An Approach to Requirements Encapsulation with Clustering

Zude Li, Quazi A. Rahman, Nazim H. Madhavji
Computer Science Department, University of Western Ontario
London, Ontario, Canada, N6A 5B7
{zli263,qrahman2,madhavji}@csd.uwo.ca

Abstract

Requirements encapsulation means organizing software requirements into a set of requirements clusters with tight cohesion along with external interfaces such that each cluster can be ultimately implemented by a functionality module. We propose an approach to encapsulating requirements which includes two steps: clustering requirements based on the similarity and associativity relations and then encapsulating each cluster by defining its external interface as stimulus-response pairs. The potential benefits of encapsulating requirements are reduced software development and maintenance costs.

1. Introduction

Requirements encapsulation has the potential to reduce software development and maintenance cost [8]. Organizing requirements into clusters and encapsulating the clusters with external interfaces can help improve transition from requirements engineering to design. In the literature, we find many approaches for clustering requirements [12, 7, 11, 1]; however few specific approaches have been proposed for encapsulating clusters. In this paper, we propose a requirements encapsulation method which involves an extensive clustering algorithm and a process to create external interfaces for those clusters.

In our approach, we organize requirements into clusters by decomposing each requirement into seven dimensions with attribute values. A dimension is an information category of requirements (e.g., Functionality, Quality, Subject, Object, etc.) within an application environment. An attribute within a dimension indicates the information granularity of requirements in the dimension. Attributes are organized hierarchically in each dimension according to their granularity on information, and partial order relation generalization (or specialization) may exist between two related attributes. The factors which compose the clustering requirements metric include three parts: (1) *priority rank of dimension* for indicating the different impacts of different dimensions to clustering, (2) *degree of similarity* which describes the relationship among subjects, objects and actions of different requirements, and (3) *degree of associativity* which indicates the cohesiveness of requirements on functionality, quality, time and location. The requirements clustering algorithm is to derive the set of requirements clusters with the clustering criterion based on the relations among requirements, dimensions, and attributes. Then for each requirements cluster, we define its external interface with stimulus-response pairs. The relationships between requirements in different clusters are maintained on the cluster level and those are encapsulated as the associations between two modules. The ultimate goal of requirements encapsulation is to make sure that each encapsulated cluster can be implemented by a functionality module.

Our requirements clustering criterion is more comprehensive than the previous methods [4, 12, 1, 8]. Also, the external interfaces of requirements clusters may contribute more to requirements-design-code reuse in practice.

The remainder of the paper is organized as follows: Section 2 discusses some related work. Section 3 proposes the notion requirements dimensions and attributes. Section 4 defines the similarity degree and associativity degree. Section 5 depicts the requirements encapsulation approach. Finally Section 6 gives a short conclusion.

2. Related Work

Two areas of research related to our work are clustering algorithms and the box-structured decomposition method. Below, we discuss the existing literature in these two areas.

The aim of requirement clustering is to organize related requirements as clusters for forming the subsystems that exhibit certain desired properties [11, 1]. It is widely used in system decomposition [4, 12], software modularization [1], requirement reuse [7, 2], requirements quality improvement [7, 3, 13], and software product line [5, 6]. Typical clustering criteria are based on the "distance" of requirements, such as *source similarity* (number of matching attributes)

[1], *external view* (usability decomposition) [4, 12], and *functionality*, *time*, and *space* [8].

The box-structured system decomposition method was first proposed by Mills [10, 9]. In this method, requirements are considered as the "*black box view*" of the system, represented as *stimulus-response* pairs. The system is implemented through stepwise stimuli identification and response specification. *Freedom* is a lightweight and customer-centric software development methodology originally developed for NASA's Space Station Freedom program [8]. *Freedom* method takes requirements as a part of the software, namely, its external interface, i.e., the black box view of the system with stimulus-response pairs (named as stimulus set) specified by customers and users. Stimulus sets are organized as *functionality tree*, which is a schematic diagram of the external interface of the software system. The main principle of *Freedom* development process is the what-is-called Requirement Encapsulation Design Rule (REDR) [8]: "*create one functionality module for each stimulus set of the functionality tree*."

It is reported in [8] that based on clustering techniques, requirements encapsulation can reduce the total cost of software over its entire life cycle by 16 to 30 percent depending on the extent to which requirements reuse is employed.

Requirements clustering techniques address the relationship between requirements. Requirements clusters contribute to requirements reuse, but they are not sufficient for design and code reuse, because they do not support either clusters' boundary definition or the relationship between clusters, which are indispensable for design and code modularization. In today's software development practice, the boundaries of clusters and the relationship between clusters are both indispensable for realizing REDR. We have attempted to address these two issues in our approach.

3. Dimensions and Attributes

A *dimension* of a requirement depicts an information category of the requirement's semantics within the application environment. For instance, `OBJECT` and `LOCATION` are two typical requirements dimensions. `OBJECT` addresses the targeted *resource* category of requirements in applications, and `LOCATION` specifies the location constraints implied in requirements. *Dimension attribute* indicates the similarity or associativity between requirements on a dimension, indicating the extent of cohesion. For instance, *Client-Data* is an attribute in dimension `OBJECT`.

Requirements may have more than one dimension, such as Subject, Object, etc. [1, 8]. In our experience, requirements can be depicted from two perspectives: *structure* and *semantics*. Each requirement is organized by some fixed components (elements), typically including *subject*, *object* and *action*, indicating *who can/may do or expect what behavior on what resource*. For example, in requirement "*User may input client data*", *User* is a subject, *Input* is an action, and *Client-Data* is an object. On the other hand, a requirement expresses the functional or nonfunctional anticipation of the emerging system, as well as it indicates some temporal and location constraints between requirements or components of a requirement. These are called semantics of requirements. As a requirement's semantics directly affects the subsequent design and implementation of this requirement, it should be considered during requirements analysis process, in particular the clustering process in this paper. We define four dimensions to capture the semantics for each requirement: *functionality*, *quality*, *time* and *location*. Overall, our dimension set (DS) contains seven dimensions that are defined below.

- `SUBJECT` (`S`). It depicts the actors category a requirement contains, i.e., *who*.
- `OBJECT` (`O`). It depicts the targeted object category a requirement contains, i.e., *to what resource*.
- `ACTION` (`A`). It depicts the behavior category a requirement contains, i.e., *do what*.
- `FUNCTIONALITY` (`F`). It depicts the functional features category a requirement addresses, i.e., *for what functional goals*.
- `QUALITY` (`Q`). It depicts the software quality capacity category a requirement addresses, i.e., *for what non-functional (quality) goals*.
- `TIME` (`T`). It depicts the temporal constraints category a requirement addresses, i.e., *when*.
- `LOCATION` (`L`). It depicts the (spatial) location environment category a requirement addresses, i.e., *where*.

In the above, `S`, `O`, and `A` are taken as the criteria factors for clustering scenarios in [1]. `F`, `T`, and `L` are used as criteria to distinguish the requirements modules (stimulus sets) in the *Freedom* method [8]. `Q` represents system-level constraints, which is expressed or implied in related requirements. We bring them together for the purpose of clustering. In our approach, the way to distinguish two requirements in an application is the (*structure*) *similarity degree* and (*semantics*) *associativity degree*, based on the degree of similarity of the constituent attributes. The structural relationship between requirements is addressed by the *Similarity relation*, indicating the requirements may have same or similar components such as subjects, objects and actions. *Similarity degree* is used in the three structure dimensions, `S`, `O`, and `A`. The set of these is named as *Similarity Dimension Set* (SDS). The semantic relationship between two or more requirements is captured by the *Associativity* relation, indicating the requirements may have similar functionalities or quality attributes, and associated temporal or location constraints. *Associativity degree* is used in the four semantics dimensions: `F`, `Q`, `T` and `L`. The set of these is named as *Associated Dimension Set* (ADS).

Let us take dimension `T` and `Q` as instances, we can define an attribute in dimension `T` as *Simultaneity*, requiring that some attributes should be handled at the same time. There exist many other attributes in `T`, such as *Sequence*, *Alternative*, *Exclusion*, etc. [5]. Dimension `Q` includes *Reliability*, *Performance*, *Portability*, etc., which can be further specialized with respect to the detail application. Different applications may emphasize on different quality features.

Example 1: consider requirements R1 and R2:
R1: *User may input client data (name, gender, birthdate)*;
R2: *Super-user can input user data (name, password)*.
In R1 and R2, *client data* and *user data* can be considered as two attributes in dimension `O`, written *Client-Data* and *User-Data*, respectively. They can be taken as two granular specializations of attribute *Data*. *Client-Data* can be further specialized into three more granular attributes: *Name*, *Gender*, and *Birthdate*. *User-Data* can be further specialized into *Name* and *Password*. On dimension `F`, R1 and R2 imply that *Input-Data* should be an attribute in `F` and *Input-Client-Data* and *Input-User-Data* can be defined as the two child attributes of *Input-Data*. On dimension `T` and `L`, it is implied that *Name*, *Gender*, and *Birthdate* are required as client data to be input at same time (i.e., *Simultaneity*) and in same location (e.g., on same GUI). Similar situation occurs with *Name* and *Password* as user data. Other dimensions can be similarly explored as above.

Generally, attributes are organized as a hierarchical structure in each dimension, called *attribute hierarchy* (AH), containing the *specialization* (or *generalization*) relations between requirements. We use $A_i \preceq_D A_j$ to indicate the specialization relation from A_i to A_j in a hierarchy AH_D (*D* is a dimension, $D \in DS$). The transitive and reflexive closure of the attribute specialization relation are written as $\preceq^*_D$. We call A_i as a *parent* (respectively, *senior*) attribute of A_j (or A_j as a *child* (respectively, *junior*) attribute of A_i) if $A_i \preceq_D$ (respectively, $\preceq^*_D$) A_j holds.

In AH, an attribute may have more than one child attribute but can have at most one parent attribute. Attributes with same name but within different dimensions or parents are different. We define the *attribute domain* (AD) of attribute A in *D*, written $AD_D(A)$, as the set of all senior attributes of A, formally, $AD_D(A) = \{A' \mid A' \preceq^*_D A\}$.

Example 2: AH_{O} (the hierarchy in dimension `O`) on R1 and R2 can be expressed as:

AH_{O}: *Data*(*Client-Data*(*Name, Gender, Birthdate*), *User-Data* (*Name, Password*))

So, we can derive: AD_{O}(*Data*) = $\{$*Data*$\}$, AD_{O}(*Gender*) = $\{$*Gender, Client-Data, Data*$\}$, etc.

Table 1. Experienced dimension prioritization

Dimension	Priority Rank
`FUNCTIONALITY(F)`	6
`ACTION(A)`	5
`OBJECT(O)`	4
`TIME(T)`, `LOCATION(L)`	3
`SUBJECT(S)`	2
`QUALITY(Q)`	1

4. Degree of Similarity and Associativity

Dimensions can be *prioritized* based on the importance for particular application. Prioritization is critical to the ultimate result of requirements clustering, since different dimensions affect the result of clustering to different extents. For instance, software quality is generally considered in a global perspective, which is measured in an integrated-system view. So the impact of dimension `Q` on requirements clustering should be much weaker than that of dimension `F` or others, as `F` implies, during the design and code phases, requirements with similar functionalities should be implemented in same functional modules.

In real requirements practice, each dimension is assigned with a numerical *priority rank* indicating the *significance level* of the dimension into which all requirements are categorized and then modularized, comparing to others. Here we try to give a rational descending rank order of the above mentioned dimensions in Table 1.

For a requirement r, we use Pr_D to return the priority rank on dimension *D*, and use $Att_D(R)$ to represent the attributes of requirement (set) R in dimension *D*. Each attribute in a dimension is assigned with a *weight* to indicate the similarity or associativity degree on the attribute. We use $Wt_D(A)$ to represent the similarity or associativity *weight* assigned on the attribute A in dimension *D*. In principle, if $A_i \preceq^*_D A_j$ holds, $Wt_D(A_i) \leq Wt_D(A_j)$.

Requirements in different applications are different, but the requirements dimension set over all applications are generally same (but just with different attributes inside). Based on the prioritized dimensions and weighted attributes, we can quantify the similarity degree ($\mathcal{SD}$) in a requirement set (RS = $\{r_1, \cdots, r_n\}$) by the formula:

$$\mathcal{SD}_{\mathsf{SDS}}(RS) = \sum_{D \in SDS} Pr_D \times Wt_D(Att_D(RS)) \quad (1)$$

$$Wt_D(Att_D(RS)) = \begin{cases} Max(Wt_D(A)),\ here\ A \in \\ \qquad AD_D(\cap_{i=1}^{n} Att(r_i)) \\ 0,\ if\ \cap_{i=1}^{n} Att(r_i) = \emptyset. \end{cases} \quad (2)$$

The associativity degree ($\mathcal{AD}$) for a requirement set can be quantified similarly to the above $\mathcal{SD}$ formula.

$$\mathcal{AD}_{\mathsf{ADS}}(RS) = \sum_{D \in ADS} Pr_D \times Wt_D(Att_D(RS)) \quad (3)$$

We use the form *attribute_name*[*weight_value*] to express the attribute (by *attribute_name*) and its similarity or associativity weight (by *weight_value*).

Example 3: continuing the above example, we firstly build the hierarchy in dimension S on R1 and R2 as

AH_{S}: *User*(*Super-User, Normal-User*)

Then we try to give the weight for each attribute in S and O with our experience:

AH_{O}: *Data*[1](*Client-Data*[2](*Name*[4], *Gender*[4], *Birthdate*[4]), *User-Data*[2] (*Name*[3], *Password*[3]));
AH_{S}: *User*[1](*Super-User*[2], *Normal-User*[2]).

We can calculate the related similarity degrees on {R1,R2}:
$\mathcal{SD}_{\mathsf{O}}$({R1,R2}) = 1, as {R1,R2} share attribute *Data*;
$\mathcal{SD}_{\mathsf{S}}$({R1,R2}) = 1, as {R1,R2} share attribute *User*.

5. Requirements Encapsulation

As mentioned in the introduction, our approach to encapsulating requirements includes two steps, first clustering requirements and then encapsulating each cluster by defining its external interface (a set of stimulus-response pairs).

5.1. Clustering

To cluster a set of various entities, a measurement criterion is important. In our approach, we define the *requirements clustering metric* ($\mathcal{RCM}$) of a requirement set (RS) as follows, based on its similarity degree $\mathcal{SD}_{\mathsf{SDS}}(RS)$ and associativity degree $\mathcal{AD}_{\mathsf{ADS}}(RS)$.

$$\mathcal{RCM}(RS) = Log(\mathcal{SD}_{\mathsf{SDS}}(RS) \times \mathcal{AD}_{\mathsf{ADS}}(RS))^C \quad (4)$$

Here C is an *adjustment factor* to balance the size of RS ($|RS|$) and the decreasing trend of $\mathcal{SD}$ and $\mathcal{AD}$ along $|RS|$ increasing[1]. For example, let C be $Log(n)$ or $\sqrt{n}$.

Based on the $\mathcal{RCM}$ criterion, we propose the *requirements clustering algorithm* to automatically organize a set of individual requirements into a set of clusters. *Requirements-Dimensions-Attributes* (RDA) relations record the attribute (with its similarity/associativity degree) of each requirement in each dimension.

Example 4: the attribute set of R1 in dimension O is {*Name, Gender, Birthdate*} [4] (the similarity degree is 4). Its complete *hierarchical path* is *Data*−*Client-Data*−{*Name, Gender, Birthdate*}[4].

We use hierarchical path to uniquely identify each attribute in a dimension, as there may exist some attributes with same name in different dimensions. We also record the hierarchical paths as the elements in the RDA table for computing convenience in the algorithm.

```
**Requirements Clustering Algorithm**
Input: RDA, $AH_D$ ($D \in DS$), RS = $\{r_1, \cdots, r_n\}$.
Output: $\{C_1^k, \cdots, C_n^k\}$. %% $C_i^k$ is the ith requirements cluster.
Body:
(1) Initialization: $k = 1; \forall i : C_i^k = r_i$;
(2) Maximal Absorption on $\mathcal{RCM}$
(3) For $i$: Build $C_i^{k+1}$: $C_i^k \subseteq C_i^{k+1}, \forall C_i'^k : C_i^k \subseteq C_i'^k \bullet$
(4)                      $\mathcal{RCM}(C_i^{k+1}) \geq \mathcal{RCM}(C_i'^k)$;
(5) Overlay Adjustment on $\mathcal{RCM}$
(6)   If $\exists i, j : C_i^{k+1} \cap C_j^{k+1} \neq \emptyset$
(7)     Find $C_i'^{k+1} \subseteq C_i^{k+1}, C_j'^{k+1} \subseteq C_j^{k+1} : C_i'^{k+1} \cap C_j'^{k+1} = \emptyset$
(8)       $\forall C_i''^{k+1} \subseteq C_i^{k+1}, C_j''^{k+1} \subseteq C_j^{k+1} \bullet \mathcal{RCM}(C_i'^{k+1}) +$
(9)       $\mathcal{RCM}(C_j'^{k+1}) \geq \mathcal{RCM}(C_i''^{k+1}) + \mathcal{RCM}(C_i''^{k+1})$;
(10)    Rebuild $C_i^{k+1} = C_i'^{k+1}, C_j^{k+1} = C_j'^{k+1}$.
```

The above algorithm can be similarly implemented with an undirected requirements graph G = (V, E) as in [4, 12]. Here, V is a finite set of vertices such that a vertex is a requirement, and E is a finite set of edges such that an edge is a relation between two requirements with the weight of $\mathcal{RCM}$ on the connected requirements. In comparison with the requirements clustering algorithms in [4, 12, 1, 8], the most significant difference with our algorithm is the clustering criterion that relies on the RDA table.

5.2. Encapsulating Clusters

To encapsulate a cluster of requirements, we need to define the cluster's *external interface*. In our approach, the term *external interface* of a requirements cluster can be defined as containing the cluster-level (or inter-cluster) relations with other clusters (i.e., requirements in other clusters) and actors (users, system procedures, etc. which have relations with the internal requirements), which are evolved from the relations between requirements during the clustering phase. Each cluster-level relation is specified as a *stimulus-response pair* to the cluster. The set of all stimulus-response pairs organizes the external interface of a requirements cluster, covering the *external behaviors* of the cluster during program modularization. The requirements in a cluster and their internal relations are implemented as in-module objects and object-level relations. While the external interface can be implemented as module-level relations through message communication. In our method, stimulus and response are defined with this format:

stimulus = (*source_requirement, trigger*)
response = (*target_requirement, response_behavior*)

[1]The $\mathcal{SD}_{\mathsf{SDS}}(RS) \times \mathcal{AD}_{\mathsf{ADS}}(RS)$ value will surely decrease when $|RS|$ increases. So we design a factor named C to avoid two situations, (1) adding a requirement into a cluster if it makes the value decreased much, and (2) not adding a requirement into a cluster if it impacts the value little.

Here *source requirement* in *stimulus* refers to the requirement that launches the *action* (named *trigger*) to another requirement (named *target requirement*). The possible behavior of target requirements responding to the trigger is called *response behavior*.

Example 5: let us consider the following requirement:
R3: *Non-medical entries are added to any client record*;
Suppose R3 and R1 are organized into two clusters by the above algorithm. In this situation, R3 has a cluster-level relation with R1, since any non-medical entries to a client record can be added only if the client record has been created, which is a temporal sequence constraint, requiring that R3 can be possibly accomplished only after R1 being accomplished. We can define the stimulus-response pair on R1 and R3 as follows:

(R1, *record-created*)−(R3, *activate-entry-option*)

Here *record-created* is the trigger that may grant R3 the capability of activating non-medical entries, as the response *activate-entry-option* indicated.

Requirements Clusters Encapsulation Process

(1) Choose a non-encapsulated cluster;
(2) For each requirement in the cluster, define its relations to the requirements not in the cluster; then transform each relationship to a stimulus-response (SR) pair;
(3) Adjust the derived SR pairs, unify the SR pairs with same subjects, objects or actions; then if necessary, add one (or more) requirement used as a unique internal interface (UII) to handle these SR pairs with the external requirements, while other internal requirements are all directly related with this requirement;
(4) Turn to step (1) if there exists any non-encapsulated clusters.

The above cluster encapsulation process is a semi-automated process. Two tasks in this process need manual effort. Those are (1) defining stimulus-response pairs, especially the naming of trigger and response behavior in each pair (2) building UII in a cluster which needs the human experience to determine its necessity. After stimulus-response pairs for each cluster are defined, we can code a module to directly implement a cluster of requirements (here we omit the design phase). The internal requirements attributes and relations can be captured as packages, classes (or objects) and members. The external interface can be addressed as (public) access methods to the module.

6. Conclusion

We describe a requirements encapsulation method as a means to reduce software development and maintenance costs. The unique features of this method are that each cluster maintains maximal independence from others and this method supports defining clusters' boundaries and the relationships between clusters. The proposed method needs to be verified with empirical analysis; what is described here are theoretical formulations of the concepts. The next step in our research plan is to conduct empirical investigation to measure its impact on the quality software development processes and requirement-design-code reuse.

Acknowledgement

We would like to thank all members of the software engineering research group in the Department of Computer Science at University of Western Ontario.

References

[1] T. N. Al-Otaiby, M. AlSherif, and W. P. Bond. Toward software requirements modularization using hierarchical clustering techniques. In *ACM Southeast Regional Conference (2) (ACMSE'05)*, pages 223–229, Kennesaw, GA, USA, 2005. ACM.

[2] D. Benavides, A. Ruiz-Cortés, P. Trinidad, and S. Segura. A survey on the automated analyses of feature models. *JISBD, José Riquelme and Pere Botella (Eds)*, 2006.

[3] A. Davis, S. Overmyer, and etc. Identifying and measuring quality in a software requirements specification. In *Proc. of 1st Int'l Software Metrics Symposium*, pages 141–152, Baltimore, MD, USA, 1993. IEEE.

[4] P. Hisa and A. T. Yaung. Another approach to system decomposition. In *The 12th Int'l Conf. on Computer Software and Applications Conference (COMPSAC'88)*, pages 75–82, Chicago, IL, USA, 1988. IEEE.

[5] K. Lee and K. C.Kang. Feature dependency analysis for product line component design. In *Proc. of the 8th Int'l Conf. on Software Reuse (ICSR'04)*, pages 69–85, 2004.

[6] Y. Lee and W. Zhao. A feature oriented approach to managing domain requirements dependencies in software product lines. In *Proc. of the 1st Int'l Multi-Symposiums on Computer and Computational Sciences (IMSCCS'06)*, 2006.

[7] W. Lim. Effects of reuse on quality, productivity, and economics. *IEEE Software*, 11(5):23–30, 1994.

[8] R. Lutowski. *Software Requirements: Encapsulation, Quality, and Reuse*. Auerbach Publisher, 2005.

[9] H. D. Mills. Stepwise refinement and verification in box-structured systems. *IEEE Computer*, 21(6):23–36, 1988.

[10] H. D. Mills, R. Linger, and A. Hevner. Box structured information systems. *IBM Systems Journal*, 26(4):395–413, 1987.

[11] O. L. Villegas, M. Ángel Laguna, and F. J. García. Reuse based analysis and clustering of requirements diagrams. In *Proc. of Int'l Workshop Conf. on Requirements Engineering: Foundation for Software Quality (REFSQ'02)*, 2002.

[12] A. T. Yaung. Design and implementation of a requirements clustering analyzer for software system decomposition. In *ACM/SIGAPP Symposium on Applied Computing: Technological Challenges of the 1990's*, pages 1048–1054, Kansas City, Missouri, USA, 1992. ACM.

[13] W. Zhang, H. Mei, and H. Zhao. Feature-driven requirements dependency analysis and high-level software design. *Requirements Engineering*, 11:205–220, 2006.