Quality-Centric Approach for Software Component Identification from Object-Oriented Code

Selim Kebir, Abdelhak-Djamel Seriai
LIRMM
University of Monpellier 2
France
Email: smkebir,seriai@lirmm.fr

Sylvain Chardigny
MGPS
Port-Saint-Louis
France
Email: s.chardigny@mgps.nfo

Allaoua Chaoui
MISC
University of Constantine
Algeria
Email: a_chaoui2001@yahoo.com

***Abstract*—Components and connectors are the main building blocks for software architectures. In the design phase of a software system, components can either be created from scratch or reused. When reused, they can either exist on component shelves or identified from existing software systems. Thus, software component identification is one of the primary challenges in component based software engineering. Typically, the identification is done by analyzing existing software artifacts. When considering object-oriented systems, many approaches have been proposed to deal with this issue by identifying a component as a high cohesive and loose coupled set of classes. However, this assumption leads to two main limitations: in one hand, the focus on simple metrics like high cohesion and loose coupling will not necessarily lead to the identification of good quality components. On the other hand, the identified components external structure (provided and required interfaces) is missing. As a result, the identified components can hardly be reused, composed, packaged and documented. To overcome these limitations, we propose in this paper an approach for identifying components based on a fitness function to measure the quality of a component. To evaluate this function, we use a semantic- correctness model defined in our previous works. Also, we propose to identify provided and required interfaces of components.**

***Keywords*-software component; object oriented; reverse engineering; reuse; quality; legacy systems reengineering;**

I. Introduction

Components and connectors constitute the main building blocks for software architectures [1]. In the design phase of a software system, components are either developed from scratch or reused [1]. When reused, they can either exist on component shelves or identified from existing systems. Thus, software component identification is one of the primary research problems in CBSE [2]. When applied to object oriented systems, software component identification results in a set of components where each one contains a set of classes. On the one hand, the relationship between classes belonging to a component must be high. On the other hand, the relationship between classes belonging to two different components must be low. Otherwise, software component identification can be performed in two different manners [2]: top-down and bottom-up. Top-down software component identification is performed by analyzing domain business models to get a set of business components. Bottom-up software component identification is performed by extracting reusable software components from existing software system source code.

Most of the software component identification techniques belong to the first category [3] [2] [4] [5]. This means that they start from semi-formal domain business models (Typically expressed in UML) and produce domain software components. This constitutes an important shortcoming like the inability to apply these approaches when domain business models are missing.

Even if these artifacts are available, in most of time, they do not express the true reality of the system due to the erosion phenomenon [6]. The reality of the system is reflected by its source code and this latter is the only artifact always available for legacy systems.

In addition, in most of the existing software component identification approaches, the result is a set of components with high intra-component cohesion and low inter- component coupling [2] [3] [4] [5] [7]. Components identified using these approaches are not necessarily of high quality due to the lack of a semantic-correctness model [8] [9] to identify the best components of the system. Another shortcoming of the existing approaches is that they identify only the classes belonging to a component without specifying its external structure (i.e. provided and required interfaces). Thus, the identified components cannot be directly used, composed or packaged for future reuse.

In this paper, we propose a bottom-up approach for software component identification from existing object- oriented source code that avoids the previous limitations.

In our previous works [8] [9], we have defined a semantic-correctness model for extracting software architectures from object-oriented source code. We rely on these results to propose a software component identification technique. Our approach is based on a mapping model between object-oriented and component-based concepts and a fitness function to measure the semantic-correctness of a component. Beyond identifying the internal structure of software components,

978-0-7695-4827-2/12 $26.00 © 2012 IEEE
DOI 10.1109/WICSA-ECSA.212.26

our approach identifies also their required and provided interfaces. In addition, we propose two strategies to identify components: explorative and requirement-driven. The former allows the identification of all potentially software components in an object oriented system. The latter allows the identification of components that meet some specific requirements.

This paper is organized as follows: Section 2 introduces the mapping model between object and component concepts and the measurement model used to evaluate the semantic-correctness of a component. In section 3, we present our component identification approach based on the above-mentioned models. In section 4, we describe how interfaces of components are identified. Next, a heuristic-based method for naming identified components and interfaces is presented in section 5. Section 6 presents the application of our approach on different systems of various sizes. In section 7, we present a comparative study with related works. Section 8 concludes the paper.

II. Mapping Object to Component

To identify software components from object-oriented source code, we base our self on an adaptation of two elements presented in our previous works [8] [9]:

- A mapping model between object-oriented concepts (i.e. classes, interfaces, packages, etc) and component based software engineering ones (i.e. components, interfaces, sub-component, etc.).
- A measurement model of semantic- correctness of a component. This model refines characteristics of a component to measurable metrics. Based on these metrics, we define a fitness function to measure the the semantic-correctness of a component.

A. Mapping model between component and object concepts

We define components as disjoint collections of classes. These collections are named shape and contain classes which can belong to different object-oriented packages (cf. Fig.1).

Each shape is composed of two sets of classes: the shape interface which is the set of classes which have a link with some classes from the outside of the shape, e.g. a method call or attribute use from the outside; and the center which is the remainder of shape. As shown in Fig.1, we assimilate component interface set to shape interface and component to shape. Figure 2 shows the component-object mapping model that we propose to handle the correspondence between object and component concepts.

B. Semantic-correctness of components

In order to measure the component semantic-correctness, we study component characteristics. This study is based on the most commonly admitted definitions of software component. Many definitions exist where each one characterizes a component somewhat differently [10]. Nonetheless, some

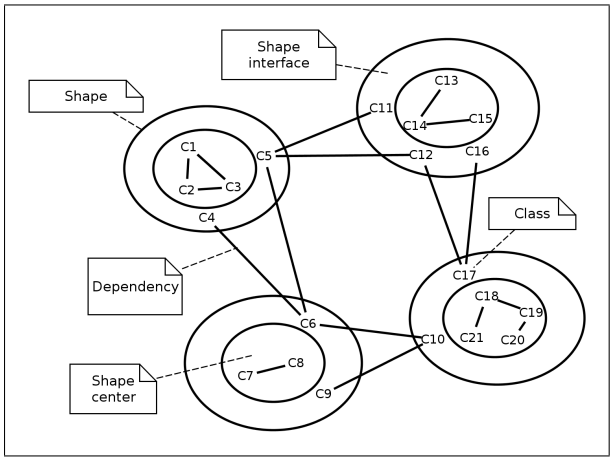


Figure 1. Component Structure

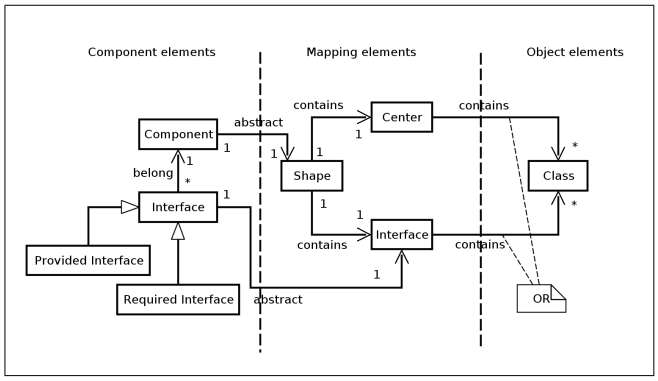


Figure 2. Object-component mapping model

important commonalities exist among the most prevalent definitions.

Szyperski defines, in [11], a component as 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. In [12], Heinemann and Councill define a component as a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard. Finally, Luer, in [13], makes a distinction between component and deployable component. He defines a component as a software element that (a) encapsulates a reusable implementation of functionnality, (b) can be composed without modification, and (c) adheres to a component model. A deployable component is a component that is (a) prepackaged, (b) independently distributed, (c) easily installed and uninstalled, and (d) self-descriptive.

In combining and refining the common elements of these definitions and others commonly accepted, we propose the

following definition of a component:

A component is a software element that (a) can be composed without modification, (b) can be distributed in an autonomous way, (c) encapsulates an implementation of functionality, and (d) adheres to a component model. Our component definition references component model. In our approach, the definition of a component model is the Luer one [13]: a model component is the combination of (a) a component standard that governs how to construct individual components and (b) a composition standard that governs how to organize a set of components into an application and how those components globally communicate and interact with each other.

As compared to the definitions of Luer and Heineman and Councill, we intentionally do not include the criterion that a component must adhere on a composition theory and the self-descriptive, pre-packaged and easy to install and uninstall properties of component. These are covered through the criterion that a component must adhere to a component model and does not need to be repeated.

In conclusion, according to our software component definition, we identify three semantic characteristics of software components: composability, autonomy and specificity.

1) From Characteristics to Properties: In the previous section, we have identified three semantic characteristics that we propose to evaluate. To do so, we adapt the characteristic refinement model given by the norm ISO-9126 [14]. According to this model, we can measure the characteristic semantic-correctness by refining it in the previous three semantic characteristics which are consequently considered as sub-characteristics. Based on the study of the semantic sub-characteristics, we refine them to a set of component measurable properties. Thus,

- A component is autonomous if it has no required interface. Consequently, the property number of required interfaces should give us a good measure of the component autonomy.
- Then, a component can be composed by means of its provided and required interfaces. However, component will be more easily composed with another if services, in each interface, are cohesive. Thus, the property average of service cohesion by component interface should be a correct measure of the component composability.
- Finally, the evaluation of the number of functionalities is based on the following statements. Firstly a component which provides many interfaces may provide various functionalities. Indeed each interface can offer different services. Thus the higher the number of interfaces is, the higher the number of functionalities can be. Secondly if interfaces (resp. services in each interface) are cohesive (i.e. share resources), they probably offer closely related functionalities. Thirdly if the code of the component is closely coupled (resp. cohesive), the different parts of the component code use each other (resp. common resources). Consequently, they probably work together in order to offer a small number of functionalities. From these statements, we refine the specificity sub characteristic to the following properties: number of provided interfaces, average of service cohesion by component interface, component interface cohesion and component cohesion and coupling.

2) From Properties to Metrics: We cannot define the metrics which measure these properties on shapes. Consequently, according to our semantic-correctness measurement model, we link the component properties to shape measurable properties.

- Firstly, according to our mapping model, component interface set is linked to the shape interface. As a result the average of the interface-class cohesion gives a correct measure of the average of service cohesion by component interface.
- Secondly the component interface cohesion, the internal component cohesion and the internal component coupling can respectively be measured by the properties interface class cohesion, shape class cohesion and shape class coupling.
- Thirdly in order to link the number of provided interfaces property to a shape property, we associate a component provided interface to each shape-interface class having public methods. Thanks to this choice, we can measure the number of provided interfaces using the number of shape interface classes having public methods.
- Finally, the number of required interfaces can be evaluated by using coupling between the component and the outside. This coupling is linked to shape external coupling. Consequently, we can measure this property using the property shape external coupling. In order to measure these properties, we need to define metrics.

The properties shape class coupling and shape external coupling require a coupling measurement. We define the metric Coupl(E) which measures the coupling of a shape E and CouplExt(E) which measures the coupling of E with the rest of classes. They measure three types of dependencies between objects: method calls, use of attributes and parameters of another class. Moreover they are percentages and are related through the equation: couplExt(E) = 100 coupl(E). Due to space limitations, we do not detail these metrics. Shape properties average of interface-class cohesion, interface-class cohesion, and shape-class cohesion require a cohesion measurement. The metric Loose Class Cohesion (LCC), proposed by Bieman and Kang [15], measures the percentage of pair of methods which are directly or indirectly connected. Two methods are connected if they use directly or indirectly a common attribute. Two methods are indirectly connected if a connected method chain connects them. This metric satisfies all our needs for the cohesion

measurement: it reflects all sharing relations, i.e. sharing attributes in object oriented system, and it is a percentage. Consequently, we use this metric to compute the cohesion for these properties. The refinement model is summarized in Fig.3.

3) Evaluation of the semantic-correctness: According to the links previously established between the sub-characteristics and the shape properties, we define the evaluation functions Spe, A and C respectively for specificity, autonomy and composability, where $nbPub(I)$ is the number of interface classes having a public method and I is the shape interface of the component shape E:

1) $Spe(E) = \frac{1}{5} \cdot (\frac{1}{|I|} \cdot \sum_{i \in I} LCC(i) + LCC(I) + LCC(E) + Coupl(E) + nbPub(I))$
2) $A(E) = couplExt(E) = 100 - coupl(E)$
3) $C(E) = \frac{1}{|I|} \cdot \sum_{i \in I} LCC(i)$

The evaluation of the semantic-correctness characteristic is based on the evaluation of each sub- characteristic. That is why we define this function as a linear combination of each fitness function of sub- characteristics (i.e. Spe, A, and C):

$$S(E) = \frac{1}{\sum_i \lambda_i} \cdot (\lambda_1 \cdot Spe(E) + \lambda_2 \cdot A(E) + \lambda_3 \cdot C(E))$$

This form is linear because each of its parts must be considered uniformly. λ_i is the weight associated with each sub function. It allows the software architect to modify, as needed, the importance of each sub-characteristic.

III. Identification of Components

According to the mapping model between component and object concepts, the content of a component matches a set of classes. Thus, in order to define the sets of classes that can belong to a component it is necessary to define a process for grouping these classes. This association must be based on a number of criteria to maximize the value of the fitness function in these groups.

In addition to the fitness function, it is necessary to define an algorithm which allows us to identify groups of classes. Among the possible algorithms, we opted for a clustering algorithm. This kind of algorithm is used for grouping elements using a similarity function. This makes it suitable for our problem insofar the fitness function defined below will play the role of a similarity function.

Clustering approaches can be classified as hierarchical or non-hierarchical. Hierarchical clustering techniques are further divided into agglomerative and divisive techniques. An agglomerative method involves a series of successive mergers whereas a divisive method involves a series of successive divisions [16]. The approach proposed here makes use of a hierarchical agglomerative clustering algorithm (cf. Algorithm 1) for grouping classes. The strength of the relationship between the classes is used as basis for clustering them. This strength is measured using the fitness function defined previously.

Algorithm 1 HierarchicalClustering(file code):Tree dendro

```
classes ← extractInformation(code);
clusters ← classes;
while (|clusters| > 1) do
  (c1, c2) ← nearestClusters(clusters);
  c3 ← Cluster(c1, c2);
  remove(c1, clusters);
  remove(c2, clusters);
  add(c3, clusters);
end while
dendro ← get(0, clusters);
return dednro;
```

The technique proceeds through a series of successive binary mergers (agglomerations), initially of individual entities (classes) and later of clusters formed during the previous stages. The classes having the highest relationship strengths are grouped first. The process continues until a cut-off point is reached. We obtain from this single cluster a dendrogram which represents the shape hierarchy. This dendrogram contains all candidate components. The presented algorithm uses the nearestClusters() function to determine which two clusters will be merged in the next step. This function returns the two clusters that maximize the value of the fitness function.

A. Explorative versus requirement-driven identification approaches

We distinguish between two types of component identification: explorative and requirement-driven identification. The explorative identification consists in using a clustering algorithm to identify all potential components of an object-oriented system. The requirement-driven identification consists in identifying components that meet some requirements. Requirements can consist, for example of list of classes or interfaces that must belong to a component.

We propose an algorithm for each type of identification. These algorithms considers that a dendrogram representation of the systems has been already extracted using the algorithm 1.

1) Explorative Identification of Software Components: In order to obtain a partition of classes, we have to select nodes among the hierarchy resulting from the previous step. This selection is done by an algorithm based on a depth-first search (Algorithm 2) which selects nodes representing shapes which will be the best components.

For each node, we compare the result of the fitness function for the node shape and for its sons. If the node result is inferior to the average of the result of its two sons, then the algorithm continues on the next node or else the node is identified to a shape, added to the partition and the

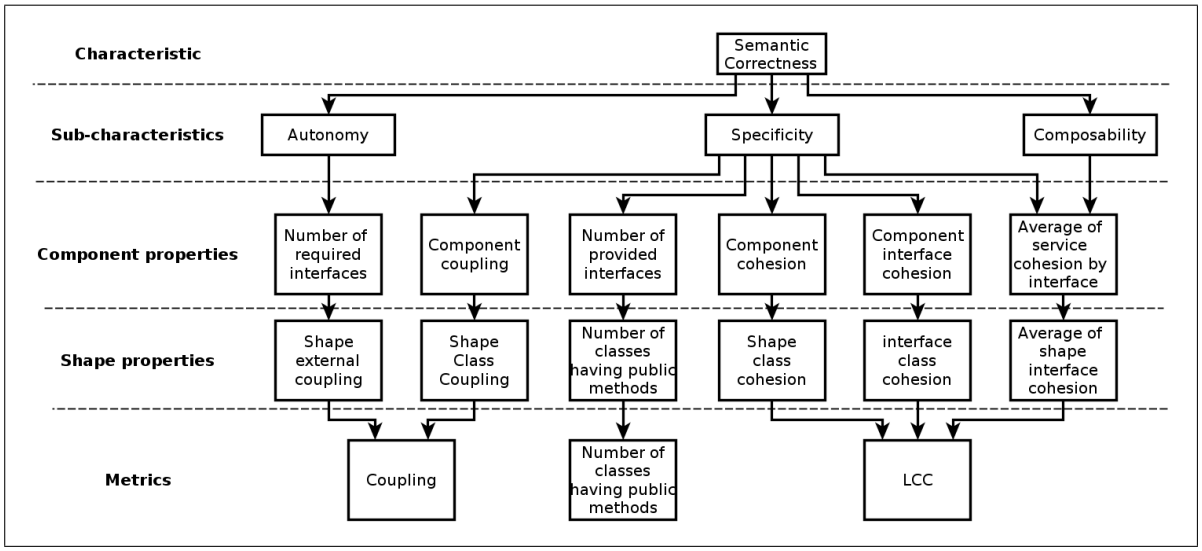


Figure 3. The refinement model for semantic-correctness

Algorithm 2 DendrogramTraversal(Tree dendro):Partition R

```
stack traversalClusters;
push(root(dendro), traversalClusters);
while (!empty(traversalClusters)) do
  Cluster father = pop(traversalClusters);
  Cluster f1 = son1(father, dendro);
  Cluster f2 = son2(father, dendro);
  if (S(father) > average(S(f1), S(f2))) then
    add(father, R);
  else
    push(f1, traversalClusters);
    push(f2, traversalClusters);
  end if
end while
return R;
```

algorithm computes the next node. In this way, good quality components will be identified while the traversal continues.

2) Requirement-driven Identification of Software Components: An architect may need to identify some components that meet some specific needs. To do this, he can define one or more key entities. These key entities can be mainly classes and/or interfaces. Our approach identifies a component around these key entities. We call this type of component identification requirement-driven because the result here is not a disjoint partition of all the system classes. The requirement-driven identification extracts a component around one or more key entities. In the case where these entities are interfaces, the process firstly extract all the classes that implement (resp. require) these provided (resp. required) interfaces. In addition to these key entities, the requirement-driven identification process accepts as input a set of constraints that the identified component must meet. For example, the architect can choose the maximal size (number of classes) of the obtained component, the quality threshold and the non-membership of a class in the desired component.

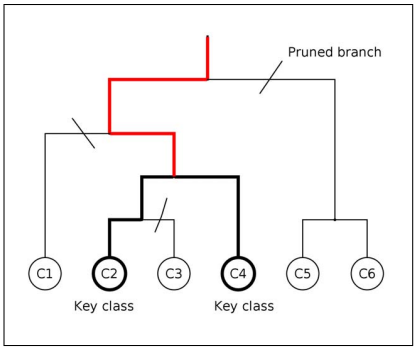
The approach that we propose for requirement-driven identification uses the dendrogram representation of the system. The architect can choose if the key entities must be grouped together from the beginning or not. Then, we proceed with the next step which consists in pruning the dendrogram to obtain a linked list that begin with the last node that contains all key entities and ends with the root of the tree.

Fig.4.a shows the result when the key entities are grouped together from the beginning. Fig.4.b shows the resulting dendrogram when the key entities are not grouped together.

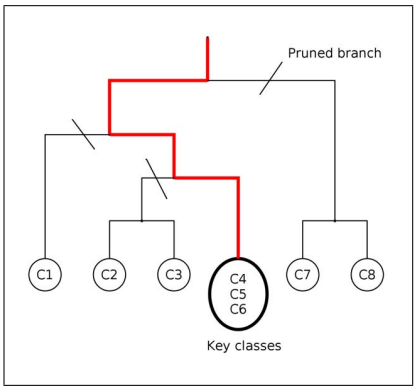
Next, the linked list produced from the previous step (In bold in Fig.4) is traversed from the bottom to the top. At each step, one or more classes are added to the component and the constraints given by the architect are evaluated. The traversal stops when one or more constraints are not satisfied. The linked list traversal algorithm is given below:

Algorithm 3 Traversal(LinkedList l, Constraints c):Cluster result

```
while (hasNext(l)) do
  l ← next(l);
  Cluster component ← node(l);
  if !(meet(component, c) then
    return previous(l);
  end if
end while
```



(a)



(b)

Figure 4. Pruned dendrogram with key entities grouped together

In the previous algorithm, $next()$ and $previous()$ functions return respectively the next node and the previous node in the linked list. $Meet(a, c)$ function returns true if the component a satisfies the constraints c.

IV. Identification of Interfaces

In our approach, a shape is a set of classes that maximizes the fitness function measuring the semantic-correctness of a component. This definition alone does not allow an identified component to be reused or assembled with other components because it only indicates the internal structure of a component and not its interface.

We propose some heuristics and a clustering algorithm to identify the required and provided interfaces of a component. As shown in Fig.1, a component shape is composed of two parts: a shape center and a shape interface. Shape center contains classes that do not have relationships with the outside. Shape interface contains classes that invoke methods from the outside and classes that have methods invoked from the outside. These classes contain the methods that represent the input of our algorithm.

A. Heuristics for interface identification

We rely on heuristics to measure the quality of interfaces. Next, we give some details of these heuristics and the metrics that they involve.

1) Heuristic 1: Method cohesion: Whether provided or required, an interface is a set of strongly-related methods with high cohesion. As in component identification, we use the LCC metric to evaluate how much a set of methods are cohesive.

2) Heuristic 2: Existing object oriented interfaces: The meanings of an interface in object-oriented design and in component-based software engineering are slightly different. We consider that if two methods belong to the same object-oriented interface, this increases their probability to belong to the same component interface. We propose to measure the number of methods belonging to the same object oriented interface using the $sameOOInterface(S : Setofmethods)$ function. This function returns the size of the greatest subset of S that contains methods belonging to the same object oriented interface divided by the size of S.

3) Heuristic 3: Correlation of method use: A provided interface is a set of services that a component offers to other components. The identification of provided interfaces in a component shape C involves the partitioning of the set of methods of Cs interface shape invoked from the outside of C. In addition to the two previous metrics (Cohesion and sameOOInterface), we consider that when some methods are invoked together many times (i.e. correlation of method use) they must belong to the same provided interface. To illustrate this, we show an example of this situation in Fig.5 where m1, m2 and m3 are three methods of a shape interface and C1 and C2 are identified component shapes.

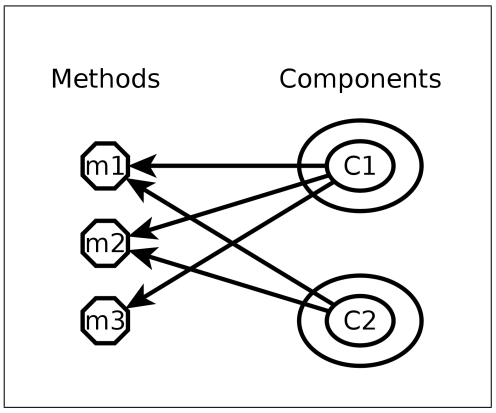


Figure 5. Methods invocations from the same component shape

The following table summarizes the number of times where each group formed from all possible combinations of the methods m1, m2 and m3 is invoked together from the same components.

Among all groups, {m1, m2} is the best because all of its methods are invoked together many times. Thus, we propose a function called $correlationOfUse(S : Setofmethods)$. This function returns the size of the greatest subset of S that contains methods invoked together as many times as

Combinations	Invocations together from the same components
{m1,m2,m3}	One time by C1
{m1,m2}	One time by C1 and one time by C2
{m1,m3}	One time by C1
{m2,m3}	One time by C1

Table I
NUMBER OF INVOCATIONS OF EACH GROUP OF METHODS TOGETHER

possible divided by the size of S.

In the same manner, for identifying required interfaces, we propose function called $correlationOfInvocation(S : Setofmethods)$. This function returns the size of the greatest subset of S that contains methods that invoke together the same component as many times as possible, divided by the size of S.

B. Definition of the fitness function and the clustering algorithm

Using the above metrics, we define two fitness functions to measure respectively the quality of provided and required interfaces. These fitness functions allows the architect to vary the weight of each sub-function using the parameter λ_i:

- $provided(M) = \frac{1}{\sum_i \lambda_i} \cdot (\lambda_1 \cdot LCC(M) + \lambda_2 \cdot sameOOInterface(M) + \lambda_3 \cdot correlationOfUse(M))$
- $required(M) = \frac{1}{\sum_i \lambda_i} \cdot (\lambda_1 \cdot LCC(M) + \lambda_2 \cdot sameOOInterface(M) + \lambda_3 \cdot CorrelationOfInvokation(M))$

Based on the definition of the previous fitness functions, the interfaces identification problem can be formulated as a clustering problem where the input is a set of methods $M = \{m_1, m_2, ..., m_n\}$ and the output is a partition of M noted $P(M) = \{p_1, p_2, ..., p_k\}$ where:

- m_i is a method belonging to a class in the shape interface.
- p_i is a subset of M.
- k is the number of identified interfaces.
- $P(M)$ does not contain empty elements: $\forall p_i, p_i \neq \phi$.
- The Union of all $P(M)$ elements is equal to M: $\bigcup p_i = M$
- The elements of $P(M)$ are pair wise disjoint: $p_i \cap p_j = \phi, if\ i \neq j$

To identify interfaces from a set of methods, we propose an algorithm based on hierarchical clustering (Algorithm 4. This algorithm proceeds in an agglomerative manner. This means that at the beginning, each cluster contains only one method, and pairs of clusters are merged at every step. The strength of relationship between methods is obtained using the fitness functions given previously for each type of interface. The result of the algorithm is presented in a dendrogram. The produced dendrogram is then traversed to produce clusters. Each cluster is a set of methods representing an interface shape. The value of the parameter t in algorithm 4 allows managing the granularity of the identified interfaces. As t increases, the resulting interfaces will be small grained.

Algorithm 4 dendrogramTraversal(Dendrogram d, double t):Partition R

if $isNotNull(d)$ **then**
  $Node\ n = nodeAsSet(d)$;
  if $f(n) > (t \times (f(son1AsSet(d)) + f(son2AsSet(d))))$ **then**
    return n;
  else
    return $dendrogramTraversal(son1(d)) \cup dendrogramTraversal(son2(d))$;
  end if
else
  return ϕ;
end if

C. Documenting identified components and interfaces

A component can be efficiently reused if its documentation (e.g. main purpose, name, interface names, etc.) is available. Thus, the need to document the identified components is important.

To reach this goal, we use an automatic heuristic to discover the purpose of an identified component. We based ourselves on the following observation: in many object oriented languages, class names are a sequence of nouns concatenated using a camel-case notation (i.e. StringBuffer, ElementFilter, etc). The first word of a class name indicates the main purpose of the class; the other words indicate a complementary purpose of the class and so on. On the other hand, an interface name should be an adjective that qualifies its implementing class. Thus, an interface name reflects one of the complementary functionalities of a given class. For example, in the Java standard API, The ArrayList class implements The Collection interface. According to the previous assertions, our heuristic identifies component name in three steps: extracting and tokenizing classes and interfaces name from identified components, weighting words and constructing the component name.

Extracting and tokenizing classes and interfaces' name from identified components: In this step, class names are split into words according to the camel-case syntax. For example: StringBuffer is split into String and Buffer.

Weighting words: In this step, a weight is affected to each extracted token. A large weight is given to tokens that are the first word of a class name. A medium weight is given to tokens that are the first word of an interface name. Finally a small weight is given to the other tokens.

In the mapping model proposed above, a component is composed of two elements: the center and the interface. Classes belonging to the former are ones that have no

relationship with the outside. In contrast, classes belonging to the latter are ones that interact with the outside. Its clear that the classes belonging to the provided interface of a component shape constitute the provided functionalities and services that it offers to other components, thus, its main purpose. On the other hand, classes belonging to the center of a component shape are less concerned with the main purpose of the component and are mainly utility classes that do not interact with the outside. Hence, a small weight will be given to tokens extracted from classes that belong to the center of a component shape and a large weight will be given to ones extracted from classes that belong to the provided interface of a component shape. For a given word, the weight is calculated as follows:

$$weight(w) = \frac{1}{\sum_i N_i} \cdot (1.0 \times (N_1 + N_2) + 0.75 \times N_3 + 0.5 \times (N_4 + N_5))$$

Where:

- N1: Number of appearance as the first word of a class name.
- N2: Number of appearance in an entity name belonging to the provided interface of a component shape.
- N3: Number of appearance as the first word of an interface name.
- N4: Number of appearance other than the first word in an entity name.
- N5: Number of appearance in an entity name belonging to the center of a component shape.

Constructing the component name: In this step, a component name is constructed using the strongest weighted tokens. The strongest weighted token is the first word of the component name; the second strongest weighted word is the second word of the component name and so on. The number of words used in a component name is chosen by the user. When many tokens have the same weight, all the possible combinations are presented to the user and he can choose the appropriate one.

We use a similar heuristic to discover interface name. Except that we use in addition the name of the methods belonging to the interface. For a given word, the weight used to construct interface name is calculated as follows:

$$weight(w) = \frac{1}{\sum_i N_i} \cdot (1.0 \times (N_1 + N_2) + 0.75 \times N_3 + 0.5 \times N_4)$$

Where:

- N1: Number of appearance as the first word of an interface name.
- N2: Number of appearance as the first word of a method name.
- N3: Number of appearance as the first word of a class name.
- N4: Number of appearance other than the first word in an entity name.

V. Case Studies

To validate our approach, we have experiment it on many open-source systems. We choose open-source systems because their documentations and source codes are publicly available. Due to space limitations, we will give the results obtained on only three systems of different sizes.

The execution time of our approach depends linearly on the size of the system because of the non-stochastic nature of the algorithms involved in each step.

In the following we will give the results obtained by applying explorative identification, requirement-driven identification and interfaces identification on different systems, followed by a discussion of the obtained results by varying the weights of each part of the fitness function.

A. Explorative Approach

We have applied our explorative component identification approach on many systems of different size. In this section we will give the results obtained on two systems: JDOM[1] , a well-known library containing 139 classes that provide a complete, Java-based solution for accessing, manipulating, and outputting XML data from Java code. Apache HTTP Components[2] , a toolset containing 368 classes focused on HTTP and associated protocols. The following table summarizes the application of our approach on the above-mentioned systems. For each system we give the number of identified components, the name and the purpose of the most prominent components, the number of classes belonging to them and the value of each sub-characteristic function.

Table II
Identified components in JDOM and Apache HTTP Components

Systems	Number of Components	Name of some components	Number of classes	Spe.	Com.	Aut.
JDOM	9	ObjectJDOM	19	0.42	0.53	0.68
		Exception	35	0.41	0.51	0.68
		SAX	21	0.46	0.55	0.61
		DOMAdapter	29	0.48	0.56	0.58
HTTP	19	HttpBasic	105	0.39	0.50	0.72
		HttpClient	59	0.49	0.50	0.69
		HandlerCookie	29	0.54	0.60	0.51
		ConnFactory	24	0.43	0.53	0.66

The purpose of the identified components can be determined according to their name and the classes that they contain. For example, DocumentJDOM contains classes responsible for creating and manipulating JDOM Documents. HttpBasic and BasicHttp contain the core HTTP classes needed by all the other components.

[1] http://www.jdom.org
[2] http://hc.apache.org

B. Requirement-driven approach

We have applied requirement-driven identification on JEval[3], a library containing 77 classes for the lexical and syntactic analysis of mathematical expressions. We have chosen *net.sourceforge.jeval.function.math.Sin* as the key entity and a maximum of 5 classes as a constraint. We have obtained the component containing the following classes and having the following values for each sub- characteristic function:

net.sourceforge.jeval.function.math.Sin net.sourceforge.jeval.function.math.ToRadians net.sourceforge.jeval.function.FunctionConstants net.sourceforge.jeval.function.math.Cos net.sourceforge.jeval.function.math.Asin	**Spe.**=0.83 **Comp.**=0.63 **Auto**.=0.43

Indeed, this component contains all the classes that have strong relationship with the key entity. According to their name, these classes abstract the evaluation of mathematical expressions composed of triangular functions.

C. Identification of Interfaces

We've applied required interface identification on the HttpClient component identified above. The result was composed of three interfaces. The following table gives the name, the number of methods and the value of the different metrics used to identify interfaces:

Table III
IDENTIFIED INTERFACES IN THE HTTPCLIENT COMPONENT

Name	Number of methods	Cohesion	Same interface	Invocation Together
Thread	8	0.50	0.25	1.00
AuthState	8	0.57	0.62	0.87
Connection	11	0.50	0.18	1.00

The Thread interface contains methods related to creating and handling threads. The AuthState interface contains methods used by the ClientHttp component to handle authentification. The Connection interface contains methods related to the establishment and the handling of HTTP connections.

D. Varying sub-characteristics weights

One advantage of our approach is the possibility to vary the weight of each sub-characteristics of the fitness function. This allows the architect to choose the importance of each characteristic of the identified components. We have applied our approach Apache HTTP Component by varying the different sub- characteristic weights of the fitness function. We have obtained the results given in Table IV.

In the first line of the table, we have given more importance to specificity and composability. The obtained components are smaller and contain few classes and thus few functionalities. This is in accordance with the definition of the specificity of a component (cf. section II.B). In the second line of the table, we have given less importance to composability. The obtained components contain many cohesive classes and thus many provided interfaces. Thus, they are less composable due to the diversity of their provided interfaces. In the third line of the table, we have given more importance to composability and autonomy. The result is composed of components having few required interfaces. This is in accordance with the definition of component autonomy (cf. section II.B).

[3]http://jeval.sourceforge.net

Table IV
RESULTS WHEN VARYING SUB-CHARACTERISTICS WEIGHTS

Spe.	Com.	Aut.	Number of components	Average number of classes	Size of the largest component
0.40	0.40	0.20	64	5.75	41
0.40	0.20	0.40	11	33	95
0.20	0.40	0.40	8	46	84

VI. RELATED WORKS

Many approaches have been proposed for the automatic identification of software components from legacy systems. The works presented in [4] [5] take as input UML diagrams and uses a clustering algorithm to identify components among the system classes. The produced clusters have high cohesion and low external coupling. The above-mentioned works relies extensively on UML diagrams while they are not always available and they may not reflect the real structure of the system. Our approach is based on the analysis of source code which remains the only software artifact that reflects the reality of the system.

Approaches presented in [3] [7] make use of only coupling and cohesion metrics to identify components. However, the use of only these two metrics can lead to poor quality components. In our approach, we have studied the semantic-correctness of components to guarantee that the identified components will be of high quality. In addition, unlike the previously cited works, our approach has the advantage to allow the architect to give more importance to some sub-characteristics.

The work presented in [3] is based on dynamic analysis of the system at runtime. It identifies software components from the execution trace of a use case. Its main drawback is the a priori knowledge of the high-level system functionalities. Also, it requires an extensive execution of many execution scenarios to involve all the classes that constitutes the system.

In [5], the authors propose to identify component external structure as object oriented interfaces. We show that component interfaces are slightly different from object ones insofar they must be a set of highly correlated methods.

VII. Conclusion

In our previous works [8] [9], we have proposed an approach to extract software architecture from an object oriented system.

In this paper, we rely on these previous works to propose a quality-centric component identification approach. Our approach gives to the architect the choice between two strategies to identify components. The first strategy is explorative. The second strategy is requirement- driven.

In addition to the identification of the internal structure of a component (i.e. the classes that constitutes it), we have proposed a method for the identification of the required and provided interfaces of a component. Also, we propose to document the identified components by automatically generating their name using some heuristics based on well accepted code conventions.

As short-term perspective, we plan to extend our approach to apply it on multiple versions/variants of the same system to obtain highly reusable components. The obtained results will guide our long-term perspective which consists in recovering software product lines.

References

[1] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[2] D. Birkmeier and S. Overhage, "On component identification approaches — classification, state of the art, and comparison," in *Proceedings of the 12th International Symposium on Component-Based Software Engineering*, ser. CBSE '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 1–18.

[3] S. Allier, H. A. Sahraoui, S. Sadou, and S. Vaucher, "Restructuring object-oriented applications into component-oriented applications by using consistency with execution traces," in *Proceedings of the 13th international conference on Component-Based Software Engineering*, ser. CBSE'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 216–231.

[4] S. D. Kim and S. H. Chang, "A systematic method to identify software components," in *Proceedings of the 11th Asia-Pacific Software Engineering Conference*, ser. APSEC '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 538–545. [Online]. Available: http://dx.doi.org/10.1109/APSEC.2004.11

[5] S. K. Mishra, D. S. Kushwaha, and A. K. Misra, "Creating reusable software component from object-oriented legacy system through reverse engineering," *Journal of Object Technology*, vol. 8, no. 5, pp. 133–152, 2009.

[6] C. Riva, "Reverse architecting: An industrial experience report," in *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*, ser. WCRE '00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 42–. [Online]. Available: http://dl.acm.org/citation.cfm?id=832307.837111

[7] H. Jain, N. Chalimeda, N. Ivaturi, and B. Reddy, "Business component identification - a formal approach," in *Proceedings of the 5th IEEE International Conference on Enterprise Distributed Object Computing*, ser. EDOC '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 183–. [Online]. Available: http://dl.acm.org/citation.cfm?id=645344.650165

[8] S. Chardigny and A. Seriai, "Software architecture recovery process based on object-oriented source code and documentation," in *Proceedings of the 4th European conference on Software architecture*, ser. ECSA'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 409–416. [Online]. Available: http://dl.acm.org/citation.cfm?id=1887899.1887937

[9] S. Chardigny, A. Seriai, M. Oussalah, and D. Tamzalit, "Extraction of component-based architecture from object-oriented systems," in *Proceedings of the Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008)*, ser. WICSA '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 285–288. [Online]. Available: http://dx.doi.org/10.1109/WICSA.2008.44

[10] I. Crnkovic, S. Sentilles, A. Vulgarakis, and M. R. V. Chaudron, "A classification framework for software component models," *IEEE Trans. Software Eng.*, vol. 37, no. 5, pp. 593–615, 2011.

[11] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[12] G. T. Heineman and W. T. Councill, Eds., *Component-based software engineering: putting the pieces together*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[13] C. Luer and A. V. D. Hoek, "Composition environments for deployable software components," Tech. Rep., 2002.

[14] ISO, "Software engineering – Product quality – Part 1: Quality model," International Organization for Standardization, Tech. Rep. ISO/IEC 9126-1, 2001.

[15] J. M. Bieman and B.-K. Kang, "Cohesion and reuse in an object-oriented system," in *Proceedings of the 1995 Symposium on Software reusability*, ser. SSR '95. New York, NY, USA: ACM, 1995, pp. 259–262. [Online]. Available: http://doi.acm.org/10.1145/211782.211856

[16] S. Johnson, "Hierarchical clustering schemes," *Psychometrika*, vol. 32, pp. 241–254, 1967, 10.1007/BF02289588. [Online]. Available: http://dx.doi.org/10.1007/BF02289588