Automating Requirements Refinement with Cross-Domain Requirements Classification

Jacob L. Cybulski
Department of Information Systems
The University of Melbourne
Parkville, Vic 3052, Australia
Phone: +613 9344 9244, Fax: +613 9349 4596
Email: j.cybulski@dis.unimelb.edu.au
WWW: http://www.dis.unimelb.edu.au/staff/jacob/

Karl Reed
School of Comp Science and Comp Engineering
La Trobe University
Bundoora, Vic 3083, Australia
Phone: +61 3 9479 1377
Email: kreed@latcs1.cs.latrobe.edu.au

Abstract

Introduction of software reuse into the development process significantly alters the life-cycle model, and has a tremendous impact on all of its software development phases. This also includes the phase of requirements engineering, its activities and the tools used in the process. To deal with the consequences of incorporating reuse in the early phases of software development, we suggest the use of a special process model, capable of simultaneously meeting the goals of software reuse and the goals of requirements engineering. The RARE (Reuse-Assisted Requirements Engineering) method proposes such a model. RARE places a particular emphasis on two aspects of requirements processing, i.e. requirements classification and domain mapping. Both of these aspects are especially helpful in the automation of requirements refinement into reusable designs.

Keywords

Requirements Refinement, Software Reuse, Information Retrieval

1. Introduction

It is commonly accepted that one of the aims of requirements engineering is to derive, validate and maintain a high-quality requirements specification document [23, p 9]. Such a document is a cornerstone of the subsequent development activities, it is considered a record of user communication with developers, a repository of user opinions and wishes, and a legal contract binding developers to deliver particular system functions. A collection of statements included in a requirements document is the ultimate deliverable of the requirements engineering process.

Requirements specifications are developed jointly by the system users and developers, the two vastly different groups of system stakeholders, each having its own needs and preferences when it comes to the form and contents of requirements documents. Developers prefer requirements to be expressed in some formal and unambiguous notation, whether diagrammatic [9, 35], model-based [17, 26], algebraic [13] or object-oriented [16]. Non-technical users of the future systems, however, prefer to convey their needs in the most familiar and flexible fashion, i.e. through the informal communication in natural language (e.g.

English). Natural language seems to be the most prevalent specifications vehicle in requirements engineering, and it also seems to be one of the least suitable for the ensuing development due to its ambiguity, imprecision and vagueness [24].

To rectify this problem, system and software development standards impose stringent quality constraints on the form and style of informal requirements specifications [19, 34]. They provide guidelines for the document structure and its contents, and they give recommendations on the degree of their precision, consistency and completeness. There exist attempts to handle informal requirements with the use of methods and tools automating or assisting analysis of natural language texts. Techniques that proved promising include those for text structuring and navigation techniques [4, 12, 22], hand-analysis of text phrases [1, 36], natural language processing [2, 3, 8, 15, 21, 27, 38], text generation from formal specifications [20, 32], or the use of knowledge-based systems [5, 6, 21, 30, 31]. Automatic techniques employed in the processing of informal requirements are usually applied after requirements have been collected and structured into a cohesive text and then passed to developers for further processing. They are used to extract the essence of requirements buried in the natural language text, or to identify cross-textual references, or to find pointers to interesting requirements, or to identify document structure, or to get just about any useful information from the illstructured and incomplete documents. Such techniques are commonly used as an act of developer's desperation.

We believe that requirements engineering must better serve its two communities of stakeholders, i.e. users and developers. To do so it must result not only in requirements documents that are easily understood by non-technical users but also in information that could support and facilitate the subsequent development phases, e.g. those concerned with the mapping of requirements specifications into software designs. We, hence, propose that the requirements engineering process should be broadened to centre not only on the tasks of requirements recording, analysis and specification but should also consider later refinement of these requirements.

RARE (Reuse-Assisted Requirements Engineering) is a requirements engineering method that defines such an extended process model. It follows a typical cycle of requirements engineering tasks [18], i.e. elaboration of needs and objectives, requirements acquisition and modelling, generation and evaluation of alternative interpretations (and possible refinements), and finally requirements verification and validation. RARE's key principle is to enhance software development by the continuous pursuit of requirements similarity, opportunities to reuse existing requirements and their refinement into reusable designs. In doing so, RARE offers designers not only a voluminous collection of requirements text but also some additional information on their possible refinement.

The issue pivotal to RARE's process model is developer's ability to effectively and efficiently assess the similarity of requirements and to be able to match such requirements against reusable design artefacts deemed suitable for their refinement. To do so we suggest automating these tasks by active utilisation of information classification and retrieval techniques.

In the following sections, we shall briefly describe the RARE method. We will explain its activities and show them as promising in support of requirements reusability and refinement. We will outline the principles of a tool providing support for the RARE method (such a tool has been developed but its details are beyond the scope of this paper), and we will illustrate its operation based on examples of requirements for a small computer Dice Game (see the left column of Table 1).

2. Requirements Gathering

The first step towards the RARE process of requirements processing is to establish a collection of requirements that could be reused from one project to another, and a collection of design artefacts that could be used to refine them (see Table 1). Together they constitute a repository of reusable artefacts. Informal requirements are normally expressed as text in natural language (such as English - D1 to D7). Design artefacts are commonly referred to by their descriptive name (A1 to A14) but are recorded in the form of diagrams, tables or highly formalised text. For each requirement statement there may be a number of design

Small System Definition Guide The Dice Game System Saturday, May 15, 1999	Design Artefacts
1. Purpose of the System The purpose of the Dice Game system is to provide entertainment to its users. **2. Scope of the System** ♦ The system will include only a single module. ♦ The system will not interface with any other external system. **3. Functions** *Enter all your user requirements in plain but simple English expressions.* D1. The system shall allow players to specify the number of dice to "roll". D2. The player shall then roll dice. D3. Each die represents numbers from 1 to 6. D4. The dice are assigned their values randomly. D5. Every time the dice are rolled, their values are assigned in a random fashion. D6. If the total of both dice is even, the user shall win.	A1. array A2. number A3. string A4. command A5. read A6. write A7. set dimension A8. set numeric value A9. random number generator A10. sum of numbers A11. compare number A12. quit program A13. you've won dialogue box A14. you've failed dialogue box ***Note:*** *these simple design artefacts are very low level of abstraction they are for demonstration purposes only.*

Table 1: Sample requirements statements (D1-D6) and design artefacts (A1-A14)

artefacts perceived as useful in its refinement. After deliberation, some of the best matching candidate artefacts will be selected and combined to form a single design, others will be rejected as inappropriate.

After a requirements document has been entered, and subsequently analysed, refined, verified and validated, it may have to undergo several cycles of correction and elaboration. We assume that a large requirements document would indeed be developed in a piecemeal fashion, perhaps by following the natural system partitions clearly reflected in the structure of the requirements template (or a master document). This approach would allow interspersing the construction of requirements documents with domain analysis, focusing on incremental and iterative identification of a domain model and its terminology.

3. Requirements Analysis

Well organised requirements documents with known structure, grammar and terminology can facilitate methodical analysis of requirements contents. In RARE, we suggest to process requirements text with a view to identifying requirements, representing their features, and characterising them by reference to domain concepts. Such analysis may involve statistical analysis of terminology used in requirements documents, parsing and skimming of text, representation and manipulation of knowledge introduced in these documents, etc. In our project, we focussed on yet another approach, i.e. nominating lexically and syntactically interesting concepts to become members of a list of domain terms characteristic to each requirement.

Table 2 shows a number of simple requirements statements drawn from the Dice Game. For each statement a list of domain terms was selected from their text. The terms were standardised, i.e. nouns were converted into singular form and verbs into infinitive tense. Some of the selected terms were characteristic to the domain of game playing, e.g. "dice", "player", "roll", "win" and "loose". Others were common to many application domains, e.g. "specify", "the number of", "represent", "random", etc. The terms were also ordered by their *relevance* to the meaning of their respective requirement. We believe that the verbs signifying the system function should be considered of the highest importance, then come the nouns that commonly refer to data objects, then come all other words found in the domain lexicon. It will be shown later that these problem domain terms can be mapped into a solution domain for the classification purposes to support requirements refinement and reuse.

We should draw the reader's attention to a pair of requirements D6 and D7. Although both relate to the game's termination condition, they were assigned characteristic terms using two different principles. Requirement D7 is quite simple and has a straightforward interpretation leading to the presented list of terms. The D6 requirement's characterisation, however, focuses on the method of detecting a winning

The Dice-Game System	Term list: Relevance:	Term 1 0.4	Term 2 0.3	Term 3 0.2	Term 4 0.1
D1. The system shall allow players to specify the number of dice to "roll".		specify	the number of	dice	player
D2. The player shall then roll dice.		roll	dice	player	then
D3. Each die represents numbers from 1 to 6.		represent	dice	number	each
D4. The dice are assigned their values randomly.		assign	value	dice	random
D5. Every time the dice are rolled, their values are assigned in a random fashion.		assign	value	random	every
D6. If the total of both dice is even, the user shall win.		total	even	win	if
D7. Otherwise the user looses.		loose	user	otherwise	

Table 2: Dice-game requirements characterisation with *lists of domain terms*

situation, rather than on the situation's end result. Such differing approaches will ultimately lead to differences in the later processing of these two requirements.

4. Requirements Refinement

The key to the successful refinement of requirements documents is in the effective mapping of requirements (defined in the problem domain) into a space of software design (part of the solution domain), of which a significant part should be reusable. This is normally done manually by skilled computer analysts and designers. However, the process can be enhanced with the use of information retrieval techniques. Such techniques usually call upon methods of classification, search, retrieval and selection of available information. In RARE, we consider a collection of reusable requirements and design artefacts as the body of retrievable information, and the text of analysed requirements as a query against this repository.

In the following sections we explain the cycle of activities recommended by RARE and illustrate them with examples drawn from the Dice Game system. We start with a description of the RARE's classification scheme that is based on the Prieto-Diaz' faceted classification [28, 29]. We then outline the similarity-based retrieval of artefacts [7, 29, 33]. Finally, we demonstrate the mapping of requirements characteristics (lists of problem domain terms) into the form that is consistent with the classification of reusable design artefacts (vectors of solution domain terms), so that they could be used in the construction of a query against the potential refinement artefacts.

4.1 Requirements and Design Classification

Classification of software artefacts (whether requirements specifications, design or code) can utilise one of several well-documented methods, e.g. keyword-based, attribute-value, enumerative or faceted [10, 29]. These methods provide similar usability and effectiveness [11], however, faceted classification seems most appropriate for our purpose. Its classification procedure allows a natural, multi-attribute view of artefact characteristics, it is simple to enforce, its search method is easy to implement, and its storage facilities can utilise a standard database technology.

Contrary to the general practice of classifying requirements using terms found in document's text [7, 14, 37, 39] (as illustrated in Table 2), we classify requirements based on terms drawn from the solution domain (see Table 3).[1] Such an approach is dictated by the need to classify design artefacts and requirements in a uniform fashion, so that it would be possible to determine their similarities and affinities. The similarity between two requirements will indicate the possibility of requirements reuse, overlap or conflict. The affinity

[1] Note that this approach does not preclude other classification and retrieval methods to be used concurrently.

	Function (weight 0.4)	Data (weight 0.3)	Method (weight 0.2)	Environ. (weight 0.1)
Design Artefacts				
A1. array	define	collection	iteration	machine
A2. number	define	number	direct	machine
A3. string	define	string	iteration	machine
A4. command	control	data	direct	user
A5. read	input	data	query	user
A6. write	output	data	report	user
A7. set dimension	define	multiplicity	elaboration	machine
A8. set numeric value	assign	number	direct	machine
A9. random number generator	calculate	number	random	machine
A10. sum of numbers	add	number	iteration	machine
A11. compare number	compare	number	choice	machine
A12. quit program	end	data	direct	machine
A13. you've won dialogue box	output	boolean	report	success
A14. you've failed dialogue box	output	boolean	report	failure
Dice Game Requirements				
D1. The system shall allow players to specify the number of dice to "roll".	define	multiplicity	elaboration	user
D2. The player shall then roll dice.	assign	value	random	instrument
D3. Each die represents numbers from 1 to 6.	define	value	iteration	instrument
D4. The dice are assigned their values randomly.	assign	value	direct	instrument
D5. Every time the dice are rolled, their values are assigned in a random fashion.	assign	value	direct	instrument
D6. If the total of both dice is even, the user shall win.	add	collection	iteration	success
D7. Otherwise the user looses.	end	boolean	choice	failure

Table 3: Classification of design artefacts and of requirements statements

between a requirement and a design artefact points to the possible refinement of a requirement. The similarity between two design artefacts provides a vehicle for the investigation of design alternatives.

Table 3 shows a selection of design artefacts (A1-14) and several requirements artefacts (D1-7). All artefacts are classified using a faceted classification scheme. Each artefact is described with a descriptor, which is a vector of descriptor values or terms, each signifying a different artefact's attribute or facet. In a faceted classification system, there exist a fixed number of such facets, they are orthogonal to each other, and each defines a number of terms that could be used as valid descriptor values. Facets can also be weighed based on their importance.

In our example, we introduced four weighted facets, i.e.

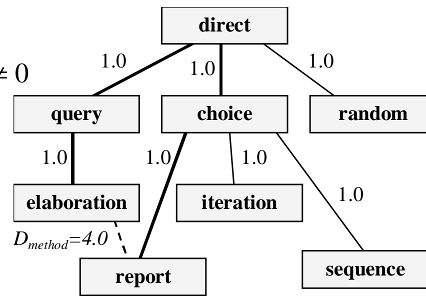
- the "function" facet (importance 0.4) which describes the operation performed by the artefact (e.g. "assign", "output" or "calculate") or its role in program computation or its development (e.g. "define");
- the "data" facet (importance 0.3) which defines data objects manipulated (e.g. "number" or "string"), their attributes (e.g. "multiplicity"), or the object types represented by the artefact ("collection");
- the "method" facet (importance 0.2) which describes how the functional artefact achieves its processing goals or what type of methods are needed to process a data artefact (e.g. "iteration", "direct" or "choice"); and,
- the "environment" facet (importance 0.1) which specifies the context of the computation (e.g. "machine" or "user"), or whether it impacts some of the external factors (e.g. "success" or "failure").

Having defined all of the facets, it is now possible to classify all of the artefacts, of either requirements or design origin. Classifying artefacts by hand could be extremely labour-intensive and totally unworkable for any practically sized requirements task [25]. The following sections offer a description of an automatic technique, which significantly reduces the effort required, making the method more tractable. With these

Equation 1: Conceptual distance calculation

$$D_f(x_1, x_n) = \begin{cases} 0 \text{ for } x_1 = x_n \\ \min \sum_{i=1}^{n-1} A_f(x_i, x_{i+1}) \text{ where } \underset{1 \le i < n}{\forall} : A_f(x_i, x_{i+1}) \neq 0 \\ \sum_{x,y \in F_f} A_f(x, y) \text{ otherwise} \end{cases}$$

$$d_f(q, a) = \frac{D_f(q, a)}{\max_{x,y} D_f(x, y)}$$



where:

$D_f(x, y)$ - conceptual distance between values "x" and "y" in facet "f"

$d_f(x, y)$ - normalised conceptual distance between values "x" and "y" in facet "f"

$A_f(x, y)$ - associative distance between facet values "x" and "y" in facet "f"

F_f - a set of "x_i" values in facet "f"

---

alterations, we believe that the faceted approach provides the most versatile form of classifying requirements and design artefacts.

4.2 Artefact Similarity

Descriptors are considered the most important artefact characteristic that can be used to distinguish one artefact from another, and which are used to determine the artefacts' similarity.

The closer the match in the corresponding descriptor values, the greater similarity of the compared artefacts. For example (see Table 3), alike data artefacts "array" and "string" have very similar sets of facet values, quite distinct from the values in the descriptors of functional artefacts "write" or "random number generator". The comparison of artefacts based on the identity of descriptor values is not very useful when we have to deal with thousands of artefacts, some of them very close semantically. Hence, we need to have a finer-grain comparison of artefact descriptors, and therefore we need a finer-grain comparison of descriptor values.

Apart from providing a collection of classification terms, the facets can also define a *conceptual distance* measure between facet values (see Figure 1). A small conceptual distance value indicates the facet values to be very close, conversely, large distance represents the two facet values to be conceptually far apart.

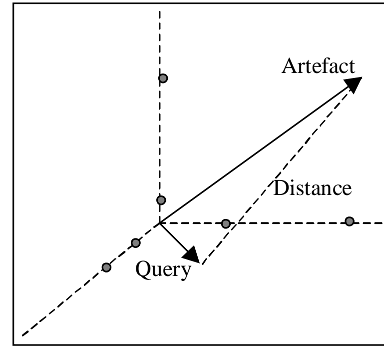
During the development of the facet structure, facet values are frequently represented as an associative network, or a connected weighted graph, of inter-related concepts (see part of Equation 1, also tabulated in Figure 3). In this model, the measure of closeness between two concepts can be defined as an *associative distance*, i.e. the weight attached to

Conceptual distance matrix for the "function" facet	add	any	arrange	assign	calculate	compare	control	count	define	do	end	execute	input	interact	none	output	start
add	0	0	3	3	1	3	4	2	4	3	5	2	5	4	6	5	5
any	0	0	0	0	0	0	0	0	0	0	0	0	0	0	6	0	0
arrange	3	0	0	2	2	2	3	3	3	2	4	1	4	3	6	4	4
assign	3	0	2	0	2	2	3	3	3	2	4	1	4	3	6	4	4
calculate	1	0	2	2	0	2	3	1	3	2	4	1	4	3	6	4	4
compare	3	0	2	2	2	0	3	3	3	2	4	1	4	3	6	4	4
control	4	0	3	3	3	3	0	4	2	1	1	2	3	2	6	3	1
count	2	0	3	3	1	3	4	0	4	3	5	2	5	4	6	5	5
define	4	0	3	3	3	3	2	4	0	1	3	2	3	2	6	3	3
do	3	0	2	2	2	2	1	3	1	0	2	1	2	1	6	2	2
end	5	0	4	4	4	4	1	5	3	2	0	2	4	3	6	4	2
execute	2	0	1	1	1	1	2	2	2	1	2	0	3	2	6	3	3
input	5	0	4	4	4	4	3	5	3	2	4	3	0	1	6	2	4
interact	4	0	3	3	3	3	2	4	2	1	3	2	1	0	6	1	3
none	6	6	6	6	6	6	6	6	6	6	6	6	6	6	0	6	6
output	5	0	4	4	4	4	3	5	3	2	4	3	2	1	6	0	4
start	5	0	4	4	4	4	1	5	3	2	2	3	4	3	6	4	0

Figure 1: "Function" facet (conceptual distance matrix)

Equation 2: Artefact distance and similarity calculation

$$dist(q,a) = \sqrt{\frac{\sum_i (w_i \times d_i(q_i,a_i))^2}{\sum_i w_i^2}}, \qquad sim(q,a) = 1 - dist(q,a)$$



Where:

dist(q, a) - normalised distance between artefacts "q" and "a"
sim(q, a) - similarity between artefacts "q" and "a"
q_i *- i-th facet value in the query vector*
a_i *- i-th facet value in the artefact vector*
w_i *- importance of the i-th facet*
$d_i(a, b)$ *- normalised distance between values in the i-th facet*

each network link. For the concepts that are not directly associated, the length of the shortest path leading from one concept to another would determine their conceptual distance [29] (cf. Equation 1). Inability to reach one facet value from another would be represented by the length of the longest possible path in a facet network (or the sum of all facet value associative distances).

After calculating the distances between all the facet values, we represent the facet structure as a conceptual distance matrix (effectively a path matrix, see Figure 1). To counter the facet size differences, this distance could also be normalised to the interval of *[0, 1]* by dividing the conceptual distance between facet values by the maximum path length within the facet. The following are two sample distance calculations between values ("add" vs. "arrange") and ("assign" vs. "arrange") in the "function" facet (see Figure 1):

$D_{function}$("add", "arrange") = 3 / 6 = 0.5000
$D_{function}$("assign", "arrange") = 2 / 6 = 0.3333

The faceted system that classifies requirements using the terms drawn from the solution domain provides an opportunity to match requirements against their potential refinement artefacts. This could be accomplished by using the requirement-descriptor as a query vector against a repository of design-artefact vectors. In doing so we calculate the query / artefact similarity using a weighted geometric distance metric (cf. Equation 2).

Table 4 shows an example of calculating similarity between two artefact descriptors. One of the descriptors represents a design artefact (data structure A1) and the other represents a requirement for the system's function (dice game D1). Some of the facet values of the two artefact descriptors are identical (e.g. "define"), some are close (e.g. "collection" vs. "multiplicity"), other are dissimilar (e.g. "iteration" vs. "elaboration"). Based on the importance of individual facets, the distances are scaled, squared and added (the conceptual distance matrices used in the calculation are shown in Figure 1, Figure 2, Figure 3 and Figure 4). As the result, the overall similarity value, 0.6859, represents the degree of match between the artefacts represented by the two descriptors.

	Compared Artefacts								
	function		**data**		**method**		**environment**		
Artefact A1:	define	(9)	collection	(4)	iteration	(5)	machine	(5)	
Requirement D1:	define	(9)	multiplicity	(7)	elaboration	(4)	user	(10)	
	Similarity Analysis								**SumSQ:**
Facet Weight:	0.40		0.30		0.20		0.1		**0.30**
Maximum Distance:	6		5		5		5		
Concept. Distance:	0		1		4		1		**Similarity:**
Norm. Distance:	0.00		0.06		0.16		0.02		**0.6859**

Table 4: Detailed similarity calculation for two artefact descriptors

Conceptual distance matrix for the "data" facet	any	boolean	character	collection	data	matrix	multiplicity	none	number	pair	single	string	value	variable	vector
any	0	0	0	0	0	0	0	5	0	0	0	0	0	0	0
boolean	0	0	2	3	2	4	4	5	2	4	1	4	1	1	1
character	0	2	0	3	2	4	4	5	2	4	1	4	1	1	4
collection	0	3	3	0	1	1	1	5	2	1	2	1	1	1	1
data	0	2	2	1	0	2	2	5	2	2	1	2	1	1	2
matrix	0	4	4	1	2	0	1	5	4	2	3	2	1	1	2
multiplicity	0	4	4	1	2	1	0	5	4	1	3	1	1	1	1
none	5	5	5	5	5	5	5	0	5	5	5	5	5	5	5
number	0	2	2	2	2	4	4	5	0	4	1	4	1	1	1
pair	0	4	4	1	2	2	1	5	4	0	3	2	1	1	2
single	0	1	1	2	1	3	3	5	1	3	0	3	1	1	3
string	0	4	4	1	2	2	1	5	4	2	3	0	1	1	2
value	0	1	1	1	1	1	1	5	1	1	1	1	0	1	1
variable	0	1	1	1	1	1	1	5	1	1	1	1	1	0	1
vector	0	1	4	1	2	2	1	5	1	2	3	2	1	1	0

Figure 2: "Data" facet (conceptual distance matrix)

Conceptual distance matrix for the "method" facet	any	choice	direct	elaboration	iteration	none	query	random	report	sequence
any	0	0	0	0	0	5	0	0	0	0
choice	0	0	1	3	1	5	2	2	1	1
direct	0	1	0	2	2	5	1	1	2	2
elaboration	0	3	2	0	4	5	1	3	4	4
iteration	0	1	2	4	0	5	3	3	2	2
none	5	5	5	5	5	0	5	5	5	5
query	0	2	1	0	3	5	0	2	3	3
random	0	2	1	3	3	5	2	0	3	3
report	0	1	2	4	2	5	3	3	0	2
sequence	0	1	2	4	2	5	3	3	2	0

Figure 3: "Method" facet (conceptual distance matrix)

Conceptual distance matrix for the "environment" facet	any	failure	game	instrument	machine	money	none	result	success	user
any	0	0	0	0	0	0	5	0	0	0
failure	0	0	2	4	2	4	5	1	2	3
game	0	3	0	3	1	3	5	2	3	2
instrument	0	4	3	0	2	2	5	3	4	1
machine	0	2	1	2	0	2	5	1	2	1
money	0	4	3	2	2	0	5	3	4	1
none	5	5	5	5	5	5	0	5	5	5
result	0	1	2	3	1	3	5	0	1	2
success	0	2	3	4	2	4	5	1	0	3
user	0	3	2	1	1	1	5	2	3	0

Figure 4: "Environment" facet (conceptual distance matrix)

After completing the calculation of similarity between the requirement's descriptor and the descriptors of all available design artefacts we could order them by their similarity to the requirement. Having done that, we would be in a position to nominate a number of reusable design artefacts that are most similar and hence possibly most useful in the refinement of the requirement statement used as the original query. The main problem with the proposed method of classification and retrieval, as described so far however, is that requirements texts use terminology of the problem domain (see Table 2) rather than that of the solution domain, which needs to be used to retrieve design artefacts (see Table 3). The majority of the existing retrieval systems treat text indexing and classification as a kind of text summarisation, hence, such schemes rely on the fact that the document terms are extracted from the body of their text.

4.3 Domain-Mapping Thesaurus

To be able to classify requirements into the common space of requirements and design artefacts, we would first have to conduct detailed analysis of their text. Having completed this analysis, we would then need to identify possible solutions to the problem stated in the requirement. Finally, candidate solutions to the problem would have to be identified, e.g. by deriving a sample design from the requirement, and only then we'd be able to classify the requirement in terms of the domain concepts associated with these candidate solutions.

Such a classification method would be purely manual, which is very much the case in the majority of the commonly used faceted classification systems, and also the source of criticisms addressed towards them [25]. Due to the large amount of labour and great attention to detail required in the process of manual

Thesaurus	**Weighed Facet Value Senses**							
Game Domain	**Function**	**0.4**	**Data**	**0.3**	**Method**	**0.2**	**Environ.**	**0.1**
1. card			value	1.0			instrument	0.5
2. coin			value	1.0			instrument	0.5
3. deal	start	1.0			iteration	0.7	game	0.3
4. dice			value	1.0			instrument	0.5
5. flip	assign	1.0	value	1.0	random	0.5		
6. loose	end	0.7					failure	1.0
7. player	interact	0.7					user	1.0
8. roll	assign	1.0	value	1.0	random	0.5		
9. shake	arrange	1.0	value	0.7	random	0.9		
10. shuffle	arrange	1.0	value	0.7	random	0.9		
11. win	end	0.7					success	1.0

General Domain	**Function**	**0.4**	**Data**	**0.3**	**Method**	**0.2**	**Environ.**	**0.1**
12. assign	assign	1.0	value	1.0	direct	0.5		
13. both			pair	1.0	sequence	0.7		
14. depict	output	1.0			direct	0.5	user	0.7
15. each			collection	0.5	iteration	1.0		
16. every			collection	0.3	iteration	1.0		
17. if			boolean	0.3	choice	1.0		
18. number			number	1.0				
19. otherwise			boolean	0.3	choice	1.0		
20. random					random	1.0		
21. represent	define	1.0						
22. specify	define	1.0			elaboration	0.4	user	0.7
23. the number of	count	0.7	multiplicity	1.0				
24. then					choice	1.0		
25. total	add	1.0	collection	0.7	iteration	0.4		
26. two			pair	1.0	sequence	0.7		
27. user	interact	0.7					user	1.0
28. value			value	1.0				

Table 5: Domain-mapping thesaurus with senses strengths

classification of artefacts, the cost of faceted classification may outweight its potential benefits. However, with adequate automation, such a technique would be more likely to be used in practice.

To resolve this problem, we developed the concept of a domain-mapping thesaurus. The thesaurus classifies all of the terms commonly found in the lexicon of a problem domain into the facets of a solution domain. Such term pre-classification helps us in automating the classification of requirements, which use this pre-classified terminology. The motivation behind this approach stems from our belief that the problem domain lexicon is much smaller as compared with the space of requirements that use its terminology. It means that the effort of classifying such a lexicon would be far smaller than the effort of classifying the great many requirements statements themselves.

The thesaurus was therefore designed as a repository of problem-domain concept descriptors *sensing* (associating or linking) facet values drawn from the solution domain. Table 5 (see previous page) shows an example of such a thesaurus. It describes two problem domains, i.e. the domain of software games and a general application domain. The "game" domain (Cf. Section 3) defines terminology specific to the objects and actions observed in card (e.g. "card", "deal" or shuffle"), dice (e.g. "dice", "roll" or "shake"), coin (e.g. "coin" and "flip"), and other generic games (e.g. "player", "win" or "loose"). The "general" domain lists common-sense concepts that occur in many different types of requirements documents (these will include such terms as "represent", "assign" or "user").

Each thesaurus term senses several facet values, which provide the interpretation of the term, give its semantics, provide hints on its design and implementation, and at the same time classify it. For instance, a coin represents a value in a game and it is the game's instrument. To flip the coin means to assign its value in a random fashion. The game can end either by winning or loosing, which represent the user's success or failure. Considering these points, the thesaurus may be designed to map the notion of a "coin" into a descriptor with facet values data="value" and environment="instrument", a "flip" concept into function="assign", data="value" and method="instrument", and the "win" concept into function="end" and environment="success".

The thesaurus terms sense each of their facet values with different strength, which is measured as a number in the interval *[0, 1]*. The *sense strength* indicates the relevance of a given facet value to the interpretation of a problem domain term. For example, the notion of a game "player" senses two facet values, i.e. environment="user" and function="interact" with strengths of 1.0 and 0.7 respectively. What this means

Requirement D1: *The system shall allow players to specify the number of dice to "roll".*					
Problem Domain		*Issue*	**Solution Domain**		
Term	**Importance**		**Alternative 1**	**Alternative 2**	**Alternative 3**
specify	40%	***solution terms:***	*function= define*	*environment= user*	*method= elaboration*
		sense strength:	1	0.7	0.4
		normal strength:	0.48	0.33	0.19
		priority:	0.41	0.39	0.38
the number of	30%	***solution terms:***	function= count	*data= multiplicity*	
		sense strength:	0.7	1	
		normal strength:	0.41	0.59	
		priority:	0.31	0.33	
dice	20%	***solution terms:***	data= value	environment= instrument	
		sense strength:	1	0.5	
		normal strength:	0.67	0.33	
		priority:	0.25	0.21	
player	10%	***solution terms:***	environment= user	function= interact	
		sense strength:	1	0.7	
		normal strength:	0.59	0.41	
		priority:	0.15	0.13	

Table 6: The process of domain mapping

Equation 3: Facet sense priority and facet value selection

$$p_{i,j} = \alpha \times w_i + \beta \times \frac{s_{i,j}}{\sum_k s_{i,k}}$$

$$f_i = f_{i,k} : p_{i,k} = \max_j p_{i,j}$$

terms		facet value senses		
w_1 t_1	$\rightarrow$	$s_{1,1}$ $f_{1,1}$	...	$s_{1,m}$ $f_{1,m}$
w_2 t_2	$\rightarrow$	$s_{2,1}$ $f_{2,1}$	...	$s_{2,m}$ $f_{2,m}$
	...		...	
w_n t_n	$\rightarrow$	$s_{n,1}$ $f_{n,1}$	...	$s_{n,m}$ $f_{n,m}$

Where:
t_i *- i-th problem domain term of some requirement*
w_i *- relevance of the i-th term in some requirement*
$f_{i,j}$ *- j-th facet value sense of term i*
$s_{i,j}$ *- j-th sense strength of facet j in term i*
$p_{i,j}$ *- facet value priority of facet j in term i*
α *- importance of the relevance factor (0.9)*
β *- importance of the sense strength (0.1)*
$\alpha + \beta = 1$

is that a domain engineer believed that being a player means being a user of the game system and being able to interact with the game. The engineer may also believe that for a player, interactivity is less important than being its user.

Although in the process of domain analysis in RARE the sense strengths are assigned subjectively to the thesaurus terms, it is possible to calculate them automatically as frequency counts of the facet values present in descriptors of refined requirements.

4.4 Domain-Mapping Process

The domain-mapping thesaurus provides the link between the terms from a problem domain (or domains) and the facet values defined in the solution domain. Now we need a method that could automate the domain-mapping process.

Let us introduce the elements of this process by means of examples (see Table 6 on the previous page). Consider a single requirement for the dice game system (see Table 1), i.e. "The system shall allow players to specify the number of dice to “roll”" (D1). The requirement statement was initially assigned a list of problem domain terms (see Table 2), which include "specify", "the number of", "dice" and "player". Each of these terms senses a few facet values from the solution domain (cf. Table 5), e.g. "specify" senses the facet values function="define", environment="user" and method="elaboration", whereas "dice" senses the facet values data="value" and environment="instrument". Since each of the problem domain terms senses several facet values, the mapping of the terms from two domains is not one-to-one. To resolve the mapping, we take advantage of two factors stored and available to the domain-mapping process, i.e.

- the term relevance to the requirement characterisation; and,
- the strength of each of the term's senses.

The two factors are combined together to calculate the *facet value priority* for the classification (cf. Equation 3). The priorities are then used to rank all possible classifications of each requirement (see Table 7). In those cases when a number of facet values share the highest priority any one of them will be used in further processing. The requirements engineer may either accept the proposed classification terms (shown in *italic*), change the suggested prioritisation, or to allocate to the facet a term, which has not been previously selected by the domain-mapping process, but which is defined in the respective facet.

4.5 Requirements Refinement

At this point in requirements processing, characteristic terms drawn from the text of individual requirements (the problem domain) can be mapped, either automatically or with the assistance of the requirements engineer, into a collection of design facet values (the solution domain) used in the classification of reusable design artefacts. This indicates that requirements descriptors can now be constructed and compared against those of design artefacts. Matching design artefacts can subsequently be used in the process of selecting the most appropriate design artefacts and their combination in requirement's refinement (cf. Section 4.2).

Requirements	Function		Data		Method		Environment	
1. The system shall allow players to specify the number of dice to “roll”.	*define* count interact	0.41 0.31 0.13	*multiplicity* value	0.33 0.25	*elaboration*	0.38	*user* instrument user	0.39 0.21 0.15
2. The player shall then roll dice.	*assign* interact	0.40 0.22	*value* value	0.40 0.34	*random* choice	0.38 0.19	*instrument* user	0.30 0.24
3. Each die represents numbers from 1 to 6.	*define*	0.46	*value* number collection	0.34 0.28 0.12	*iteration*	0.16	*instrument*	0.30
4. The dice are assigned their values randomly.	*assign*	0.40	*value* value value	0.40 0.37 0.25	*direct* random	0.38 0.19	*instrument*	0.21
5. Every time the dice are rolled, their values are assigned in a random fashion.	*assign*	0.40	*value* value collection	0.40 0.37 0.11	*direct* random iteration	0.38 0.28 0.17		
6. If the total of both dice is even, the user shall win.	*add* end	0.41 0.22	*collection* boolean	0.39 0.11	*iteration* choice	0.38 0.17	*success*	0.24
7. Otherwise the user looses.	*end* interact	0.40 0.31	*boolean*	0.20	*choice*	0.26	*failure* user	0.42 0.33

Table 7: Resolution of requirements classification terms

Although the descriptors of requirements and designs can be compared for their similarity, the text and the names of the two artefact types may differ dramatically (e.g. see Table 8). We therefore suggest abandoning the term "similarity" to describe requirements/design relationships, in favour of "affinity" - a more precise notion in this context. *The Concise Oxford Dictionary* (1982 edition) defines *affinity* as:

> "*n.* relationship, relations, esp. by marriage; structural resemblance (between animals, plants, languages); (fig.) similarity of character suggesting relationship, family likeness; liking, attraction; person having attraction for another; (Chem.) tendency of certain substances to combine with others."

Requirements/design affinity is, therefore, an attraction of one artefact type to another. The affinity concept goes beyond the superficial characteristics of the compared artefacts (e.g. terminology or syntax). It is their tendency to match their deep semantic features, and to combine them in the refinement process.

Table 8 reports affinity values between the dice game requirements and a range of reusable design artefacts. The same values correspond to the similarity of requirements and design descriptors as derived automatically (without any analyst’s interference) from the requirements terminology. All the figures in **bold** indicate the highest affinity of design artefacts (rows) to individual requirements (columns). According to the table, specification of the dice number (D1) signals setting the dimension of a collection (A7), rolling dice (D2, D4 and D5) points to assigning the dice value (A8), calculating the score (D6) associates with the sum of numbers (A10), etc.

The highest affinity value does not always suggest the only artefact that could be used in requirement refinement. For example, requirement D1 could be refined with the use of two artefacts, one being an array (A1), and the other having ability to set its dimension (A7). D4 matched not only an artefact setting the numeric value to a dice (A8), but also a random number generator (A9) and a number comparator (A11). Some of the requirements have no single matching refinement artefact, e.g. D3 associates with quite a few useful high affinity artefacts (A1, A3, A2 or A4), nevertheless, none of them matches the required feature completely. Others missed the artefact that we would consider as the most important, e.g. it is anticipated that D6 would have high affinity with the program termination function (A12), however, due to the a particular selection of the requirement’s characteristic terms, the affinity with A12 is unfortunately low.

Requirements:		D1	D2	D3	D4	D5	D6	D7
Design Artefacts:		The system shall allow players to specify the number of dice to "roll".	The player shall then roll dice.	Each die represents numbers from 1 to 6.	The dice are assigned their values randomly.	Every time the dice are rolled, their values are assigned in a random fashion.	If the total of both dice is even, the user shall win.	Otherwise the user looses.
A1	array	0.686	0.554	**0.868**	0.585	0.592	0.508	0.498
A2	number	0.537	0.605	0.803	0.612	0.619	0.442	0.562
A3	string	0.686	0.554	0.868	0.585	0.592	0.496	0.420
A4	command	0.641	0.610	0.694	0.617	0.619	0.469	0.717
A5	read	0.568	0.479	0.559	0.494	0.496	0.335	0.436
A6	write	0.484	0.454	0.590	0.479	0.480	0.355	0.450
A7	set dimension	**0.963**	0.554	0.680	0.585	0.592	0.417	0.385
A8	set numeric value	0.410	**0.849**	0.585	**0.868**	**0.890**	0.544	0.456
A9	random number generator	0.388	0.723	0.554	0.714	0.723	0.659	0.442
A10	sum of numbers	0.282	0.554	0.496	0.585	0.592	**0.769**	0.345
A11	compare number	0.388	0.687	0.605	0.714	0.723	0.562	0.461
A12	quit program	0.548	0.490	0.585	0.496	0.501	0.360	**0.758**
A13	you've won dialogue box	0.350	0.436	0.566	0.460	0.480	0.293	0.502
A14	you've failed dialogue box	0.350	0.436	0.566	0.460	0.480	0.289	0.508

Table 8: Requirements affinity calculation

Using the RARE IDIOM method we've identified possible refinements of requirements defined for the "Dice" game system. Table 9 summarises our findings. For each dice-game requirement it shows the two best matching design artefacts that could be used in the refinement of this requirement. Note that compared with D7, requirement D6 does not match artefact A12 (quit program). This is due to the difference in the selection of two artefact term lists (cf. Table 2) - one selects the verb as the most important term, while the other chooses the noun (contrary to the recommentations). Overall, however, the example illustrates that affinity analysis of requirements provides a good indication of refinement and design reuse opportunities.

5. Summary and Conclusions

This paper outlined a method of enhancing reuse at the requirements level. It introduced a requirements engineering process model, RARE (Reuse-Assisted Requirements Engineering), which incorporates reuse activities at its core. RARE provides a method, which combines faceted classification, similarity and affinity analyses, and a domain-mapping thesaurus. The method gives developers important refinement information, which is not available with other requirements engineering approches, but which could effectively guide them in further processing of the collected requirements, hence, leading to the design based on reusable software components. The complete example presented in this paper, although small due to the limitation of available space, demonstrates that the method is practical and feasible.

We should also mention that RARE was implemented in an experimental system IDIOM (Informal Document Interpreter, Organiser and Manager). The prototype focuses on the issue of requirements classification, refinement and reuse. At this stage of research, RARE IDIOM does not fully cover the entire requirements engineering process. In the future, however, we would like to incorporate RARE IDIOM features into an existing requirements management system or a CASE environment.

RARE IDIOM is still undergoing trials with professional and student groups in usability experiments. The experiments so far were used to identify and rectify some of the weaknesses of the tool's user interface

The Dice-Game System	Artefact 1	Artefact 2
D1. The system shall allow players to specify the number of dice to "roll".	A7. set dimension	A1. array
D2. The player shall then roll dice.	A8. set numeric value	A9. random number generator
D3. Each die represents numbers from 1 to 6.	A1. array	A2. number
D4. The dice are assigned their values randomly.	A8. set numeric value	A9. random number generator
D5. Every time the dice are rolled, their values are assigned in a random fashion.	A8. set numeric value	A9. random number generator
D6. If the total of both dice is even, the user shall win.	A10. sum of numbers	A9. random number generator
D7. Otherwise the user looses.	A12. quit program	A4. command

Table 9: Refinement opportinity for the "Dice" game system

and its workflow. We are also conducting benchmarks to compare RARE IDIOM's information retrieval capabilities against those offered by other retrieval tools. Our tests demonstrate that RARE IDIOM clearly stands out in its unique ability to match text across different domains of discourse, the feature that we regard as pivotal to the successful refinement of system and software requirements.

6. References

[1] Abbott, R.J. (1983): *Program design by informal English descriptions.* Communications of the ACM. **26**(11): p. 882-894.

[2] Aguilera, C. and D.M. Berry (1990): *The use of a repeated phrase finder in requirements extraction.* Journal of Systems and Software. **13**(3): p. 209-230.

[3] Balzer, R.M., N. Goldman, and D. Wile (1978): *Informality in program specifications.* IEEE Trans. on Software Eng. **SE**(4): p. 94-103.

[4] Bigelow, J. (1988): *Hypertext and CASE.* IEEE Software: p. 23-27.

[5] Borgida, A., S. Greenspan, and J. Mylopoulos (1985): *Knowledge representation as the basis for requirements specifications.* IEEE Computer: p. 82-90.

[6] Bruno, W.F., G. Narayanaswami, M. Aoyama, and C.K. Chang (1988): *A knowledge based system approach to the development of a system functional requirement specification processor*. in *12th Annual International Computer Software & Applications Conference*: Computer Society Press of the IEEE, p. 387-394.

[7] Castano, S. and V. De Antonellis (1994): *The F3 Reuse Environment for Requirements Engineering.* ACM SIGSOFT Software Engineering Notes. **19**(3): p. 62-65.

[8] Dankel, D.D., M.S. Schmalz, and K.S. Nielsen (1994): *Understanding natural language software specifications*. in *Fourteen International Avignon Conference, AI'94*. Paris, France: Ec-2, p. .

[9] Davis, A.M. (1993): *Software Requirements: Objects, Functions and States*. 2nd ed. Upper Saddle River, New Jersey: Prentice Hall.

[10] DoA (1992): *The Army Strategic Software Reuse Plan*, Report ODISC4, Department of the Army, Office of the Directcors of Information Systems for Command, Control, Communications and Computers: Washington, DC.

[11] Frakes, W.B. and T.P. Pole (1994): *An empirical study of representation methods for reusable software components.* IEEE Transactions on Software Engineering. **20**(8): p. 617-630.

[12] Garg, P.K. and W. Scacchi (1990): *Hypertext system to manage software life-cycle documents.* IEEE Software. **7**(3): p. 90-98.

[13] Gaudel, M.C. (1991): *Algebraic specifications*, in *Software Engineer's Reference Book*, J.A. McDermid, Editor. Butterworth-Heinemann Ltd: Oxford, U.K. p. 22/1-10.

[14] Girardi, M.R. and B. Ibrahim (1994): *A similarity measure for retrieving software artefacts*. in *6th Int. Conf. on Software Engineering and Knowledge Engineering*. Jurmala Latvia, p. 478-485.

[15] Goldin, L. and D.M. Berry (1994): *AbstFinder, a prototype abstraction finder for natural language text for use in requirement elicitation: design, methodologies, and evaluation*. in *The First International Conference on Requirements Engineering*. Colorado Springs, Colorado: IEEE Computer Society Press, p. 84-93.

[16] Greenspan, S., J. Mylopoulos, and A. Borgida (1994): *On formal requirements modeling languages: RML revisited*. in *16th International Conference on Software Engineering*. Sorrento, Italy: IEEE Computer Society Press, p. 135-147.

[17] Hayes, I.J., C.B. Jones, and J.E. Nicholls (1994): *Understanding the differences between VDM and Z.* ACM SIGSOFT Software Engineering Notes. **19**(3): p. 75-81.

[18] Hofmann, H.F. (1993): *Requirement Engineering: A Survey of Methods and Tools*, 93.05, Institut fŸr Informatik der Universitat ZŸrich.

[19] IEEE/EIA (1998): *Information Technology - Software Life Cycle Processes*, IEEE/EIA Standard 12207, IEEE.

[20] Jacobs, P.S. (1987): *Knowledge-intensive natural language generation.* Artificial Intelligence. **33**: p. 325-378.

[21] Johnson, W.L., M.S. Feather, and D.R. Harris (1991): *The KBSA requirements/specification facet: ARIES*. in *6th Annual Knowledge-Based Software Engineering Conference*. Syracuse, New York, USA: IEEE Computer Society Press, p. 48-56.

[22] Kaindl, H. (1993): *The missing link in requirements engineering.* ACM SIGSOFT Software Engineering Notes. **18**(2): p. 30-39.

[23] Kotonya, G. and I. Sommerville (1998): *Requirements Engineering: Processes and Techniques*. Chichester, England: Wiley.

[24] Meyer, B. (1985): *On formalism in specifications.* IEEE Software: p. 6-26.

[25] Mili, H., E. Ah-Ki, R. Godin, and H. Mcheick (1997): *Another nail to the coffin of faceted controlled-vocabulary component classification and retrieval.* Software Engineering Notes. **22**(3): p. 89-98.

[26] Monahan, B. and R. Shaw (1991): *Model-based specifications*, in *Software Engineer's Reference Book*, J.A. McDermid, Editor. Butterworth-Heinemann Ltd: Oxford, U.K. p. 21/1-37.

[27] Nanduri, S. and S. Rugaber (1995): *Requirements validation via automatic natural language parsing.* Journal of Management Information Systems. **12**(2): p. 9-19.

[28] Prieto-Diaz, R. (1989): *Classification of reusable modules*, in *Software Reusability: Concepts and Models*, T.J. Biggerstaff and A.J. Perlis, Editors. Addison-Wesley Pub. Co.: New York, NY. p. 99-123.

[29] Prieto-Diaz, R. and P. Freeman (1987): *Classifying software for reusability.* IEEE Software. **4**(1): p. 6-16.

[30] Puncello, P.P., P. Torrigiani, F. Pietri, R. Burlon, B. Cardile, and M. Conti (1988): *ASPIS: a knowledge-based CASE environment.* IEEE Software: p. 58-65.

[31] Ramesh, B. and V. Dhar (1992): *Supporting systems development using knowledge captured during requirements engineering.* Report to appear in IEEE Transactions on Software Engineering: p. 1-25.

[32] Salek, A., P.G. Sorenson, J.P. Tremblay, and J.M. Punshon (1994): *The REVIEW system: From formal specifications to natural language*. in *The First International Conference on Requirements Engineering*. Colorado Springs, Colorado: IEEE Computer Society Press, p. 220-229.

[33] Salton, G. (1989): *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer*. Readings, Massachusetts: Addison-Wesley Pub. Co.

[34] V-Model-97 (1997): *Development Standard for IT Systems of the Federal Republic of Germany*, General Directive 250/1, BWB IT I 5: Koblenz, Germany.

[35] Wieringa, R.J. (1996): *Requirements Engineering: Frameworks for Understanding*. Chichester, UK: John Wiley & Sons.

[36] Wilkinson, N.M. (1995): *Using CRC Cards*: SIGS.

[37] Wood, D.P., M.G. Christel, and S.M. Stevens (1994): *A multimedia approach to requirements capture and modeling*. in *The First International Conference on Requirements Engineering*. Colorado Springs, Colorado: IEEE Computer Society Press, p. 53-56.

[38] Young, M. and K. Reed (1992): *Identifying reusable components in software requirements specifications to develop a natural language-like SRS language with a CRNLP*, Technical Report TR005, Amdahl Australian Intelligent Tools Programme: Bundoora, Vic, Australia.

[39] Zarri, G.P. (1988): *Knowledge acquisition for large knowledge bases using natural language analysis techniques.* Expert Systems for Information Management. **1**(2): p. 85-109.