Reverse Engineering Reusable Software Components from Object-Oriented APIs

Anas Shatnawi[1], Abdelhak-Djamel Seriai[1], Houari Sahraoui[2], and Zakarea Al-Shara[1]

[1] UMR CNRS 5506, LIRMM, University of Montpellier, Montpellier, France
shatnawi, seriai, alshara@lirmm.fr
[2] DIRO, University of Montreal, Montreal, Canada
sahraoui@iro.umontreal.ca

Abstract. Object-oriented Application Programing Interfaces (APIs) support software reuse by providing pre-implemented functionalities. Due to the huge number of included classes, reusing and understanding large APIs is a complex task. Otherwise, software components are accepted to be more reusable and understandable entities than object-oriented ones. Thus, in this paper, we propose an approach for reengineering object-oriented APIs into component-based ones. We mine components as a group of classes based on the frequency they are used together and their ability to form a quality-centric component. To validate our approach, we experimented on 100 *Java* applications that used *Android* APIs.

Keywords: Reuse· reusability· understandability· reengineering· reverse engineering· component· API· object-oriented· frequent usage pattern.

1 Introduction

Nowadays, the development of large and complex software applications is based on reusing pre-existing functionalities instead of developing them from scratch [1, 2]. Application Programming Interfaces (APIs) are recognized as the most commonly used repositories supporting software reuse [1]. APIs provide a pre-implemented, tested and high quality set of functionalities [2, 3]. Consequently, they increase software quality and reduce the effort spent on coding, testing and maintenance activities [2].

In the case of object-oriented APIs, e.g. *Standard Template Libraries* in *C++* or *Java SDK*, the functionalities are implemented as object-oriented classes. It is well known that reusing and understanding large APIs, such as *Java SDK* which contains more than 7.000 classes, is not an easy task [4, 5]. On the other hand, classes of an API are used following specific usage patterns, in order to provide services to software applications [6–8]. For example, in the *Android* API, *Activity*, *GroupView*, *Context*, *LayoutInflater* and *View* are the classes needed to create a simple activity which contains an empty view [9]. Consequently, many approaches have been proposed, such as [7, 10, 11], to facilitate the understandability and the reusability of APIs by discovering Frequent Usage Patterns

(FUPs) of APIs. This is based on the API usage history of software applications (i.e. API clients). Despite the value of FUPs, these are not sufficient to provide a high degree of API reusability and understandability. These are used as guides for reusing API classes and are not themselves reusable entities [12].

Otherwise, software components are admitted to be more reusable and understandable entities than object-oriented ones [13]. It is because components are considered coarse-grained software entities, while object-oriented classes are considered fine-grained ones. In addition, components define their required and provided interfaces. This means that the component dependencies are more understandable compared to the dependencies among objects. Consequently, many approaches have been proposed to identify components from object-oriented software applications such as [14–16]. These approaches aim at mining components by analyzing the source code of software applications. In this context, dependencies between classes are only realized via calls between methods, sharing types, etc. Nevertheless, no approach has been proposed to identify components from object-oriented APIs. In this context, we distinguish two kinds of dependencies. The first one is that classes are structurally dependent. The second one is that some classes need to be reused together to implement a functionality. This kind of dependencies can not be identified by only analyzing the source code, but also needs the analysis of how software applications use the API classes. For example, in the *Android* API, *Activity* and *Context* classes are structurally and behaviorally independent, but they have to be used together to build android applications. This means that classes frequently used together are more favorable to belong to the same component.

In this paper, we propose an approach that aims at recovering software components from object-oriented APIs. This does not only improve the reusability of APIs themselves, but also supports component-based reuse techniques by providing component based APIs. The approach exploits specificity of API entities by statically analyzing the source code of both APIs and their software clients to identify groups of API classes that are able to form components. Our assumption is based on the probability of classes to be reused together by API clients on the one hand, and on the structural dependencies between classes on the other hand. In order to validate the proposed approach, we experimented it on a set of 100 *Java* applications that use three *Android* APIs. The evaluation shows that structuring object-oriented APIs as component-based ones improves the reusability and the understandability of these APIs.

This journal paper is an extended version of our conference paper published in [17]. The extension includes providing better analysis of the problem, more explanation about the proposed solution (i.e. algorithms, figures and examples), structuring component interfaces, more related works, discussions and threats to validity.

The rest of this paper is organized as follows. Section 2 presents the background needed to understand our approach. Section 3 shows the foundation of our approach. Then, in Section 4 we present the identification of classes composing component interfaces. Section 5 presents how APIs are organized

as component-based libraries. Experimentation and results of our approach are discussed through three APIs case studies in Section 6. Next, related works are discussed in Section 7. Finally, concluding remarks and future directions are presented in Section 8.

2 Background

2.1 Component Quality Model: the ROMANTIC Approach

In this paper, we rely on the component quality model proposed in our previous works related to the ROMANTIC[3] approach [15, 18]. In *ROMANTIC*, we have proposed a set of metrics to measure the ability of a group of classes in a software application to form a component. These metrics are defined based on the component quality characteristics that are driven from the component definitions: *Composability*, *Autonomy* and *Specificity*. *Composability* of a component refers to its ability to be composed through its interfaces without any modification. *Autonomy* means that a component can be reused in an autonomous way because it encapsulates the strongly dependent functionalities. *Specificity* refers to the fact that a component implements a limited number of closed functionalities, which makes it a coarse-grained entity.

Similar to the software quality model ISO 9126 [19], we proposed to refine the characteristics of the component into sub-characteristics. Next, the sub-characteristics are refined into the properties of the component (e.g. number of required interfaces). Then, these properties are mapped to the properties of the group of classes from which the component is identified (e.g. group of classes coupling). Lastly, these properties are refined into object-oriented metrics (e.g. coupling metric). This quality refinement model is shown in Figure 1. According to this model, a quality function has been proposed to measure the component quality. This quality function is used as a similarity metric for a hierarchal clustering algorithm [15, 18] as well as in search-based algorithms [20] to partition the object-oriented classes into groups; where each group represents a component.

2.2 Frequent Usage Patterns

In the domain of data mining, a Frequent Usage Pattern (FUP) is defined as a set of items, subsequences or substructures that are frequently used together by customers [21]. It provides information that helps decision makers (e.g. customer shopping behavior) by mining associations and correlations among a set of items in a huge data set. An example of FUP mining is a market basket analysis. In this example, the customer buying habits are analyzed to identify items that are frequently bought together in the customer shopping baskets, for instance, milk and bread form a FUP when they bought frequently together. The identification

[3] ROMANTIC: Re-engineering of Object-oriented systeMs by Architecture extractioN and migraTIon to Component based ones.

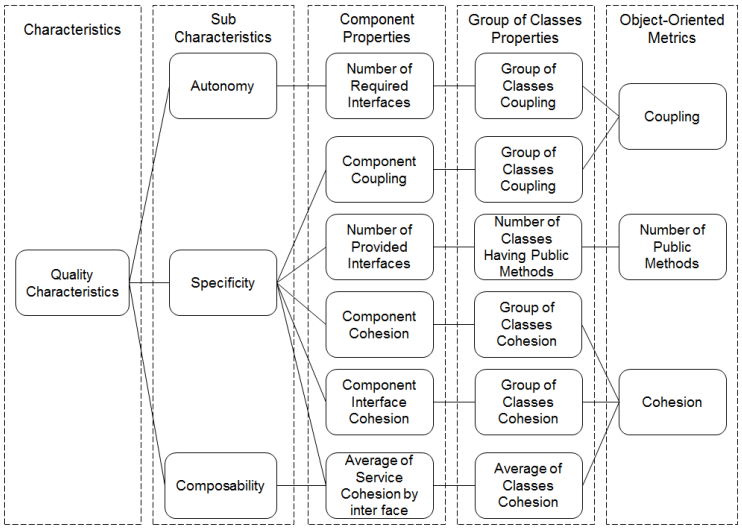


Fig. 1. From component characteristics to object-oriented metrics

of FUP is based on *Support* quality metric that is used to measure the interestingness of a set of items. *Support* refers to the probability of finding a set of items in the transactions. For example, the value of 0.30 *Support*, means that 30% of all the transactions contain the target item set. The following equation refers to *Support*:

$$S(E1, E2) = P(E1 \cup E2) \tag{1}$$

Where *E1*, *E2* are sets of items; S refers to *Support*; P refers to the probability.

3 The Proposed Approach Foundations

The goal of our approach is at reengineering object-oriented APIs to component-based ones. This is done in two directions. The first one is the identification of groups of classes that can be considered as the object-oriented implementation of the API components. The second one is the identification of how these components can be organized as component-based APIs.

3.1 Component Identification

We view a component as a group of API classes that provides coarse grained services to clients of an API. Classes that have direct links (e.g. method call, attribute access) with classes implementing other components compose the interfaces of the component. Provided interfaces of a component are defined as a

group of methods implemented by classes composing these interfaces. Required interfaces of a component are defined as a group of methods invoked by the component and provided by other components.

The identification of groups of classes composing components is based on two kinds of dependencies; usage-pattern-based and source-code-based ones. Usage-pattern-based is related to the way that software applications use these groups of classes. It refers to observations made based on the analysis of previous usages of APIs. We consider that classes frequently used together are more favorable belonging to a single or a few number of components. This is realized through Frequent Usage Patterns (FUPs) that identify recurring patterns, composed of classes frequently used together. Classes composing FUPs represent the gateways to access the API services. Thus, they are used to guide the identification of classes composing the provided interfaces of components. Classes composing a FUP may be related to different services that have been used together. Therefore, they can be mapped to be a part of different component interfaces. Classes of a component interface can be very dependent on other classes that are not directly used by clients of the API. These are identified based on source-code-based dependencies. It implies that the component identification process is driven by the identification of its provided interfaces. To this end, the analysis of structural dependencies between classes is used to identify classes forming the core of the component. It is used to form a quality-centric component. This is achieved based on the three quality characteristics that should be satisfied by the group of classes forming the component; *Composability*, *Autonomy* and *Specificity*. To this end, we rely on the component quality model presented by the ROMANTIC approach [15, 18].

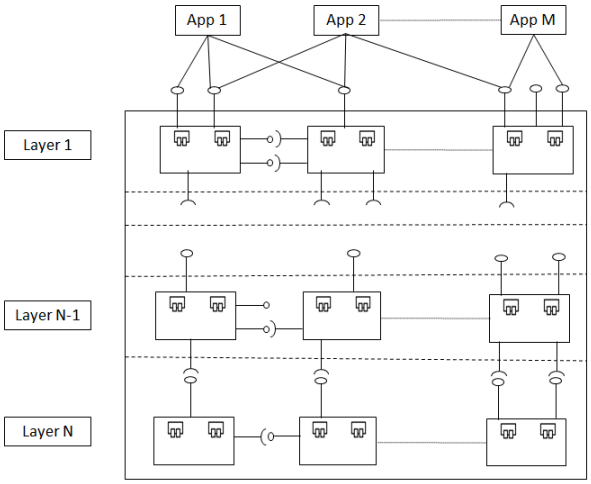


Fig. 2. Multi-layers component-based API

3.2 API as a Library of Components

We organize the API in layers of components. These layers describe how API components are vertically and horizontally organized. We consider that each layer contains components providing services to components of the layer above and requiring services from components of the layer below.

Classes constituting an API can be categorized into two types. The first one is made up of classes that are directly reused by software applications. These represent the implementation of accessible-services of the API (provided to software applications). Thus, components that are identified corresponding to these classes constitute the first layer of the API (i.e. the layer accessed by software applications). The second one is composed of classes representing the rest of API classes. These can also be divided into two categories. The first includes classes providing services to the first layer components. These represent the implementation of components constituting the second layer. In the same manner, components composing the other layers are identified. Based on that, we organize component-based APIs as a set of layers describing how their components are organized. Figure 2 shows our point of view regarding the API organization.

3.3 Principles and Mapping Model

Based on the observations made in the previous sub-sections, the proposed approach can be summarized according to the following principles:

- In object-oriented APIs, a component is identified as a group of classes.
- To reengineer the entire object-oriented API into a component-based one, each class of the API is mapped to be part of at least one component. Each class is mapped either as a class of the component interfaces or as a part of the internal classes of the component.
- Classes frequently used together by software applications provide accessible-user services of the API. Thus, they are used to guide the identification of classes composing the provided interfaces of components. These are identified based on FUPs.
- As a FUP can be composed of classes providing multiple services, its classes can be mapped to be a part of different component interfaces.
- A class of an API can be a part of several FUPs and can participate in implementing multiple services. Consequently, a class can be mapped into multiple component interfaces.
- The identification of classes forming the core of the components is driven by the identification of its provided interfaces.
- The analysis of structural dependencies between classes is used to identify classes forming the core of the component.
- Classes that are not used by software applications are used to structure components of the API layers.
- In a component-based API, the components are vertically and horizontally organized in terms of layers based on the required and provided services between the components.

Based on these principles, we propose a mapping model, shown in Figure 3, that maps class-to-component through FUPs.

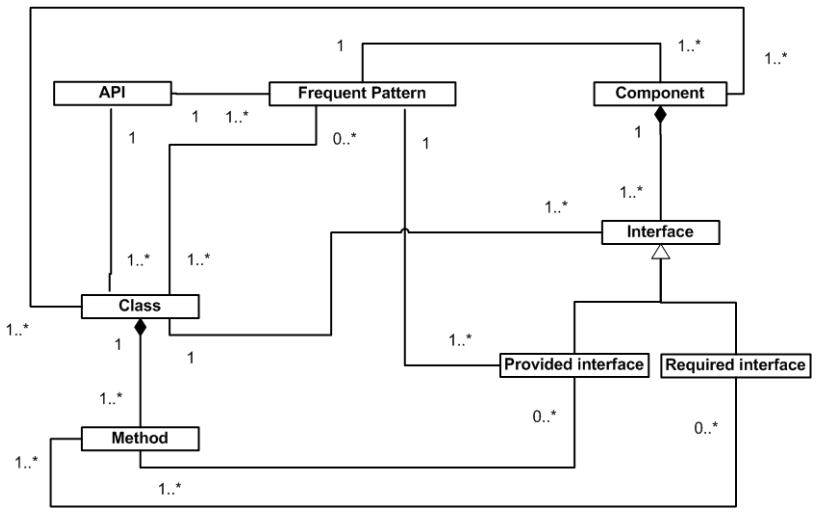


Fig. 3. Mapping class to component through FUP

3.4 Identification Process

We propose the following process to identify components from object-oriented APIs (see Figure 4):

- **Identification of frequent usage patterns.** FUPs are identified by analyzing the interactions between the API and its application clients.
- **Identification of the interfaces of components.** We partition the set of classes of each FUP into subgroups, where each one is considered as related to the provided interfaces of one component (c.f. Figure 5). The partitioning is based on criteria related to structural dependencies, lexical similarity and the frequency of simultaneous reuse.
- **Identification of internal classes of components driven by their provided interfaces.** Classes composing the provided interfaces of a component form the starting point for identifying the rest of the component classes. To identify these classes we rely on the analysis of structural dependencies between classes in the API with those forming the interfaces. We check if these classes are able to form a quality-centric component.
- **Organizing API as Layers of Components.** As each class of the API must be a part of at least one component, we associate classes that do not compose any of the already identified components to new ones. According to that, we organize component-based APIs as a set of layers. This organization is use-driven. The first layer is composed of components that are used by

the software clients, while the second layer is composed of components that provide services used by components of the first layer, and so on. As a result, the API is structured in N layers of components.

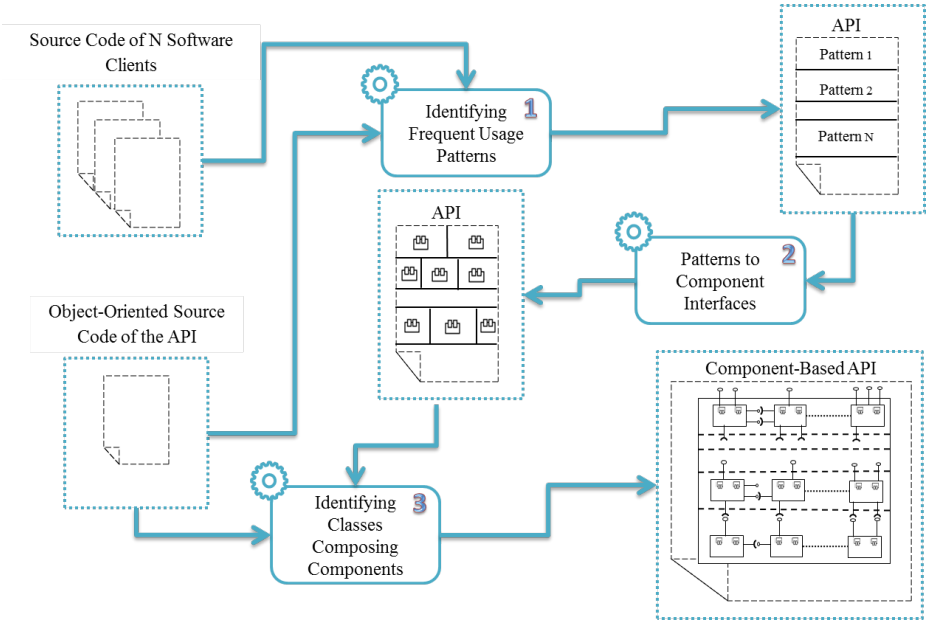


Fig. 4. The process of mining components from an object-oriented API

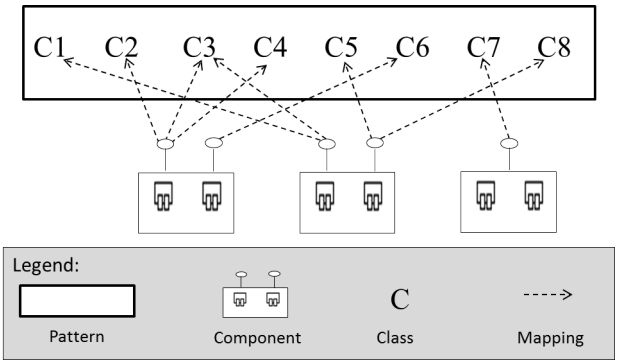


Fig. 5. Identification of provided interfaces of API components from FUPs

4 Identification of Component Interfaces

The identification of classes forming an API component is driven by the identification of classes composing the provided interfaces of this component. Classes composing these interfaces are those directly accessed by the clients of the API. Classes belonging to the same interface are those frequently used together. Therefore, they are identified from frequent usage patterns. Classes of the API composing frequent usage patterns are identified based on the analysis of how API classes were used by the API clients. API classes used together constitute transactions of usage.

4.1 Extracting Transactions of Usage

A transaction of usage is a set of interactions between an API and a client of this API. These interactions consist of calling methods, accessing attributes, inheritance or creating an instance object based on a class of the API. They are identified by statically analyzing the source code of both the API and its clients. Transactions are different depending on the choice of which are the API clients. This choice directly affects the type of the resulting patterns. Multiple options are possible, a client can be either a class, a group of classes or the whole application. Figure 6 illustrates these situations. Firstly, if we consider that a transaction corresponding to a class composing a client application, then *{C2, C5}*, *{C3}*, *{C5}*, *{C7}* and *{C7}* are the set of transactions that will be identified based on the first client application. Secondly, if a transaction corresponds to a group of classes from the client application, then *{C2, C5, C3}*, *{C5, C7}* and *{C7}* are the set of transactions that will be identified considering the first client application. Thirdly, if a transaction corresponds to the whole client application, then *{C2, C5, C3, C7}* is the transaction that will be identified considering the first client application.

In our approach, we consider as an API client a group of classes related to the same application functionality. The idea is that classes corresponding to the same application functionality use API classes related to correlated API functionalities. We identify groups of classes related to the same application functionalities as components of this application. This is done thanks to ROMANTIC approach defined in our previous work [15]. As a result, a transaction is a set of API classes such that each one is used by at least one class of the client component classes. Figure 7 shows an example. Algorithm 1 shows the process of transaction identification. It starts by partitioning each software client into components. Then, for each component, it identifies API classes that are reused by the component classes.

4.2 Mining Frequent Usage Patterns of Classes

In the previous step, the interactions of application clients with the API are identified as transactions. Based on these transactions, we identify FUPs. A FUP is defined as a set of API classes that are frequently used together by

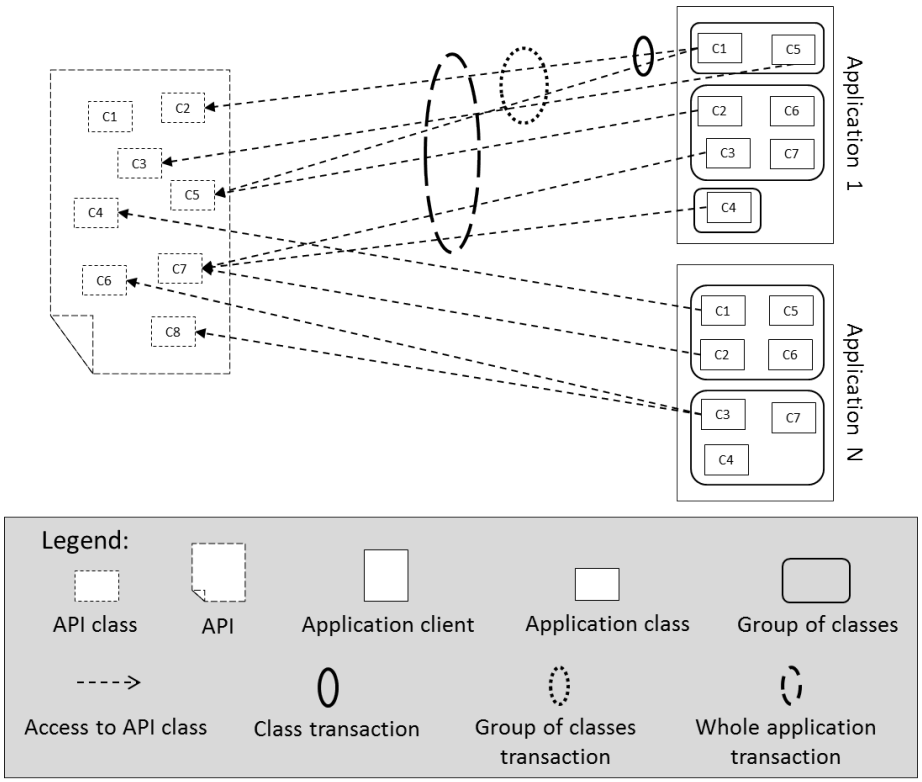


Fig. 6. Transactions based on clients

Algorithm 1: Identifying Transactions

Input: Source Code of a Set of Software Clients(*Clients*), API Source Code(*API*)
Output: A Set of Transactions(*trans*)
for *each client* ∈ *Clients* **do**
  | *components*.add(ROMANTIC(*client*.sourceCode));
end
for *each com* ∈ *components* **do**
  | *transaction* = ∅;
  | **for** *each class* ∈ *com* **do**
  | | *transaction*.add(class.getUsedClasses(*API*.sourceCode));
  | **end**
  | *trans*.add(*transaction*);
end
return *trans;*

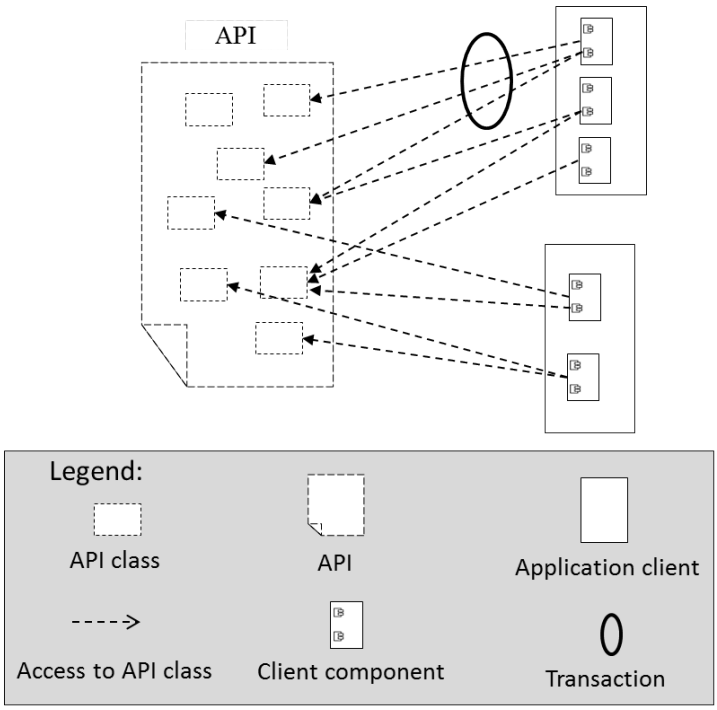


Fig. 7. Client components and corresponding transactions

client components. A group of classes is considered as a frequent pattern if it reaches a predefined threshold of interestingness metric. This metric is known as *Support*. The *Support* refers to the probability of finding a set of API classes in the transactions.

FUPs Mining Algorithms: an Analysis The identification of groups of classes forming FUPs can be done based on several algorithms. One of them is the Brute-Force algorithm [21] that identifies all possible groups of classes. Then, it prunes groups that do not reach the predefined *Support* threshold value. However, this algorithm is computationally prohibitive since that the identification of all groups, corresponding to N classes, needs 2^N time complexity [21]. Another algorithm is the Apriori algorithm that utilizes the property of *anti-monotone* [21], which means that if a group of classes is considered as infrequent, then all of its supersets must be infrequent as well. Thus, they do not need to be generated. However, this algorithm still has to generate the candidate groups of classes. For instance, suppose that we have 10^4 frequent groups of classes of size *1*, it requires to generate about 10^7 groups of size *2*. Furthermore, it needs to generate about 10^{30} groups of size *10*. Thus, this algorithm does not work in the situation where low *Support* threshold values are selected [22]. Another

algorithm is the Frequent-Pattern Growth (FP Growth) algorithm [22]. In this algorithm, there is no need to produce the candidate groups. Instead, it uses a divide-and-conquer technique to mine FUPs. It firstly builds a special data structure called Frequent-Pattern tree (FP-tree). This tree is used to compress information of class associations. Then, FP Growth divides the FP-tree into a collection of databases, such that each one is related to one frequent group of classes.

Table 1. An example of transactions composed of API classes

Transaction ID	List of Classes
T1	C1, C2, C5
T2	C2, C4
T3	C2, C3
T4	C1, C2, C4
T5	C1, C3
T6	C2, C3
T7	C1, C3
T8	C1, C2, C3, C5
T9	C1, C2, C3

Table 2. Classes ordering inside the transactions

Transaction ID	Ordered Classes
T1	C2, C1, C5
T2	C2, C4
T3	C2, C3
T4	C2, C1, C4
T5	C1, C3
T6	C2, C3
T7	C1, C3
T8	C2, C1, C3, C5
T9	C2, C1, C3

Frequent-Pattern Growth Algorithm Among the presented algorithms, FP Growth is the best one since that it outperforms the others in terms of time and space complexity [21]. Thus, we mine FUPs based on the FP Growth. To better understand how FP Growth works, we provide an illustrative example. In this example, we have *9* transactions presented in Table 1. The algorithm starts by building the FP-tree corresponding to these transactions. To this end, it first scans the transactions to find the frequency of each API class. In our example, the frequencies of *C1, C2, C3, C4* and *C5* are respectively *6, 7, 6, 2* and *2*.

Then, the classes are sorted in a descending order according to their frequency values. That is *C2, C1, C3, C4, C5*. Next, the classes inside the transactions are ordered according to their frequency values (see Table 2). Then, the tree is built based on the ordered transactions as follows: starting from the root of the tree, which is labeled by a *NULL* value, each transaction is added as a branch in the tree, such that the class which has the highest frequency is added first and so on. In the example, the order is *C2, C1, C5* for the first transaction. Whenever a branch shares a common prefix with an already added branch, we only increment the frequency of the shared nodes. Figure 8 explains the process of building the FP-tree.

Based on the FP-tree, the algorithm extracts conditional pattern bases and a conditional FP-tree for each frequent class. Conditional pattern bases consist of the collection of paths that co-located with the suffix pattern, while the conditional FP-trees are the subtrees that generate the pattern. For example, the conditional pattern bases corresponding to *C5* is {{*C2:1, C1:1*}, {*C2:1, C1:1, C3: 1*} }, thus the conditional FP-tree is $\langle C2 : 2, C1 : 2\rangle$. Paths that do not reach the predefined threshold value are rejected. For example, if the threshold is *2*, the path $\langle C2 : 2, C1 : 2, C31\rangle$ is excluded since its frequency is *1*. The set of FUPs identified from our example is {*{C2, C1, C5}*, *{C2, C4}*, *{C2, C1, C3}*, *{C2, C1}*}.

Less Commonly Used Classes The use of the *Support* threshold separates the classes of API used by application clients into two groups according to whether they belong to at least one FUP or not. Classes that do not belong to any of the identified FUPs are the less commonly used classes. As each API class that belongs to a transaction is a class that has been accessed by the clients of the API, therefore it must be a part of the classes composing the interfaces of at least one component. We propose assigning each class of the less commonly used classes to the pattern holding the maximum *Support* value when they are merged together.

4.3 Identifying Classes Composing Component Interfaces from Frequent Usage Patterns

We identify classes composing component interfaces from those composing FUPs. Each FUP is partitioned into a set of groups, where each group represents a component interface.

FUP Partitioning Fitness Function Classes are grouped together according to three heuristics that measure the probability of a set of classes to be a part of the same interface.

1. **Frequency of simultaneous use:** classes composing a FUP are regarded differently depending on the frequency of their simultaneous reuse by software applications. As much as classes are reused together, the probability

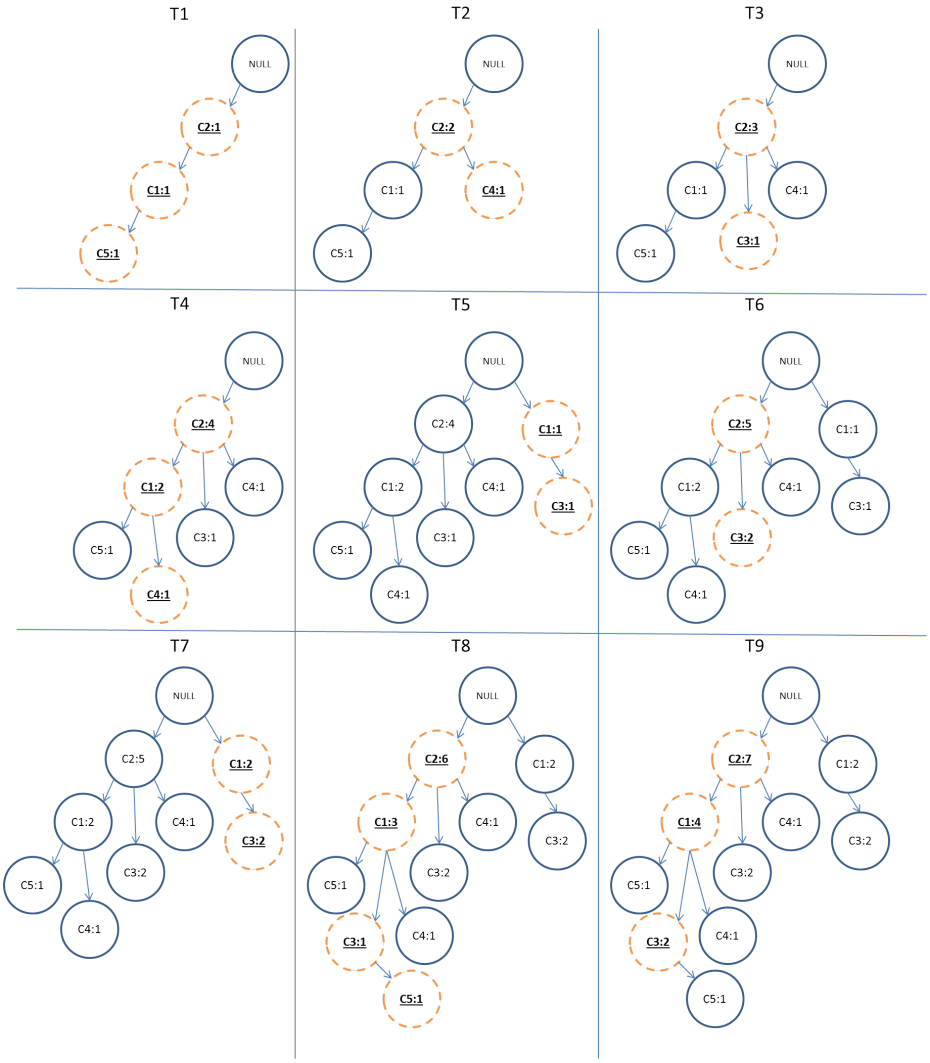


Fig. 8. Process of building the FP-tree

that these classes providing related services is higher. Therefore, we rely on *Support* metric to measure the association frequency of a set of classes.

2. **Cohesion:** a group of classes that accesses and shares the same data (e.g. attributes) is probably related to the same service. Thus, we consider that the cohesion of a group of classes is an indication of their functional proximity. To this end, we use LCC metric [23] to measure the cohesion of a set of classes. We select LCC since it measures both direct and indirect dependencies between the classes.

3. **Lexical similarity:** in most cases, classes of an API are well-documented (i.e. the identifier names are meaningful). Thus, their identifier names indicate to the offered services. Therefore, a group of classes having similar identifier names is likely to belong to the same service. To this end, we utilize *Conceptual Coupling* metric [24] to measure classes' lexical similarity based on the semantic information obtained from the source code, encoded in identifiers and comments.

Based on the above heuristics, we propose a fitness function, given below, measuring the ability of a group of classes to form a component interface.

$$IQ(E) = \frac{1}{\sum_i \lambda_i} \cdot (\lambda_1 \cdot LCC(E) + \lambda_2 \cdot CC(E) + \lambda_3 \cdot S(E)) \tag{2}$$

Where:

- E is a set of object-oriented classes
- *LCC(E)* is the *Cohesion* of E
- *CC(E)* is *Conceptual Coupling* of E
- *S(E)* is the *Support* of E
- λ_1, λ_2, and λ_3 are weight values, situated in [0-1]. These are used by the API expert to weight each characteristic as needed.

FUP Partitioning Algorithm The fitness function defined in the previous section is used to partition each FUP into groups of classes using a hierarchical clustering algorithm. This algorithm consists of two steps. The first one aims to build a binary tree, called dendrogram. This dendrogram provides a set of candidate clusters by presenting a hierarchical representation of classes' similarity. Figure 9 shows an example of a dendrogram tree, where C_i refers to $Class_i$. The second step aims at traveling through the built dendrogram, in order to extract the best clusters, representing a partition.

To build a dendrogram, the algorithm starts by considering each individual class as an initial leaf node in a binary tree. Next, the two most similar nodes are grouped into a new one, i.e. as a parent of them. For example, in Figure 9, the C_2 and C_3 classes are grouped. This is continued until all nodes are grouped in the root of the dendrogram. Algorithm 2 presents the procedure used to gather similar classes onto a dendrogram. It takes a set of classes as an input. The result of this algorithm is a hierarchical tree representation of candidate clusters.

To identify the best clusters, a depth first search algorithm is used to travel through the dendrogram tree. It starts from the tree root to find the cut-off points. It compares the similarity of the current node with its children. 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 function value. Otherwise, the algorithm continues through its children. Algorithm 3 presents the procedure used to extract clusters of classes from a dendrogram. The result of this algorithm is a set of clusters, where each contains classes corresponding to a component interface.

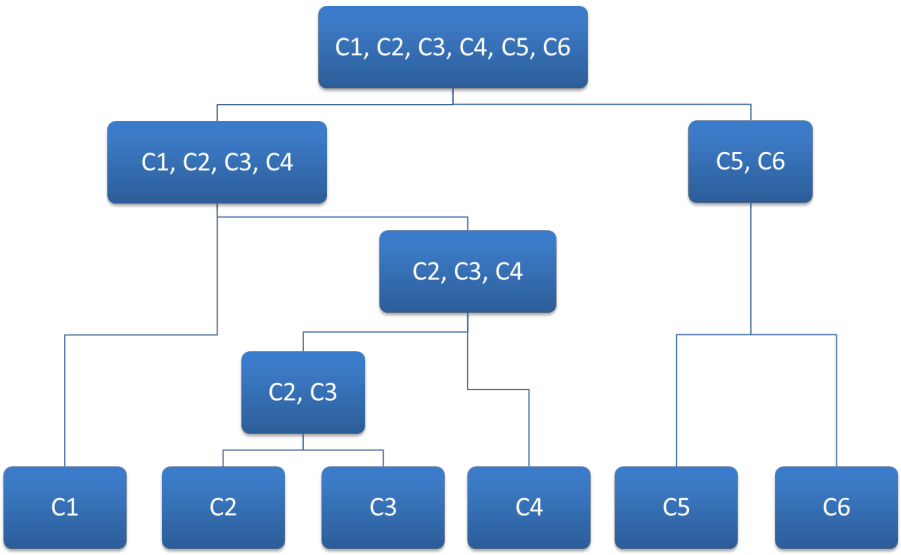


Fig. 9. An example of a dendrogram tree

Algorithm 2: Building Dendrogram

Input: A Set of Classes Composing FUP(FUP)
Output: Dendrogram Tree ($dendrogram$)
BinaryTree $dendrogram = FUP$;
while *(|dendrogram| > 1)* **do**
  $c1, c2$ = mostLexicallySimilarNodes($dendrogram$);
  c = newNode($c1, c2$);
  remove($c1, dendrogram$);
  remove($c2, dendrogram$);
  add($c, dendrogram$);
end
return *dendrogram;*

4.4 Structuring Component Interfaces

An interface provided by an API component is composed of public methods selected from classes which are identified in the previous step as composing the component interfaces. However, methods of required interfaces of an API component are those called inside one of its classes and defined in classes of other components. These two sets of methods constitute the initial search-space to identify component interfaces. The identification process is based on the following heuristics to partition this search-space into sub-groups; where each represents an interface:

Algorithm 3: Dendrogram Traversal

```
Input: Dendrogram Tree(dendrogram)
Output: A Set of Clusters of Component Interfaces(clusters)
Stack traversal;
traversal.push(dendrogram.getRoot());
while (! traversal.isEmpty()) do
    Node father = traversal.pop();
    Node left = dendrogram.getLeftSon(father);
    Node right = dendrogram.getRightSon(father);
    if similarity(father) > (similarity(left) + similarity(right) / 2) then
        clusters.add(father)
    else
        traversal.push(left);
        traversal.push(right);
    end
end
return clusters;
```

- Methods that belong to the same interface have a high probability of being used together. Consequently, we consider that methods frequently called together have a higher probability of belonging to the same component interface.
- A group of methods belonging to the same object-oriented interface has a higher probability of belonging to the same component interface.
- A group of methods having a high cohesion and a high lexical similarity has a high probability of belonging to the same component interface.

Based on these heuristics, we define a similarity function that measures the quality of a set of methods to form a component interface. We use *Support*, *LCC* [23] and *Conceptual Coupling* [24] metrics to respectively measure the frequency of use, cohesion and lexical similarity of a set of methods. This function is defined as follows:

$$Interface(M) = \frac{1}{\sum_i \lambda_i} \cdot (\lambda_1 \cdot LCC(M) + \lambda_2 \cdot CS(M) + \lambda_3 \cdot S(M) + \lambda_4 \cdot SOI(M)) \tag{3}$$

Where:

- M is a set of methods.
- $LCC(M)$, $CS(M)$, $S(M)$, and $SOI(M)$ respectively refers to the cohesion, *Cosine* similarity, *support* and the association with the same object-oriented interface(1 if yes, else 0) of M.
- λ_1, λ_2, λ_3 and λ_4 are weight values, situated in [0-1]. These are used by the API expert to weigh each characteristic as needed.

Based on this similarity function, we partition methods of the above search-space into clusters based on a hierarchical clustering algorithm. Each cluster contains a set of methods forming an interface.

5 API as Library of Components

5.1 Identifying Classes Composing Components

As we mentioned before, the component identification process is driven by the identification of its provided interfaces. This means that API classes forming a component are identified in relation to their structural dependencies with the classes forming provided interfaces of the component. Thus, classes having either direct or indirect links with the interface ones compose the search-space of classes that may be added to the component. The selection of a group of classes, from the search-space, is based on the measurement of the quality of the component, when they are included.

To identify the best group of classes that can serve as the implementation of a component providing the identified interfaces, we investigate all subsets of candidate classes. Then, the set that maximizes the component quality is selected. However, this requires an exponential time complexity to identify all subsets (i.e. NP-hard problem). Thus, we present a heuristic-based technique that identifies near-optimal groups of classes of the corresponding optimal ones.

The identification of these classes is done gradually. In other words, we start to form the group of classes composing the interface ones, and then we add other classes to form a component based on the component quality measurement model. Classes having either direct or indirect links with the interface ones represent the candidate classes to be added to the component. At each

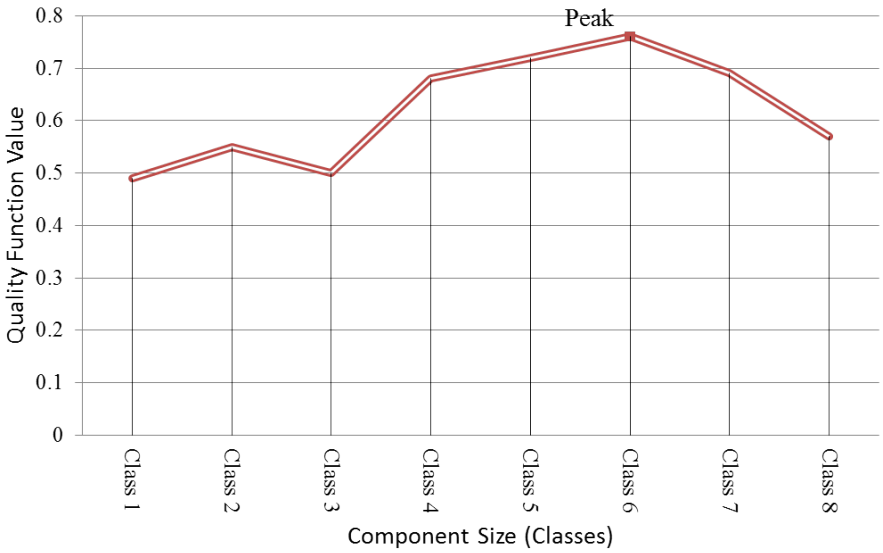


Fig. 10. Identifying Classes Composing Components

step, we add a new API class. This is selected based on the quality value of the component, formed by adding this class to the ones already selected. The class that maximizes the quality value is selected in this step. This is done until all search-space classes are investigated.

Each time we add a class, we evaluate the component quality. Then, we select the peak quality value to decide which classes form the component. This means that we exclude classes added after the peak value. As an example, $Class7$ and $Class8$ in Figure 10 are excluded from the resulting component because they were added after the quality value reached the peak. Algorithm 4 illustrates the process of identifying classes composing a component. In this algorithm, Q refers to the quality fitness function.

Algorithm 4: Identifying classes composing components

```
Input: Sets of Provided Interface Classes(interfaces), API Source Code(API)
Output: A Set of Components(components)
for each inter in interfaces do
    comp = inter.getClasses();
    bestComp = comp;
    searchSpace = API.getConnectedClasses(inter);
    while (|searchClasses| > 1 ) do
        c = Q.getMaximizeClass(searchSpace, comp);
        searchSpace.remove(c);
        comp = comp ∪ c;
        if Q(comp)) > Q(bestComp) then
            bestComp = comp;
        end
    end
    components.add(bestComp);
end
return components;
```

5.2 Organizing API as Layers of Components

As we previously mentioned, the API is structured in N layers of components. To identify components of layer L, we rely on components of layer $L-1$. We proceed similarly to the identification of the components of the first layer. We use required interfaces of the components already identified in layer $L-1$ to identify the interfaces provided by components in layer L. This continues until reaching a layer where its components either do not require any interface or they require ones already identified. Figure 11 shows an example that illustrates how the components composing each layer are identified, where Figure 11.a presents an object oriented API, Figure 11.b shows how the first layer components are identified, Figure 11.c explains the second layer component identification and Figure 11.d shows the resulting component-based API.

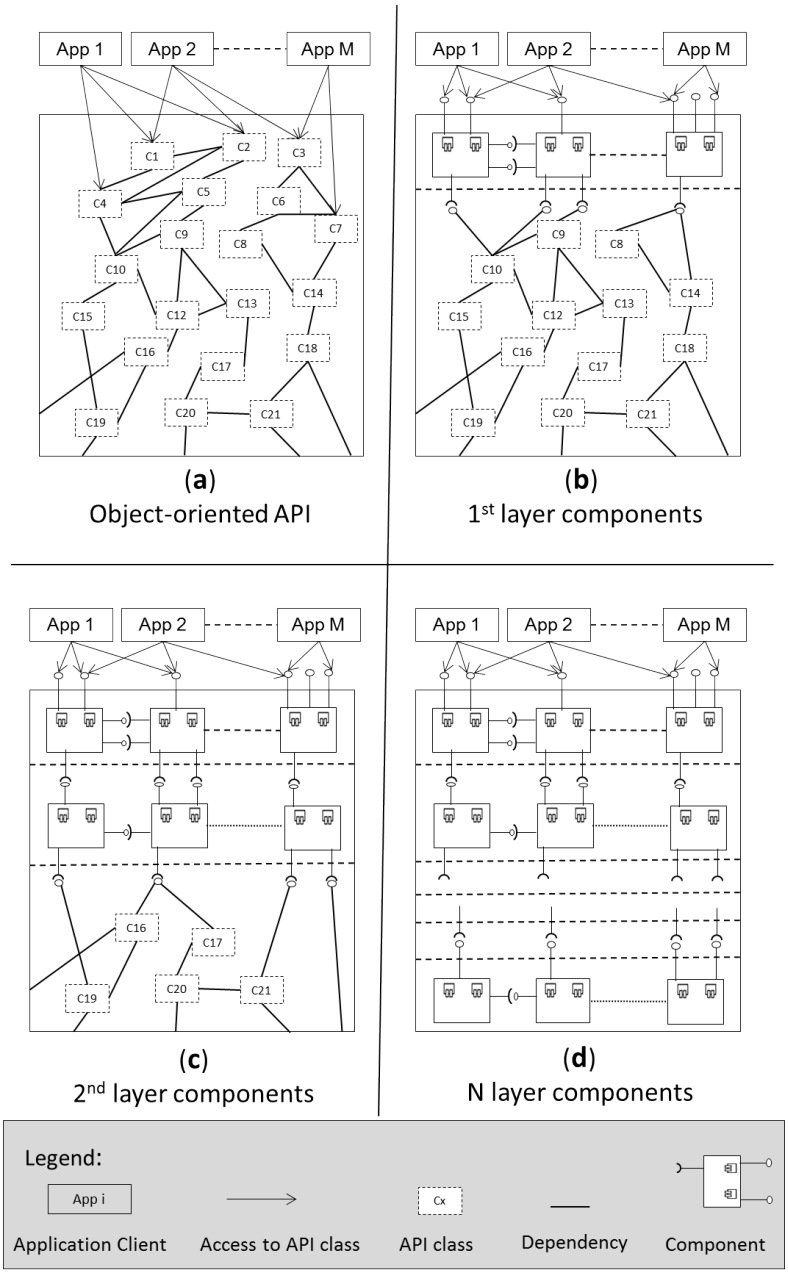


Fig. 11. Identifying component-based API as layers of components

Each interface that is defined as required for a component of layer $L-1$ is considered as provided by a component of layer L except ones provided by the already identified components. The identification of these interfaces is similar to the identification of provided interfaces of the first layer. Thus, we consider that each component (already identified) in layer $L-1$ is a client of the rest of API classes. This means that we collect a set of transactions, such that each transaction consists of classes that are used by a component in layer $L-1$. These transactions are used to identify FUPs based on the FP Growth algorithm. Similar to the first layer, each FUP is divided into groups of classes composing provided interfaces of components in layer L. The partitioning is based on (i) the cohesion of classes, (ii) the lexical similarity of these classes and (iii) the frequency of their simultaneous use. Analogously to the identification of the components of the first layer, the other classes composing each component are identified starting from classes composed of its already identified provided interfaces. Algorithm 5 shows the procedure that identifies component-based API as a set of layers composed of components.

Algorithm 5: Organizing API as Layers of Components

Input: Source Code of a Set of Application Clients(*AppClients*), API Source Code(API)
Output: Component-Based API as Layers of Components($CBAPI$)
$clients$ = *AppClients*;
$layerIndex$ = 1;
while *(|API| > 1)* **do**
  $transactons$ = extractTransactions($clients$, API);
  $FUPs$ = FPGrowth($transactons$, $SupportThreshold$);
  for *each pattern* $\in FUPs$ **do**
    ▷*IQ refers to Equation 2*
    $ProvideInterfaces = ProvideInterfaces \cup$clustering($pattern$, IQ);
  end
  ▷*Identifying classes composing components*
  $components$ = Algorithm4($providedInterfaces, API$);
  $CBAPI$.addLayer($layerIndex$, $components$);
  $layerIndex = layerIndex + 1$;
  $API = API$ - $components$.getClasses();
  $clients = components$;
end
return $CBAPI$;

6 Experimentation and Results

6.1 Experimental Design

Data Collection We collected a set of 100 *Android − Java* applications from open-source repositories[4]. The average size of these applications in terms of number of classes is 90. Table 3 presents the names of these applications. These applications are developed based on classes of the *android* APIs[5]. In our experimentation, we focus on three of these APIs. The first one is the *android.view* composed of 491 classes. This API provides services related to the definition and management of the user interfaces in android applications. The second API is the *android.app* composed of 361 classes. This API provides services related to creating and managing android applications. The last API is the *android* that is composed of 5790 classes. This API includes all of the android services.

Table 3. The selected android applications

ADW Launcher	APV	ARMarker	ARviewer	Alerts
Alogcat	AndorsTrail	AndroMaze	AndroidomaticKeyer	AppsOrganizer
AripucaTracker	AsciiCam	Asqare	AugmentRealityFW	AussieWeatherRadar
AutoAnswer	Avare	BansheeRemote	BiSMoClient	BigPlanetTracks
BinauralBeats	Blokish	BostonBusMap	CH-EtherDroid	CVox
CalendarPicker	CamTimer	ChanImageBrowser	CidrCalculator	ColorPicker
CompareMyDinner	ConnectBot	CorporateAddressBook	Countdown	CountdownTimer
CrossWord	CustomMaps	DIYgenomics	Dazzle	Dialer2
DiskUsage	DistLibrary	Dolphin	Doom	DriSMo
DroidLife	DroidStack	Droidar	ExchangeOWA	FeedGoal
FileManager	FloatingImage	Gcstar	GeekList	GetARobotVPNFrontend
GlTron	GoHome	GoogleMapsSupport	GraphView	HeartSong
Hermit	Historify	Holoken	HotDeath	Introspy
LegoMindstroms	Lexic	LibVoyager	LiveMusic	LocaleBridge
Look	LookSocial	MAME4droid	Macnos	Mandelbrot
Mathdoku	MediaPlayer	Ministocks	MotionDetection	NGNStack
NewspaperPuzzles	OnMyWay	OpenIntents	OpenMap	OpenSudoku
Pedometer	Phoenix	PhotSpot	Prey	PubkeyGenerator
PwdHash	QueueMan	RateBeerMobile	AlienbloodBath	SuperGenPass
SwallowCatcher	Swiftp	Tumblife	VectorPinball	WordSearch

Research Questions and Evaluation Method The approach is evaluated on the collected software applications and APIs. We identify client components independently for each software application. Each component in software is considered as a client of the APIs to form a transaction of classes. Then, we mine Frequent Usage Patterns (FUPs) from the identified transactions. Next, from classes composing each FUP, we identify classes composing a set of component interfaces. Then, we identify all component classes starting from ones composing

[4] *sourceforge.net*, *code.google.com*, *github.com*, *gitorious.org*, and *aopensource.com*
[5] We select android API level 14 as a reference

their interfaces. Lastly, the results related to component-based APIs obtained based on our approach are presented.

We evaluate the obtained components by answering the three following research questions.

- **RQ1: Do the Resulting Component-Based APIs Reduce the Understandability Efforts?** This research question aims at measuring the saved efforts in the API understandability that are brought by migrating object-oriented APIs into component-based ones.
- **RQ2: Are the Mined Components Reusable?** As our approach aims at mining reusable components, we evaluate the reusability of the resulting component. This is based on measuring how much related classes are grouped into the same components.
- **RQ3: Is the Identification of Provided Interfaces Based on FUPs Useful?** The proposed approach identifies the provided interfaces of the components based on how clients have used the API classes (i.e. FUPs). Thus, this research question evaluates how much benefit the use of FUPs brings by comparing components identified by our approach with the ones identified without taking FUPs into account.

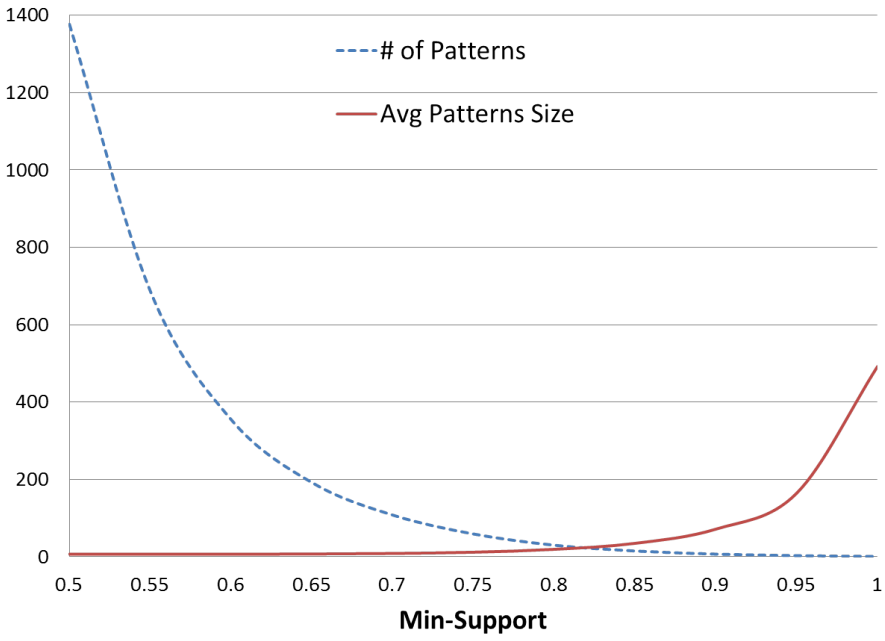


Fig. 12. Changing the support threshold value to mine FUPs in *android* API

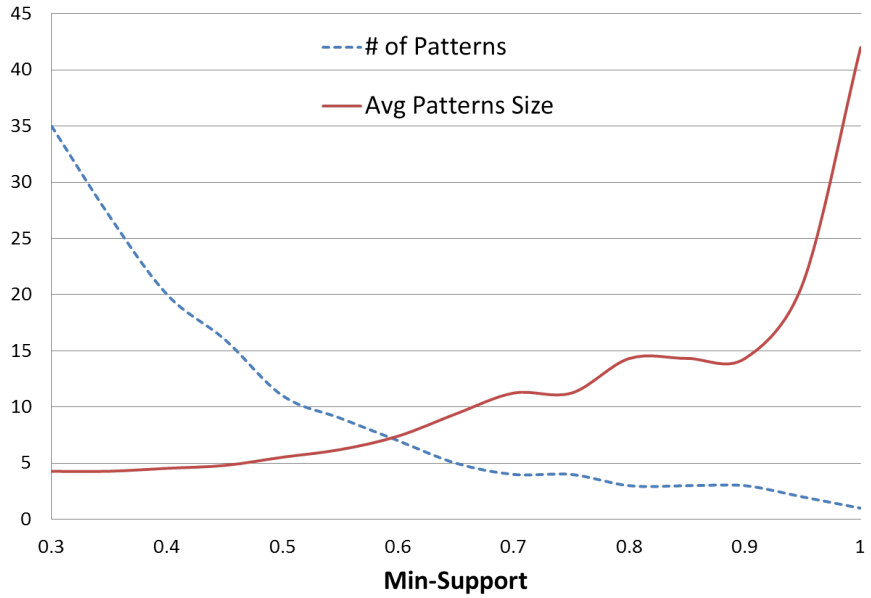


Fig. 13. Changing the support threshold value to mine FUPs in *view* API

6.2 Results

Intermediate Results and Identified Components The average number of client components identified from each software is 4.5 and the average number of classes composing each component is 18.73. Table 4 shows the average number of transactions per software application ($ANTIC$), the average transaction size in terms of classes (ATS), and the percentage of components that have used the API (PCU). The last column of this table shows an example of transactions.

The results show that *android*, *view*, and *app* APIs have been used respectively by only 54%, 29% and 32% of client components. In addition, we note that each client component has used the API classes intensively compared to the number of classes composing it. For example, the transaction size is 17.91 classes for the *view* API, where the average number of classes per component is 18.73. This is due to the fact that classes that serve the same service in software applications, and consequently depend on the same API classes, are grouped together in the same client component.

The identification of FUPs relies on the value of the *Support* threshold. The number and the size of the mined FUPs depend on this value. For all application domains where FUPs are used (e.g. data mining), this value is determined by domain experts. In our approach, to help API experts to determine this value,

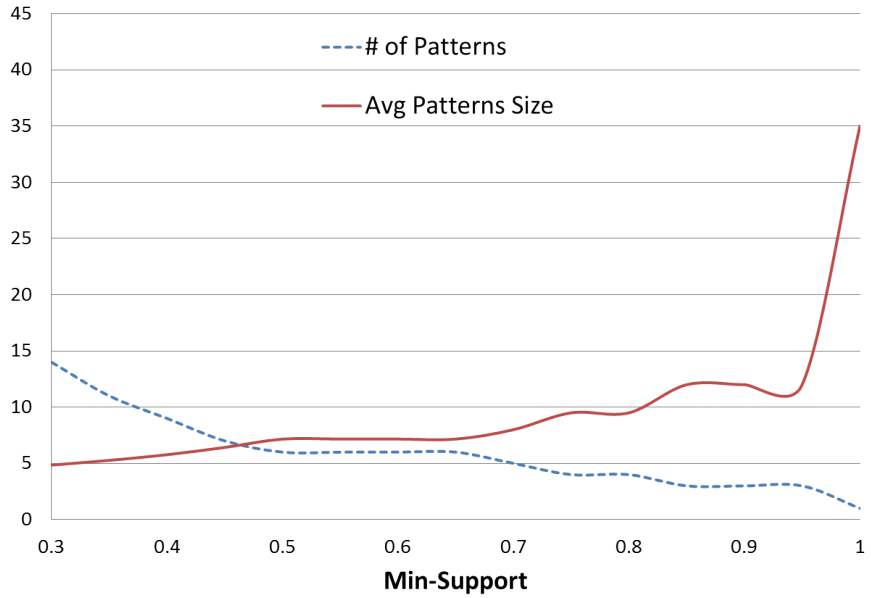


Fig. 14. Changing the support threshold value to mine FUPs in *app* API

Table 4. The Identification of Transactions

API	ANTIC	ATS	PCU	Example
android	2.61	64.82	0.54	Bitmap, Path, Log, Activity, Location, Canvas, Paint, ViewGroup, MotionEvent, View, TextView, GestureDetector
view	1.51	17.91	0.29	MenuItem, Menu, View, ContextMenu, WindowManager, MenuInflater, Display, LayoutInflater
app	1.58	10.90	0.32	ProgressDialog, Dialog, AlertDialog, Activity, ActionBar, Builder, ListActivity

we assign the *Support* threshold values situated in [30%-100%]. We give for each *Support* value the number of the mined FUPs and the average size of the mined FUPs for each API. Figure 12, Figure 13 and Figure 14 respectively refer to the results of the *android*, the *view* and the *app* APIs. The results show that the number of mined FUPs is directly proportional to the *Support* value, while the average size of the mined FUPs is inversely proportional.

Based on their knowledge of the API, API experts can select the value of the *Support*. For example, if the known average number of API classes used together to implement an application service is N, then the experts can choose the *Support* value corresponding to FUPs having N as the average size. Based on

the obtained results and our knowledge of android APIs, [6] we select the *Support* threshold values as 60%, 45%, and 45% respectively for the *android*, the *view* and the *app* APIs.

Table 5 shows examples of the mined FUPs. For instance, the FUP related to *view* API contains 10 classes. The analysis of this FUP shows that it corresponds to three services: animation (*Animation* and *AnimationUtils* classes), view (*Surface, SurfaceView, SurfaceHolder, MeasureSpec, ViewManager* and *MenuInflater* classes), and persistence of the view states (*AbsSavedState* and *AccessibilityRecord* classes). These services are dependent. Animation service needs the view service and the data of animation view needs to be persistent.

Table 5. Examples of the Mined FUPs

API	Example
android	Intent, Context, Log, SharedPreferences, View, TextView, Toast, Activity, Resources
view	Surface, Animation, AnimationUtils, AccessibilityRecord, ViewManager, MenuInflater, AbsSavedState, SurfaceView, SurfaceHolder, MeasureSpec
app	Dialog, Activity, ProgressDialog

In Table 6, we present the results of interface identification in terms of the average number of component interfaces identified from a FUP ($ANCIP$), the average number of classes composing component interfaces ($ACIS$) and the total number of component interfaces in the API ($TNCI$). The last column of this table presents examples of component interfaces identified from the FUPs given in Table 5.

The results show that FUPs contain classes corresponding to a different set of services. On average, each FUP is divided into *1.57*, *2.17* and *2.5* services, such that each service is provided by *5.62*, *2.94* and *4* classes respectively for *android*, *view* and *app* APIs. Figure 15 shows an instance of partitioning a FUP into component interfaces from *view* API. The analysis of classes composing the identified component interfaces shows that they are related to three services; animation, view and persistent of the view states.

Table 6. Identification of Component Interfaces from FUPs

API	ANCIP	ACIS	TNCI	Examples
android	1.57	5.62	232	Activity, View, TextView, Toast
view	2.17	2.94	19	Surface, SurfaceView, SurfaceHolder
app	2.50	4	10	Dialog, ProgressDialog

[6] The authors of this paper are experts on the android APIs

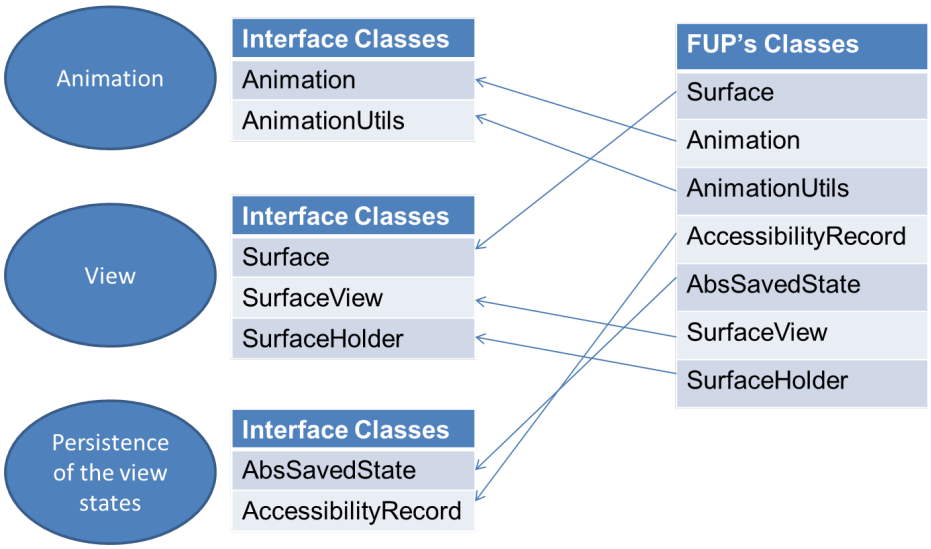


Fig. 15. An instance of partitioning a FUP into component interfaces from *view* API

Table 7 presents the results related to the mined components composing the first API layer. For each API, we give the number of the mined components (NMC) and the average number of classes composing the mined components (ACS). The last column of this table shows examples of classes composing components initially identified from classes composing provided component interfaces presented in Table 6. The results show that the services offered by classes of *android*, *view* and *app* APIs are identified as 232, 19 and 10 components respectively. This means that developers only require to interact with these components to get the required services from these APIs.

Table 7. Identifying Classes Composing Components

API	NMC	ACS	Example
android	232	19.99	Activity, View, TextView, Toast, Drawable, GroupView, Window, Context, ColorStateList, LayoutInflater
view	19	7.49	Surface,SurfaceView, SurfaceHolder, MockView, Display, CallBack
app	10	5.86	Dialog, ProgressDialog, AlertDialog

Table 8 shows the final results obtained from our approach. For each API, we firstly give the size of the API in terms of the number of object-oriented classes composing the API and the number of the identified components. Secondly, we present the total number of used entities (classes and respectively components) by the software clients. The results show that classes involved in providing related

services are grouped into one component. Furthermore, the total number of cohesive and decoupled services is identified for each API. For instance, *android* API consists of 497 components (coarse-grained services), while *view* and *app* APIs contain 43 and 55 components respectively.

Table 8. The Final Results

API Name	API Entity	API size	No. of used Entities
android	*OO*	5790	491
	CB	497	54
view	*OO*	491	42
	CB	43	17
app	*OO*	361	45
	CB	55	5

Answering Research Questions ***RQ1: Do the Resulting Component-Based APIs Reduce the Understandability Efforts?*** The efforts spent to understand an API is directly proportional to the complexity of the API. This complexity is related to the number of API elements and the individual element's complexity. On the one hand, the reduction in the number of elements composing the API is obtained by grouping classes collaborating to provide one coarse-grained service into one component. The results show that the average number of identified components for the studied APIs is 11% (((497/5790) + (43/491) + (55/361)) /3) of the number of classes composing the APIs. This means that the API size is significantly reduced by mapping class-to-component. On the other hand, the reduction in the individual element complexity is done by migrating object-oriented APIs into component-based ones. Meaning, components define their required and provided interfaces, while object-oriented classes at least do not define required interfaces (e.g. a class may call a large number of methods belonging to a set of classes without an explicit specification of these dependencies). The results show that the average number of used components for the APIs is 4% (((54/491) + (17/42) + (5/45)) /3) of the number of used classes. This means that the effort spent to understand API entities is significantly reduced in the case of software applications developed based on API components compared to the development based on API classes. Note that, developers only need to understand the component interfaces, but not the whole component implementation.

RQ2: Are the Mined Components Reusable? We consider that the reusability of a software component is related to the number of used classes among all ones composing the software component. Thus, we calculate the reusability of the component based on the ratio between the numbers of used classes composing the component to the total number of classes composing the component. To prove that our resulting component-based APIs could be generalized to another independent set of client applications, we rely on $K - fold$

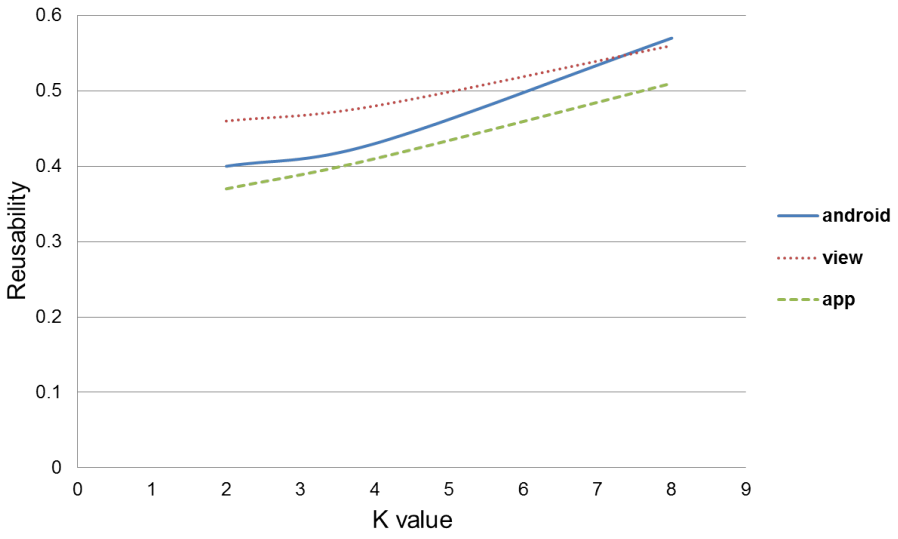
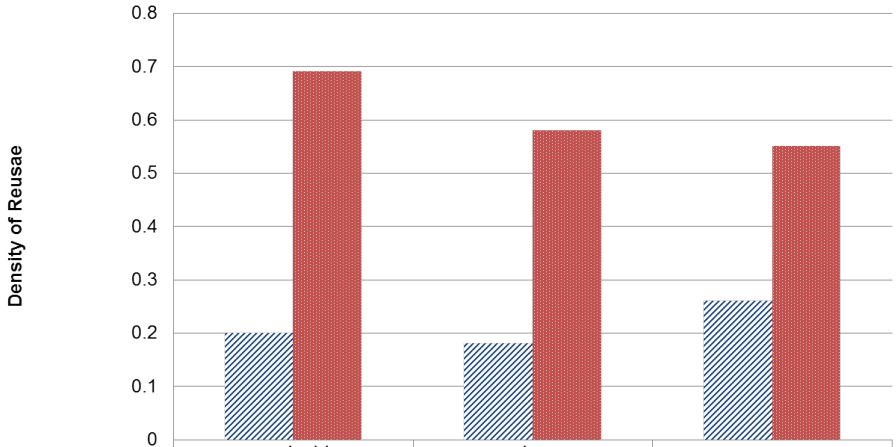


Fig. 16. Reusability validation results

cross validation method [21]. The main idea is to evaluate the model using independent client applications. Thus, K-fold divides the set of client applications into K parts. Then, the identification process is applied K times by considering, each time, $K-1$ different parts for the identification process and by using the remaining part to measure the reusability. Next, we take the average of all K trial results. In our experiment, we set K to 2, 4, and 8.

Figure 16 presents the results of this measurement. These results show that the reusability results are distributed in a disparate manner. The reason behind this dispersion is the size of the train and test data as well as the size of the API. For instance, the average reusability for the *app* API is 37% when the number of train clients is 50 application clients, while it is 51% when the number of train clients is 88 application clients. Thus, the reusability of the components increases when the number of train client applications increases. The results show that our approach identifies reusable components, where the average reusability for all APIs is 47%.

RQ3: Is the Identification of Provided Interfaces Based on FUPs Useful? To prove the utility of using FUPs during the identification process, we compare the components mined based on our approach with ones mined using the ROMANTIC approach, which does not take FUPs into consideration. This is based on the density of use of the component provided interfaces by application clients. The density refers to the ratio between the number of used interface classes to the total number of interface classes for each component. Figure 17 shows the average density for the two identification approaches. These results



	android	view	app
ROMANTIC	0.2	0.18	0.26
Our Approach	0.69	0.58	0.55

Fig. 17. Density validation results

show that our approach outperforms the ROMANTIC approach. For instance, the application clients need to reuse a larger number of components mined based on the ROMANTIC with less density of provided interface classes compared to components mined based on our approach. For instance, the average usage density of classes composing provided interfaces of ROMANTIC components is 21%, while it is 61% for components mined by our approach for all APIs.

6.3 Threats to Validity

Our proposed approach is subjected to two types of threats: internal validity and external validity.

Threats to Internal Validity There are five aspects to be considered regarding the internal validity. These are as follows:

1. The validations of understandability and reusability of the resulted component-based APIs are not directly measured. On the one hand, the understandability is measured through the complexity of the resulted API, while in some cases a complex API can be understandable if it is well documented. However, for the same API, the understandability of a complex version is worse than the understandability of a less complex one, even if both versions are already documented.
   On the other hand, the reusability is measured based on the number of used classes among the ones composing the components. Although the reusability

of components needs to be measured based on their interfaces, this provides an indication of how the component interfaces will be reused by the future software clients.
2. We use FPGrowth algorithm to mine FUPs. Nevertheless, this algorithm has a limitation of ignoring those classes whose patterns' support values do not reach the support threshold (i.e. less commonly used classes). Thus, some of the API classes may not be presented by a FUP. However, we attach each class of them to a FUP holding the maximum support value when it is added. This guarantees that each API class used by software applications is attached to at least one FUP.
3. As our approach is use-driven, the results depend on the quality and the number of usages of the API. This means that identified FUPs rely on the considered software clients. Therefore, the identification of provided interfaces and their corresponding components depends on API clients. Consequently, it is essential to select clients having the largest number of usages of the API.
4. In the case of facing a NP-hard problem, we rely on heuristic algorithms instead of optimal algorithms. This affects the accuracy of the results. However, these heuristics guarantee near-optimal solutions, such as clustering algorithms.
5. We select a static analysis technique to identify dependencies between the classes. However, this analysis affects our results in two axes. The first one is that it does not address polymorphism and dynamic binding. However, in object-oriented programming, the most important dependencies are realized through method calls and access attributes. Thus, not dealing with polymorphism and dynamic binding does not have a high impact on the general results of our approach. The second one is that it does not differ from the used and unused source code. This may provide noise dependencies. However, such situations rarely occur in well designed and implemented software. In contrast, dynamic analysis addresses all of these limitations. But the challenge with dynamic analysis is to identify all use cases of the software.

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

1. The presented approach is experimented via APIs that are implemented by *Android* programming language. However, the obtained results can be generalized for any object-oriented API. The reason behind this generalization is the fact that all object-oriented languages (e.g. *Java*, *C++* and *C#*) are structured in terms of classes and their relationships are realized through method calls, access attributes, etc.
2. The way that API classes are reused together may strongly depend on the choice of the subject software applications, i.e. different software applications may use API classes following different patterns, ending up in different components. In order to prove that our resulting component-based APIs could be generalized into other independent sets of software applications,

we rely on $K - fold$ cross validation method. The cross validation presents whether components identified on n-K software applications can be reused by K software applications.

7 Related Work

To the best of our knowledge, no approach has been proposed to identify components from object-oriented APIs. However, we present three research areas that are related to our approach. The first one aims at identifying components by analyzing object-oriented software applications. The second area aims to identify features from software applications. The third one is related to mining frequent patterns of API usage.

7.1 Identifying Software Components from Software Applications

APIs and software applications are different compared to relationships between classes composing them. In the case of object-oriented applications, classes composing them are structural and behavioral dependent to provide the expected services. For instance, these dependencies are realized via calls between methods, sharing types, etc. For APIs, we distinguish two kinds of dependencies. On the one hand, classes are structural and behavioral dependent, to provide reusable services for software applications. On the other hand, some classes needs to be reused together, i.e. simultaneously, by software applications to implement API services (e.g. $JFrame$ and $Layout$ classes in $java.swing$ API). This kind of dependencies can not be identified by only analyzing the API source code, but also needs the analysis of how software applications use the API classes.

Dependencies between classes composing object-oriented applications are exploited by numerous approaches that aim to identify components from object-oriented applications [25, 26]. In [27], Detten et al. presented the Archimetrix approach, which aims at mining the architecture of legacy software. It relies on a clustering algorithm to partition the system classes into components. This algorithm depends on name resemblance, coupling and cohesion metrics as a fitness function. In [15], Kebir et al. presented an approach to extract components from a single object-oriented software system. Classes composing the extracted components form a partition. Mined components are considered as a part of the component-based architecture of the corresponding software. In [16] Allier et al. depended on dynamic dependencies between classes to recover components. Based on the use case diagram, the execution trace scenarios are identified. Classes that frequently occur in the execution traces are grouped into a single component. Cohesion and coupling metrics are also taken into account during the identification process. Weinreich et al. proposed, in [28], an approach to recover multi-view architecture models of software applications implemented based on service oriented architecture. The authors classified software artifacts based on the information from source code, configuration files and binary codes.

In [29], the author extracted the architecture of an object-oriented software using the fast community detection algorithm. Also, a performance evaluation of fast community and five clustering algorithms is applied. The authors converted the object oriented elements into a graph representation. Then, the algorithms are applied to identify the most connected component within the graph. Lastly, software architects analyze and evaluate the resulted architecture. In [30], an approach has been presented to mine reusable components from a set of similar software applications. A component is considered as more reusable, when it is reused many times by the software applications. The authors firstly identified components independently from each software application. Then, based on the lexical similarity between the classes composing these components, they identified reusable ones. In [31], an approach was presented to visually analyze the distribution of variability and commonality among the source code of product variants. The analysis includes multi-level of abstractions (e.g. line of code, method, class, etc.). This aims to facilitate the interpretation of variability distribution, to support identifying reusable entities.

7.2 Feature Mining from Software Applications

The difference between feature mining and component identification arises from the difference between a feature and a component. The difference is also in the goals and nature of the process. A feature is a non-structural element that provides "user-visible aspect, quality, or characteristic of a software system or systems"[32]. It does not have any interfaces that represent the interaction between features, rather than component interfaces, required and provided ones. In addition to that, features and components belong to different levels of abstraction, where software requirements are abstracted at a high level as features, while a component represents an architectural element at the design level.

There are many approaches presented to address feature location and feature identification. These aims to identify program units such as methods, or classes that represent features. In [33], a survey of them is presented. These approaches identify features based on the analysis of single software application, such as [34–36], and multiple software applications, such as [37, 38].

7.3 Mining Frequent Patterns of API Usage

FUPs are observations made based on the analysis of previous uses of APIs. They aim to help users of APIs by identifying recurring patterns, composed of API elements frequently used together. FUPs and components serve reuse needs in two different ways. Components are entities that can be directly reused and integrated into software applications, while FUPs are guides for reuse and not entities for reuse. In addition, components and FUPs are structurally different. Related to *Specificity* characteristic, classes composing a component serve a coherent body of services, while a FUP may be related to different services. Concerning *Autonomy* characteristic, dependencies of component's classes are

mostly internal, which forms an autonomous entity. FUP's can be very dependent on other API classes that are not directly used by clients of APIs. Concerning *Composability* characteristic, a component is structured and reused via interfaces, while FUPs are not directly reusable entities.

Several approaches have been proposed to mine FUPs based on the analysis of API clients. Robillard et al. provide a survey of these approaches [8]. These approaches can be mainly classified based on four main criteria. The first one is related to the goal, which can be either giving examples and recommendations of how to use API entities such as [10, 5], supporting the documentation of APIs like [10, 7], or improving the bug detection task such as [11]. The second criterion is related to pattern ordering, where some approaches mine ordered patterns like [10, 7], while other ones mine unordered patterns such as [11, 39]. The third one concerns the granularity of the elements composing patterns. For examples, in [10, 7], the approaches mine patterns composed of methods, and the approach in [39] mines patterns composed of classes. The fourth one related to the technique that is used to identify the patterns. The used technique can be association rules mining like [39], clustering algorithms such as [7] or a heuristic defined by the authors such as [10, 11]. Some approaches combine many techniques, e.g., Unddin et al. used Principle Component Analysis with clustering algorithm [5], and Buse and Weimer combined the clustering algorithm with their own proposed heuristic [40].

8 Conclusion and Future Work

8.1 Conclusion

In this paper, we presented an approach that aims to mine software components from object-oriented APIs. This is based on static analysis of the source code of both the APIs and their software clients, in order to analyze the way that the software clients have used the API classes. The component identification process is use-driven. It implies that components are identified starting from classes composing their interfaces. Classes composing the provided interface of the first layer components compose FUPs. Then, the API is organized by a set of layers, where each layer composes of components providing services to the others composing the above layer, and so on.

To validate the presented approach, we experimented it by applying on a set of open source *Java* applications as clients for three android APIs. The validation is done through three research questions. The first one is related to the understandability, while the second indicates to the reusability. The results show that our approach improves the reusability and the understandability of the API. The third research question aims at compare our approach with a traditional component identification approach. The results prove that our approach outperforms the traditional one.

8.2 Future Work

There are many future directions that are indicated by this research. These include:

1. **Migrating the identified object-oriented components into existing component models.** Components are identified as clusters of object-oriented classes representing their implementation. This constitutes the first step of the reengineering process of object-oriented software into component-based software. Thus, we plan to extend our approach by transforming the object-oriented implementation of the identified components into an equivalent component-based one, such as OSGi [41] and Fractal [42].
2. **Developing a visual environment.** The presented approach can be extended by providing a visual environment, such that domain experts can supervise the approach steps and modify the obtained results when needed.
3. **Experimenting with large number of case studies.** The selection of API client applications affects the resulted component-based API. Thus, we plan to extend the evaluation of the proposed approach by conducting more case studies in order to further test the approach and to generalize the results as well.
4. **Validating our approach by human experts.** The results of the presented approach are validated based on heuristic measurements that we proposed. To better validate our approach, we plan to validate the results using the help of human experts.

References

1. W.B. Frakes and Kyo Kang. Software reuse research: status and future. *IEEE Transactions on Software Engineering*, 31(7):529–536, 2005.
2. M.F. Zibran, F.Z. Eishita, and C.K. Roy. Useful, but usable? factors affecting the usability of apis. In *18th Working Conf. on Reverse Engineering (WCRE)*, pages 151–155, 2011.
3. M. Monperrus, M. Eichberg, E. Tekes, and M. Mezini. What should developers be aware of? an empirical study on the directives of api documentation. *Empirical Software Engineering*, 17(6):703–737, 2012.
4. Homan Ma, R. Amor, and E. Tempero. Usage patterns of the java standard api. In *13th Asia Pacific Software Engineering Conf. APSEC 2006*, pages 342–352, 2006.
5. G. Uddin, B. Dagenais, and M. P. Robillard. Temporal analysis of api usage concepts. In *Proc. of the 2012 Inter. Conf. on Software Engineering*, ICSE 2012, pages 804–814, Piscataway, NJ, USA, 2012. IEEE Press.
6. Mithun Acharya, Tao Xie, Jian Pei, and Jun Xu. Mining api patterns as partial orders from source code: from usage scenarios to specifications. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 25–34. ACM, 2007.
7. J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang. Mining succinct and high-coverage api usage patterns from source code. In *Proc. of the 10th Working Conf. on Mining Software Repositories*, MSR '13, pages 319–328, Piscataway, NJ, USA, 2013. IEEE Press.

8. M.P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford. Automated api property inference techniques. *IEEE Transactions on Software Engineering*, 39(5):613–637, 2013.
9. Google. Api guides (http://developer.android.com/reference/packages.html), 2015.
10. J.E. Montandon, H. Borges, D. Felix, and M.T. Valente. Documenting apis with examples: Lessons learned with the apiminer platform. In *20th Working Conf. on Reverse Engineering (WCRE)*, pages 401–408, 2013.
11. M. Monperrus, M. Bruch, and M. Mezini. Detecting missing method calls in object-oriented software. In *European Conf. on Object-Oriented Programming ECOOP*, volume 6183 of *Lecture Notes in Computer Science*, pages 2–25. Springer Berlin Heidelberg, 2010.
12. W. Maalej and M.P. Robillard. Patterns of knowledge in api reference documentation. *IEEE Transactions on Software Engineering*, 39(9):1264–1282, 2013.
13. C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Pearson Education, 2002.
14. S. Mishra, D. S. Kushwaha, and A. K. Misra. Creating reusable software component from object-oriented legacy system through reverse engineering. *Journal of object technology, 8(5), 133-152, 2009.*
15. S. Kebir, A.-D. Seriai, S. Chardigny, and A. Chaoui. Quality-centric approach for software component identification from object-oriented code. In *Joint Working IEEE/IFIP Conf. and European Conf. on Software Architecture (WICSA)/(ECSA), 2012*, pages 181–190, 2012.
16. S. Allier, S. Sadou, H. Sahraoui, and R. Fleurquin. From object-oriented applications to component-oriented applications via component-oriented architecture. In *2011 9th Working IEEE/IFIP Conf. on Software Architecture (WICSA)*, pages 214–223. IEEE, 2011.
17. Anas Shatnawi, Abdelhak Seriai, Houari A. Sahraoui, and Zakarea Al-Shara. Mining software components from object-oriented apis. In *Software Reuse for Dynamic Systems in the Cloud and Beyond - 14th International Conference on Software Reuse, ICSR 2015, Miami, FL, USA, January 4-6, 2015. Proceedings*, pages 330–347, 2015.
18. S. Chardigny, A. Seriai, M. Oussalah, and D. Tamzalit. Extraction of component-based architecture from object-oriented systems. In *Seventh Working IEEE/IFIP Conf. on Software Architecture (WICSA)*, pages 285–288, 2008.
19. ISO. Software Engineering – Product Quality – Part 1: Quality Model. Technical Report ISO/IEC 9126-1, International Organization for Standardization, 2001.
20. S. Chardigny, A.-D. Seriai, M. Oussalah, and D. Tamzalit. Search-based extraction of component-based architecture from object-oriented systems. In *2nd European Conf. in Software Architecture (ECSA)*, volume 5292 of *Lecture Notes in Computer Science*, pages 322–325. Springer Berlin Heidelberg, 2008.
21. J. Han, M. Kamber, and J. Pei. *Data mining: concepts and techniques*. Morgan kaufmann, 2006.
22. Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In *ACM SIGMOD Record*, volume 29, pages 1–12. ACM, 2000.
23. J. M. Bieman and B.-K. Kang. Cohesion and reuse in an object-oriented system. In *Proc. of the 1995 Symposium on Software Reusability*, SSR '95, pages 259–262, New York, NY, USA, 1995. ACM.
24. D. Poshyvanyk and A Marcus. The conceptual coupling metrics for object-oriented systems. In *22nd IEEE Inter. Conf. on Software Maintenance (ICSM), 2006*, pages 469–478, Sept 2006.

25. J. Garcia, I. Ivkovic, and N. Medvidovic. A comparative analysis of software architecture recovery techniques. In *IEEE/ACM 28th Inter. Conf. on Automated Software Engineering (ASE)*, pages 486–496, Nov 2013.
26. Stéphane Ducasse and Damien Pollet. Software architecture reconstruction: A process-oriented taxonomy. *Software Engineering, IEEE Transactions on*, 35(4):573–591, 2009.
27. M. von Detten, M. C. Platenius, and S. Becker. Reengineering component-based software systems with archimetrix. *Software & Systems Modeling*, pages 1–30, 2013.
28. R. Weinreich, C. Miesbauer, G. Buchgeher, and T. Kriechbaum. Extracting and facilitating architecture in service-oriented software systems. In *Joint Working IEEE/IFIP Conf. on Software Architecture (WICSA) and European Conf. on Software Architecture (ECSA)*, pages 81–90, Aug 2012.
29. Ural Erdemir, Umut Tekin, and Feza Buzluca. Object oriented software clustering based on community structure. In *2011 18th Asia Pacific Software Engineering Conference (APSEC)*, pages 315–321. IEEE, 2011.
30. A. Shatnawi and A.-D. Seriai. Mining reusable software components from object-oriented source code of a set of similar software. In *IEEE 14th Inter. Conf. on Information Reuse and Integration (IRI)*, pages 193–200, 2013.
31. S. Duszynski, J. Knodel, and M. Becker. Analyzing the source code of multiple software variants for reuse potential. In *Proc. of WCRE*, pages 303–307. IEEE, 2011.
32. K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, DTIC Document, 1990.
33. B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process*, 25(1):53–95, 2013.
34. Giuliano Antoniol and Y-G Guéhéneuc. Feature identification: a novel approach and a case study. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pages 357–366. IEEE, 2005.
35. Kunrong Chen and Václav Rajlich. Case study of feature location using dependence graph. In *In Proceedings of the 8th International Workshop on Program Comprehension*, 2000.
36. Robertas Damaševičius, Paulius Paškevičius, Eimutis Karčiauskas, and Romas Marcinkevičius. Automatic extraction of features and generation of feature models from java programs. *Information Technology And Control*, 41(4):376–384, 2012.
37. Yinxing Xue. Reengineering legacy software products into software product line based on automatic variability analysis. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 1114–1117. ACM, 2011.
38. Tewfik Ziadi, Luz Frias, Marcos Aurélio Almeida da Silva, and Mikal Ziane. Feature identification from the source code of product variants. In *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, pages 417–422. IEEE, 2012.
39. M. Bruch, T. Schäfer, and M. Mezini. Fruit: Ide support for framework understanding. In *Proc. of the 2006 OOPSLA Workshop on Eclipse Technology eXchange*, eclipse '06, pages 55–59, New York, NY, USA, 2006. ACM.
40. R. P. L. Buse and W. Weimer. Synthesizing api usage examples. In *Proc. of the 2012 Inter. Conf. on Software Engineering*, ICSE 2012, pages 782–792, Piscataway, NJ, USA, 2012. IEEE Press.

41. A. L. C. Tavares and M. T. Valente. A gentle introduction to osgi. *SIGSOFT Softw. Eng. Notes*, 33(5):8:1–8:5, August 2008.
42. E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, and J.-B. Stefani. The fractal component model and its support in java. *Software: Practice and Experience*, 36(11-12):1257–1284, 2006.