A Systematic Method to Identify Software Components

Soo Dong Kim and Soo Ho Chang
Department of Computer Science
Soongsil University, Seoul, Korea
sdkim@comp.ssu.ac.kr shchang@otlab.ssu.ac.kr

Abstract

In component-based development (CBD), component is the basic unit for reuse and it provides a relatively coarse-grained functionality. A component typically consists of several related objects, where they collaborate in order to carry out system operations. Hence it is essential in CBD to identify components with high cohesion and low coupling. In this article, we propose a systematic UML-based method to identify components. Our approach makes use of clustering algorithms, metrics, decision rules and a set of heuristics. We assume that an object-oriented model for a target domain is available. The method takes these artifacts and transforms them into components in a seamless way.

1. Motivation

In CBD, component is the basic unit for reuse and it provides a relatively coarse-grained functionality. A component typically consists of related objects which collaborate in order to carry out system operations [1][2][3]. Hence it is an essential activity in CBD to group related objects into components in such way that the identified components have high cohesion and low coupling with others.

Much of the research on identifying components emphasizes the principle of functional independence but does not provide systematic and metric-based formal approaches. This simplifies the process of component identification, but it is a significant departure from defining optimal components.

In this article, we propose a UML-based technique to identify components. Our approach makes use of clustering algorithms, metrics, decision rules and a set of heuristics. We assume that an object-oriented model for a target domain is available, including use case model, object model and dynamic model. The method takes these artifacts and transforms them into components in a seamless way.

It begins, in section 2, with a survey of representative component identification methods. Section 3 discusses the criteria used to define our proposed process and the criteria that identified components should possess. Section 4 defines the overall process and elaborates the steps in details. Section 5 shows how the proposed process and resulting components meet the criteria specified.

2. Related Work

2.1. Lee, et al [4]

This method measures the inter-class relationships in terms of *create, retrieve, update* and *delete* (CRUD), and uses a clustering algorithm of shifting rows and columns which is also found in information engineering. In addition, it uses dynamic coupling metric between objects to measure the potential number of messages exchanged. It suggests basic techniques to resolve conflicts in clustering. However, the method does not consider functional dependency among use cases, but it only considers data dependency among classes.

2.2. Sugumaran, et al [5]

This work proposes a process to identify components from process requirements using domain model and object libraries. Main factors used in component identification are conceptual objects, processes and behavior navigation. The process includes activities and high-level guidelines to identify components from abstraction to implementation level, but does not include specific rules, metrics, or algorithms for component clustering.

2.3. Cheesman and Daniels [6]

This work has a wide scope of CBD in general, and it includes a method to identify components using use cases and business type models. Inter-class relationship is used as the main factor for identifying components.

Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04)
1530-1362/04 $ 20.00 IEEE

The core class serves as the center of each clustering, and responsibility derived from use cases is used to guide the process. This work provides conceptual and high level guidelines, and it relies largely on domain experts in applying the guidelines.

2.4. Jain, et al [7]

This method uses static and dynamic relationship between classes to group related classes into components. Static relationship measures the relationship strength using different weights, and dynamic relationship measures the frequency of message exchanges at runtime. Then, it combines two relationships to compute the resulting strength of relationship between classes. However, the guidelines for giving the weight values and counting the number of message passing are not referred. Hence, the method sets a conceptual framework of component clustering.

3. Design Criteria

3.1. Our Approach

In contrast to the related works in section 2, we focus on providing a process and detailed steps. Metrics are used whenever applicable, although not all the engineering activities can be carried out with metrics. If metrics are not applicable, we try to provide more specific rules or guidelines rather than simply giving general principles.

3.2. Criteria for Process

In defining the process, we focus on some essential criteria that the process should adhere to. The criteria are defined by considering the fundamental concepts of software engineering and properties of well-defined components.

The first criterion is that the process must include a systematic sequence of steps, in that the ordering of steps should be logical and easy to apply in practice. Hence, a later step can comfortably rely on the artifacts of previous steps.

The second criterion is that the process should specify work instructions in details and so that they can be effectively applied in practice. Majority of work we surveyed provide high-level and general guidelines, lacking the practical applicability and engineering preciseness.

The third guideline is the traceability among artifacts throughout the whole process. An artifact should be derived seamlessly from artifacts produced from previous steps, and every element of an artifact has a trace to element(s) of other artifacts. It is also desirable to define the templates or formats for various artifacts so that engineers can apply the process effectively.

3.3. Criteria for Identified Components

To develop our process, we define desirable properties that identified components should possess. Components are more inter-organizational reuse in a domain rather than in-house reuse [1] . Hence, they should provide a common functionality in a target domain and consider variability in the common functionality as well. Since this research assumes the availability of object-oriented modeling artifacts, the commonality and variability analysis is assumed to be carried out and specified during object-oriented modeling by using an approach like [8].

A component should be relatively large-grained so that it provides a cohesive functionality to the clients without heavily referring to other components. This is the principle of functional independency in software design. Hence, we argue that a component should have a high cohesion and a low coupling with others.

The specification of identified components should include functional elements as well as data elements, to be effectively implemented. The functional element specifies the behavior of components, and the data element of components specifies the objects residing in components. Furthermore, functional elements are used to derive the interface of components and data elements are used to specify the classes to be implemented.

4. Process and Instructions

The process proposed for component clustering consists of four steps as in figure 1. We assume that the fundamental artifacts of object-oriented modeling such as use case model, object model and dynamic model expressed in sequence diagram or collaboration diagram are available.

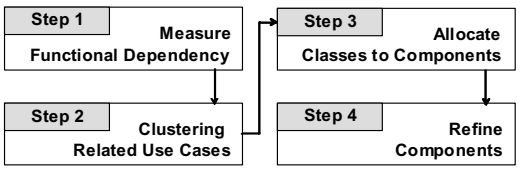

Figure 1. The Overall Process

In our approach, we consider three types of relationships in identifying components. In steps 1 and 2, functional dependency between use cases is used as the fundamental means to cluster related functions. And, the dependencies are measured and related use

cases are clustered. In step 3, functionality-to-data relationships expressed in a dynamic model such as sequence diagram are taken to assign related classes to candidate components. In step 4, dependency or coupling between classes is used to verify and refine the identified components. If there are two closely related classes which are separated into two components, it is identified and refined in this step..

4.1. Step 1. Measuring Functional Dependency

A software module or any unit for reuse should provide a cohesive functionality, i.e., it must be functionally independent [5, 6, 7]. Since a component is larger-grained than an object in terms of services provided, its functionality typically consists of several related use cases. For example, *Account Management* component has a functionality which can be characterized by multiple use cases such as *deposit, withdraw, getBalance, transfer, etc.* Often, The collection of these services exceeds the service provided by a single class.

Based on this observation, the first step for identifying components is to assess the functional dependencies between use cases and to cluster closely related use cases into a component. The degree of functional dependency between two uses cases may not be mechanically measured due to the informal nature of functional description. However, we propose the following metrics, guidelines and criteria to more effectively measure the functional dependency between two use cases, UC_i and UC_j.

- Criterion concerning Sub-Systems
  Most large-scaled systems consist of several sub-systems, and use cases belonging to the same sub-system are more closely related than the other case. If both UC_i and UC_j belong to the same sub-system, the measure for this criterion M_1 gets the value of 1. Otherwise M_{11} gets 0. If the target system is small enough that sub-systems are not defined, then M_{11} also gets 1.

- Criterion concerning Actors
  Use cases initiated or invoked by the same actor are more closely related than the other case. This is based on the observation that an actor usually plays a homogeneous role in the system and so the use cases used by the actor have some degree of close relationship. In measuring this relationship, we should consider that a use case may be initiated by more than one actor. The measure for this criterion, M_2, is defined as the following;

$$M_{12} = \frac{(\text{actors initiating } U_i \cap \text{actors initiating } U_j)}{(\text{actors initiating } U_i \cup \text{actors initiating } U_j)}$$

  In the metric, M_{12} measures the proportion of actors which initiate both U_i and U_j, against to the set of all related actors. A higher value of M_{12} indicates a higher degree of sharing use cases by same actor(s). The range of M_{12} is 0..1. If M_{12} is 1, U_i and U_j have same actor(s), but if M_{12} is 0, then there is no actor initiating both use cases.

- Criterion concerning Shared Data
  Use cases manipulating the same set of data are more closely related than other cases. This is based on the observation that there exist in general operations specific and intrinsic to certain data and so the operations manipulating the same data have some degree of relationship. By examining the functionality of use cases, one can derive the data or objects manipulated by each use case, i.e. classes related to each use are identified. The degree of commonality on data manipulated by two use cases M_{13} can be more accurately expressed using a metric rather than simply representing it as a boolean value or a scale. We suggest the following metric;

$$M_{13} = \frac{(\text{classes manipulated by } U_i \cap \text{classes manipulated by } U_j)}{(\text{classes manipulated by } U_i \cup \text{classes manipulated by } U_j)}$$

  The range, minimum value and maximum value of this metric are same as those for M_{12}.

- Criterion concerning Use Case Relationship
  A measure, M_{14}, represents the degree of coupling between two use cases. Three kinds of relationships between use cases exist; «include», «extend» and generalization. If UC_i and UC_j are specialized from a generalized use case, then M_{14} is 1. If UC_i and UC_j are related with «extend», then M_{14} gets 1. If UC_i includes UC_j with «include», then the relationships between UC_j and use cases other than UC_i should also be considered. For example, UC_j may also be included by UC_k. In this case, if UC_i has a relatively strong relationship with UC_k, then M_{14} gets a value of 1. If UC_i is independent from UC_k, UC_j should be placed in a separate utility component.

In addition to the four criteria which we believe are the most relevant to use case clustering, additional criteria can be defined producing M_{15}, M_{16}, and so on. Different criteria have a different significance and impact on clustering.

Moreover we use a weighting system. Each criterion may be assigned with a different weight value, and the

sum of all weights must be 1. For instance, one may assign 0.2, 0.3, 0.25, and 0.25 for the above four criteria respectively. As a tip, a weight of subsystem depends on the expected number of use cases in the subsystem. As the weight, a subsystem has one component or several components.

Based on the given criteria, we define a metric to compute the functional dependency between two use cases UC_i and UC_j, $FD_{i,j}$, as follows; $FD_{ij} = \sum (M_k * W_k)$ where M_k is the measure on the k^{th} criterion for the two use cases, and W_k is the weigh value for the k^{th} criterion. Then the range of FD_{ij} is between 0 and 1. We use a functional dependency worksheet to record measures and to compute the metrics as in Table 1.

Table 1. Functional Dependency Worksheet

Measure Criteria		UC_i		UC_{i+1}		…
		M_k	W_k*M_k	M_k	W_k*M_k	
UC_i	Sub-system					
	Actor					
	Data Shared					
	Relationship					
	FD		∑		∑	
…						

4.2. Step 2. Clustering Use Cases

Once the functional dependencies are computed for every pair of use cases in step 1, we now define candidate components by clustering related use cases. To do this, we use a variation of the clustering algorithm which is commonly used in information engineering[9]. From the worksheet in table 1, we create a summary table of functional dependency for a case study conducted in table 2.

Table 2. Summary Matrix of Functional Dependency

	UC1	UC2	UC3	UC4	UC5	UC6	UC7	UC8
UC1	1	0.4	0.3	0.2	0.9	0.8	0.2	0.3
UC2		1	0.4	0.5	0.4	0.2	0.8	0.4
UC3			1	0.8	0.1	0.3	0.5	0.9
UC4				1	0.4	0.3	0.3	0.7
UC5					1	0.7	0.4	0.3
UC6						1	0.1	0.1
UC7							1	0.3
UC8								1

The clustering algorithm uses a row and column shifting method and it requires a constant value *t* to identify rows and columns to be shifted. For given value of t, the algorithm performs the best possible clustering. As t increases, smaller grained components are acquired. However, the value of t is chosen in computed values of functional dependency by domain expert. Different values of t can be applied to this step, and different clustering results can be examined by domain experts or developers to select the most optimal result. This is further illustrated in step 4.

As an example, let us take 0.7 for the value of t. For the use case UC_1, use cases in the same row of which FD_{ij} value is greater than the value of t are shifted column-wise and row-wise toward UC_1, i.e. UC_5 and UC_6 are shifted toward UC_1 as shown in table 3.

Table 3. Matrix after Shifting for UC_1

	UC1	UC5	UC6	UC2	UC3	UC4	UC7	UC8
UC1	1	0.9	0.8	0.4	0.3	0.2	0.2	0.3
UC5		1	0.7	0.4	0.1	0.4	0.4	0.3
UC6			1	0.2	0.3	0.3	0.1	0.1
UC2				1	0.4	0.5	0.8	0.4
UC3					1	0.8	0.5	0.9
UC4						1	0.3	0.7
UC7							1	0.3
UC8								1

After repeating necessary shifting for all use cases, we finally have the matrix as shown in table 4.

Table 4. The Final Matrix after All Shifting

	UC1	UC5	UC6	UC2	UC7	UC3	UC4	UC8
UC1	1	0.9	0.8	0.4	0.2	0.3	0.2	0.3
UC5		1	0.7	0.4	0.4	0.1	0.4	0.3
UC6			1	0.2	0.1	0.3	0.3	0.1
UC2				1	0.8	0.4	0.5	0.4
UC7					1	0.5	0.3	0.3
UC3						1	0.8	0.9
UC4							1	0.7
UC8								1

Now the remaining task is to identify the set of adjacent use cases which have FD_{ij} values that are greater than t_1. And, each set of clustered use cases makes up a candidate component. In this way, we can derive three components from the matrix after all shifting as shown in table 4. The first component includes use cases 1, 5 and 6, the second component includes use cases 2 and 7, and the third component includes uses cases 3, 4 and 8.

A potential problem during clustering occurs when we may not have a clear-cut clustering. A use case can be shifted several times if the use case has more than one functional dependency FD_{ij} which is grater than t. This may yield a clustering result with use cases shared by sets, and as the result, clear-cut clustering is not possible. The situation is illustrated in Figure 2.

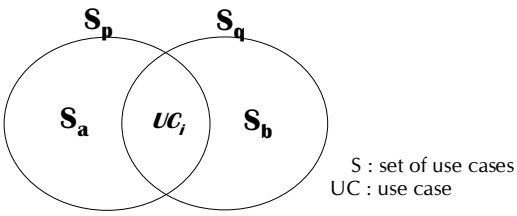

Figure 2. Relationship between Sets of Use cases

In the figure, S_p and S_q are clustered sets of use cases from the table 4. S_a and S_b are sets of use cases included exclusively into S_p and S_q respectively. A use case UC_i is the intersection of S_a and S_b. That is, S_p is union of S_a and $\{UC_i\}$, and S_q is also a union of S_b and $\{UC_i\}$.

We may focus on the relationship among S_a, S_b and UC_i to derive a difference as follows; We define a *degree of coupling* between a set and a use case as *Coupling*(S_a, UC_i); If UC_j is an element of S_a, *Coupling*(S_a, UC_i) is $\sum FD_{ij}$ for all i. We calculate a *difference of (Coupling*(S_a, UC_i) and *Coupling*(S_b, UC_i)*)*. If the difference is greater then 0, UC_i should be included in S_p. If the difference is less then 0, UC_i should be included in S_q. If the difference is equal to 0, then the use case may be allocated arbitrary or by domain experts.

4.3. Step 3. Allocate Classes to Components

This step is to allocate classes to each component by using the dynamic model expressed in sequence diagrams. As shown in Figure 3, a component typically includes several use cases, the dynamic behavior of each use case is depicted by a sequence diagram, and a sequence diagram specifies a set of participating objects/classes. The relationship among component, use case, class, and sequence diagram is as following Figure 3.

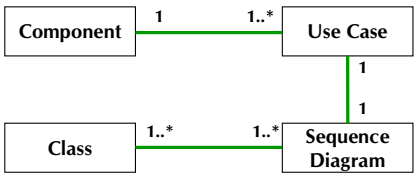

Figure 3. Relationship between Components and Classes

For each component identified in step 2, we locate sequence diagrams for use cases included in the component. Then, the participating classes shown in a sequence diagram are assigned to the corresponding component. We use the class assignment matrix as in Table 3.

Table 3. Class Assignment Matrix

	Component A	Component B	Component C
Class 1		❍	
Class 2			❍
Class 3	❍		❍
Class 4		❍	
Class 5	❍		
Class 6	❍		
Class7		❍	
Class 8			❍
…			

Even if the measure of functional dependencies is carried out relatively accurate, the assignment of classes into components may yield a conflict where a class is assigned into more than a component.

For the conflicts, we first compute an entity dependency between a component and a class, and then compare the entity dependency of one conflicted class for one component to that of another component. In order to compute the entity dependency, we suggest criteria as followings.

– **The number of message passing**: through tracing a sequence diagram, we can estimate the number of dynamic message passes between classes for each use case at runtime[7]. We represent the criterion, M_{21}, as following;

$$M_{21} = \sum \text{the number of message passing to } C_i, \text{ for all use cases in } Com_j$$

Weight for role of an entity class: An entity class is used to manipulate data such as Create, Retrieve,

Update, and Delete. These functions have weight for their importance like to *Shared Data*, a criterion on Section 4.1, so the case of M_{22} can be computed as following;

$M_{22} = \sum$ *weight of role of* C_i *in all use cases in* Com_j

- **Relationship between Classes**: There are five types of relationships between two classes; *inheritance, association, dependency, aggregation* and *composition* as shown in Figure 4.

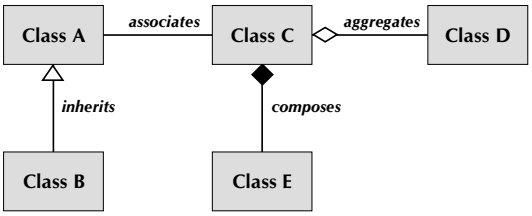

Figure 4. Different Relationships between Classes

In general, super classes and sub-classes are closely related through inheritance, especially by protected items, two classes with association are also highly related because their instances have links each other for navigation. Dependency is relatively lower than association since dependency specifies potential method invocation. Composition is a stronger form of aggregation since it shares its lifetime with part objects. Based on this observation, one has to give different weight for each type of relationship. Instead of defining constant values for the weights, we define the following comparison on different relationships in terms of relationship strength[10].

0 < Dependency < Aggregation < Association, Inheritance < Composition < 1

Using this relative weight values, we can compute the inter-class relationship between a pair of classes. Similar to the step 2, we can define a constant value, t, to determine where two classes should be put together in a component or not.

$M_{23} = \sum$ *weight of relationship between* C_i *and other classes in* Com_j

From the criteria, we make a formula to compute entity dependency $ED_{ij} = \sum (M_k * W_k)$ between a class C_i and a component Com_j.

4.4. Step 4. Selecting Optimal Component Identification

By applying steps 1 through 3, candidate components are identified. The value of t used in the process determines the number of components and their granularity. Hence, it is desirable to apply different value of t to generate different clustering results and to let developers to choose an optimal clustering result using certain criteria. The step 4 is to choose an optimal configuration from different clustering results generated by applying different t values.

- **Range of t**: The range of t is between the minimum value and maximum value of all measured functional dependencies in step 1. In the example of table 2, the range of t is from 0.1 to 0.9.

Effect of t: As shown in figure 5, a smaller value of t produces components with the larger granularity, but the total number of components is decreased. In contrast, a larger value of t produces finer grained components which are close to objects, but the total number of components is increased.

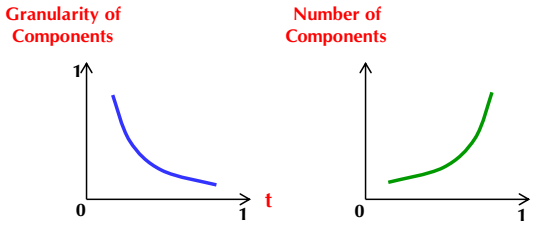

Figure 5. Effects of the value t_1

Selection of t: This relationship is depicted in figure 6 which combines two graphs of figure 5. It is possible to elicit an optimal value of t which produces components that satisfy selection criteria such as economical value, granularity of components and number of components. By applying well-defined criteria, an optimal value of t can be determined resulting in components with adequate granularity. As noted as a shaded rectangle in figure 6, the optimal range of t will be near the intersection of two curves. It is impractical to define a single set of selection guidelines applicable to all projects; rather it can be set by considering characteristics of each project.

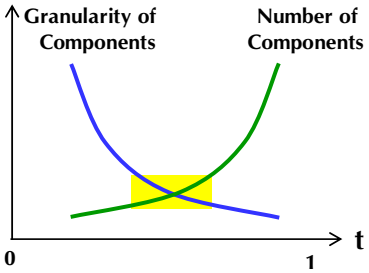

Figure 6. Optimal Range of t Value.

5. Assessment

This proposed method is expected to yield better component clustering result than the result by simply relying on experience and heuristics. However, it is still desirable that domain experts and modeling engineers examine the clustering result and refine the clustering as needed.

The proposed process includes four steps and their ordering follows the relevant impacts of criteria used in clustering components. That is, the steps 1 and 2 consider the functional relationship between use cases to derive the initial set of candidate components. This conforms to the second criterion for identified components as illustrated in section 3.3. The step 3 of the process is to include classes with each component, and it uses the message invocation information in sequence diagrams. This shows that the artifact of step 3 can be traced to other artifacts. Including this step, all the steps and artifacts are specified with traceability in mind. A component specification should include both functional and data elements as in section 3.3, and it is satisfied by steps 3 and 4.

In table 6, we compared our approach to the four related works described in section 2, using the criteria stated in section 3.

Table 6. Comparison to Other Approaches

	Lee, et al [3]	Sugumaran, et al [4]	Cheesman, et al [5]	Jain, et al [6]	Our Process
Well defined sequence of steps or activities	●	●	❍		●
Detailed level of instructions including Metrics	●		❍	❍	●
Traceability of Artifacts	❍				❍
High Cohesion and Low Coupling	❍		❍	❍	●
Specify both functional and data elements	●	❍	❍		●
Considering functional dependency	❍	●	●		●
Considering functionality-to-data relationships	❍		●	●	●
Considering object-to-object relationship	●	❍	●	●	●

● – Supports, ❍ - Partially Supports

As shown in the table, our proposed approach satisfies majority of criteria specified. Especially, the approach considers all three types of relationships; use case to use case, use case to class, and class to class relationships.

6. Concluding Remarks

In component-based development (CBD), component is the basic unit for reuse and it provides a relatively coarse-grained functionality. A component typically consists of several related objects, where they collaborate in order to carry out system operations. Hence it is an essential development activity in CBD to identify and design components with high cohesion and low coupling.

In this article, we proposed a UML-based method to identify and design components. Our approach makes use of clustering algorithms, metrics, decision rules and a set of heuristics. We assume that the object-oriented model for the target domain is available including use case model, object model and dynamic model. The method takes these artifacts and transforms them into components in a seamless way. Especially our approach utilizes metrics wherever possible, and provides instructions as specific as possible. Using the propose method, we believe that high quality components can be identified and design.

References

[1] Heineman, G. T., Councill, W. T., *Component-Based Software Engineering*, Addison-Wesley, 2001

[2] Szyperski, C., Component Software Beyond Object-Oriented Programming, Addison-Wesley, 2002

[3] Crnkovic, I. and Larsson, M, *Building Reliable Component-Based Software Systems*, Artech House, 2002.

[4] Lee, S., Yang, Y., Cho, E., Kim, S., and Rhew, S., "COMO: A UML-Based Component Development Methodology", Proceedings of the IEEE Sixth Asia Pacific Software Engineering Conference, Dec. 1999.

[5] Sugumaran, V., Tanniru, M., Storey, and Veda. C., "Identifying Software components from process requirements using domain model and object libraries", Proceeding of the 20th ACM international conference on information system, 1999.

[6] Cheesman, J. and Daniels, J., *UML Components*, Addison-Wesley, 2000.

[7] Jain, H. and Chalimeda, N., "Business Component Identification – A Formal Approach", roceedings of the IEEE Fifth International Enterprise Distributed Object Computing Conference, 2001.

[8] Atkinson, C., Component-based Product Line engineering with UML, Addison-Wesley, 2002.

[9] James,M.,*Information Engineering*, Prentice Hall,1989.

[10] Cho, Eun Sook, Kim, Soo Dong, and, Rhew, Sung Yul, "Domain Analysis and Preliminary Design Technique for Component Development", *International Journal of Software Engineering and Knowledge Engineering*, Vol. 14, No. 2, p.221-254, World Scientific Publishing Co., May 2004.