ReSIde: Reusable Service Identification from Software Families

Anas Shatnawi, Abdelhak-Djamel Seriai, Houari Sahraoui, Tewfik Ziadi, Abderrahmene Seriai

ReSIde: Reusable Service Identification from Software Families

Anas Shatnawi[a,c,1], Abdelhak Seriai[a], Houari Sahraoui[b], Tewfik Ziadi[c], Abderrahmene Seriai[b]

[a]*LIRMM, University of Montpellier, Montpellier, France*
[b]*GEODES, University of Montreal, Montreal, Quebec, Canada*
[c]*LIP6, Sorbonne University, Paris, France*

Abstract

The clone-and-own approach becomes a common practice to quickly develop Software Product Variants (SPVs) that meet variability in user requirements. However, managing the reuse and maintenance of the cloned codes is a very hard task. Therefore, we aim to analyze SPVs to identify cloned codes and package them using a modern systematic reuse approach like Service-Oriented Architecture (SOA). The objective is to benefit from all the advantages of SOA when creating new SPVs. The development based on services in SOA supports the software reuse and maintenance better than the development based on individual classes in monolithic object-oriented software. Existing service identification approaches identify services based on the analysis of a single software product. These approaches are not able to analyze multiple SPVs to identify reusable services of cloned codes. Identifying services by analyzing several SPVs allows to increase the reusability of identified services. In this paper, we propose *ReSIde* (**Re**usable **S**ervice **Ide**ntification): an automated approach that identifies reusable services from a set of object-oriented SPVs. This is based on analyzing the commonality and the variability between SPVs to identify the implementation of reusable functionalities corresponding to cloned codes that can be packaged as reusable services. To validate ReSIde, we have applied it on three product families of different sizes. The results show that the services identified based on the analysis of multiple product variants using ReSIde are more reusable than services identified based on the analysis of singular ones.

Keywords: software reuse, service-oriented reengineering, reverse engineering, variability, software families, object-oriented source code

1. Introduction

It is a common practice that software developers rely on the *clone-and-own* approach to deal with custom-tailored software [1, 2]. New software products are developed by copying and modifying codes corresponding to functionalities from existing software to meet the requirement of new needs of new customers. The resulting software products are considered Software Product Variants (SPVs) because they share features and differ in terms of others [1]. The existence of this phenomenon has been proved by empirical studies like [2] [3].

For *monolithic object-oriented SPVs*, managing the software reuse and maintenance of the cloned codes is a very hard task [4]. For reuse, e.g., it is hard to identify reusable codes from the monolithic object-oriented implementation of these SPVs [5]. For maintenance, e.g., it is difficult to propagate updates for fixing bugs related to the implementation of the cloned codes. Therefore, we are interested in analyzing SPVs to identify cloned codes and package them using a modern systematic reuse approach like Service-Oriented Architecture (SOA). The objective is to benefit from all the advantages of SOA when creating new SPVs. With SOA, SPVs are defined in terms of flexible architectures composed of a set of independent coarse-grained services that implement reusable functionalities across several SPVs, and *clearly* define their external dependencies in an explicit way through their provided and required interfaces.
One of the most important steps for reengineering monolithic object-oriented SPVs to SOAs is the identification of reusable services corresponding to cloned codes of reusable functionalities across several SPVs. Moreover, the identification of reusable services is an efficient way to supply service-based libraries.

Existing service identification approaches identify services based on the analysis of a single software product [6, 7, 8, 9]. These existing approaches partition the object-oriented implementation to disjoint groups of classes where each group is the implementation of a potential service. As these approaches only analyze single products, the identified services may be useless in other software products and consequently their reusability is not guaranteed. In addition, these approaches are not able to analyze multiple

Email addresses: anas.shatnawi@lip6.fr (Anas Shatnawi), abdelhak.seriai@lirmm.fr (Abdelhak Seriai), sahraouh@iro.umontreal.ca (Houari Sahraoui), tewfik.ziadi@lip6.fr (Tewfik Ziadi), seriai.abdelrahmene@gmail.com (Abderrahmene Seriai)
[1]Corresponding author

SPVs to identify reusable services related to cloned functionalities and their related codes. In fact the probability of reusing a service in a new software product is proportional to the number of software products that have already used it [10, 11]. Thus, mining software services based on the analysis of a set of SPVs contributes to identify reusable services. Nonetheless, this has not been investigated in the literature. Identifying services by analyzing multiple SPVs makes it possible to improve the reusability of services to reduce the effort when developing new software products (by reuse) and to reduce the maintenance effort by making it possible to propagate any change to a service across all of the products that reuse this service.

In this paper, we propose *ReSIde* (**Re**usable **S**ervice **Ide**ntification): an automated approach that identifies reusable services from a set of similar object-oriented SPVs. ReSIde analyzes the commonality and the variability between the object-oriented source code of multiple SPVs to identify the implementation of reusable functionalities corresponding to cloned codes. These identified functionalities are intended to be packaged as reusable services that can be reused across multiple products. ReSIde is motivated by the fact that services identified based on the analysis of several existing SPVs will be more useful (reusable) for the development of new SPVs than services identified from singular ones.

To validate ReSIde, we have applied it on three open-source product families of different sizes (i.e., small, medium and large-scale ones). We propose an empirical measurement to evaluate the reusability of the identified services. According to this measurement, the results show that the reusability of the identified services using ReSIde is better than the reusability of those identified from singular software.

The idea of analyzing multiple SPVs to identify reusable components was introduced in our conference paper [12]. In relationship with this conference paper, this journal paper addresses the identification of services and not software components. Also, it includes additional contents in terms of:

1. Proposition of a deep analysis of the problem of identifying reusable services from multiple SPVs.
2. Proposition of more details and deep analysis of the proposed solution, e.g., by giving more details about used algorithms and illustrating the solution based on new examples and figures.
3. Adding a new case study that is Health Watcher and consequently extending the evaluation.
4. Presentation of new detailed results and new analysis of their relevance.
5. Adding threats to validity discussions.
6. Important extension of related work analysis and classification.
7. The analysis of the research and practical implications of the obtained results.

The rest of this paper is organized as follows. Section 2 presents a background needed to understand our approach. In Section 3, we provide the foundations of ReSIde. Section 4.1 discusses how ReSIde identifies potential services from each SPV. In Section 4.2, we present the identification of similar services between different SPVs. Reusable services are recovered from the similar ones in Section 4.3. Section 4.4 presents how ReSIde structures the service interfaces. The evaluation results are discussed in Section 5 and Section 6. In Section 7, we present the related works to our approach. A conclusion of this paper is presented in Section 8.

2. Background: service quality model

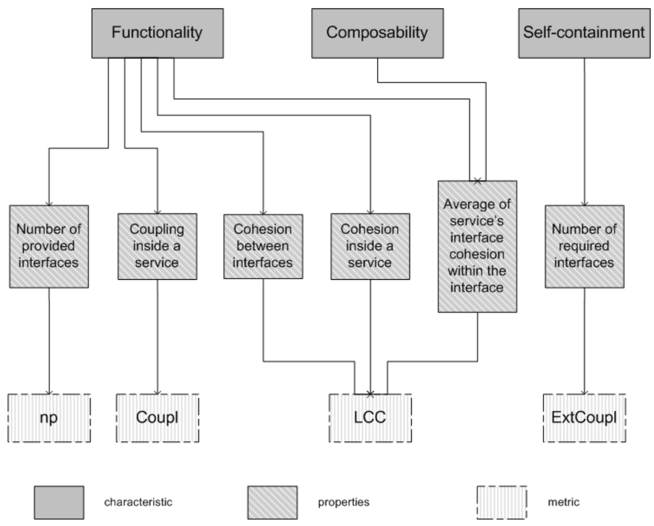

Figure 1: Service quality model

In this section, we discuss the service quality model proposed in our previous work [6] and which is reused in this paper to evaluate the quality of a cluster of classes to form a quality-centric service based on the structural dependencies between these classes. From the service structure point of view, any group of classes can form a service. Therefore, we need a measurement to distinguish good services from bad ones. To do so, we use this quality fitness function to identify only groups of classes that could form high quality services.

To define this service quality model, we studied the existing definition of services in the literature and identified three quality characteristics that should be measured to evaluate the quality of a group of classes to form a quality-centric service. These characteristics are: (i) the *coarse-grained of functionalities* implemented by the cluster of classes, (ii) the *composability* of the cluster of classes to be reused through their interfaces without any modification, and (iii) the *self-containment* of the the cluster of classes.

As presented in Figure 1, we perform similar to the ISO9126 quality model [13] to refine these three characteristics to a number of service properties that can be measured using a number of object-oriented metrics (e.g., *self-containment* is refined to the number of required interfaces by a given service).

The service quality model identifies service characteristics and refine them as metrics. However, to identify services, we need to put these metrics as function that can be computed to output a numerical value for evaluating the semantic of a service (i.e. what a service is) based on its implementation composed of a cluster of object-oriented classes. Therefore, we defined a quality fitness function (QFF) based on our service quality model where its input is a cluster of classes (E), and its output is a value, situated in [0–1], corresponding to the quality of this cluster of classes to form a quality-centric service. This QFF is represented by Equation 1 based on the linear combination of the three quality characteristics: Functionality (Fun), Composability (Comp) and Self-Containment (SelfCon).

$$QFF(E) = \frac{1}{\sum_{i=1}^{3} \lambda_i} \cdot (\lambda_1 \cdot Fun(E) + \lambda_2 \cdot Comp(E) + \lambda_3 \cdot SelfCont(E)) \quad (1)$$

Where λ_i are parameters used by the practitioners to weight each characteristic.

The Functionality (Fun), the Composability (Comp) and the Self-Containment (SelfCon) of a group of classes (E) are measured based on Equation 2, Equation 3 and Equation 4 respectively.

$$Fun(E) = \frac{1}{5} \cdot (np(E) + \frac{1}{I} \sum_{i \in I} LCC(i) + \mathrm{LCC(I)} + \mathrm{Coupl(E)} + \mathrm{LCC(E)}) \quad (2)$$

$$Comp(E) = \frac{1}{I} \sum_{i \in I} LCC(i) \quad (3)$$

$$SelfCont(E) = ExtCoupl(E) \quad (4)$$

Where *np(E)* is the number of provided interfaces based on the number of public methods in E. *LCC(i)* (Loose Class Cohesion) [14] is the average of the cohesion of a group of methods composing the service interfaces. These methods are the public methods implemented in the identified classes of the service. *LCC(i)* is calculated based on the percentage between the number of links among these methods and the total number of possible links among these methods. *LCC(I)* is the cohesion between interfaces. LCC(E) is the cohesion inside a service. *Coupl(E)* (Coupling) measures the level of connectivity (e.g., method calls, attribute accesses) of a group of classes with the reaming classes of the software. *ExtCoupl* (External Coupling) measures the coupling of a given potential service with other services (1 - Coupl).

In this paper, we use this quality fitness function as a black box component. It worths to note that it can be replaced by any service quality fitness function following the needs of the software engineers. Please refer to [6] for more details regarding the service quality model and its quality fitness function.

3. ReSIde foundations

3.1. Illustrative example

We present in Figure 2 an illustrative example to easy understand the foundations of our approach. We have 2 SPVs that are developed based on the clone-and-own approach.

SPV1 includes 5 classes that implement functionalities related to the photo management. SPV2 cloned SPV1 and extends it based on 10 additional classes. These classes aim to improve existing functionalities and to add other functionalities related to the music management.
Our goal is to identify reusable services based on the analysis of source codes respectively of theses two SPVs.

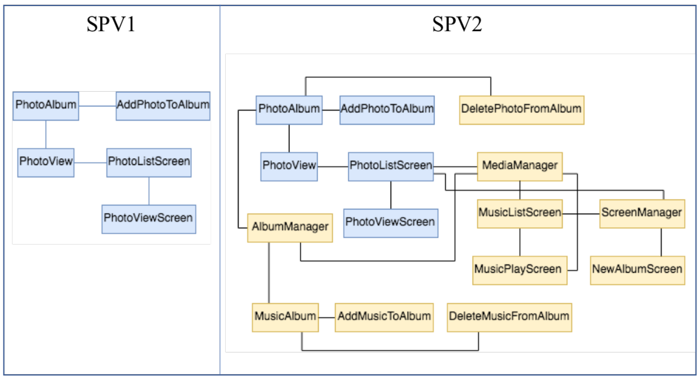

Figure 2: Illustrative example of two SPVs

3.2. ReSIde principles

ReSIde aims to identify reusable services based on the analysis of the object-oriented source code of similar SPVs. To identify services from the source code of object-oriented software, we propose an object-to-service mapping model that maps the object-oriented elements (classes and methods) to SOA ones (services and interfaces). We present this mapping model in Figure 3. We define a service in terms of a cluster of object-oriented classes. A service implements a set of functionalities provided using its interfaces. Each interface is defined in terms of a set of object-oriented methods implemented in the classes of the service. The provided interfaces of a service are defined as a group of methods implemented by classes composing the service and accessed by classes of other services. The required interfaces of a service are defined as a group of methods invoked by classes of the service and implemented in the classes of other services.

To identify a cluster of classes frequently appear together in several SPVs to implement the same functionalities, we rely on two types of dependencies: the *co-existence together* and the *structural object-oriented* dependencies. The *co-existence together dependencies* measure how much a cluster of classes are reused together in the same subset of SPVs. These are used to guarantee that the resulting services are reusable across different SPVs. The *structural object-oriented dependencies* evaluate the quality of a cluster of classes to form a quality-centric service based on the service quality model proposed in our previous work (c.f. Section 2). These are used to guarantee that we produce quality-centric services.

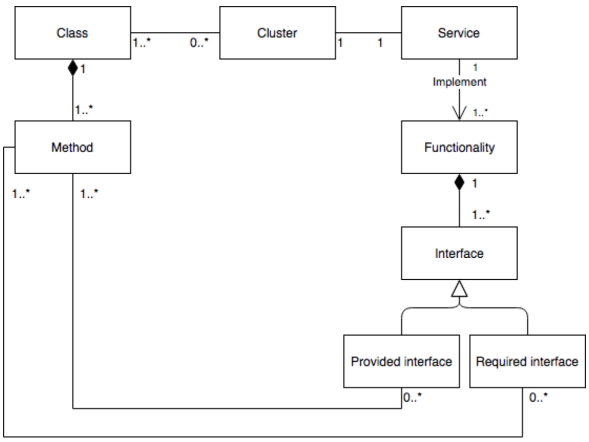

Figure 3: Object-to-service mapping model

We summarize the principles of ReSIde as follows.

- ReSIde defines a service in terms of a cluster of object-oriented classes. E.g., the PhotoAlbum service could be formed based on the *PhotoAlbum, AddPhotoToAlbum, AlbumManager* and *DeletePhotoFromAlbum* classes.

- A reusable service is the one identified in several SPVs. E.g., the PhotoAlbum service is identified in the two SPVs in our illustrative example.

- *Co-existence together dependency* between classes is used to identify reusable services already reused in several SPVs. E,g., the *PhotoAlbum* and *AddPhotoToAlbum* co-exist together in the two SPVs.

- Object-oriented dependency between classes is used to identify quality-centric services. E.g., the *PhotoAlbum* and *AddPhotoToAlbum* are cohisive based on their method calls and attribute accesses.

- ReSIde analyzes the commonality and the variability between SPVs to identify reusable services.

- Classes composing a reusable service should implement one or more coarse-grained functionalities in several SPVs. E.g., the *PhotoAlbum, AddPhotoToAlbum* and *DeletePhotoFromAlbum* classes implement the cohesive PhotoAlbum service.

- A class can belong to many services since this class could contribute to implement different functionalities by participating with different groups of classes. E.g., the *AlbumManager* class is a part of the PhotoAlbum service (*AlbumManager, PhotoAlbum, AddPhotoToAlbum, DeletePhotoFromAlbum*) and the MusicAlbum service (*AlbumManager, MusicAlbum, AddMusicToAlbum, DeleteMusicFromAlbum*).

- A provided interface of a service is a group of methods that are accessed by classes composing other services.

- A required interface of a service is a set of methods used by classes composing this service and belonging to other services' classes.

3.3. ReSIde process

Based on what we mentioned before, we propose a process presented in Figure 4 to identify reusable services from a set of SPVs. This process consists of four main steps.

1. **Identification of potential services in each SPV.** We analyze each SPV independently to identify all potential services composing each SPV. To identify quality-centric potential services, we rely on object-oriented dependencies between classes to evaluate their quality. We consider that any set of classes could form a potential service if and only if it has an accepted value following the quality fitness function of the quality model presented in Section 2.
   In our illustrative example, we identify the 5 clusters of classes corresponding to potential services in SVP1.

   (a) *PhotoAlbum, AddPhotoToAlbum.*
   (b) *PhotoAlbum, AddPhotoToAlbum, PhotoView.*
   (c) *PhotoListScreen, PhotoViewScreen.*
   (d) *PhotoListScreen, PhotoViewScreen, PhotoView.*
   (e) *PhotoViewScreen, PhotoView.*

   These clusters of classes are obtained based on the strength of the structural dependencies among the classes.
2. **Identification of similar services between different SPVs.** Due to the similarity between the SPVs, their identified potential services could provide similar functionalities. Similar services are those providing mostly the same functionalities and differ compared to few others. Thus, we identify similar services from all potential ones identified from different SPVs. To this end, we cluster the services into groups based on the lexical similarity among classes

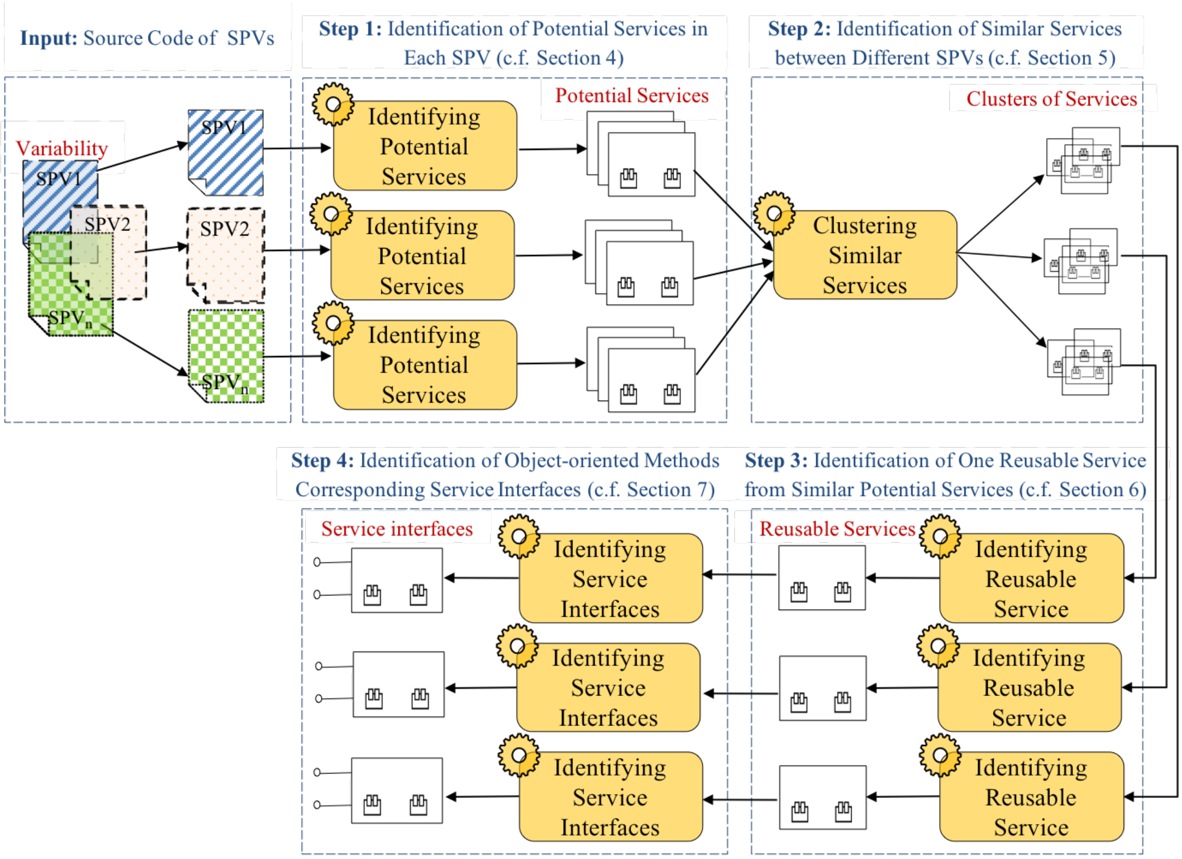

Figure 4: The process of reusable services identification from multiple SPVs

composing the services based on the cosine similarity metric [15].

Considering our illustrative, we identify the cluster of [*PhotoAlbum, AddPhotoToAlbum, PhotoView*] in SPV1 as similar to the cluster of [*AlbumManager, PhotoAlbum, AddPhotoToAlbum, PhotoView*] in SPV2.

3. **Identification of one reusable service from similar potential services.** Similar services identified from different SPVs are considered as variants of one service because they provide mostly the same functionalities. Therefore, from a cluster of similar services, we identify one common service that is representative of this cluster of similar services and it is considered as the most reusable one compared to the members of the analyzed cluster. We rely on the *co-existence together dependencies* and the *structural object-oriented dependencies* to identify the classes composing this common service. *The co-existence together dependencies* are identified based on the percentage of services containing the classes. The *structural object-oriented dependencies* are based on the quality fitness function of the quality model presented in Section 2.

   For example, we identify the group of [*PhotoAlbum, AddPhotoToAlbum, PhotoView*] as implementation of the resuable service from the similar clusters of [*PhotoAlbum, AddPhotoToAlbum, PhotoView*] in SPV1 and [*AlbumManager, PhotoAlbum, AddPhotoToAlbum, PhotoView*] in SPV2.

4. **Identification of object-oriented methods corresponding to service interfaces.** Only classes constituting the internal structures, i.e., the implementation, of the reusable services are identified in the previous steps. However a service is used based on its provided and required interfaces. Thus, we structure service interfaces, required and provided ones, based on the analysis of the dependencies (e.g., method calls, attribute accesses) between services in order to identify how they interact with each others.

4. ReSIde in depth

4.1. Identification of potential services in each software product variant

We view a potential service as a cluster of object-oriented classes, where the corresponding value of the quality fitness function is satisfactory (i.e., its quality value is higher than a predefined quality threshold). Thus, our analysis consists of extracting any set of object-oriented classes that can be formed as a potential service. Such that the overlapping between the services is allowed.

4.1.1. *Method to identify potential services*

Identifying all potential services needs to investigate all subsets of classes that can be formulated from the source code. Then, the ones that maximize the quality fitness function are selected. Nevertheless, this is considered as NP-hard problem as the computation of all subsets requires an exponential time complexity ($O(2^n)$). To this end, we propose a heuristic-based technique that aim to extract services that are good enough ones compared to the optimal potential services. We consider that classes composing a potential service are gradually identified starting from a core class that participates with other classes to contribute functionalities. Thus, each class of the analyzed SPV can be selected to be a core one. Classes having either direct or indirect link with it are candidates to be added to the corresponding service.

4.1.2. *Algorithm to identify potential services*

Algorithm 1 illustrates the process of identifying potential services. In this algorithm, Q refers to the quality fitness function and *Q_threshold* is a predefined quality threshold. The selection of a class to be added at each step is decided based on the quality fitness function value obtained from the formed service. Classes are ranked based on the obtained value of the quality fitness function when it is gathered to the current group composing the service. The class obtaining the highest quality value is selected to extend the current group (c.f. lines 7 and 8). We do this until all candidate classes are grouped into the service (c.f. lines 6 to 11). The quality of the formed groups is evaluated at each step, i.e., each time when a new class is added. We select the peak quality value to decide which classes form the service (c.f. lines 10 and 11). This means that we exclude classes added after the quality fitness function reaches the peak value since they minimize the quality of the identified service. For example, in Figure 5, *the 7th and the 8th added classes* are putted aside from the group of classes related to *service 2* because when they have been added the quality of the service is decreased compared to the peak value. Thus, classes retained in the group are those maximizing the quality of the formed service. After identifying all potential services of such a SPV, the only ones retained are services that their quality values are higher than a quality threshold that is defined by software architects (c.f. lines 12 and 13). For example, in Figure 5, suppose that the predefined quality threshold value is *70%*. Thus, *Service 1* does not reach the required threshold. Therefore, it should not be retained as a potential service. This means that the starting core class is not suitable to form a service.

```
   Input: Object-Oriented Source Code(OO)
   Output: A Set of Potential Services(PS)
1  classes = extractInformation(OO);
2  for each c in classes do
3      service = c;
4      candidateClasses =
         classes.getConnectedClasses(c);
5      bestService = service;
6      while (|candidateClasses| >= 1) do
7          c1 = getNearestClass(service,
             candidateClasses);
8          service = service + c1;
9          candidateClasses = candidateClasses - c1;
10         if Q(service)) > Q(bestService) then
11             bestService = service;
           end
       end
12     if Q(bestService) > Q_threshold then
13         PS = PS + bestService;
       end
   end
14 return PS
```

Algorithm 1: Identifying Potential Services

4.2. *Identification of similar services between different software product variants*

We define similar services as a set of services providing mostly the same functionalities and differing in few ones. These can be considered as variants of the same service.

4.2.1. *Method to identify similar services*

SPVs are usually developed using the clone and own technique. Thus, we consider that classes having similar names implement almost the same functionalities. Although some of the composed methods are overridden, added or deleted, the main functionalities are still the same ones from the architectural point of view. Therefore, the similarity as well as the difference between services are calculated based on the object-oriented classes composing these services. Thus, similar services are those sharing the majority of their classes and differing considering the other ones.

Groups of similar services are built based on a lexical similarity metric. Thus, services are identified as similar compared to the strength of similarity links between classes composing them. A survey of text similarity metrics is conducted in [16]. Practitioners could use any of these similarity metrics based on their needs. For our experimentation, we selected the cosine similarity metric because it is based on the angle between vectors instead of points [15]. Following this cosine similarity metric each service is considered as a text document, which consists of a list of service classes' names. The similarity between a set of services is calculated based on the ration between the number of shared classes to the total number of distinguished classes.

4.2.2. *Algorithm to identify similar services*

We use a hierarchical clustering technique to gather similar services into groups. This hierarchical clustering technique consists of two algorithms. The first algorithm

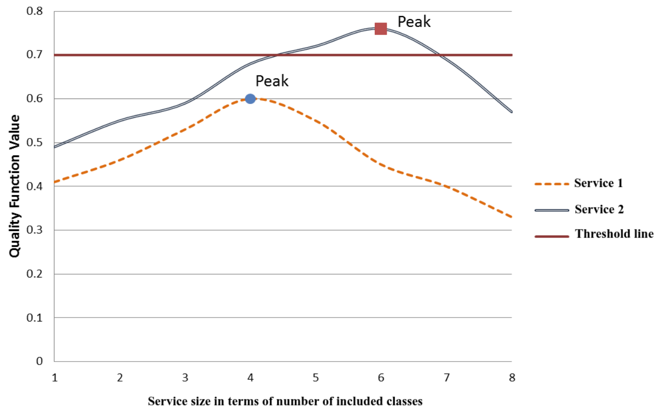

Figure 5: Forming potential services by incremental selection of classes

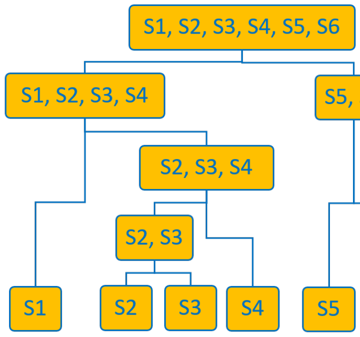

Figure 6: An example of a dendrogram

Input: Potential Services(PS)
Output: Dendrogram ($dendrogram$)
1 Dendrogram $dendrogram = PS$;
2 **while** $(|dendrogram| > 1)$ **do**
3 $c1, c2 =$ mostLexicallySimilarNodes($dendrogram$);
4 $c =$ newNode($c1, c2$);
5 remove($c1, dendrogram$);
6 remove($c2, dendrogram$);
7 add($c, dendrogram$);
end
8 **return** $dendrogram$

Algorithm 2: Building Dendrogram of Similar Services

aims at building a binary tree, called *dendrogram*. This dendrogram provides a set of candidate clusters by presenting a hierarchical representation of service similarity. Figure 6 shows an example of a dendrogram, where S_i refers to $Service_i$. The second algorithm aims at traveling through the built dendrogram, in order to extract the best clusters, representing a partition.

To build a dendrogram of similar services, we rely on Algorithm 2. It takes a set of potential services as an input. The result of this algorithm is a dendrogram representing candidate clusters, similar to Figure 6. The algorithm starts by considering individual services as initial leaf nodes in a binary tree, i.e., the lowest level of the dendrogram in Figure 6 (c.f. line 1). Next, the two most similar nodes are grouped into a new one, i.e., as a parent of them (c.f. lines 3 and 4). For example, in Figure 6, the S_2 and S_3 are grouped. This is continued until all nodes are grouped in the root of the dendrogram (c.f. lines 2 to 7).

To identify the best clusters, we rely on Algorithm 3 that uses a depth first search technique to travel through the dendrogram. It starts from the dendrogram root node to find the cut-off points (i.e., the highest node holding *S1, S2, S3, S4, S5, S6* in Figure 6). It compares the similarity of the current node with its children (c.f. lines 4 to 7). For example, the node holding *S1, S2, S3, S4* and the node holding *S5, S6* in Figure 6. If the current node has a similarity value exceeding the average similarity value of its children, then the cut-off point is in the current node where the children minimize the quality fitness function value (c.f. lines 7 and 8). Otherwise, the algorithm recursively continues through its children (c.f. lines 9 to 11). The results of this algorithm are a collection of clusters, where each cluster groups a set of similar services (c.f. line 12).

Input: Dendrogram($dendrogram$)
Output: A Set of Clusters of Potential Services($clusters$)
1 Stack $traversal$;
2 $traversal$.push($dendrogram$.getRoot());
3 **while** (! $traversal.isEmpty()$) **do**
4 Node $father = traversal$.pop();
5 Node $left = dendrogram$.getLeftSon($father$);
6 Node $right = dendrogram$.getRightSon($father$);
7 **if** $similarity(father) > (similarity(left) + similarity(right)\ /\ 2)$ **then**
8 $clusters$.add($father$)
9 **else**
10 $traversal$.push($left$);
11 $traversal$.push($right$);
end
end
12 **return** $clusters$

Algorithm 3: Dendrogram Traversal to Identify Cluster of Similar Services

4.3. Identification of one reusable service from similar potential services

As previously mentioned, similar services are considered as variants of a common one. Thus, from each cluster of similar services, we extract a common service which is considered as the most reusable compared to the members of the analyzed group.

4.3.1. Method to identify reusable service based on similar ones

Classes composing similar services are classified into two types. The first one consists of classes that are shared by these services. We call these classes as *Shared* classes. In Figure 7, *C3, C4, C8* and *C9* are examples of *Shared* classes in the three services belonging to the cluster of similar services. The second type is composed of other classes that are diversified between the services. These are called as *Non-Shared* classes. *C1, C2, C5, C6, C7* and *C10* are examples of *Non-Shared* classes in the cluster of similar services presented in Figure 7.

As *Shared* classes are identified in several SPVs to be part of one service, we consider that *Shared* classes form the core of the reusable service. Thus, *C3, C4, C8* and *C9* should be included in the service identified from the cluster presented in Figure 7. However, these classes may not form a correct service following our quality fitness function. Thus, some *Non-Shared* classes need to be added to the reusable service, in order to keep the service quality high. The selection of a *Non-Shared* class to be included in the service is based on the following criteria:

- The quality of the service obtained by adding a *Non-Shared* class to the core ones. This criterion is to increase the service quality. Therefore, classes maximizing the quality fitness function value are more preferable to be added to the service.
- The density of a *Non-Shared* class in a cluster of similar services. This refers to the occurrence ratio of the class compared to the services of this group. It is calculated based on the number of services including the class to the total number of services composing the cluster. We consider that a class having a high density value contributes to build a reusable service because it keeps the service belonging to a larger number of SPVs. For example, in Figure 7, the densities of *C2* and *C1* are respectively *66% (2/3)* and *33% (1/3)*. Thus, *C2* is more preferable to be included in the service than *C1*, as *C2* keeps the reusable service belonging to two SPVs, while *C1* keeps it belonging only to one SPV.

As results of the clone-and-own approach, classes of identified services could have different implementations across various SPVs. These different implementations of the same cloned class across SPVs should be merged by creating one suitable and representative abstraction that allows the variability configuration, e.g., using the preprocessing annotations. In literature, we identify potential approaches to be reused for merging the several implementations of cloned methods/classes [17] [18].

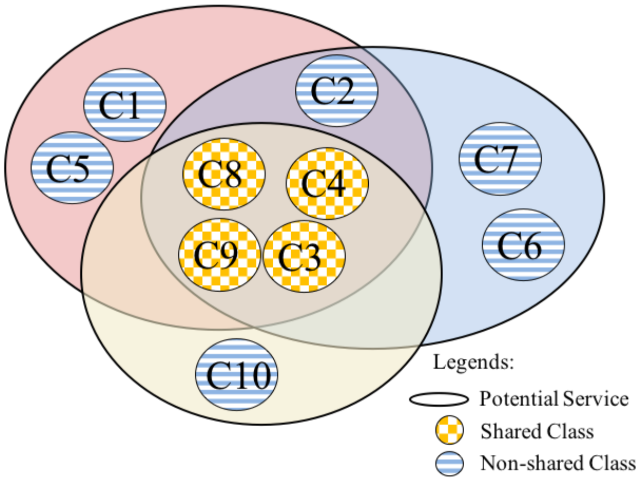

Figure 7: An example of a cluster of three similar services

4.3.2. Algorithm to identify reusable service based on similar ones

Based on the method given in the previous section, an optimal solution requires identifying all subsets of a collection of classes which represents an NP-complete problem (i.e., $O(2^n)$). This algorithm is not scalable for a large number of *Non-Shared* classes (e.g., *10 Non-Shared* classes need *1024* operations, while *20* classes need *1048576* operations).

Therefore, we propose to identify the optimal solution only for services with a small number of *Non-Shared* classes. Otherwise, we rely on a near-optimal solution. In the following subsections, we discuss two algorithms to identify an optimal solution and a near-optimal one respectively.

Algorithm providing optimal solution for reusable service identification. Algorithm 4 computes an optimal reusable service from similar ones, where Q refers to the service quality fitness function, *Q_threshold* refers to the predefined quality threshold and *D_threshold* refers to the predefined density threshold. First, for each cluster of similar services, we extract all candidate subsets of classes among the set of *Non-Shared* ones (c.f. lines 1 to 8). Then, the subsets that reach a predefined density threshold are only selected (c.f. line 12). The density of a subset is the average densities of all classes in this subset. Next, we evaluate the quality of the service formed by grouping core classes with classes of each subset resulting from the previous step (c.f. lines 13 and 14). Thus, the subset maximizing the quality value is grouped with the core classes to form the reusable service. Only services with a quality value higher than a predefined threshold are retained (c.f. lines 15 to 17).

Algorithm providing near-optimal solution for reusable service identification. We defined a heuristic algorithm pre-

```
Input: Clusters of Services(clusters)
Output: A Set of Reusable Services(RC)
for each cluster ∈ clusters do
    shared = cluster.getFirstservice().getClasses;
    allClasses = ∅;
    for each service ∈ cluster do
        shared = shared ∩ service.getClasses();
        allClasses = allClasses ∪ service.getClasses();
    end
    nonShared = allClasses − shared;
    allSubsets = generateAllsubsets(nonShared);
    reusableService = shared;
    bestService = reusableService;
    for each subset ∈ allSubsets do
        if Density(subset) > D_threshold then
            if Q(reusableService ∪ subset)) > Q(bestService) then
                bestService = reusableService ∪ subset;
            end
        end
    end
    if Q(bestService) >= Q_threshold then
        add(RC,bestService);
    end
end
return RC
```

Algorithm 4: Optimal Solution for Reusable Service Identification

```
   Input: Clusters of Services(clusters)
   Output: A Set of Reusable Services(RC)
1  for each cluster ∈ clusters do
2      shared = cluster.getFirstservice().getClasses;
3      allClasses = ∅;
4      for each service ∈ cluster do
5          shared = shared ∩ service.getClasses();
6          allClasses = allClasses ∪ service.getClasses();
       end
7      nonShared = allClasses − shared;
8      reusableService = shared;
9      for each class ∈ nonShared do
10         if Density(class) < D_threshold then
11             nonShared = nonShared - class;
           end
       end
12     while (|nonShare| > 0) do
13         if Q(reusableService ∪ nonShare) >= Q_threshold then
14             add(RC,reusableservice);
15             break;
16         else
17             removeLessQualityClass(nonShare, shared);
           end
       end
   end
18 return RC
```

Algorithm 5: Near-Optimal Solution for Reusable Service Identification

sented in Algorithm 5, where Q refers to the quality fitness function, *Q_threshold* refers to the predefined quality threshold and *D_threshold* refers to the predefined density threshold. First of all, *Non-Shared* classes are evaluated based on their density. The Classes that do not reach a predefined density threshold are rejected (c.f. lines 9 to 11). Then, we identify the greater subset that reaches a predefined quality threshold when it is added to the core classes. To identify the greater subset, we consider the set composed of all *Non-Shared* classes as the initial one (c.f. lines 9 to 11). This subset is grouped with the core classes to form a service. If this service reaches the predefined quality threshold, then it represents the reusable service (c.f. lines 12 to 15). Otherwise, we remove the *Non-Shared* class that reduces the quality of the service when this *Non-Shared* class is added to the corresponding core classes (c.f. line 17). We do this until a service reaching the quality threshold or the subset of *Non-Shared* classes becomes empty (c.f. line 12).

4.4. Identification of object-oriented methods corresponding to service interfaces

A service is used based on its provided and required interfaces. For object-oriented services, the interaction between the services is realized through object-oriented method calls (i.e., method invocations). A service provides its services through a set of object-oriented methods that can be called by the other services that require functionalities of this service. Thus, the provided interfaces are composed of a set of public methods that are implemented by classes composing this service. On the other hand, required interfaces are composed of methods that are invoked by classes of this service and belong to classes of other services (i.e., the provided interfaces of the other services). The identification of service interfaces is based on grouping a set of object-oriented methods into a set of service interfaces. We rely on the following heuristics to identify these interfaces:

Object-oriented methods belonging to the same object-oriented entities. In object-oriented, methods implementing cohesive functionalities are generally implemented by the same object-oriented entities (e.g., object-oriented interface, abstract class and concrete class). Therefore, we consider any object-oriented entity that groups together a set of methods as an indicator of high probability that these methods belong to the same service interface. We propose

Algorithm 6 to measure how much a set of methods M belongs to the same service interface. This algorithm calculates the size of the greatest subset of M which consists of methods that belong to the same object-oriented class or interface (c.f. lines 1 to 4). Then, it divides the size of the greatest subset by the size of M (c.f. line 5) and returns SI as a final return value (c.f. line 6).

Input: A Set of Methods(M), a Set of Object-Oriented Entities(OOI)
Output: Same Object-Oriented Entity Value (SI)
1 $sizeGreatest = |M \cap OOI.\text{getFirstInterface().getMethods()}|$;
2 **for** *each* $interface \in OOI$ **do**
3   **if** $|M \cap interface.getMethods()| > sizeGreatest$ **then**
4         $sizeGreatest = |M \cap interface.\text{getMethods()}|$;
    end
end
5 $SI = sizeGreatest \ / \ M.\text{size()}$;
6 **return** SI

Algorithm 6: Same Object-Oriented Entity (SI)

Object-oriented method cohesion. Methods access the same set of attributes to participate to provide the same services. Thus, cohesive methods have more probability to belong the same service interface than those that are not. To measure how much a set of methods is cohesive, we use the *Loose Class Cohesion (LCC)* metric [14]. We select LCC because it measures direct and indirect dependencies between methods. Please refer to [14] for more details about LCC.

Method lexical similarity. The lexical similarity of methods probably indicates to similar implemented services. Therefore, methods having a lexical similarity likely belong to the same interface. To this end, we utilize *Conceptual Coupling* metric [19] to measure methods lexical similarity based on the semantic information obtained from the source code, encoded in identifiers and comments.

Method correlation of usage: when a service provides functionalities for another service, it provides them through the same object-oriented entities (e.g., object-oriented interface, abstract class and concrete class). Thus, methods that have got called together by object-oriented classes the other services are likely to belong to the same service interface. To this end, we propose Algorithm 7 to calculate the Correlation of Usage (CU) of a given set of methods M. It is based on the size of the greatest subset of M that has got called together by the same service (c.f, lines 2 to 4). The final value of CU is the percentage between the identified size of the greatest subset and the size of M (c.f, line 5).

Input: A Set of Methods(M), a Set of services($Services$)
Output: Correlation of Usage Value(CU)
1 $sizeGreatest = |M \cap Services.\text{getFirstservice().getCalledMethods()}|$;
2 **for** *each* $service \in Services$ **do**
3   **if** $|M \cap service.getCalledMethods()| > sizeGreatest$ **then**
4         $sizeGreatest = |M \cap interface.\text{getCalledMethods()}|$;
    end
end
5 $CU = sizeGreatest \ / \ M.\text{size()}$;
6 **return** CU

Algorithm 7: Correlation of Usage (CU)

According to these heuristics, we define a fitness function for measuring the quality of a group of methods M to form a service interface. We rely on a set of parameters (i.e., λ_i) to allow architects to weight each characteristic as needed. The values of these parameters are situated in [0-1]. The selection of values of these parameters is based on the knowledge of architects about the SPVs. Furthermore, architects could use these parameters to analyze the relationships between each characteristic and the quality of the obtained service interfaces by changing the values of parameters. Once architects identify the best values based on a set of test cases of service interfaces, they could generalize these values to the remaining of the SPVs in the same family.

$$Interface(M) = \frac{1}{\sum_i \lambda_i} \cdot (\lambda_1 \cdot SI(M) + \lambda_2 \cdot LCC(M) + \lambda_3 \cdot CS(M) + \lambda_4 \cdot CU(M)) \quad (5)$$

Based on this fitness function, we use a hierarchical clustering technique to partition a set of public methods into a set of clusters, where each cluster is considered as a service interface. The hierarchical clustering technique constructs a dendrogram of similar public methods like Algorithm 2. Then, it extracts the best clusters of public methods using a depth first search technique similar to Algorithm 3.

5. Evaluation

5.1. Data collection

To evaluate ReSIde, we collect three sets of software product families that are Mobile Media[2] [20], Health Watcher[3], and ArgoUML[4] [21].

[2]Available at http://homepages.dcc.ufmg.br/∼figueiredo/spl/icse08
[3]Available at http://ptolemy.cs.iastate.edu/design-study/#healthwatcher
[4]Available at http://argouml-spl.tigris.org/

Mobile Media (MM) is a SPL that manipulates music, video and photo on mobile phones. It is implemented using Java. In our experimentation, we considered, as SPVs, a set of *8* products derived by [20] as representative of all features of the SPL. The average size of a product is *43.25* classes. Health Watcher product variants implement a set of web-based software applications that offers services related to managing health records and customer complaints. We consider 10 SPVs written in Java. On average, each SPV is composed of *137.6* classes. ArgoUML (AL) is a UML modeling tool. It is developed in Java as a software product line. We applied ReSIde on *9* products generated and used in [22]. Each SPV contains *2198.11* classes on average.

Our method to select these software families is based on four factors. First, we consider covering different sizes of software families to test the scalability of ReSIde with different system sizes; MM as a small-scale software (43.25 classes per SPV), HW as medium-scale software (137.6 classes per SPV), and AL as a large-scale one (2198.11 classes per SPV). Second, we consider software families that were already used by other researchers in the domain of reverse engineering of software product lines such as [23, 22, 24, 25]. Third, the suitability of the case studies to identify services that were reused in the implementation of several SPVs. Fourth, the availability of their source code.

5.2. Research questions and their methodologies

We aim to answer four Research Questions (RQs) as follows.

5.2.1. RQ1: What are good threshold values to identify potential services from each SPV?

Goal. As the selection of threshold values affects both the quality and the number of the identified potential services, the aim of this RQ is to help software architects selecting proper threshold values to consider a group of classes forming a potential service or not.

Methodology. To support software architects choosing a proper threshold value, we assign the quality threshold values situated in *[0%, 100%]*. The goal of changing the threshold values is to explicitly identify the general relationship between the number of identified services and the selected threshold values. To do this, we need to explore all the values of the interval [0%-100%]. To segment this interval, we can start from 0% and increment using any value (1%, 1.55%, 5%, 8,09%, 23%, etc.). We rely on two strategies applied successively.
The first strategy is to explore the threshold values based on a 5% increment. We consider that 5% is a fair empirical increment for two reasons. First it provides a finite number of values that are distributed uniformly compared to this interval (i.e. 0%, 5%, 10%, 15%... 100%). Second, the variation of values obtained based on successive increments allows the interpolation of other unconsidered values. As soon as we identify an interesting interval based on the number of identified services and the maximum value of the quality fitness function, we apply *the second strategy* which consists of exploring the values in this interval by a finer increment which is 1. We show the impact of threshold values on the average number of identified services for each software family of SPVs.

5.2.2. RQ2: What potential services implement similar functionalities across different SPVs?

Goal. The goal of this RQ is to study the characteristics of potential services identified as similar across SPVs.

Methodology. We applied the second step of ReSide to cluster similar potential services based a hierarchical clustering technique. For each case study, we identify the number of clusters, the average number of services in the identified clusters, the average number of *Shared* classes in these clusters, the average value of the *Functionality* characteristic, the average value of the *Self-containment* characteristic, and the average value of *Composability* characteristic of the *Shared* classes in these clusters.

5.2.3. RQ3: What are the reusable services identified based on ReSide?

Goal. The aim of this RQ is to to study the characteristics of reusable services identified from clusters of similar potential services.

Methodology. We rely on the third step of ReSide to extract one reusable service from each cluster of potential services. For each case study, we identify the number of the identified services, the average service size in terms of number of included classes, and the average value of the *Functionality*, the *Self-containment*, and the *Composability* of the identified services.

5.2.4. RQ4: What is the reusability of services identified based on ReSide?

Goal. This RQ evaluates the improvement of the reusability of services identified based on the analysis several SPVs using ReSide compared to the reusability of services identified based on the analysis of singular software.

Methodology. To validate the reusability of services identified by the ReSIde approach, we propose a validation process presented in Figure 8. This process consists of three main steps as follows:

1. **Dividing SPVs into K parts:** To prove that our validation can be generalized for other independent SPVs, we depend on *K-fold* cross validation method [15]. In data mining, *K-fold* is widely used to validate the results of a mining model. The main idea is to evaluate the model using an independent data set. Thus, K-fold divides the data set into two parts: train data, and test data. On the one hand, train data are used to learn the mining model. On the

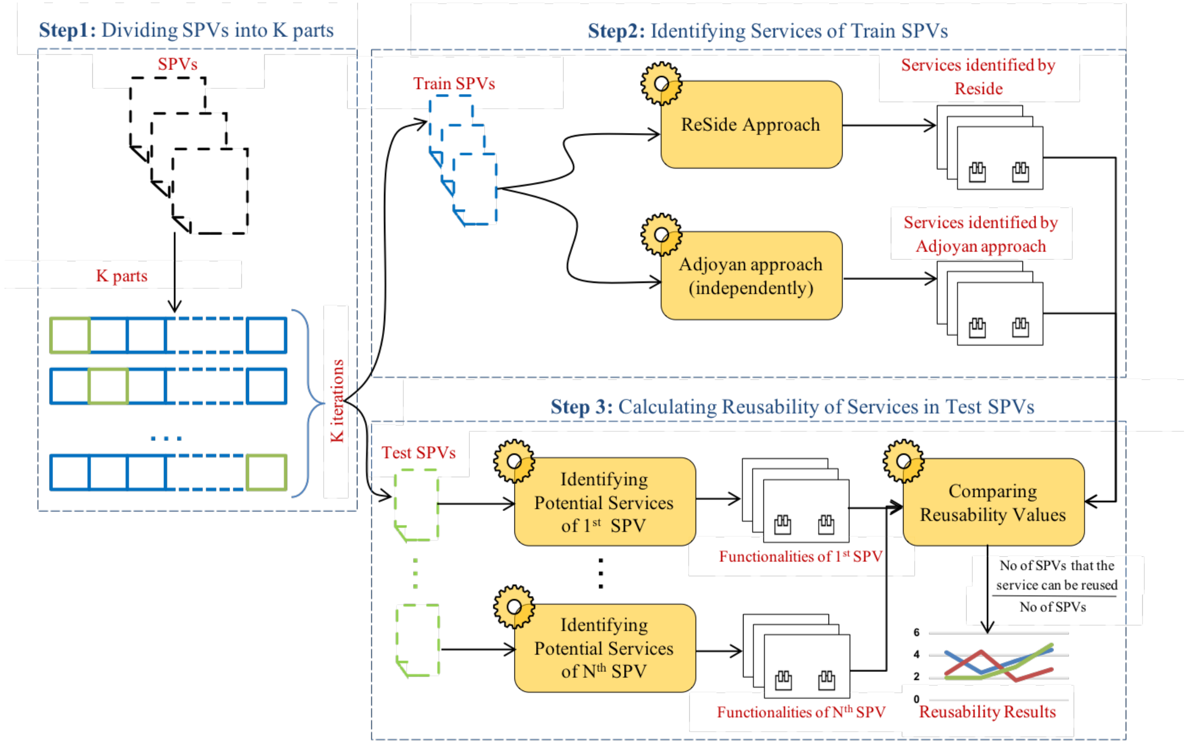

Figure 8: The process of validating the reusability of services identified by ReSIde

other hand, test data are then used to validate the mining model. To do so, K-fold divides the data set into K parts. The validation is applied K times by considering K-1 parts as train data and the other one as test data. We validate ReSIde by dividing the SPVs into K parts. Then, we only identify services from the train SPVs (i.e., *K-1* parts). Next, we validate the reusability of these services in the test SPVs. We evaluate the result by assigning *2, 4* and *8* to the K at each run of the validation.

2. **Identifying services of train SPVs:** To compare the reusability of services identified based on the analysis of multiple SPVs versus singular SPV, we identified services using the ReSIde approach and a traditional service identification approach that analyzes only singular SPVs independently. We selected Adjoyan et al. approach [6] due to the availability of the tool of its implementation.

3. **Calculating the reusability of services in test SPVs:** We consider that the reusability of a service is evaluated based on the number of SPVs that the service can be reused in. For a collection of SPVs, the reusability is calculated as the ratio between the number of SPVs that can reuse the service to the total number of SPVs in the test part. A service can be reused in a SPV if it provides functionalities required by this SPV. We analyze the functionalities of each SPV in the test part to check if an identified service provides some of these functionalities. The functionalities required by a SPV are identified based on the potential services extracted from this SPV using the first step of ReSIde in Section 4.3. The validation results are calculated based on the average of all K trails.

5.3. Results

5.3.1. RQ1: What are good threshold values to identify potential services from each SPV?

The results obtained from MM, HW and AL case studies are respectively shown in Figure 9, Figure 10 and Figure 11, where the values of the threshold are at the X-axis, and the average numbers of the identified services in a SPV are at the Y-axis.

The results show that the number of the identified services is lower than the number of classes composing the SPVs, for low threshold values. The reason behind that is the fact that some of the investigated classes produce the same service. For example, *InvalidPhotoAlbumNameException* and *InvalidImageFormatException* produce the same service, when they are considered as the core for identifying potential services.

Moreover, the results show that the number of identified services is the same for all quality threshold values in this interval *[0%, 55%]* for the three case studies. This means that selecting a value in this interval does not make sense as it does not make a distinction between services having diverse quality values.

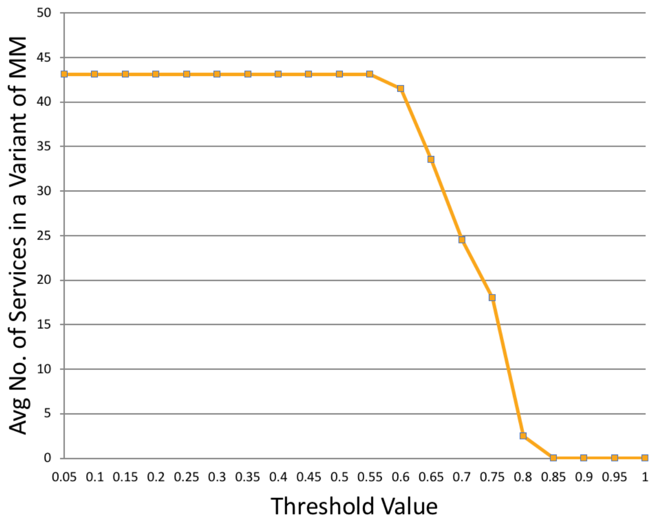

Figure 9: Changing threshold value to extract potential services from MM

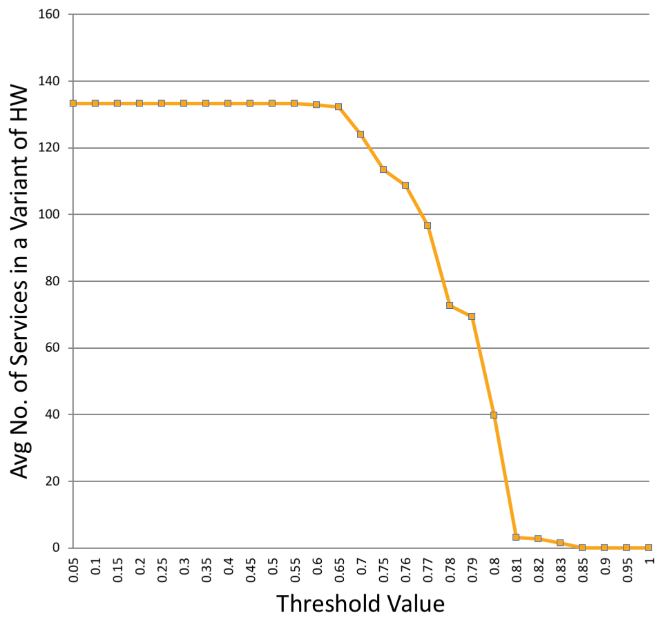

Figure 10: Changing threshold value to extract potential services from HW

The results of incrementing 5% each time allow us to identify the interesting intervals as [65%, 70%], [75%, 80%] and [80%, 85%] respectively for MM, HW and AL case studies. Thus, any value in these intervals can be selected as a threshold to be considered respectively for each case study. We rely on the number of functionalities of the analyzed SPVs to select threshold values in these intervals. We use the number of classes of SPVs as indicators for the number of functionalities implemented in the SPVs (direct proportion). We assign 70%, 77% and 83% as threshold values respectively for MM, HW and AL case studies. Table 1 shows the detail results obtained based on these threshold values. It presents the total number of potential services *(TNOPS)* identified based on the analysis of all SPVs, the average size of these services *(ASOS)* in terms on number of included classes, the average value of the *Functionality* characteristic *(AF)*, the average value of the *Self-containment* characteristic *(ASC)* and the average value of the *Composability* characteristic *(AC)*.

Table 1: The results of potential services extraction

Family Name	TNOPS	ASOS	AF	ASC	AC
MM	24.50	6.45	0.56	0.71	0.83
HW	96.6	5.55	0.61	0.76	0.99
AL	811	11.38	0.64	0.83	0.89

TNOPS: total number of potential services.
ASOS: average size of potential services in classes.
AF: average value of the *Functionality* characteristic of potential services.
ASC: average value of the *Self-containment* characteristic of potential services.
AC: average value of the *Composability* characteristic of potential services.

Figure 12 presents an example of a potential service extracted from AL. This service is identified by considering *GoClassToNavigableClass* as the core class. The quality fitness function reaches the peak value when we add 18 classes to this identified service. We find that classes added later reduce the quality of the identified service. Therefore, we reject classes added after the 18th class to be part of this identified service.

5.3.2. RQ2: What potential services implement similar functionalities across different SPVs?

Table 2 presents the results of the process of grouping similar potential services into clusters. For each case study, it shows the number of clusters *(NOC)*, the average number of services in the identified clusters *(ANOC)*, the average number of *Shared* classes in these clusters *(ANSC)*, the average value of the *Functionality* characteristic *(AFS)*, the average value of the *Self-containment* characteristic *(ASCS)*, and the average value of *Composability* characteristic of the *Shared* classes *(ACS)* in these clusters. The results show that SPVs sharing a bunch of similar services. For instance, each SPV of MM has *24.5* services in average. These services are grouped into *42* clusters. This means that each SPV shares *5.38* services with the other SPVs, in average. Thus, a reusable service can be identified from these services. In the same way, AL SPVs share *5.26* services. Table 3 shows an example of a cluster of similar services identified from AL case study, where *X* refers to that a class is a member in the corresponding SPV. In this example, we note that the services have *5 Shared* classes. These classes have been identified to be part of the same service in *9* SPVs of AL. Thus, they can be considered as core classes to form a reusable service that is reused in the *9* SPVs.

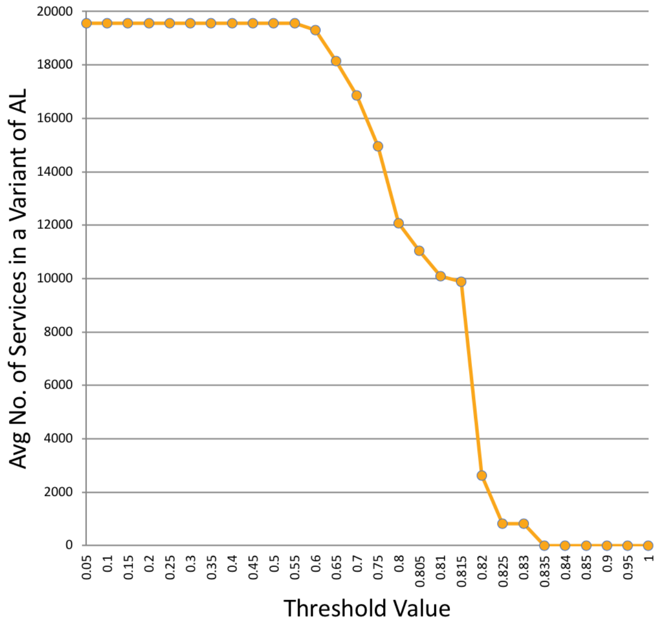

Figure 11: Changing threshold value to extract potential services from AL

Table 2: The results of service clustering

Family Name	NOC	ANOC	ANSC	AFS	ASCS	ACS
MM	42	5.38	5.04	0.59	0.71	0.89
HW	504	6.17	5.33	0.62	0.74	0.99
AL	325	5.26	8.67	0.57	0.87	0.93

NOC: the number of clusters.
ANOC: the average number of services in the identified clusters.
ANSC: the average number of *Shared* classes in these clusters.
AFS: the average value of the *Functionality* of the *Shared* classes in the identified clusters.
ASCS: the average value of the *Self-containment* of the *Shared* classes in the identified clusters.
ACS: the average value of *Composability* of the *Shared* classes in the identified clusters.

5.3.3. RQ3: What are the reusable services identified based on ReSide?

Table 4 summarizes the final set of reusable services identified using ReSIde. Based on our experimentation, we assign *50%* to the density threshold value. For each product family (i.e., a set of SPVs), we present the number of the identified services *(NOIS)*, the average service size in terms of number of included classes *(ASS)*, and the average value of the *Functionality (AF)*, the *Self-containment (ASC)*, and the *Composability (AC)* of the identified services. The results show that some of the identified clusters do not produce reusable services. For instance, in Mobile Media, the *42* clusters produce only *39* services. This means that three of the clusters are not able to form reusable services. The reason behind that is one of the following two situations. The first one is that the selection of threshold density causes to remove classes that are important to constitute the service, and hence, the service was rejected because it did not exceed the quality threshold value. The second one is that the produced service is already identified from another cluster, therefore, the service is removed to avoid the redundancy.

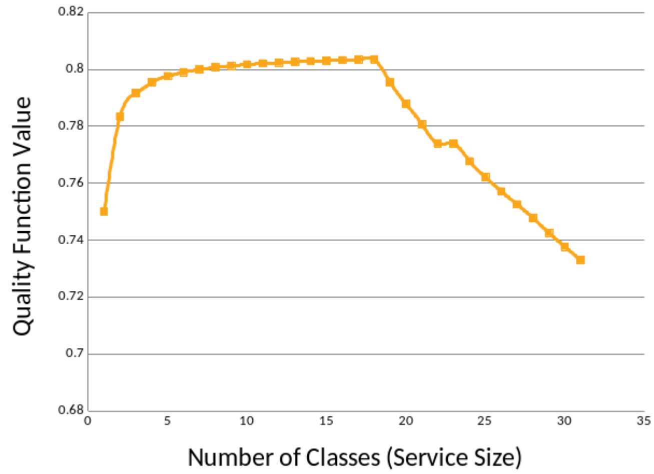

Figure 12: An instance of a potential service extracted from AL case study

Table 3: An instance of a cluster of similar potential services in AL

Class Name \ SPV No.	1	2	3	4	5	6	7	8	9
ArgoEventTypes	X	X	X	X	X	X	X	X	X
JWindow	X	X	X	X	X	X	X	X	X
TabFigTarget	X	X	X	X	X	X	X	X	X
FileConstants	X	X	X	X	X	X	X	X	X
OclAPIModelInterpreter	X	X	X	X	X	X	X	X	X
StreamSource	X				X	X	X		X
SortedListModel	X		X		X	X	X		
BooleanSelection2		X	X	X				X	

Table 4: The final set of identified reusable services.

Family Name	NOIS	ASS	AF	ASC	AC
MM	39	5.61	0.58	0.74	0.90
HW	443	6.90	0.63	0.75	0.99
AL	324	9.77	0.61	0.84	0.84

NOIS: the number of the identified reusable services.
ASS: the average size of identified reusable services in terms of number of included classes.
AF: the average value of the *Functionality* of identified reusable services.
ASC: the average value of the *Self-containment* of identified reusable services.
AC: the average value of the *Composability* of identified reusable services.

Table 5 shows examples of a set of reusable services that are identified based on the analysis of Mobile Media. Where, *NOV* refers to the number of SPVs that contain the service, *NOC* represents the number of classes that form the service. *S, A* and *C* respectively represent the *Functionality*, the *Self-containment*, and the *Composability* of each service. As it is shown in Table 5, the second service provides two functionalities, which are *Add Constants Photo Album*, and *Count Software Splash Down Screen*. The former one deals with adding a photo to an album. The letter is dedicated to the splash screen service.

Table 5: Some services

Description of the functionalities	NOV	NOC	S	A	C
New Constants Screen Album Image	6	6	0.59	0.75	0.94
Add Constants Photo Album Count Software Splash Down Screen	8	10	0.57	0.75	0.89
Base Image Constants Album Screen Accessor List Controller Image Interface Thread	6	9	0.67	0.50	0.85

NOV: the number of SPVs that contain the services.
NOC: the number of classes that form the services.
F: the value of the *Functionality* characteristic of the services.
S: the value of the *Self-containment*characteristic of the services.
C: the value of the *Composability* characteristic of the services.

5.3.4. RQ4: What is the reusability of services identified based on ReSide?

The results obtained from MM, HW and AL case studies are respectively presented in Figure 13, Figure 14 and Figure 15. These results show that the reusability of the services which are identified from a collection of similar software is better than the reusability of services which is identified from singular software. We note that the reusability is increased when the number of *K* is increased. The reason is that the number of train SPVs is increased compared to the test SPVs. For example, there is only one test SPV when *K=8*. We note that the difference between the reusability results of the two approaches is increased as well as the number of train SPVs is increased.

The slight difference between the reusability results for small K comes from the nature of our case studies, where these case studies are very similar. Consequently, the resulting services are closely similar. In other words, there are many groups of similar services containing exactly the same classes. This yields a reusable service that is identical to cluster services. Therefore, the reusability has the same value for all of these services. However, ReSIde remains outperforming the traditional service identification approach proposed by Adjoyan et al. [6]. In Table 5.3.4, we provide a conceptional comparison between the ReSide and Adjoyan approaches based on seven attributes.

6. Discussion

6.1. Deployment of the identified services

ReSide currently reverse engineers the structural implementation of reusable services in terms of groups of object-oriented classes. To complete the reengineering to SOA, these groups of classes need to be transformed and packaged based on existing service-oriented models. Therefore, we plan in our near future work to extend ReSide where reusable web services (e.g., generate *WSDL* files) and REST services can be generated from these groups of clusters identified in this paper. In this context, we need to deal with direct dependencies between different services, exception handling of Java programs and the instantiation of services.

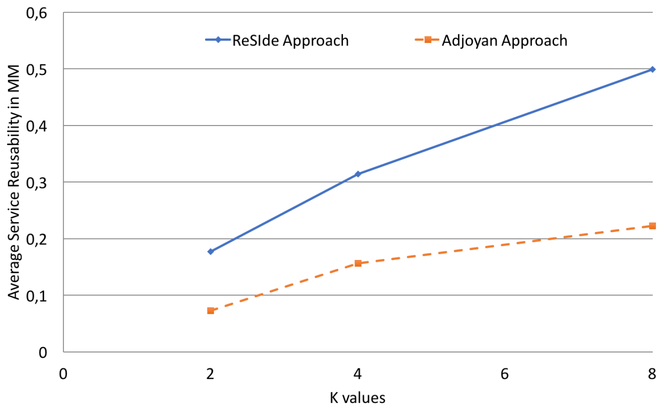

Figure 13: The results of reusability validation of MM services

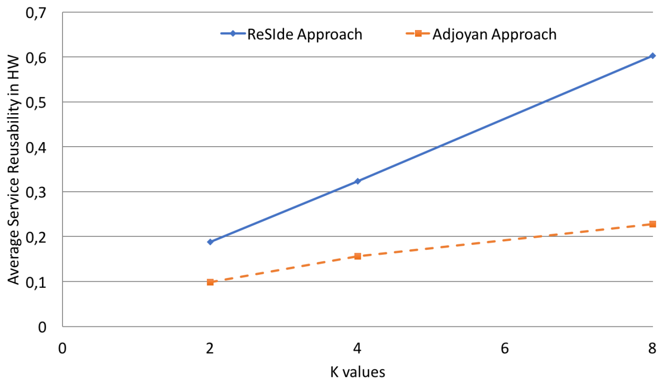

Figure 14: The results of reusability validation of HW services

6.2. The adaption of our approach for SPVs that already applied SOA

Although our approach is designed for object-oriented SPVs, it can be adapted for SPVs that already applied SOA from the start. This adaptation is based on ignoring the first step of our approach (i.e., Identification of Potential Services in Each SPV) where we identify a set of services from the object-oriented implementation of each SPV. This means that we provide as input for the second step of our approach the already implemented services corresponding to the SOA of SPVs.

6.3. Threats to validity

ReSIde is concerned by two types of threats to validity. These are internal and external.

6.3.1. Threats to internal validity

There are three aspects to be considered regarding the internal validity. These are as follows.

Table 6: Comparisons between ReSide and Adjoyan's approaches

Attributes	ReSide approach	Adjoyan's approach
Goal	Service identification	Service identification
Input artifacts	Multiple software product variants	Single software product
Target development paradigm	Object-oriented	Object-oriented
Quality metrics	Structural and co-existence together dependencies	Structural dependencies
Used Algorithms	Authors' defined heuristic and clustering algorithms	Clustering algorithm
Service interface identification	Yes	No
Output	Clusters of classing corresponding to services	Clusters of classing corresponding to services

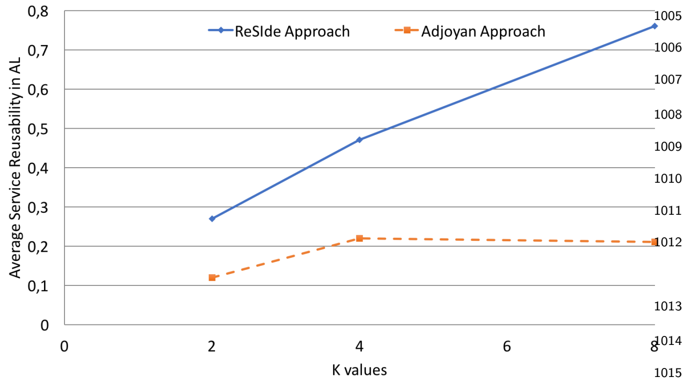

Figure 15: The results of reusability validation of AL services

1. The input we used to evaluate our approach are Abstract Syntax Trees (ASTs) of the analyzed SVPs. These ASTs obtained based on Eclipse Java Development Tools[5] (JDT) API reflect static dependencies between source code entities. This means that all dependencies in the source code will be equally considered regardless if they are really existed or not compared to the program execution scenarios. As dependencies between source code entities are used to compute the value of the fitness function that evaluates the quality of clusters of classes to form services, then the precision of the identified services can be impacted. I.e., the fitness function may consider some dependencies that are not materialized in the program execution scenarios.
   Also, the ASTs do not consider polymorphism and dynamic binding. Consequently some dependencies are not captured (e.g., Java reflection dependencies). This impacts the recall of the identified services.
2. We use a hierarchical clustering algorithm to group similar services. We use this hierarchical clustering algorithm because it does not need to specify the number of clusters in advance as we do not know the number of services to be identified in advance. However, it provides a near optimal solution of the partitioning. Other grouping techniques may provide more accurate solutions, such as search-based algorithms. This will be a future extension of ReSIde to implement simulated annealing and genetic algorithms.
3. Due to the lack of models that measure the reusability of object-oriented services, we propose our own empirical measurement to validate the reusability of the identified services. This can threat the reusability validation results.

6.3.2. Threats to external validity

There are two aspects to be considered regarding the external validity. These are as follows:

1. ReSIde is experimented via SPVs that are implemented by *Java*. As other object-oriented languages (e.g., *C++*, *C#*) include other concepts than Java (e.g., templates and preprocessor directives in C++), we need to develop new parsers to handle these new concepts properly to allow ReSide to work with these other languages.
2. Only three case studies have been collected in the experimentation (Mobile Media, Health Watcher and ArgoUML). However these are used in several research papers that address the problem of migrating SPVs into software product line. On average, the selected case studies obtained the same results. We do not claim that our results can be generalized for other similar case studies without testing ReSIde with a large number of case studies. This will be a logical extension of our work.

6.4. Research implications

As research implication, we find that services identified based on the analysis of multiple SPVs are more reusable than ones identified from singular software. We can generalize this conclusion to all domain of software reuse. E.g., when a collection of SPVs is available, it will be suitable to recover reusable entities (e.g., components, microservices, modules) by analyzing commonality and variability across these SPVs.

6.5. Practical implications

As it is mentioned in the motivation of the paper, SOA improves the management of reuse and maintenance of cloned SPVs. However, it provides limited customizability of services as they are used as black-boxes. Furthermore,

[5]https://www.eclipse.org/jdt/

the performance of the identified services may be negatively impacted as the service technologies add communication layers between services. This impact can be amplified if the identified services rely on heavy data exchange. The security challenge maybe emerged if the identified services are going to be accessed by third-party applications.

7. Related work

In this section, we discuss four research areas crosscutting with ReSIde. These are service identification, component identification, software product line architecture recovery and feature identification research areas. We decide to also include the latter three research areas because we find that they share with service identification similar input artifacts (e.g., source code) and technical analysis processes (e.g., reverse engineering, clustering algorithm), but with different conceptual identification goals (i.e., service vs component vs feature).

7.1. Service identification

Several service identification approaches have been proposed to identify services based on the analysis of object-oriented software [6, 7, 8, 9]. According to the life cycle of service identification approaches, we classify approaches presented in the literature based on four axes; the goal, the input, the applied process and the resulting output of service identification approaches. In our classification, we select approaches based on two criteria. The first one focuses on the approaches that are frequently cited since they are considered as the most known approaches presented in the state-of-the-art. The second one is related to the comprehension of the classification axes. This means that we select approaches that cover all of the classification axes to give concrete examples of these classification axes.

The goal of an identification approach can be: understanding, reuse, construction, evolution, analysis or management [26]. Software understanding is supported by providing a high level of abstraction describing the system structure. Reuse is supported by providing a coarse-grain software entities that can be easily decoupled from the system and deployed in another one. Construction is guided by explaining how software components interact with each other through their interfaces. A better comprehension of the outcome changes is provided to software maintainers. Thus, they can be more precise in estimating cost of modifications of the software evolution. Software analysis is enriched by understanding the dependencies provided by software architectures. Managing the development tasks get success, when a clear view of the system structure is provided [26].

The input of a service identification approach can be source codes [6, 27, 7, 28, 29], data bases [30, 31, 32], execution and log traces [33, 34, 35], business process models [36, 37], knowledge of human expertises [27, 38], documentations [39, 40, 41, 42] or a combination of these input artifacts [29, 43, 33].

The process of service identification approaches aims to cluster elements of input artifacts into services. Existing approaches uses several algorithm including *clustering algorithm*, *genetic algorithm* [27, 44], *Formal Concept Analysis* [45, 32, 46] or *user-defined* heuristics [27, 8]. These algorithms rely on various quality characteristics in their fitness functions to maximize the service quality of identified clusters. Such quality characteristics are *loose coupling* [6, 27, 28, 47, 40, 43, 48, 49, 41], *cohesion* [6, 27, 28, 47, 40, 43, 48, 49, 37], *service granularity level* [47, 40, 41, 43, 48, 49], *self-containment* [6, 41],*composability* [6] and *interoperability* [42].

The output of these service identification approaches is normally clusters of classes where each cluster represents the implementation of one service. Some approaches propose to package these clusters to form web service [6], REST services [7, 50] or microservices [8, 51, 49].

Nevertheless all of these existing service identification approaches perform the identification based on the analysis of only one single software product. Services are identified as group of object-oriented elements that have strong object-oriented dependencies without considering the global reusability of these elements together in other software products. Therefore, the identified services may be useless in other software products and consequently their reusability is not guaranteed. In our evaluation results where we compare our approach with a traditional one, we find that the reusability of services identified by considering multiple software products outperforms the reusability of services identified using the traditional service identification approaches.

7.2. Component identification

Following the definitions of services [52] [53] [54] [55] [56] and software components [57][6], [58][7] [59][8], we find that services are very similar to software components in terms of their characteristics (*loose coupling*, *reusability*, *autonomy*, *composability*, etc.). However, we can distinguish between services and components based on two aspects: the granularity level and the deployment technologies and models. Services start at higher level of abstractions compared to components. A service can be a part of a business process (at the requirement level), an architectural element (at the design level) and a function (at the implementation level). Components appear only at the design level and the implementation level in terms of

[6]A component is "abstract, self-contained packages of functionality performing a specific business function within a technology framework. These business components are reusable with well-defined interfaces" [57].

[7]A component is "a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties" [58].

[8]A component is "a software element that (a) encapsulates a reusable implementation of functionality, (b) can be composed without modification, and (c) adheres to a component model" [59].

architectural elements and functions. Services and components are different in terms of deployment technologies and models that are used to technically implement them. For examples, services can be *web-services* [60], *micro-services* [61], *REST services* [62], etc., while components can be *OSGi* [63], *Fractal* [64], *SOFA* [65], etc. These have variations in their specification that make the implementation of services and components varied and respectively their provided and required interfaces.

Therefore, we can see that service identification and component identification are very similar in terms of identifying architectural elements that represent main reusable functionalities, but they are different in the way of packaging these functionalities following SOA or component-based models and deployment technologies (e.g., REST services vs OSGi components).

Many approaches were proposed to identify components from object-oriented software such as [66, 67, 68, 69, 70, 71]. These approaches mined components from single software that limits the reusability of identified components.

In [66], Kebir et al. extracted the component-based architecture based on partitioning classes into clusters corresponding to components. The partitions are based on static code dependencies.

Mishra et al. [67] also extracted the component-based architecture. Components are extracted based on information realized in use cases, sequence diagrams, and class diagrams. Unlike source code, sequence diagrams, use cases, and class diagrams are not always available. Hamza [72] identified components from requirements and use cases using formal concept analysis. He focused on the component stability rather than the component reusability.

Allier et al. [68] depended on dynamic dependencies between software classes to extract a component-based architecture. They relied on the use-cases to identify the execution trace scenarios. Classes that frequently occur in the execution traces are grouped into a component.

Liu et al. [73] identified interfaces of identified components. Similar to our approach, they defined a component interface as a group of methods belonging to the cluster of classes of an identified component. They relied on process mining tools to analyze the direct connections between the clusters of classes.

7.3. Software product line architecture identification

SPLA recovery approaches are similar to our approach in terms of the analysis of the variability between software products. However, compared to our service identification approach, they are different in their goal which is the identification of one architecture model that describes the design of a set of software products in the same family of software product line [74] based on variation points and variants between architectural-elements [75, 76, 77]. In other words, their focus is more on the understandability of the design of the family of products than the reusability of identified services.

Technically, SPLA recovery approaches identify component-based architecture (not potential reusable services) from each software product as disjoint clusters of classes that describe the system structure of this product. Then, they analyze the variability between the recovered architectures to identify variation points and variants using *clone detection algorithm* [78, 79, 80], *clustering algorithm* [25], or *user-defined* heuristics [81, 82]. They identify different aspects of SPLA such as mandatory components [25], optional components [25], component variability [78, 81, 25, 79, 82, 80], variability dependencies between different components [25, 79, 82], and feature model of architecture variability [25].

7.4. Feature identification

The distinguish between service identification and feature identification approaches comes from the differences between the concepts of a service and a feature. We have different goals and details processes to achieve these goals. Following Kang et al [83], a feature is defined as a non-structural element that provides user visible aspects. On the other hand, a service is a structural (architectural) element that could implement either user visible or invisible aspects. Furthermore, services and features belong to various abstraction levels. Features abstract software requirements at a high level (e.g., requirement level), and services are architectural elements at the design level. Please note that services can be used to represent the implementation of features at the design level. The mapping model could be many-to-many, many-to-one, one-to-many or one-to-one depending on the software engineers' decisions. Technically, features does not have any interfaces that represent the interaction between each others, rather than service interfaces, required and provided ones.

Feature identification has been investigated by many approaches. These aims to identify program units such as methods, or classes that represent features related only to user visible aspects, and without considering the position of these feature at the design level. Dit et al. [84] provided a survey of feature identification approaches. Features were identified based on the analysis of single software product like [85, 86, 87], or based on the analysis of multiple software products by exploring the commonality and the variability between these products like [88, 89, 90, 91].

8. Conclusion

In this paper, we presented *ReSIde* (**Re**usable **S**ervice **Ide**ntification): an automated approach that identifies reusable services based on the analysis of a set of similar object-oriented SPVs. ReSIde identifies *reusable* functionalities of cloned codes that can be qualified as services across multiple SPVs based on the analysis of the commonality and the variability between the source code ofthese SPVs. We consider that identifying service based on the analysis of multiple SPVs provides more guarantee for the reusability of the identified services, compared to the analysis of singular SPV.

ReSIde firstly identifies all potential services from each SPV independently. A potential service is defined based on a group of object-oriented classes identified gradually based on one core class. Groups of classes have quality values exceeding a pre-defined threshold value are considered as potential services. Then, ReSIde explores the commonality and the variability between the identified potential services to identify ones that are shared between different SPVs. The identification of these shared services is based on a clustering algorithm. From each cluster of services, ReSIde extracts one common service that represents that most reusable and quality-centric service in this cluster.

To validate ReSIde, we applied it on three case studies of product variants of three different sizes; 8 products of Mobile Media as a small-scale software (43.25 classes per product), 10 products of Health Watcher as medium-scale software (137.6 classes per product), and 9 products of ArgoUML as a large-scale one (2198.11 classes per product). The results demonstrated the applicability of ReSIde on the selected three case studies and the capability to identify reusable services that can be existed in multiple SPVs. We proposed an empirical measurement method to evaluate the reusability of the identified services. The results of this measurement method showed that the reusability of the services identified based on ReSIde is better than the reusability of those identified based on the analysis of singular software product.

As future directions, we plan to extend ReSIde to transform the object-oriented implementation of identified services into truly deployable services by making them adhere to one or more SOA models such as Web services or microservices. Also, we want to evaluate the reusability of the identified services based on the human expert knowledge.

References

[1] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, A. Egyed, Enhancing clone-and-own with systematic reuse for developing software variants, in: Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on, IEEE, pp. 391–400.

[2] J. Businge, M. Openja, S. Nadi, E. Bainomugisha, T. Berger, Clone-based variability management in the android ecosystem, in: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, pp. 625–634.

[3] A. Shatnawi, T. Ziadi, M. Y. Mohamadi, Understanding source code variability in cloned android families: an empirical study on 75 families, in: 2019 26th Asia-Pacific Software Engineering Conference (APSEC), IEEE, pp. 292–299.

[4] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, K. Czarnecki, An exploratory study of cloning in industrial software product lines, in: Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on, IEEE, pp. 25–34.

[5] W. Jin, T. Liu, Y. Cai, R. Kazman, R. Mo, Q. Zheng, Service candidate identification from monolithic systems based on execution traces, IEEE Transactions on Software Engineering (2019).

[6] S. Adjoyan, A. Seriai, A. Shatnawi, Service identification based on quality metrics - object-oriented legacy system migration towards SOA, in: The 26th International Conference on Software Engineering and Knowledge Engineering, Hyatt Regency, Vancouver, BC, Canada, July 1-3, 2013., pp. 1–6.

[7] R. Rodríguez-Echeverría, F. Maclas, V. M. Pavón, J. M. Conejero, F. Sánchez-Figueroa, Generating a rest service layer from a legacy system, in: Information System Development, 2014, pp. 433–444.

[8] M. Gysel, L. Kölbener, W. Giersche, O. Zimmermann, Service cutter: A systematic approach to service decomposition, in: ESOCC, pp. 185–200.

[9] M. Abdellatif, A. Shatnawi, Y.-G. Guéhéneuc, H. Mili, J. Privat, Toward service identification to support legacy object-oriented software systems migration to soa.

[10] M. Gasparic, A. Janes, A. Sillitti, G. Succi, An analysis of a project reuse approach in an industrial setting, in: Software Reuse for Dynamic Systems in the Cloud and Beyond, Springer, 2014, pp. 164–171.

[11] J. Sametinger, Software engineering with reusable components, Springer Science & Business Media, 1997.

[12] A. Shatnawi, A.-D. Seriai, Mining reusable software components from object-oriented source code of a set of similar software, in: IEEE 14th International Conference on Information Reuse and Integration (IRI 2013), IEEE, pp. 193–200.

[13] I. Iso, Iec 9126-1: Software engineering-product quality-part 1: Quality model, Geneva, Switzerland: International Organization for Standardization (2001).

[14] J. M. Bieman, B.-K. Kang, Cohesion and reuse in an object-oriented system, in: ACM SIGSOFT Software Engineering Notes, volume 20, ACM, pp. 259–262.

[15] J. Han, M. Kamber, J. Pei, Data mining, southeast asia edition: Concepts and techniques, Morgan kaufmann, 2006.

[16] W. H. Gomaa, A. A. Fahmy, A survey of text similarity approaches, International Journal of Computer Applications 68 (2013) 13–18.

[17] K. Narasimhan, C. Reichenbach, J. Lawall, Cleaning up copy–paste clones with interactive merging, Automated Software Engineering 25 (2018) 627–673.

[18] G. P. Krishnan, N. Tsantalis, Unification and refactoring of clones, in: CSMR-WCRE, IEEE, pp. 104–113.

[19] D. Poshyvanyk, A. Marcus, The conceptual coupling metrics for object-oriented systems, in: Software Maintenance, 2006. ICSM'06. 22nd IEEE International Conference on, IEEE, pp. 469–478.

[20] E. Figueiredo, N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F. Dantas, et al., Evolving software product lines with aspects, in: ACM/IEEE 30th International Conference on Software Engineering (ICSE'08), IEEE, pp. 261–270.

[21] M. V. Couto, M. T. Valente, E. Figueiredo, Extracting software product lines: A case study using conditional compilation, in: 15th European Conference on Software Maintenance and Reengineering (CSMR2011), IEEE, pp. 191–200.

[22] H. Eyal Salman, A.-D. Seriai, C. Dony, Feature-level change impact analysis using formal concept analysis, International Journal of Software Engineering and Knowledge Engineering 25 (2015) 69–92.

[23] L. P. Tizzei, C. M. Rubira, J. Lee, An aspect-based feature model for architecting component product lines, in: 2012 38th Euromicro Conference on Software Engineering and Advanced Applications, IEEE, pp. 85–92.

[24] J. Martinez, N. Ordoñez, X. Tërnava, T. Ziadi, J. Aponte, E. Figueiredo, M. Valente, Feature location benchmark with argouml spl, in: Systems and Software Product Line Conference (SPLC).

[25] A. Shatnawi, A.-D. Seriai, H. Sahraoui, Recovering software product line architecture of a family of object-oriented product variants, Journal of Systems and Software 131 (2017) 325–346.

[26] D. Garlan, Software architecture: A roadmap, in: Proceedings of the Conference on The Future of Software Engineering, ICSE '00, ACM, New York, NY, USA, 2000, pp. 91–101.

[27] H. Jain, H. Zhao, N. R. Chinta, A spanning tree based approach to identifying web services, International Journal of Web Services Research 1 (2004) 1.

[28] S. Alahmari, E. Zaluska, D. De Roure, A service identification framework for legacy system migration into soa, in: Services Computing (SCC), 2010 IEEE International Conference on, IEEE, pp. 614–617.

[29] H. M. Sneed, C. Verhoef, S. H. Sneed, Reusing existing object-oriented code as web services in a soa, in: Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA), 2013 IEEE 7th International Symposium on the, IEEE, pp. 31–39.

[30] D. Saha, Service mining from legacy database applications, in: Web Services (ICWS), 2015 IEEE International Conference on, IEEE, pp. 448–455.

[31] Y. Baghdadi, Reverse engineering relational databases to identify and specify basic web services with respect to service oriented computing, Information systems frontiers 8 (2006) 395–410.

[32] C. Del Grosso, M. Di Penta, I. G.-R. de Guzman, An approach for mining services in database oriented applications, in: 11th European Conference on Software Maintenance and Reengineering, IEEE, pp. 287–296.

[33] A. Fuhr, T. Horn, V. Riediger, Using dynamic analysis and clustering for implementing services by reusing legacy code, in: Reverse Engineering (WCRE), 2011 18th Working Conference on, IEEE, pp. 275–279.

[34] B. Upadhyaya, Y. Zou, F. Khomh, An approach to extract restful services from web applications, International Journal of Business Process Integration and Management 7 (2015) 213–227.

[35] S. Mani, V. S. Sinha, N. Sukaviriya, T. Ramachandra, Using user interface design to enhance service identification, in: Web Services, 2008. ICWS'08. IEEE International Conference on, IEEE, pp. 78–87.

[36] E. Sosa, P. J. Clemente, J. M. Conejero, R. Rodríguez-Echeverría, A model-driven process to modernize legacy web applications based on service oriented architectures, in: 2013 15th IEEE International Symposium on Web Systems Evolution (WSE), IEEE, pp. 61–70.

[37] M. J. Amiri, S. Parsa, A. M. Lajevardi, Multifaceted service identification: Process, requirement and data, Computer Science and Information Systems 13 (2016) 335–358.

[38] H. M. Sneed, Integrating legacy software into a service oriented architecture, in: Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on, IEEE, pp. 11–pp.

[39] L. Aversano, L. Cerulo, C. Palumbo, Mining candidate web services from legacy code, in: 10th International Symposium on Web Site Evolution, IEEE, pp. 37–40.

[40] M. Nakamur, H. Igaki, T. Kimura, K. Matsumoto, Identifying services in procedural programs for migrating legacy system to service oriented architecture, Implementation and Integration of Information Systems in the Service Sector (2012) 237.

[41] Z. Zhang, H. Yang, Incubating services in legacy systems for architectural migration, in: Software Engineering Conference, 2004. 11th Asia-Pacific, IEEE, pp. 196–203.

[42] H. Sneed, Migrating to web services: A research framework, in: Proceedings of the International.

[43] Z. Zhang, R. Liu, H. Yang, Service identification and packaging in service oriented reengineering., in: SEKE, volume 5, pp. 620–625.

[44] M. Abdelkader, M. Malki, S. M. Benslimane, A heuristic approach to locate candidate web service in legacy software, International Journal of Computer Applications in Technology 47 (2013) 152–161.

[45] Z. Zhang, H. Yang, W. C. Chu, Extracting reusable object-oriented legacy code segments with combined formal concept analysis and slicing techniques for service integration, in: 2006 Sixth International Conference on Quality Software (QSIC'06), IEEE, pp. 385–392.

[46] F. Chen, Z. Zhang, J. Li, J. Kang, H. Yang, Service identification via ontology mapping, in: 2009 33rd Annual IEEE International Computer Software and Applications Conference, volume 1, IEEE, pp. 486–491.

[47] R. S. Huergo, P. F. Pires, F. C. Delicato, Mdcsim: A method and a tool to identify services, IT Convergence Practice 2 (2014) 1–27.

[48] R. S. Huergo, P. F. Pires, F. C. Delicato, A method to identify services using master data and artifact-centric modeling approach, in: Proceedings of the 29th Annual ACM Symposium on Applied Computing, ACM, pp. 1225–1230.

[49] L. Baresi, M. Garriga, A. De Renzis, Microservices identification through interface analysis, in: European Conference on Service-Oriented and Cloud Computing, Springer, pp. 19–33.

[50] F. J. Frey, C. Hentrich, U. Zdun, Capability-based service identification in service-oriented legacy modernization, in: Proceedings of the 18th European Conference on Pattern Languages of Program, ACM, p. 10.

[51] G. Mazlami, J. Cito, P. Leitner, Extraction of microservices from monolithic software architectures, in: Web Services (ICWS), 2017 IEEE International Conference on, IEEE, pp. 524–531.

[52] D. K. Barry, Web Services, Service-oriented Architectures, and Cloud Computing: The Savvy Manager's Guide, Morgan Kaufmann, 2003.

[53] M. Nakamura, H. Igaki, T. Kimura, K.-i. Matsumoto, Extracting service candidates from procedural programs based on process dependency analysis, in: Services Computing Conference, 2009. APSCC 2009. IEEE Asia-Pacific, IEEE, pp. 484–491.

[54] A. Erradi, S. Anand, N. Kulkarni, Soaf: An architectural framework for service definition and realization, in: Services Computing, 2006. SCC'06. IEEE International Conference on, IEEE, pp. 151–158.

[55] A. Brown, S. Johnston, K. Kelly, Using service-oriented architecture and component-based development to build web service applications, Rational Software Corporation (2002).

[56] T. O. Group, Service oriented architecture, in: https://www.opengroup.org/soa/source-book/togaf/soadef.htm.

[57] G. Baster, P. Konana, J. E. Scott, Business components: A case study of bankers trust australia limited, Commun. ACM 44 (2001) 92–98.

[58] C. Szyperski, Component software: beyond object-oriented programming, Pearson Education, 2002.

[59] C. Lüer, A. Van Der Hoek, Composition environments for deployable software components, Citeseer, 2002.

[60] G. Alonso, F. Casati, H. Kuno, V. Machiraju, Web services, in: Web Services, Springer, 2004, pp. 123–149.

[61] D. Namiot, M. Sneps-Sneppe, On micro-services architecture, International Journal of Open Information Technologies 2 (2014) 24–27.

[62] C. Riva, M. Laitkorpi, Designing web-based mobile services with rest, in: Service-Oriented Computing-ICSOC 2007 Workshops, Springer, pp. 439–450.

[63] O. Alliance, Osgi service platform, release 3, IOS Press, Inc., 2003.

[64] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, J.-B. Stefani, The fractal component model and its support in java, Software: Practice and Experience 36 (2006) 1257–1284.

[65] F. Plasil, D. Balek, R. Janecek, Sofa/dcup: Architecture for component trading and dynamic updating, in: Configurable Distributed Systems, 1998. Proceedings. Fourth International Conference on, IEEE, pp. 43–51.

[66] S. Kebir, A.-D. Seriai, S. Chardigny, A. Chaoui, Quality-centric approach for software component identification from object-oriented code, in: 2012 Joint Working IEEE/IFIP Conference on Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), IEEE, pp. 181–190.

[67] S. Mishra, D. S. Kushwaha, A. K. Misra, Creating reusable software component from object-oriented legacy system through reverse engineering., Journal of object technology 8 (2009) 133–152.

[68] S. Allier, H. A. Sahraoui, S. Sadou, S. Vaucher, Restructuring object-oriented applications into component-oriented applications by using consistency with execution traces, in: Component-Based Software Engineering, Springer, 2010, pp. 216–231.

[69] A. Shatnawi, A.-D. Seriai, H. Sahraoui, Z. Alshara, Reverse engineering reusable software components from object-oriented apis, Journal of Systems and Software 131 (2017) 442–460.

[70] A. Shatnawi, H. Shatnawi, M. A. Saied, Z. A. Shara, H. Sahraoui, A. Seriai, Identifying components from object-oriented apis based on dynamic analysis, arXiv preprint arXiv:1803.06235 (2018).

[71] Z. Alshara, A.-D. Seriai, C. Tibermacine, H. L. Bouziane, C. Dony, A. Shatnawi, Materializing architecture recovered from object-oriented source code in component-based languages, in: European Conference on Software Architecture, Springer, pp. 309–325.

[72] H. S. Hamza, A framework for identifying reusable software components using formal concept analysis, in: Sixth International Conference on Information Technology: New Generations (ITNG), 2009, IEEE, pp. 813–818.

[73] C. Liu, B. van Dongen, N. Assy, W. van der Aalst, Component interface identification and behavioral model discovery from software execution data, in: International Conference on Program Comprehension, pp. 1–10.

[74] P. Clements, L. Northrop, Software product lines: practices and patterns (2002).

[75] H. Gomaa, Designing software product lines with uml, in: Software Engineering Workshop - Tutorial Notes, 2005. 29th Annual IEEE/NASA, pp. 160–216.

[76] C. Lima, Product line architecture recovery: an approach proposal, in: Proceedings of the 39th International Conference on Software Engineering Companion, IEEE Press, pp. 481–482.

[77] M. Zahid, Z. Mehmmod, I. Inayat, Evolution in software architecture recovery techniques—a survey, in: Emerging Technologies (ICET), 2017 13th International Conference on, IEEE, pp. 1–6.

[78] T. Mende, F. Beckwermert, R. Koschke, G. Meier, Supporting the grow-and-prune model in software product lines evolution using clone detection, in: 12th European Conference on Software Maintenance and Reengineering (CSMR), IEEE, pp. 163–172.

[79] R. Koschke, P. Frenzel, A. P. Breu, K. Angstmann, Extending the reflexion method for consolidating software variants into product lines, Software Quality Journal 17 (2009) 331–366.

[80] R. Kolb, D. Muthig, T. Patzke, K. Yamauchi, A case study in refactoring a legacy component for reuse in a product line, in: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM 2005), IEEE, pp. 369–378.

[81] M. Pinzger, H. Gall, J.-F. Girard, J. Knodel, C. Riva, W. Pasman, C. Broerse, J. G. Wijnstra, Architecture recovery for product families, in: Software Product-Family Engineering, Springer, 2004, pp. 332–351.

[82] K. C. Kang, M. Kim, J. Lee, B. Kim, Feature-oriented re-engineering of legacy systems into product line assets–a case study, in: Software Product Lines, Springer, 2005, pp. 45–56.

[83] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, A. S. Peterson, Feature-oriented domain analysis (FODA) feasibility study, Technical Report, DTIC Document, 1990.

[84] B. Dit, M. Revelle, M. Gethers, D. Poshyvanyk, Feature location in source code: a taxonomy and survey, Journal of Software: Evolution and Process 25 (2013) 53–95.

[85] G. Antoniol, Y.-G. Guéhéneuc, Feature identification: a novel approach and a case study, in: 21st IEEE International Conference on Software Maintenance (ICSM'05), IEEE, pp. 357–366.

[86] K. Chen, V. Rajlich, Case study of feature location using dependence graph, in: Program Comprehension, 2000. Proceedings. IWPC 2000. 8th International Workshop on, IEEE, pp. 241–247.

[87] R. Damaševičius, P. Paškevičius, E. Karčiauskas, R. Marcinkevičius, Automatic extraction of features and generation of feature models from java programs, Information Technology and Control 41 (2012) 376–384.

[88] Y. Xue, Reengineering legacy software products into software product line based on automatic variability analysis, in: Proceedings of the 33rd International Conference on Software Engineering, ACM, pp. 1114–1117.

[89] T. Ziadi, L. Frias, M. A. A. da Silva, M. Ziane, Feature identification from the source code of product variants, in: Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on, IEEE, pp. 417–422.

[90] A. Ra'Fat, A. Seriai, M. Huchard, C. Urtado, S. Vauttier, H. E. Salman, Feature location in a collection of software product variants using formal concept analysis, in: International Conference on Software Reuse, Springer, pp. 302–307.

[91] J. Carbonnel, M. Huchard, A. Gutierrez, Variability representation in product lines using concept lattices: feasibility study with descriptions from wikipedia's product comparison matrices, in: FCA&A 2015, co-located with 13th International Conference on Formal Concept Analysis (ICFCA 2015), volume 1434.