) \tag{8}$$

where $w_{seq} + w_{com} + w_{sem} = 1$ and their values denote the weight of each similarity measure.

3.3 Interface coupling

Although best service design practice suggests that operations in service interface should be cohesive, e.g., operate on the same set of data, however, some interactions can arise between different service interfaces. This is because, typically, operations of a service may contribute to either single business abstractions or some other semantically

[5] http://nlp.stanford.edu/software/corenlp.shtml.

meaningful concepts such as a data entity or another abstraction in the problem domain, and therefore coupling between service interfaces is sometimes unavoidable.

Definition 9 (*Coupling*) We define the *Coupling* between two service interfaces si_1 and si_2 as the average similarity between all possible pairs of operations from si_1 and si_2. Formally, the coupling, Cpl, is defined as follows:

$$Cpl(si_1, si_2) = \frac{\sum_{\forall op_i \in si_1, \forall op_j \in si_2} Sim(op_i, op_j)}{\mid si_1 \mid \times \mid si_2 \mid} \tag{9}$$

where $\mid si_1 \mid$ denotes the number of operations in the interface si_1, and $Sim(op_i, op_j)$ is defined as the weighted sum of the different operations similarity measures defined in the previous section:

$$Sim(op_i, op_j) = w_{seq} * S_{seq}(op_i, op_j) + w_{com} * S_{com}(op_i, op_j) + w_{sem} * S_{sem}(op_i, op_j) \tag{10}$$

where $w_{seq} + w_{com} + w_{sem} = 1$ and their values denote the weight of each similarity measure.

It is worth noticing that while the contributions of S_{seq} and S_{com} to coupling are intuitive, the contribution of S_{sem} is equally important. Indeed, such kind of semantic relatedness is referred to as conceptual coupling in object-oriented design (Poshyvanyk and Marcus 2006). It has been shown that the conceptual coupling captures new dimensions of coupling, which are not captured by existing coupling measures; hence it can be used to complement the existing metrics as pointed out by Poshyvanyk and Marcus (Poshyvanyk and Marcus 2006). For example, the two operations *createKeyPair* and *deleteKeyPair* from the *AmazonEC2PortType* Web service are semantically related, and it is relevant to keep them in the same interface.

4 The proposed approach: *WSIRem*

This section shows how Web service interface remodularization is formulated as a multi-objective optimization problem. We first present an overview of our approach, *WSIRem*, then we provide the details of our adaptation of NSGA-II to our problem.

4.1 *WSIRem* overview

Given a set of service operations there are many ways in which the module boundaries can be drawn leading to different possible modularizations of the service abstractions. The problem is a graph partitioning problem, which is known to be NP hard and therefore seems suited to a metaheuristic search-based approach (Mkaouer et al. 2015; Praditwong et al. 2011).

Figure 2 shows the overall architecture of the *WSIRem* approach to the Web service interface remodularization problem. *WSIRem* aims at exploring a large search space to find a set of optimal remodularization solutions, by grouping together all collections of operations that have high cohesion into separate interfaces.

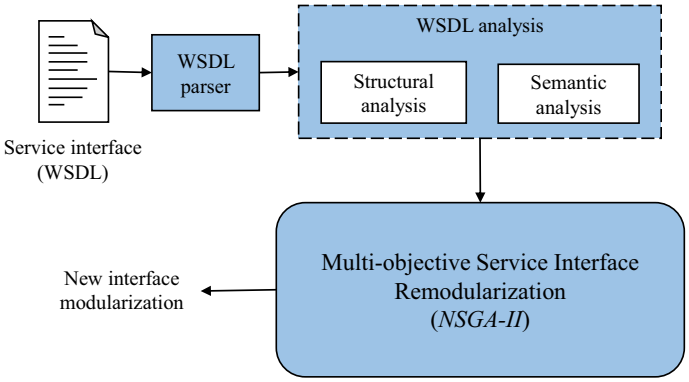


Fig. 2 Overall *WSIRem* architecture

WSIRem takes as input a Web service interface WSDL file/url to be improved. Then, *WSIRem* parses the WSDL sources by tree walking up the XML hierarchy. It then analyses the parsed WSDL through (1) a structural analysis to extract both sequential and communicational operations similarity (as described in Sect. 3.2), and (2) a semantic analysis in order to extract semantic relationships between operations (as described in Sect. 3.2). The extracted information will be used in an optimization process based on the non-dominated sorting genetic algorithm (NSGA-II) (Deb et al. 2002) to generate remodularization solutions. An optimal modularization solution should find the best trade-off between the following objectives (*i*) maximizing cohesion, (*ii*) minimizing coupling, (*iii*) minimizing the modification degree.

As output, the result of *WSIRem* should be a set of interfaces, one for each distinct abstraction, and each one containing the complete set of operations that operates on that abstraction.

To manipulate instances of this kind, *WSIRem* start by (*i*) creating a set of new empty interfaces, and (*ii*) assigning each operation to a unique interface. A modularization solution should assign each operation to exactly one interface, and have no empty interfaces. Then, *WSIRem* uses NSGA-II in order to find the best modularization solution that provide the best trade-off between our four objective functions.

4.2 NSGA-II overview

One of the widely used multi-objective search techniques is NSGA-II (Deb et al. 2002) that has shown good performance in solving several software design problems (Harman et al. 2012; Harman 2007; Ouni et al. 2013, 2016a, 2015b, 2017c).

A high-level view of NSGA-II is depicted in Algorithm 1. NSGA-II takes as input a WSDL url or file, and recommends as output a set of modularization solutions. The algorithm starts by randomly creating an initial population P_0 of individuals encoded using a specific representation (line 1). Then, a child population Q_0 is generated from the population of parents P_0 (line 2) using genetic operators (crossover and mutation). Both populations are merged into an initial population R_0 of size N (line 5). *Fast-*

non-dominated-sort (Deb et al. 2002) is the technique used by NSGA-II to classify individual solutions into different dominance levels (line 6). Indeed, the concept of non-dominance consists of comparing each solution x with every other solution in the population until it is dominated (or not) by one of them. According to Pareto optimality: "A solution x_1 is said to dominate another solution x_2, if x_1 is no worse than x_2 in all objectives and x_1 is strictly better than x_2 in at least one objective". Formally, if we consider a set of objectives f_i, $i \in 1..n$, to maximize, a solution x_1 dominates x_2:

$$\textit{iff } \forall i,\ f_i(x_2) \leqslant f_i(x_1) \text{ and } \exists j \mid f_j(x_2) < f_j(x_1)$$

The whole population that contains N individuals (solutions) is sorted using the dominance principle into several fronts (line 6). Solutions on the first Pareto-front F_0 get assigned dominance level of 0. Then, after taking these solutions out, *fast-non-dominated-sort* calculates the Pareto-front F_1 of the remaining population; solutions on this second front get assigned dominance level of 1, and so on. The dominance level becomes the basis of selection of individual solutions for the next generation. Fronts are added successively until the parent population P_{t+1} is filled with N solutions (line 8). When NSGA-II has to cut off a front F_i and select a subset of individual solutions with the same dominance level, it relies on the crowding distance (Deb et al. 2002) to make the selection (line 9). The crowding distance mechanism is used to promote diversity within the population by sorting the population according to each objective function value in ascending order of magnitude. The crowding distance is based on the assumption that solutions within a cubic have the same crowding distance, which has no contribution to the convergence of the algorithm (Deb et al. 2002). Thereafter, the front F_i to be split, is sorted in descending order (line 13), and the first (N- $|P_{t+1}|$) elements of F_i are chosen (line 14). Then, a new population Q_{t+1} is created using selection, crossover and mutation (line 15). This process will be repeated until reaching the last iteration according to stop criteria (line 4). The stop criteria is generally set out based on reaching a specif number of iterations, fitness evaluations or fitness values. For more details about NSGA-II, inetrested reasers are invited to refer to Deb et al. (2002).

4.3 NSGA-II adaptation

To adapt a search algorithm to a specific problem, the following elements should be defined: (*i*) solution representation and the generation of initial population, (*ii*) fitness function to evaluate candidate solutions according to each objective, (*iii*) change operators to generate new individuals using genetic operators (crossover and mutation). In the following we describe these elements.

4.3.1 Solution representation

A candidate solution to the problem is a service modularization, i.e., a set of interfaces, each exposing a set of cohesive operations. A valid solution assigns each service operation to exactly one interface, and has no empty interfaces. To this end, we adopt

Procedure 1 High level pseudo code for NSGA-II

```
Create an initial population P_0
Create an offspring population Q_0
t = 0
while stopping criteria not reached do
    R_t = P_t ∪ Q_t
    F = fast-non-dominated-sort(R_t)
    P_{t+1} = ∅ and i = 1
    while | P_{t+1} | + | F_i | ⩽ N do
        Apply crowding-distance-assignment(F_i)
        P_{t+1} = P_{t+1} ∪ F_i
        i = i + 1
    end while
    Sort(F_i, ≺ n)
    P_{t+1} = P_{t+1} ∪ F_i[N− | P_{t+1} |]
    Q_{t+1} = create-new-pop(P_{t+1})
    t = t+1
end while
```

Fig. 3 Solution encoding

op_	1	2	3	4	5	6	7
si_	1	2	3	1	3	2	2

the *label-based integer* encoding (Hruschka et al. 2009) where a candidate solution is represented as an *integer array* of n positions, where n is the number of operations exposed in a service. Each position corresponds to a specific operation. The integer values in the array represent the interface to which the operations belong. For instance, the modularization example in Fig. 1 is encoded as shown in Fig. 3 where the operations op 1 and 4 belong to the same service interface si labeled with 1; operations 2, 6, and 7 belong to the interface 2, and operations 3 and 5 belong to the interface 3.

The initial population is completely random where a max number of interfaces n is fixed, then each operation in the service is randomly assigned to a unique interface. Furthermore, we define the parameter $minSize$ as minimum number of operations per interface. Indeed, we fixed $minSize$ at 2, as typically a core business abstraction requires at least two operations.

4.3.2 Objective functions

The quality of each candidate modularization solution is defined by a fitness function that evaluates multiple objective and constraint dimensions. Each objective dimension refers to a specific value that should be either minimized or maximized for a solution to be considered "better" than another solution. In our approach, we optimize the following four objectives:

1. **Cohesion** The cohesion objective function is a measure of the overall cohesion of a candidate interface modularization. This objective function corresponds to the average cohesion score of each interface in a Modularization $\mathcal{M}$ and is computed as follows:

$$Cohesion(\mathcal{M}) = 1 - \frac{\sum_{\forall si \in \mathcal{M}} LoC(si)}{\mid \mathcal{M} \mid} \tag{11}$$

where $LoC(si_i)$ denotes the total interface lack of cohesion given by Eq. 10, and $\mid \mathcal{M} \mid$ is the total number of interfaces in the modularization $\mathcal{M}$.

2. **Coupling** The coupling objective function measures the overall coupling among interfaces in a modularization $\mathcal{M}$. This objective function corresponds to the average coupling score between all possible pairs interfaces in a the modularization $\mathcal{M}$ in a service and is calculated as follows:

$$Coupling(\mathcal{M}) = \frac{\sum_{\substack{\forall (si_i, si_j) \in \mathcal{M} \\ si_i \neq si_j}} Cpl(si_i, si_j)}{\frac{|\mathcal{M}| \times (|\mathcal{M}|-1)}{2}} \tag{12}$$

where $Cpl(si_i, si_j)$ denotes the coupling between the interfaces si_i and si_j given by Eq. 9, and $\mid \mathcal{M} \mid$ is the total number of interfaces in the modularization $\mathcal{M}$. Typically, coupling among service interfaces should be minimized as this indicates that each interface covers separate functionality aspects.

3. **Interface modification degree (IMD)** This objective function refers to the total number of modifications to be applied to the original interface to obtain the new modularization $\mathcal{M}$. We measure the design deviation as the number of operations and interfaces that would need to be moved, extracted or merged, according to the MoJoFM metric (Wen and Tzerpos 2004; Paixao et al. 2018). Given two different interface modularizations A and B of the same Web service, MoJoFM(A, B) accounts for the proportional number of *Move* and *Join* operations that are necessary to transform A in B, such as presented in Eq. 4.

$$\text{MoJoFM(A, B)} = \left(1 - \frac{mno(A, B)}{max(mno(\forall A, B))}\right) \times 100\% \tag{13}$$

In this paper, a Move operation represents moving a service operation from its original interface to another interface in the Web service, while the Join operation represents the merge of two interfaces. The distance between A and B is the minimal number of operations that transform A in B, computed by *mno(A, B)*, and this value is normalized by the maximum distance between any possible modularization partitioning of the Web service (denoted by $\forall A$) and B (Wen and Tzerpos 2004). The higher the value of MoJoFM, the lower is the design deviation degree. Our modification degree objective function is calculated based on MoJoFM and reflects how much the original interface needs to be changed to adopt the modular optimized solution. Given the original service interface A, and a candidate solution B, the design deviation of the solution B is calculated as follows:

$$\text{IMD(A, B)} = 100\% - \text{MoJoFM(A, B)} \tag{14}$$

This objective function is be minimized as high levels of design deviation might take away the service from its original architecture and design. Moreover, minimizing the modifications degree would be interesting to ensure that an improvement

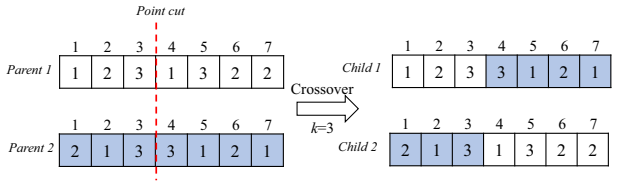


Fig. 4 Crossover operator

in terms of cohesion and/or coupling should not be accepted unless it is higher than the applied modification. That is, with the IMD objective, if two solutions have the same cohesion and coupling quality, then the solution with the smallest modification degree will be selected. On the other hand, a high interface modification degree might also lead to several clients affected by the changes which undermine the industrial uptake and popularity of the Web service. Thus, our goal is to improve as much as possible the design quality while reducing the design modification degree.

One can notice that these objective functions are conflicting by nature making service interface remodularization more challenging to find the best balance between coupling and cohesion. On the other hand, increasing the interface modification degree (IMD) might result in a large number of interfaces, leading to several scattered core abstractions. This makes service interface modularization a non-trivial decision-making task for developers.

4.3.3 Change operators

Population-based search algorithms deploy crossover and mutation operators to improve the fitness of the solutions in the population in each iteration. Change operators such as crossover and mutation aim to drive the search towards near-optimal solutions, i.e., remodularizations.

The *crossover* operator is responsible for creating new solutions based on already existing ones, e.g., re-combining solutions into ones. In our adaptation, we use a single, random cut-point crossover to construct offspring solutions. It starts by selecting and splitting at random two-parent solutions. Then crossover creates two child solutions by putting, for the first child, the first part of the first parent with the second part of the second parent, and vice versa for the second child. An example of crossover is depicted in Fig. 4.

The *mutation* operator is used to introduce slight random changes into candidate solutions. This operator guides the algorithm into areas of the search space that would not be reachable through recombination alone and avoids the convergence of the population towards a few elite solutions. With Web service interface remodularization, we use a mutation operator that picks at random one or more positions from their integer array and replaces them by other ones randomly as shown is Fig. 5.

Note that, to be valid, crossover and mutation operators should ensure that (*i*) each operation is assigned to a unique interface, and (*ii*) each interface should contain more than one operation ($minSize = 2$). The first constrait is assured by our solution

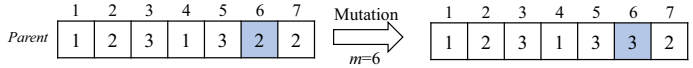


Fig. 5 Mutation operator

encoding where each operation has only one index in the vector encoding. The second constrait is checked by a repair function function to assure that each interface has at least two operations after applying the crossover and mutation as described in Procedure 2. For each interface si in the service S that has a number of operations less than $minSize$ (counted by the function $countOp()$ in line 3), a random operation from a different interface (lines 5–7) will be selected and assigned to the current interface si (line 8). The current solution S will be updated accordingly (line 10–11) and the process will be repeated until there is no remaining interfaces si_i with a number of operations less than $minSize$. In the situation where the number of interfaces is high after the change operator (i.e., the number of interfaces is higher than 50% of the number of operations $|S_O| > \frac{|S|}{2}$), the solution S will be discarded and the change operator will be re-applied (lines 13–14).

Procedure 2 Repair function

Input: Solution $S = \{si_1, si_2, \ldots, si_n\}$, $S_O = \{op_1, op_2, \ldots, op_m\}$, $minSize = 2$
Output: Repaired solution S_r
1: $checked = false$
2: **for all** $si_i \in S$ **do**
3: **while** $countOp(si_i) < minSize$ **do**
4: $checked = false$
5: **repeat**
6: $op = getRandomOpIndex(S_O)$
7: **until** $getSI(op) \neq si_i$
8: $setIndex(op, si_i)$
9: **end while**
10: $update(S, si_i)$
11: $checked = true$
12: **end for**
13: **if** $checked = false$ **then**
14: $mutation(S)$
15: **end if**

5 Evaluation

In this section, we present the results of our evaluation for the proposed approach. The purpose of this study is to investigate how well our *WSIRem* approach provides modularization solutions and compare it with available state-of-the-art approaches by Athanasopoulos et al. (2015), and Ouni et al. (2016).

The experimental materials used in our study is publicly available in a comprehensive replication package.[6]

To the best of our knowlege, Athanasopoulos et al. (2015) and Ouni et al. (2016) are the only existing techniques that have attempted to automate the service interface remodularization problem. In Athanasopoulos et al. (2015), the authors proposed a cohesion-based approach that iteratively split a service interface using a greedy algorithm without considering the coupling between the generated interfaces. In the rest of the paper we refer by *Greedy* to denote the approach proposed in Athanasopoulos et al. (2015). In Ouni et al. (2016), the authors have proposed an approach called *SIM* based on graph partitioning to find related operations that are likely to define new interfaces. However, both approaches are based on traditional algorithms greedy search and graph partitioning for this problem. Our empirical study is performed through three different types of evaluations:

1. *Evaluation with design metrics*: we evaluate the impact of the suggested remodularizations by our approach on the interface design quality in terms of cohesion, coupling and modularity.
2. *Evaluation with interface partitioning correctness*: We compare our remodularization results with those manually performed by developers in terms of precision and recall. The goal is to see if our technique can actually identify new abstractions which were improperly embedded in the original interface.
3. *Evaluation with developers*: We asked independent developers to evaluate each of the modularizations provided by our approach in practice, and provide their opinion on the applicability of each solution using five-point Likert scale (Chisnall 1993).

For each evaluation, we (*i*) present the research questions we set out to answer, (*ii*) describe the analysis methods we used to answer these questions, and (*iii*) present and discuss the obtained results.

As multi-objective algorithms such as NSGA-II return a set of optimal solutions, we used the *Knee point* mechanism (Zhang et al. 2015; Deb et al. 2002; Harman et al. 2012; Kessentini and Ouni 2017) which is a commonly used technique to automatically select an optimal solution from the Pareto front. The *knee point* is a solution from the Pareto front where a small change in one fitness exhibits a large change in another. The *knee point* denotes interesting areas of the Pareto front that provide a high trade-off among the considered objectives.

5.1 Evaluation with design metrics

This experiment aims at assessing the design improvement that a candidate remodularization suggested by *WSIRem* will bring to service interfaces comparing to *Greedy* (Athanasopoulos et al. 2015) and *SIM* (Ouni et al. 2016). We address the following research questions:

– **RQ1** To what extent can *WSIRem* improve service interface design quality?

[6] https://github.com/ouniali/AmazonYahooBenchmark.

Table 1 Experimental benchmark overview

Service interface	Provider	ID	#operations	LoC_{seq}	LoC_{com}	LoC_{sem}
AutoScalingPortType	Amazon	I1	13	0.98	0.96	0.79
MechanicalTurkRequesterPortType	Amazon	I2	27	0.84	0.91	0.85
AmazonFPSPorttype	Amazon	I3	27	0.97	0.92	0.93
AmazonRDSv2PortType	Amazon	I4	23	0.96	0.91	0.58
AmazonVPCPortType	Amazon	I5	21	0.96	0.93	0.82
AmazonFWSInboundPortType	Amazon	I6	18	0.96	0.93	0.73
AmazonS3	Amazon	I7	16	0.97	0.89	0.75
AmazonSNSPortType	Amazon	I8	13	0.97	0.96	0.84
ElasticLoadBalancingPortType	Amazon	I9	13	0.97	0.93	0.72
MessageQueue	Amazon	I10	13	0.98	0.98	0.81
AmazonEC2PortType	Amazon	I11	87	0.98	0.97	0.93
KeywordService	Yahoo	I12	34	0.93	0.84	0.91
AdGroupService	Yahoo	I13	28	0.94	0.84	0.65
UserManagementService	Yahoo	I14	28	0.97	0.96	0.91
TargetingService	Yahoo	I15	23	0.96	0.74	0.74
AccountService	Yahoo	I16	20	0.98	0.92	0.88
AdService	Yahoo	I17	20	0.89	0.79	0.88
CompaignService	Yahoo	I18	19	0.91	0.83	0.91
BasicReportService	Yahoo	I19	12	0.99	0.91	0.92
TargetingConverterService	Yahoo	I20	12	0.80	0.84	0.53
ExcludedWordsService	Yahoo	I21	10	0.81	0.72	0.54
GeographicalDictionaryService	Yahoo	I22	10	0.99	0.79	0.65

5.1.1 Subject services

To evaluate our approach, we conducted our experiment on a benchmark of 22 real-world services provided by Amazon[7] and Yahoo.[8] We selected services that are identified as *god object Web service* antipatterns (Ouni et al. 2017b; Palma et al. 2014) with interfaces exposing at least 10 operations. We chose these Web services because their WSDL interfaces are publicly available, and they were previously studied in the literature (Fokaefs et al. 2011; Athanasopoulos et al. 2015; Ouni et al. 2016). Table 1 presents our used benchmark.

5.1.2 Analysis method

To answer **RQ1**, we assess the design improvement that a candidate remodularization suggested by *WSIRem* will achieve comparing to *Greedy* (Athanasopoulos et al. 2015), and *SIM* (Ouni et al. 2016). Historically, software engineers have conceived metric

[7] http://aws.amazon.com/.

[8] http://developer.searchmarketing.yahoo.com/docs/V6/reference/.

suites as valuable tools to estimate the quality of their software artifacts (Perepletchikov et al. 2010; Athanasopoulos and Zarras 2011; Perepletchikov et al. 2007b).

Our evaluation is based on Cohesion, Coupling, and Modularity metrics. For *Cohesion*, we use the complement of the average of three widely used lack of cohesion metrics: lack of sequential cohesion (LoC_{seq}), lack of communicational cohesion (LoC_{com}), and lack of semantic cohesion (LoC_{sem}), as given by Eq. 11. *Coupling* refers to the average coupling values between all possible pairs of interfaces (cf. Eq. 12). Finally, *Modularity* evaluates the balance between coupling and cohesion by combining them into a single measurement. The aim is to reward increased cohesion with a higher *Modularity* score and to punish increased coupling with a lower *Modularity* score. It has been proven that the higher the value of *Modularity*, the better the quality of the modularization (Stewart et al. 2006). The *Modularity* metric is computed as the average of the overall cohesion and coupling.

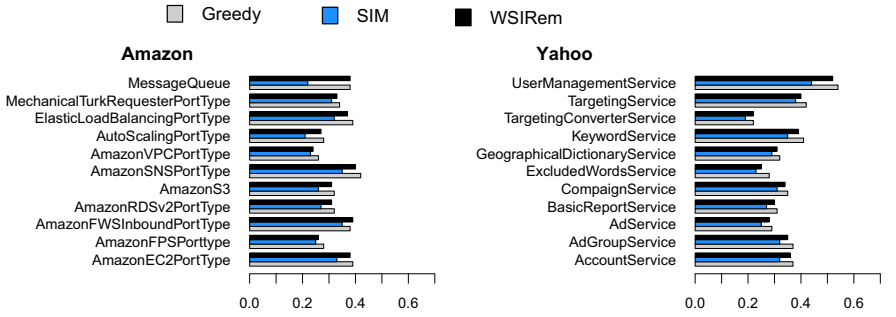
For each of these three metrics, we report the quality improvement (QI), i.e., the difference value before and after remodularization, $QI_{Cohesion}$, $QI_{Coupling}$, and $QI_{Modularity}$.

Statistical tests method To measure and analyze the obtained results, we used the Wilcoxon signed rank test in a pairwise fashion (Demšar 2006; Cohen 1988) in order to detect significant performance differences between the different approaches (*WSIRem*, *SIM*, and *Greedy*) under comparison. We used the Wilcoxon test because it is a non-parametric statistical test method that does not require the data sets to follow a normal distribution since it operates on values' ranks instead of operating on the values themselves. We set the confidence limit, α, at 0.01. Moreover, we investigate the effect size using Cliff's delta (d) (Cliff 1993). The effect size is considered:

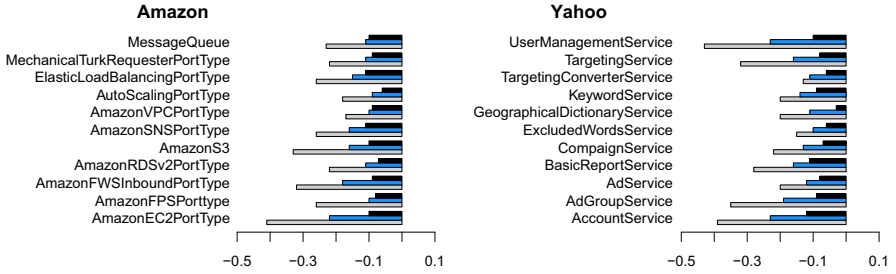
$$\begin{cases} Negligible & if \mid d \mid < 0.147 \\ Small & if \ 0.147 \leq \mid d \mid < 0.33 \\ Medium & if \ 0.33 \leq \mid d \mid < 0.474 \\ Large & if \ \mid d \mid \geq 0.474 \end{cases} \tag{15}$$

5.1.3 Results for RQ1

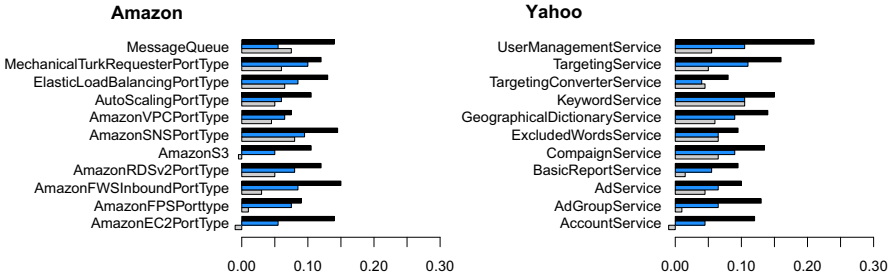
Figure 6 and Table 2 report the results achieved by *WSIRem*, *Greedy*, and *SIM* in terms of cohesion, coupling and modularity. Looking at Fig. 6a, we can see that, for almost all the Amazon and Yahoo interfaces, the cohesion is remarkably improved by the three approaches. In particular, we observed that the cohesion improvement achieved by *Greedy* was 0.35 which is slightly better than both *WSIRem* and *SIM* having 0.33 and 0.29, respectively. For Amazon services, we observed no statistically difference ($\alpha = 0.013$) between *WSIRem* and *Greedy* with a 'small' Cliffs d effect size as reported in Table 2. As for the Yahoo services, we obtained a statistically significant difference between cohesion improvements achieved by *WSIRem* and *SIM*, but with a 'negligible' effect size. On the other hand, *WSIRem* achieved better results in terms of



(a) Cohesion Improvement $QI_{Cohesion}$ achieved by WSIRem, SIM and Greedy.



(b) Coupling Improvement $QI_{Coupling}$ achieved by WSIRem, SIM and Greedy.



(c) Modularity Improvement $QI_{Modularity}$ achieved by WSIRem, SIM and Greedy.

Fig. 6 Quality improvements achieved by each of *WSIRem*, *SIM*, and *Greedy* in terms of cohesion, coupling and modularity in terms of cohesion, coupling and modularity. Higher $QI_{Cohesion}$, $QI_{Modularity}$ and $QI_{Coupling}$ scores mean better performance

cohesion than *SIM* with a 'large' effect size. Thus, overall, the obtained results show comparable cohesion improvements between both *WSIRem* and *Greedy*, while *SIM* achieves lower results.

By contrast, in terms of coupling, *WSIRem* achieved much lower coupling results (the lower the coupling is, the better is the remodularization) with an average score of -0.08 for both Amazon and Yahoo, compared to SIM and Greedy having -0.14 and -0.26 respectively. Furthermore, from Table 2, we observe that the coupling results are statistically different with a 'large' effect size. Thus, Greedy turns out to be the worst technique in terms of coupling. It is worth notice that, it is natural

Table 2 Statistical tests for the difference in terms of cohesion, coupling, and modularity of WSIRem, SIM and Greedy, for Amazon and Yahoo services

Approaches compared	Service interfaces	Cohesion		Coupling		Modularity	
		p value	Effect size (d)	p value	Effect size (d)	p value	Effect size (d)
WSIRem versus SIM	Amazon	< 0.01	0.479 large	< 0.01	0.702 large	< 0.01	0.801 large
	Yahoo	< 0.01	0.231 small	< 0.01	0.9 large	< 0.01	0.768 large
WSIRem versus greedy	Amazon	0.013	−0.148 small	< 0.01	0.909 large	< 0.01	0.892 large
	Yahoo	< 0.01	−0.14 negligible	< 0.01	0.909 large	< 0.01	0.876 large
SIM versus greedy	Amazon	< 0.01	−0.409 medium	< 0.01	0.531 large	< 0.01	0.51 large
	Yahoo	< 0.01	−0.41 medium	< 0.01	0.69 large	< 0.01	0.691 large

Wilcoxon signed rank test, and Cliff's delta (d)

that the coupling with be decreased as the original services have a single interface (thus its Coupling = 0). Consequently, any interface partitioning will result in some connections between interfaces due to the semantic similarity that is unlikely to be equals to zero, and also due to some shared (primitive) data types in messages and complex types. As reported in Fig. 6b, *Greedy* turns out to be the worst technique as the remodularization is driven only by cohesion, thus ignoring the coupling. On the other hand, *WRIRem* was able to remarkably reduce the coupling for all the services. Improvement of cohesion usually comes at the expense of increase in coupling and vice versa.

We assume that a candidate remodularization is a good design solution if the improvement of cohesion is significantly greater than the deterioration of coupling. This balance is captured by the *Modularity* metric as reported in Fig. 6c. For the 22 services, a modularity improvement was achieved by *WSIRem* with an average of 0.12, while *Greedy* and *SIM* turn out to be less effective with an average of 0.04 and 0.07, respectively. In addition, we noticed that *Greedy* produced three modularizations (for `AmazonS3`, `AmazonEC2PortType`, and `AccountService`) with deteriorated modularity due to the high coupling resulted in the new interfaces. Overall, our approach achieved significantly better results (with 'large' effect size) in terms interface modularity design improvement.

Hence, we expected an increase of cohesion (desired effect) due to the reorganization of different operations exposed in the original interface into several smaller cohesive interfaces. However, it was also expected to notice an increase in terms of coupling (side effect), since splitting an interface into several interfaces typically results in an increment of the total dependencies between these interfaces. For these reasons, coupling and cohesion should be measured together to make a proper judgment on the complexity and quality of the identified interfaces, using our *Modularity* metric.

Furthermore, Fig. 7 shows a fragment of the *WSIRem* remodularization for the Amazon EC2 interface described in Sect. 2.2. We noticed that its operations are better partitioned into several cohesive interfaces, where each interface exposes operations for a specific abstraction: instance, address, volume, security, snapshot and image managements, etc. To get more qualitative sense, RQ2 assesses the results from a developer's perspective.

5.2 Evaluation with interface partitioning correctness

For this part of the evaluation, we assess whether *WSIRem* is able to identify new business abstractions, which were improperly embedded in another interface. More precisely, a business abstraction refers to an entity, e.g., interface, having all the relevant aspects included and none of the extraneous ones such as an *OrderService*, or *ShipmentService* (Dudney et al. 2003).

Our aim is to answer the following research question:

– **RQ2** Does *WSIRem* able to identify appropriate partitioning of distinct business abstractions?

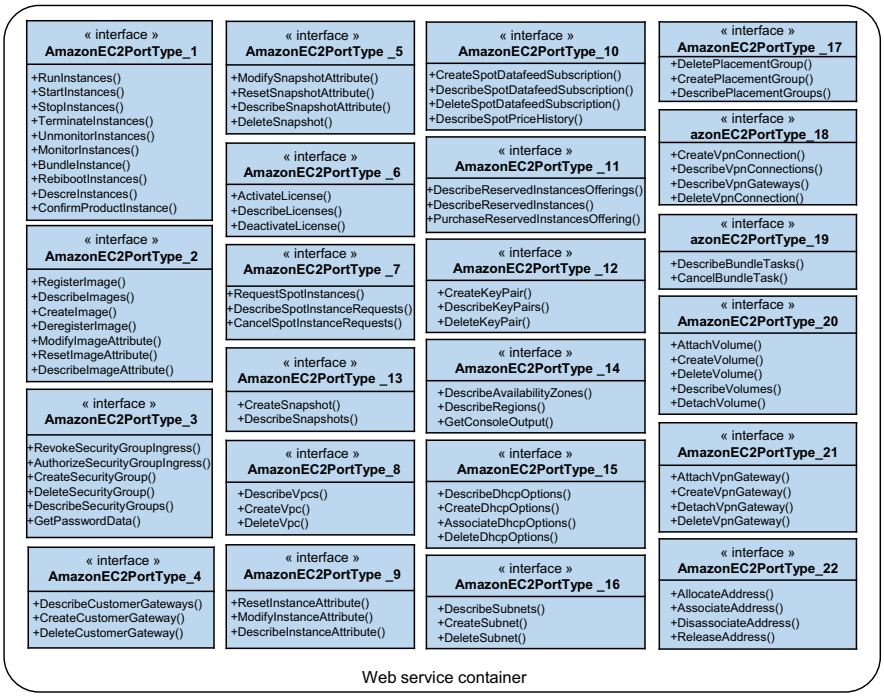


Fig. 7 The resulted remodularization obtained with WSIRem for the `AmazonEC2PortType` service interface

5.2.1 Analysis method

To answer **RQ2**, we asked a group of independent developers to manually decompose each of the studied interfaces (*cf.* Table 1) in order to identify groups of operations that represent distinct core abstractions. We provided each participants with the documentation of each considered service API and its high level view using a graphical representation of the service as a single class diagram exposing its list of operations and data types. We provided the participants with the necessary instructions on the output format as well as the formal process description of service interface partitioning refactoring as described in Dudney et al. (2003). The abstractions identified by the developers were considered as the ground truth, allowing the calculation of the precision and recall of our approach.

We compute he precision and recall scores as follows:

$$Precision = \frac{TP}{TP + FP} \tag{16}$$

$$Recall = \frac{TP}{TP + FN} \tag{17}$$

where TP (*True Positive*) corresponds to an interface identified by the independent developer and also by the proposed approach; FP (*False Positive*) corresponds an

interface identified by the proposed approach, but not by the independent developer; FN (*False Negative*) corresponds to an interface identified by the independent developer, but not by the proposed approach.

Note that we computed TP, FP and FN at a fine-grained level, meaning that the interface identified by the proposed approach and by the independent developer should match with a Jaccard similarity of at least 80% in terms of their operations.

5.2.2 Participants

Our evaluation involved 14 independent volunteer participants including 6 industrial developers and 8 graduate students in Software Engineering (3 MSc and 5 PhD candidates). We first gathered information about the participant's background. All participants are familiar with service-oriented development and SOAP Web services with an experience ranging from 4 to 9 years. The participants were unaware of the techniques *WSIRem*, *Greedy*, and *SIM*, neither the particular research questions, in order to guarantee that there will be no bias in their judgment.

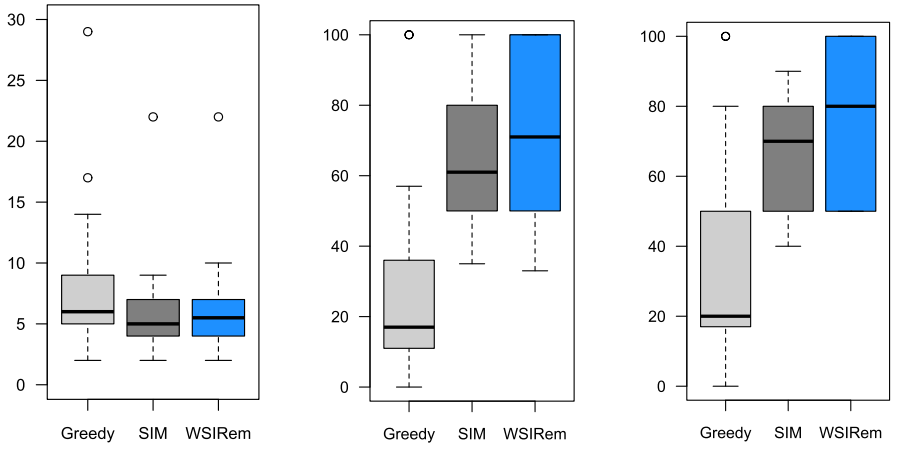
5.2.3 Results for RQ2

Table 3 and Fig. 8 report the results for RQ2 in terms of number of generated interfaces, precision and recall of each of *WSIRem*, *SIM*, and *Greedy*. As it can be observed from the table and figure, for the 22 services, *WSIRem* achieved an average number of interface split (i.e., modularization size) of 6.14, an average precision of 73% and an average recall of 77% with respect to the expected results obtained from the developers. We consider that these values of precision and recall are high since a deviation between the proposed solution and the expected one may not be an indication of some wrong recommendations but it could be just another possible good solution. In fact, there is no single good remodularization solutions but multiple ones. As for the state-of-the-art techniques, we observed that *SIM* achieved an average of 64% and 66% of precision and recall, respectively, while having an average of 5.86 in terms of interfaces splits. On the other hand, we noticed that *Greedy* tends to produce more split interfaces with an average of 8.22, thus generating smaller interfaces. Indeed, smaller interfaces tend to have higher cohesion. This resulted in low precision and recall with an average of 27% and 33%, respectively. We noticed that, *Greedy* generated in many cases, several interfaces with only one operation. Such fine-grained interfaces will make the service more complex and limit its reusability as core abstractions will be split into several smaller and scattered interfaces. Such interfaces may include other antipattern types such as fine-grained, chatty or data Web services (Ouni et al. 2017b).

Indeed, we observe from the obtained results the valuable benefits of considering the three heuristics IMD, cohesion and coupling that would prevent generating very small, i.e., fine-grained or chatty, interfaces which require typically high IMD. In general, fine-grained and chatty interfaces often requires several coupled Web service interfaces to complete an abstraction (Dudney et al. 2003). Therefore, finding the best trade-off between cohesion, coupling and IMD would help avoiding such undesirable interfaces.

Table 3 Comparison results of *WSIRem* and *Greedy* in terms of (a) number of generated interfaces, (b) precision and (c) recall

Provider	Interface	WSIRem			Greedy			SIM		
		# interfaces	Precision (%)	Recall (%)	# interfaces	Precision (%)	Recall (%)	# interfaces	Precision (%)	Recall (%)
Amazon	AmazonEC2PortType	22	82	90	29	14	18	22	82	90
	MechanicalTurkRequesterPortType	7	100	100	17	0	0	7	80	85
	AmazonFPSPorttype	10	80	89	11	27	30	9	60	70
	AmazonRDSv2PortType	6	67	80	5	20	20	5	66	70
	AmazonVPCPortType	6	100	100	6	0	0	6	85	80
	AmazonFWSInboundPortType	6	67	67	5	40	33	6	60	58
	AmazonS3	4	75	60	5	17	20	5	65	50
	AmazonSNSPortType	5	40	50	6	17	20	4	35	40
	ElasticLoadBalancingPortType	4	50	67	4	0	0	3	40	55
	MessageQueue	4	50	50	6	50	60	4	45	50
	AutoScalingPortType	4	50	67	6	17	33	3	45	55
Yahoo	KeywordService	8	78	88	9	11	14	7	68	70
	AdGroupService	9	100	100	9	100	80	9	100	100
	UserManagementService	7	100	100	14	36	71	7	80	90
	TargetingService	6	67	80	8	13	20	5	62	70
	AccountService	6	100	100	14	7	17	6	90	90
	AdService	5	60	50	3	25	17	6	55	45
	CompaignService	3	67	50	7	14	25	4	60	40
	BasicReportService	5	100	100	7	57	80	5	80	85
	TargetingConverterService	2	100	100	2	50	60	2	100	100
	ExcludedWordsService	3	33	50	4	33	50	2	50	70
	GeographicalDictionaryService	3	33	50	4	0	0	2	50	50
Average		6.14	72.68	76.73	8.23	27.18	33.09	5.86	64.00	66.05



(a) # of interfaces. (b) Precision. (c) Recall.

Fig. 8 Boxplots for the comparison results of *WSIRem*, *SIM*, and *Greedy* in terms of **a** number of generated interfaces, **b** precision, and **c** recall. Higher precision and recall scores mean better performance

Another interesting observation from Table 3 is that *WSIRem* has successfully identified remodularization solutions with 100% of precision and recall in 7 out of the 22 services, while *SIM* succeeded to do so only twice, and none for *Greedy*. Furthermore, we noticed that *Greedy* failed in identifying correct remodularization in 4 cases out of 22 with 0% of precision and recall.

Finally, we identify a main drawback of the *Greedy* approach from our perspective, that driving Web service interface modularization with only cohesion metrics would not be enough. The obtained results suggests that other design aspects such as coupling and size of interfaces are as important aspects as cohesion to drive service interface remodularization.

5.3 Evaluation with developers

To further evaluate our approach, we assessed the remodularization solutions from the developers' point of view. Our aim is to answer the following research question:

– **RQ3** Does *WSIRem* identifies 'useful' interface remodularization solutions from a developer's point of view?

5.3.1 Analysis method

To answer **RQ3**, we asked the fourteen participants involved in RQ2 to evaluate the usefulness of the remodularization solutions for each of the 22 cases. For each service, the participants evaluated four different solutions: (*i*) the remodularization provided by *WSIRem*, (*ii*) the remodularization provided by *Greedy*, (*iii*) the remodularization provided by SIM, and (*iV*) a random remodularization. The random remodularization

option is considered as a 'sanity check' to make sure whether the participants have seriously answered this study, as a random remodularization does not make sense.

To this end, we used a survey hosted in *eSurveyPro*,[9] an online Web application. Specifically, for each modularization solution, we provide a high-level description of each service interface before and after remodularization using UML classes (as represented in Fig. 7). Then, the participants was asked to answer the following question for each remodularization solution:

"Does the new modularization improve the understandability of the service?"

Possible answers follow a five-point Likert scale (Chisnall 1993) to express their level of agreement: 1: *Strongly disagree*, 2: *Disagree*, 3: *Neutral*, 4: *Agree*, 5: *Fully agree*. Note that the Web application used for our survey allowed our participants to save and complete the study in multiple rounds within a maximum of 7 days available to respond. At the end of the 7th day, we collected the 14 complete questionnaires.

5.3.2 Results for RQ3

Figure 9 and Table 4 report the results achieved by our study for the developers evaluation. From the table, we observe that for all the studied services, the participants rated the *WSIRem* interfaces with an average score of 3.93, while an average of 3.06, 2.5, and 1.32 was recorded for SIM, *Greedy*, and the random interfaces, respectively. This results indicate that the remodularization solutions suggested by *WSIRem* are more adjusted to developers needs than those of *SIM* and `Greedy`. Moreover, on top of the 22 cases, participants identified two services, `GeographicalDictionaryService` and `AutoScalingPortType` where the original interface is relatively understandable even without remodularization, but they suggested that an early remodularization would be preferable to avoid potential difficulties in future releases of the Web service with additional operations.

As it can be noticed from Fig. 9, the rating scores differ from one developer to another. Indeed, since this is a subjective decision process, it is normal that not all the developers have the same opinion. Thus, it is important to study the level of agreement between developers. To address this issue, we evaluated the level of agreement using Fleiss's Kappa coefficient κ (Fleiss 1971), which measures to what extent the developers agree when rating the resulted interface modularizations. The analysis of the results indicate that the Kappa coefficient assessments is 0.63, which is characterized as "*substantial agreement*" by Landis and Koch (1977). This obtained score makes us more confident that our defined objectives for Web service interfaces remodularization makes sense from software developer's perspective.

To further analyze the results, it is worth to note that during the evaluation, we discovered some common operations provided by different Amazon Web services. For example, we found that `AmazonVPCPortType` and `AmazonEC2PortType` have several common operations including *CreateVpc()*, *DescribeVpcs()*, *DeleteVpc()*, *DeleteVpnConnection()*, *CreateVpnGateway()* and *DeleteVpnGateway()*. More interestingly, some generated interfaces from both `AmazonVPCPortType` and `Amazon`

[9] http://www.esurveyspro.com.

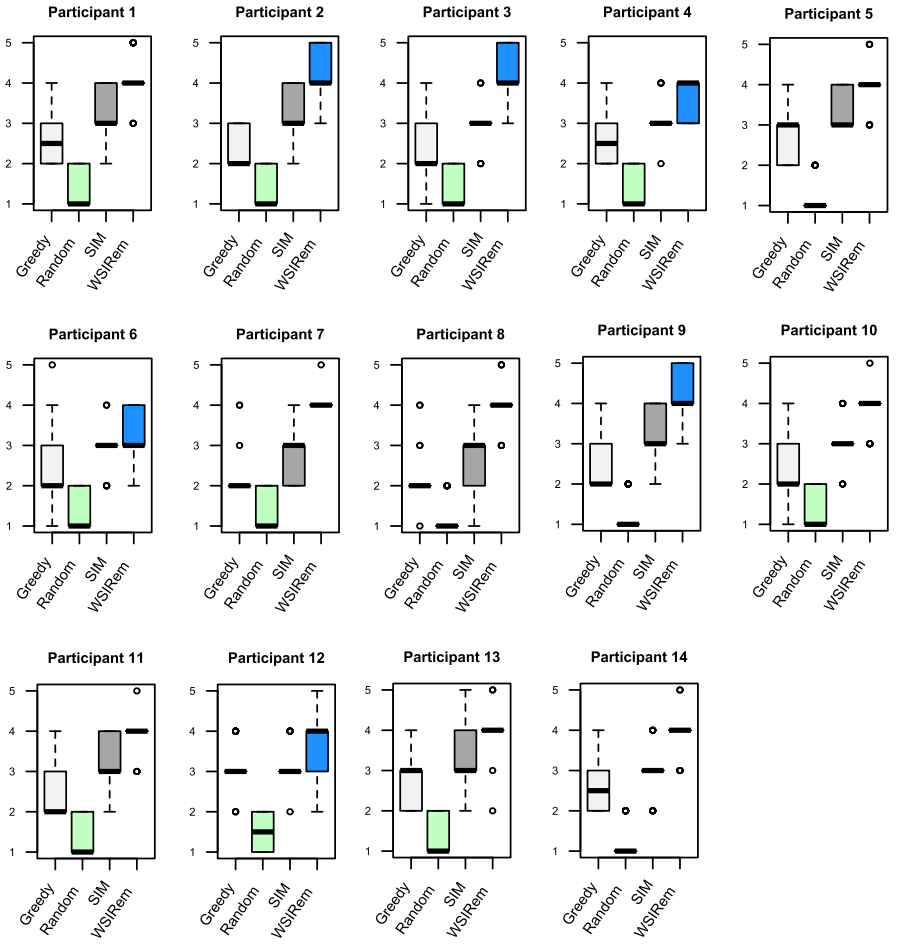


Fig. 9 Developer's evaluation of the interface remodularizations for *WSIRem*, *SIM*, *Greedy*, and *Random* modularizations. High rating scores mean better performance

`EC2PortType` expose exactly the same operations. Although this redundancy can be related to some business constraints, best design practices in SOA suggest that common core abstraction can be implemented in separate service interfaces, making them easier to maintain, evolve and reuse.

An interesting point was that some participants confirmed that the interfaces suggested by *WSRem* tend to be more appropriately sized and describe distinct abstractions with less overlap. We asked one of the participants to comment on his decision for the generated Amazon EC2 interfaces, his answer was: "*This new interface structure is more understandable to me, as it was previously very difficult to follow and understand a bench of 87 operations exposed in a single interface. I strongly recommend the original provider to restructure his service, to allow the service to be reused more effectively*".

Table 4 Developer's evaluation of the interface remodularizations for *WSIRem*, *SIM*, *Greedy*, and *Random* modularization for each service

Provider	Interface	WSIRem	SIM	Greedy	Random
Amazon	AmazonEC2PortType	4.00	2.92	2.57	1.28
	MechanicalTurkRequesterPortType	4.28	3.14	2.42	1.5
	AmazonFPSPorttype	4.07	2.92	2.42	1.28
	AmazonRDSv2PortType	3.92	3.14	2.35	1.21
	AmazonVPCPortType	3.85	3.07	2.5	1.35
	AmazonFWSInboundPortType	4.07	3.07	2.28	1.14
	AmazonS3	4.14	3.21	2.21	1.28
	AmazonSNSPortType	3.71	3.00	2.14	1.35
	ElasticLoadBalancingPortType	3.78	2.85	2.21	1.28
	MessageQueue	3.92	2.85	2.35	1.42
	AutoScalingPortType	3.78	3.35	2.71	1.14
Yahoo	KeywordService	3.92	3.07	2.5	1.5
	AdGroupService	3.78	3.21	2.85	1.21
	UserManagementService	3.85	3.14	2.64	1.5
	TargetingService	4	3.21	2.57	1.21
	AccountService	3.85	3.14	2.71	1.42
	AdService	3.78	3.21	2.78	1.21
	CompaignService	4.07	3.21	2.92	1.35
	BasicReportService	3.92	3.07	2.57	1.35
	TargetingConverterService	4.00	2.57	2.35	1.64
	ExcludedWordsService	3.85	2.92	2.35	1.14
	GeographicalDictionaryService	3.85	3.07	2.5	1.23
Average		3.93	3.06	2.50	1.32

Moreover, we noticed that in most of the cases, *Greedy* approach tend to split core abstractions into many interfaces. For instance, in the Amazon EC2 interface, operations related to image management was dispersed through many other interfaces: operations *RegisterImage()* and *DescribeImages()* are assigned to a new interface, *DescribeImageAttribute()* is in another interface, *CreateImage()* is in another interface, *ResetImageAttribute()*, *DeregisterImage()* and *ModifyImageAttribute()* are in another interface along with other unrelated operations (Athanasopoulos et al. 2015). We asked another participant comment on this remodularization, his answer was: "*Such scattered abstractions will result in several connections between interfaces for no benefit as a large number of suggested interfaces are not representing core abstractions*". On the other hand, most of the identified interfaces expose operations related to different core abstractions. For instance, for the same Amazon EC2 service, a suggested interface by *Greedy* contains *DetachVolume()*, *AttachVolume()* and *DescribeInstanceAttribute()*. Results show that this design is unlikely to be desirable for developers. Moreover, the obtained results suggest that coupling is as important metric as cohesion to drive Web service interface remodularization.

6 Threats to validity

It is widely recognised that several factors can bias the validity of empirical studies. In this section, we discuss the different threats that may affect our study.

Construct threats to validity can be related to the measurements performed to address the research questions. In our study we measured the increase in cohesion, coupling and mudularity result in the remodularization suggested by our approach. To measure cohesion, we employed three well-established quality metrics, i.e., LoC_{seq} for the lack of sequential cohesion, LoC_{com} for the lack of communicational cohesion, LoC_{sem} for the lack of conceptual cohesion. We provide a significant improvement to the latter by employing WordNet and a lemmatization technique to get more sense about the semantic information embodied in operations names. We also employed a coupling metric to measure the relatedness between service interfaces based on sequential, communicational and conceptual measures. We used equal weights in the experiments (the average of the three measures). Our approach has the advantage to provide an extra flexibility to the user to configure these weights based on his needs/preferences. As future work, we plan to conduct an empirical study with different values to assess the importance of each of them.

Although achieving an improvement in terns of metrics is necessary to show that an automated approach is consistent, we also performed developer study. Another threat to construct validity can be related to the set of ground truth to calculate precision and recall with remodularizations performed manually by developers. Indeed, since, for each interface, there is no single good remodularization solution, we used the Jaccard similarity at 80% to measure the similarity between a candidate interface and a manually identified interface. However, a deviation with the expected remodularization solution provided by the developer may not indicate that part of our recommendations are wrong but it could be just another possibility to remodularize the interface.

Internal threats to validity can be related to the knowledge and expertise of the human evaluators. Inadequate knowledge could lead to limited ability to assess the quality of an interface. We mitigate this threat by selecting participants having from 4 to 9 years experience with service-oriented development and familiar with SOAP Web services. Additionally, we plan to ask a large number of experienced professionals on software quality assessment and software refactoring to provide their expert opinion. Moreover, to avoid bias in the experiment none of the authors have been involved in this evaluation. In addition, we randomized the ordering in which the *WSIRem*, *Greedy* and random remodularizations were shown to the participants, to mitigate any sort of learning or fatigue effect.

Threats to external validity can be related to the studied services. Although we used 22 real-world Web services provided by Amazon and Yahoo, from different application domains and ranging from 10 to 87 operations, we can not generalize our results to other services and other paradigms, e.g., REST services.

7 Related work

This section reviews the related literature in three different areas (*i*) Web service antipatterns, (*ii*) software refactoring, and (*iii*) software modularization.

7.1 Web service antipatterns

Detecting antipatterns, i.e., bad design and implementation practices, in Web services and SOA (service-oriented architecture) is a relatively new field. The first book in the literature was written by Dudney et al. (2003) and provides informal definitions of a set of Web service antipatterns. More recently, Rotem-Gal-Oz described the symptoms of a range of SOA antipatterns (Rotem-Gal-Oz 2012). Furthermore, Král and Zemlicka (2009) listed seven "popular" SOA antipatterns that violate accepted SOA principles. Recently, Ouni et al. (2015a, 2016, 2017b) proposed a search-based approach to automatically detect Web service antipatterns including the god object Web service, fine-grained Web service, ambiguous Web service and semantically unrelated operations in port types. The proposed approach uses genetic programming to identify Web service interfaces that present symptoms of poor design practices. Moha et al. (2012) have proposed a rule-based approach called SODA for SCA systems (Service Component Architecture). Later, Palma et al. (2014) extended this work for Web service antipatterns in SODA-W. The proposed approach relies on declarative rule specification using a domain-specific language (DSL) to specify/identify the key symptoms that characterize an antipattern using a set of WSDL metrics. In other work (Torkamani and Bagheri 2014), the authors presented a repository of 45 general antipatterns in SOA, to support developers to work with clear understanding of patterns in phases of software development and so avoid many potential problems. Mateos et al. (2015b) have proposed an interesting approach towards generating WSDL documents with less antipatterns using text mining techniques. In this paper, we assume that such a bad practice is detected and we focus on fixing it. Recently, De Renzis et al. (2017) presented a practical metric to quantify readability in WSDL documents using WordNet to estimate their semantics along with a set of best practices to be used during the development of WSDL documents to improve their readability. Hirsch et al. (2017) assessed an approach called Gapidt to remove discoverability anti-patterns in code-first Java Web Services using text mining techniques in the source code level. However, the proposed aproaches does not provide guidance for solving service interface granularity problems.

7.2 Software refactoring

Software refactoring have been practiced for more many years (Fowler 1999). It is widely recognized that, if applied well, refactoring brings a lot of advantages to improve software readability, maintainability and flexibility. One of the first attempts to address service interface refactoring was by Athanasopoulos et al. (2015) (*Greedy*). Although their approach was able to improve cohesion, it is not perfectly adjusted to the developers' needs (Athanasopoulos et al. 2015). However, coupling between

interfaces is not considered which results in many cohesive but highly connected interfaces. Our approach addresses explicitly this issue limitation to improve the modularization quality. In another study, Rodriguez et al. (2013, 2010) and Coscia et al. (2014) provided a set of guidelines for service providers to avoid bad practices while writing WSDLs. Based on some heuristics, the authors detected eight bad practices in the writing of WSDL for Web services. Ouni et al. (2016) proposed a graph partitioning based approach, namely SIM for Web service interface remodularization to find groups of cohesive operations representing different abstractions, which has been later extended by refining the obtained remodularization solutions using a multi-objective search algorithm (Ouni et al. 2018). Recently, Wang et al. (2018) proposed an optimization-based approach to improve the design quality of Web service interfaces and reduce antipatterns while putting the developer in the loop using an interactive approach. Daagi et al. (2017) proposed a Web service decomposition approach using formal concept analysis to identify potential groups of structurally related operations in a Web service interface. However, the identified interfaces suffer from a couple of limitations including a low balance between coupling and cohesion, as well as a high design deviation in the new interfaces. Indeed, it has been shown in the litterature that heuristic-based approaches are most fitted for such combinatorial problems. We thus address explicitly these limitations in this paper to improve the design quality of Web service interfaces and produce more efficient solutions.

A lot of efforts has been devoted to refactoring of object-oriented (OO) applications. Our approach is more closely similar to *Extract Class* refactoring in OO systems, which employs metrics to split a large class into smaller, more cohesive classes (Fowler 1999). Romano et al. (2014); Romano and Pinzger (2014) proposed a single objective genetic algorithm-based approach to address the problem of Interface Segregation Principle (ISP) violation in ”fat” APIs (Application Programming Interfaces). The proposed approach aims at splitting Fat API interfaces into smaller interfaces exposing only the methods invoked by groups of clients. However, applying the ISP is a challenging task when fat interfaces are invoked differently by many clients or when the number of available clients is not representative, or when there is no available clients for new services. Moreover, the proposed approach is a signle objective one ignoring the cohesion and coupling aspects that are widely recognized as basic measures to assess software modularity (Paixao et al. 2018; Mkaouer et al. 2015; Praditwong et al. 2011). Bavota et al. (2011, 2014) have proposed a similar approach to split a large class into smaller cohesive classes using structural and semantic similarity measures. Fokaefs et al. (2012) proposed an automated extract class refactoring approach based on a hierarchical clustering algorithm to identify cohesive subsets of class methods and attributes. However, the *Extract Class* refactoring is not applicable in the context of Web services as typically the Web service source code is not publicly available, and the development paradigm, used technologies and metrics are different. Furthermore, the proposed approach is a deterministic approach which may not be efficient in solving combinatorial problems, such as the service interface remodularization one as pointed out by many researchers (Praditwong et al. 2011; Mkaouer et al. 2015).

7.3 Software modularization

Several studies addressed the problem of clustering and remodularization of OO applications in terms of packages organization. Anquetil and Lethbridge (1999) used cohesion and coupling of modules within a decomposition of OO systems to evaluate its quality. Maqbool and Babri (2007) used hierarchical clustering in the context of software architecture recovery and modularization. On the other hand, Mancoridis et al. (1998) proposed the first search-based approach to address the problem of software modularization using a single objective approach. Harman et al. (2002) used a genetic algorithm to improve subsystems decomposition by combining several quality metrics including coupling, cohesion, and complexity. Similarly, Seng et al. (2005) treated the remodularization task as a single-objective optimization problem using genetic algorithm to reduce violations of design principles. Later, Abdeen et al. (2009) proposed a heuristic search-based approach for automatically optimizing (i.e., reducing) the dependencies between packages of a software system by moving classes between packages. Recently, Mkaouer et al. (2015) have proposed a multi-objective approach to finding optimal remodularization solutions that improve the structure of packages, minimize the number of changes, preserve semantics coherence, and reuse the history of changes. Paixao et al. (2015) proposed a search-based software clustering approach for Kate programs using two optimization strategies: Maximizing Clusters (MCA), and Equal-size Clusters (ECA). The proposed approach achieves an interesting improvement in terms of cohesion and coupling. However, the new generated modularization might require extensive changes to be applied, taking the system away from its original design. Despite these advances in OO systems modularization, still this problem in its enfancy in the context of Web services and service-based software systems.

8 Conclusions and future work

In this paper, we have proposed a multi-objective search-based approach, *WSIRem* to Web service interface remodularization and evaluated it on 22 real-world Web services. Our approach used NSGA-II to find the best trade-off between cohesion, coupling, number of interfaces, and the average number of operations per interface. A comprehensive empirical evaluation of the proposed approach was performed from design metrics and developers perspective. Our results provides evidence that *WSIRem* performs significantly better than state-of-the art techniques in terms of design quality improvement. Moreover, we evaluated our approach with 14 developers since we strongly believe that an automated approach must suggest remodularization solutions considered relevant by developers. Our results show that *WSIRem* provides remodularization solutions that are (*i*) more adjusted to the developers' expectations with more than 73% of precision and 77% of recall, and (*ii*) able to improve the service understandability from developer's point of view.

In the future work, we plan to conduct a comparative study between NSGA-II and other existing multi-objective search algorithms such as multi-objective particle swarm optimization (MOPSO), indicator-based evolutionary algorithm (IBEA), and

multi-objective ant calony optimization (MOACO) to assess the performance of each search method in different contexts. More interestingly, we plan to extend *WSIRem* to a multi-objective interactive optimization, where we put the developer in the loop. We aim at improving the remodularization quality by driving the search process by another objective function that encapsulates the particular preferences of the developer.

Acknowledgements This work is partially supported by NSERC Discovery Grant (Grant No. RGPIN-2018-05960), and by the Research Start-up (2) 2016 Grant G00002211 funded by UAE University.

References

Abdeen, H., Ducasse, S., Sahraoui, H., Alloui, I.: Automatic package coupling and cycle minimization. In: 16th Working Conference on Reverse Engineering, pp. 103–112. IEEE (2009)

Anquetil, N., Lethbridge, T.C.: Experiments with clustering as a software remodularization method. In: 6th Working Conference on Reverse Engineering, pp. 235–255. IEEE (1999)

Athanasopoulos, D., Zarras, A.: Fine-grained metrics of cohesion lack for service interfaces. In: IEEE International Conference on Web Services (ICWS), pp. 588–595 (2011)

Athanasopoulos, D., Zarras, A.V., Miskos, G., Issarny, V.: Cohesion-driven decomposition of service interfaces without access to source code. IEEE Trans. Serv. Comput. **8**(JUNE), 1–18 (2015)

Bavota, G., De Lucia, A., Oliveto, R.: Identifying extract class refactoring opportunities using structural and semantic cohesion measures. J. Syst. Softw. **84**(3), 397–414 (2011)

Bavota, G., De Lucia, A., Marcus, A., Oliveto, R.: Automating extract class refactoring: an improved method and its evaluation. Empir. Softw. Eng. **19**(6), 1617–1664 (2014)

Budgen, D.: Software Design. Addision-Wesley, Reading (1999)

Card, D.N., Glass, R.L.: Measuring Software Design Quality. Prentice-Hall Inc., Englewood Cliffs (1990)

Chisnall, P.M.: Questionnaire design, interviewing and attitude measurement. J. Mark. Res. Soc. **35**(4), 392–393 (1993)

Cliff, N.: Dominance statistics: ordinal analyses to answer ordinal questions. Psychol. Bull. **114**(3), 494 (1993)

Cohen, J.: Statistical Power Analysis for the Behavioral Sciences. Academic Press, London (1988)

Coscia, J.L.O., Mateos, C., Crasso, M., Zunino, A.: Refactoring code-first web services for early avoiding WSDL anti-patterns: approach and comprehensive assessment. Sci. Comput. Program. **89**, 374–407 (2014)

Crasso, M., Rodriguez, J.M., Zunino, A., Campo, M.: Revising WSDL documents: why and how. IEEE Internet Comput. **14**(5), 48–56 (2010)

Daagi, M., Ouni, A., Kessentini, M., Gammoudi, M.M., Bouktif, S.: Web service interface decomposition using formal concept analysis. In: IEEE International Conference on Web Services (ICWS), pp. 172–179. IEEE (2017)

De Renzis, A., Garriga, M., Flores, A., Cechich, A., Mateos, C., Zunino, A.: A domain independent readability metric for web service descriptions. Comput. Stand. Interfaces **50**, 124–141 (2017)

Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Trans. Evolut. Comput. **6**(2), 182–197 (2002)

Demšar, J.: Statistical comparisons of classifiers over multiple data sets. J. Mach. Learn. Res. **7**, 1–30 (2006)

Dudney, B., Krozak, J., Wittkopf, K., Asbury, S., Osborne, D.: J2EE Antipatterns. Wiley, New York (2003)

Fleiss, J.L.: Measuring nominal scale agreement among many raters. Psychol. Bull. **76**(5), 378 (1971)

Fokaefs, M., Mikhaiel, R., Tsantalis, N., Stroulia, E., Lau, A.: An empirical study on web service evolution. In: IEEE International Conference on Web Services (ICWS), pp. 49–56 (2011)

Fokaefs, M., Tsantalis, N., Stroulia, E., Chatzigeorgiou, A.: Identification and application of extract class refactorings in object-oriented systems. J. Syst. Softw. **85**(10), 2241–2260 (2012)

Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Longman Publishing Co., Inc., Reading (1999)

Haesen, R., Snoeck, M., Lemahieu, W., Poelmans, S.: On the definition of service granularity and its architectural impact. In: Bellahsène, Z., Léonard, M. (eds.) Advanced Information Systems Engineering, pp. 375–389. Springer (2008)

Harman, M.: The current state and future of search based software engineering. In: Future of Software Engineering (FOSE), pp. 342–357 (2007)
Harman, M., Hierons, R.M., Proctor, M.: A new representation and crossover operator for search-based optimization of software modularization. GECCO **2**, 1351–1358 (2002)
Harman, M., Mansouri, S.A., Zhang, Y.: Search-based software engineering: trends, techniques and applications. ACM Comput. Surv. (CSUR) **45**(1), 11 (2012)
Hirsch, M., Rodriguez, A., Rodriguez, J.M., Mateos, C., Zunino, A.: Spotting and removing wsdl anti-pattern root causes in code-first web services: a thorough evaluation of impact on service discoverability. Comput. Stand. Interfaces **56**, 116–133 (2017)
Hruschka, E.R., Campello, R.J., Freitas, A., De Carvalho, A.C., et al.: A survey of evolutionary algorithms for clustering. IEEE Trans. Syst. Man Cybern. C Appl. Rev. **39**(2), 133–155 (2009)
Kessentini, M., Ouni, A.: Detecting android smells using multi-objective genetic programming. In: International Conference on Mobile Software Engineering and Systems (MobileSoft), pp. 122–132 (2017)
Král, J., Zemlicka, M.: Popular SOA antipatterns. In: Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns, pp. 271–276 (2009)
Landis, J.R., Koch, G.G.: The measurement of observer agreement for categorical data. Biometrics **33**(1), 159–174 (1977)
Mancoridis, S., Mitchell, B.S., Rorres, C., Chen, Y.F., Gansner, E.R.: Using automatic clustering to produce high-level system organizations of source code. In: IWPC, vol. 98, pp. 45–52. Citeseer (1998)
Maqbool, O., Babri, H.A.: Hierarchical clustering for software architecture recovery. IEEE Trans. Softw. Eng. **33**(11), 759–780 (2007)
Masoud, H., Jalili, S.: A clustering-based model for class responsibility assignment problem in object-oriented analysis. J. Syst. Softw. **93**, 110–131 (2014)
Mateos, C., Crasso, M., Rodriguez, J.M., Zunino, A., Campo, M.: Measuring the impact of the approach to migration in the quality of web service interfaces. Enterp. Inf. Syst. **9**(1), 58–85 (2015a)
Mateos, C., Rodriguez, J.M., Zunino, A.: A tool to improve code-first web services discoverability through text mining techniques. Softw. Pract. Exp. **45**(7), 925–948 (2015b)
Mkaouer, W., Kessentini, M., Shaout, A., Koligheu, P., Bechikh, S., Deb, K., Ouni, A.: Many-objective software remodularization using NSGA-III. ACM Trans. Softw. Eng. Methodol. (TOSEM) **24**(3), 17:1–17:45 (2015)
Moha, N., Palma, F., Nayrolles, M., Conseil, B.J., Guéhéneuc, Y.G., Baudry, B., Jézéquel, J.M.: Specification and detection of SOA antipatterns. In: Service-Oriented Computing, pp. 1–16. Springer (2012)
Newcomer, E.: Understanding Web Services: XML Soap, and UDDI. Addison-Wesley Professional, Reading (2002). Wsdl
Ouni, A., Kessentini, M., Sahraoui, H., Boukadoum, M.: Maintainability defects detection and correction: a multi-objective approach. Autom. Softw. Eng. **20**(1), 47–79 (2013)
Ouni, A., Gaikovina Kula, R., Kessentini, M., Inoue, K.: Web service antipatterns detection using genetic programming. In: Proceedings of the 2015 on Genetic and Evolutionary Computation Conference, GECCO'15, pp. 1351–1358 (2015a)
Ouni, A., Kessentini, M., Sahraoui, H., Inoue, K., Hamdi, M.S.: Improving multi-objective code-smells correction using development history. J. Syst. Softw. **105**, 18–39 (2015b)
Ouni, A., Kessentini, M., Sahraoui, H., Inoue, K., Deb, K.: Multi-criteria code refactoring using search-based software engineering: an industrial case study. ACM Trans. Softw. Eng. Methodol. (TOSEM) **25**(3), 23 (2016a)
Ouni, A., Salem, Z., Inoue, K., Soui, M.: SIM: an automated approach to improve web service interface modularization. In: International Conference on Web Services (ICWS), pp. 91–98 (2016b)
Ouni, A., Daagi, M., Kessentini, M., Bouktif, S., Gammoudi, M.M.: A machine learning-based approach to detect web service design defects. In: IEEE International Conference on Web Services (ICWS), pp. 532–539. IEEE (2017a)
Ouni, A., Kessentini, M., Inoue, K., O Cinneide, M.: Search-based web service antipatterns detection. IEEE Trans. Serv. Comput. **10**(4), 603–617 (2017b)
Ouni, A., Kessentini, M., Ó Cinnéide, M., Sahraoui, H., Deb, K., Inoue, K.: More: a multi-objective refactoring recommendation approach to introducing design patterns and fixing code smells. J. Softw. Evolut. Process **29**(5), e1843 (2017c)
Ouni, A., Wang, H., Kessentini, M., Inoue, K., Bouktif, S.: A hybrid approach for improving the design quality of web service interfaces. ACM Trans. Internet Technol. (TOIT) **19**(1), 1–24 (2018)

Paixao, M., Harman, M., Zhang, Y.: Multi-objective Module Clustering for Kate, pp. 282–288. Springer International Publishing, Cham (2015)
Paixao, M., Harman, M., Zhang, Y., Yu, Y.: An empirical study of cohesion and coupling: balancing optimization and disruption. IEEE Trans. Evolut. Comput. **22**(3), 394–414 (2018)
Palma, F., Moha, N., Tremblay, G., Guéhéneuc, Y.G.: Specification and detection of SOA antipatterns in web services. In: Avgeriou, P., Zdun, U. (eds.) Software Architecture, pp. 58–73. Springer (2014)
Perepletchikov, M., Ryan, C., Frampton, K.: Cohesion metrics for predicting maintainability of service-oriented software. In: 7th International Conference on Quality Software, pp. 328–335 (2007a)
Perepletchikov, M., Ryan, C., Frampton, K., Tari, Z.: Coupling metrics for predicting maintainability in service-oriented designs. In: 18th Australian Software Engineering Conference, pp. 329–340 (2007b)
Perepletchikov, M., Ryan, C., Frampton, K., Schmidt, H.: Formalising service-oriented design. J. Softw. **3**(2), 1–14 (2008)
Perepletchikov, M., Ryan, C., Tari, Z.: The impact of service cohesion on the analyzability of service-oriented software. IEEE Trans. Serv. Comput. **3**(2), 89–103 (2010)
Poshyvanyk, D., Marcus, A.: The conceptual coupling metrics for object-oriented systems. In: IEEE International Conference on Software Maintenance (ICSME), pp. 469–478 (2006)
Praditwong, K., Harman, M., Yao, X.: Software module clustering as a multi-objective search problem. IEEE Trans. Softw. Eng. **37**(2), 264–282 (2011)
Rodriguez, J.M., Crasso, M., Zunino, A., Campo, M.: Automatically detecting opportunities for web service descriptions improvement. In: Software Services for e-World, pp. 139–150. Springer (2010)
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)
Romano, D., Pinzger, M.: Analyzing the evolution of web services using fine-grained changes. In: IEEE International Conference on Web Services (ICWS), pp. 392–399 (2012)
Romano, D., Pinzger, M.: A genetic algorithm to find the adequate granularity for service interfaces. In: 2014 IEEE World Congress on Services (SERVICES), pp. 478–485. IEEE (2014)
Romano, D., Raemaekers, S., Pinzger, M.: Refactoring fat interfaces using a genetic algorithm. In: 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 351–360. IEEE (2014)
Rotem-Gal-Oz, A.: SOA Patterns. Manning Publications, Shelter Island (2012)
Seng, O., Bauer, M., Biehl, M., Pache, G.: Search-based improvement of subsystem decompositions. In: Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation, pp. 1045–1051. ACM (2005)
Stewart, K.J., Darcy, D.P., Daniel, S.L.: Opportunities and challenges applying functional data analysis to the study of open source software evolution. Stat. Sci. **21**, 167–178 (2006)
Torkamani, M.A., Bagheri, H.: A systematic method for identification of anti-patterns in service oriented system development. Int. J. Electr. Comput. Eng. **4**(1), 16–23 (2014)
Wang, H., Kessentini, M., Ouni, A.: Interactive refactoring of web service interfaces using computational search. IEEE Trans. Serv. Comput. (TSC) (2018). https://doi.org/10.1109/TSC.2017.2787152
Wen, Z., Tzerpos, V.: An effectiveness measure for software clustering algorithms. In: International Workshop on Program Comprehension, pp. 194–203 (2004)
Zhang, X., Tian, Y., Jin, Y.: A knee point-driven evolutionary algorithm for many-objective optimization. IEEE Trans. Evolut. Comput. **19**(6), 761–776 (2015)

Affiliations

Sabrine Boukharata[1] · Ali Ouni[1] · Marouane Kessentini[2] · Salah Bouktif[3] · Hanzhang Wang[4]

[1] ETS Montreal, University of Quebec, Montreal, QC, Canada

2 Computer and Information Science Department, University of Michigan, Ann Arbor, MI, USA

3 College of Information Technology, UAE University, Al Ain, UAE

4 eBay, San Jose, CA, USA