Sixth International Conference on Software Reuse, Vienna, Austria, June 27-29, 2000.

Version 1.0 (September 28, 1999)

Requirements Classification and Reuse: Crossing Domain Boundaries

Jacob L. Cybulski
Dept of Information Systems
The University of Melbourne
Parkville, Vic 3052, Australia
Email: j.cybulski@dis.unimelb.edu.au

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

Abstract

A serious problem in the classification of software project artefacts for reuse is the natural partitioning of classification terms into many separate domains of discourse. This problem is particularly pronounced when dealing with requirements artefacts that need to be matched with design components in the refinement process. In such a case, requirements can be described with terms drawn from a problem domain (e.g. games), whereas designs with the use of terms characteristic for the solution domain (e.g. implementation). The two domains have not only distinct terminology, but also different semantics and use of their artefacts. This paper describes a method of cross-domain classification of requirements texts with a view to facilitate their reuse and their refinement into reusable design components.

Keywords

Requirements Refinement, Reuse, Information Retrieval

1. Introduction

Reuse of development work-products in the earliest phases of software life-cycle, e.g. requirements engineering and architectural design, is recognised to be beneficial to software development [1]. In the past, claims were made that early reuse improves resource utilisation [24] and produces higher quality of software products [20]. It was also suggested that early reuse encourages software development environment that facilitates a more systematic approach to software reuse [14, 31]. Such an environment will most often be automated, leading to tool support in early phases of a project [34], hence, yielding improved reuse in the subsequent development phases [34, 43]. The objective of requirements reuse is to identify descriptions of (large-scale) systems that could either be reused in their entirety or in part, with the minimum modifications, thus, reducing the overall development effort!

Informal requirements specification are commonly used in the early phases of software development. Such documents are usually produced in natural language (such as English), and in spite of many problems in their handling, they are still regarded as one of the most important communication medium between developers and their clients [11]. The lack of formality, structure and ambiguity of natural language makes requirements documents difficult to represent and process, not to mention their effective reuse. To overcome this problem, requirements statements need to be processed in a unique fashion to accommodate reuse tasks, which include analysis of existing requirements, their organisation into a repository of reusable requirements artefacts, and their synthesis into new requirements documents. Several methods, techniques, tools and methodologies were suggested as useful in supporting these tasks.

There are three major approaches to requirements reuse (see Table 1), i.e. text processing, knowledge management and process improvement. The first approach focuses on the text of requirements, its parsing, indexing, access and navigation. Such approaches rely

Table 1: Approaches to Requirements Reuse

	Approaches	Researchers/Groups
Text	parsing specifications	Allen & Lee [3]
	natural languages processing	Naka [33], Girardi & Ibrahim [19]
	use of hypertext	Kaindl, Kaiya [21, 22]; Garg & Scacci [18]
	finding repeated phrases	Aguilera & Berry [2]
	assessment of similarity	Fugini, et al. [5, 17]
Knowledge	taxonomic representation	Wirsing, et al. [42] Johnson & Harris [20]
	logic-based specifications	Puncello, et al. [37]
	analogical reasoning	Maiden & Sutcliffe [30]
	knowledge-based systems	Lowry [26] Tamai [41], Borgida [6] Lubars & Harandi [28, 29] Zeroual [44]
	analysis patterns	Fowler [13]
	domain mapping	Simos, et al. [39] Cybulski & Reed [10]
	domain analysis	Prieto-Diaz [35] Frakes, et al. [15] Kang, et al. [23] Simos [40]
Process	reuse-based process	Kang, et al. [24]
	meta- and working models	Castano, Bubenko [7, 8]
	wide-spectrum reusability	Lubars [27]
	family of requirements	Lam [25]
	reuse-based maintenance	Basili [4]
	CASE-support of early reuse	Poulin [34]

heavily on the natural language grammars and lexicons, statistical text analysers, and hypertext. The second approach aims at elicitation, representation and use of knowledge contained in requirements documents, and reasoning about this captured knowledge. These methods commonly focus on the modelling of a problem domain, they utilise knowledge acquisition techniques and elaborate modelling methods. Sometimes they also utilise knowledge-based systems and inference engines. The last approach aims at changing development practices to embrace reuse. We believe that the most successful method of requirements reuse should address all three above-mentioned aspects of requirements handling.

Informal requirements do not impose any rigid syntax or semantics of their texts. Their form is natural and easily understood by all of the stakeholders, despite their potential complexity. Their content is very rich, however, it may also be ambiguous, imprecise and incomplete, and hence confusing. We, therefore, suggest that reuse of informal requirements texts should not overly rely on the informal documents' grammar or their semantics, which may both be very difficult to deal with. Instead we suggest that the methods applicable to informal requirements documents should focus on their lexical and structural properties. A method embracing these principles, RARE (Reuse-Assisted Requirements Engineering), was therefore proposed and subsequently implemented in a prototype tool, IDIOM (Informal Document Interpreter, Organiser and Manager). In what follows, we illustrate the RARE IDIOM approach by identifying commonalities in two simple "requirements documents" for two common games of chance. This will provide both a short description of the RARE method and will demonstrate the power of an IDIOM tool.

2. The RARE Method

The RARE method centres on the tasks of requirements recording, analysis and refinement. It is based on the commonly accepted understanding that the main purpose of requirements engineering is to prepare a high quality requirements specification document.[1] RARE's key principle is to promote use of requirements analysis techniques that lead to the discovery and addition of reuse information to the requirements documents with a view to enhance the subsequent development stages. RARE suggests a focus on three aspects of requirements analysis.

1. Identification and replacement of new requirements with those drawn from existing documents that have already been processed and refined into a collection of designs (Cf. Figure 1a).
2. Identification of requirements for the new software product that can be refined with reusable design components, created in the process of developing some other software system (Cf. Figure 1b).
3. Identification of similarities between requirements drawn from specifications in the same application domain or software system, indicating either reuse potential, conflict or redundancy. Similarity cross-referencing may contribute to the improved quality of the resulting designs and ultimately the entire software product (Cf. Figure 1c).

To address these aspects, we propose to utilise :-

1) cross-domain requirements classification to detect requirements similarity, and
2) affinity of requirements to design artefacts to support refinement of requirements.

We will explain the need for cross-domain classification

[1] Which means the document is clear, precise, unambiguous and consistent.

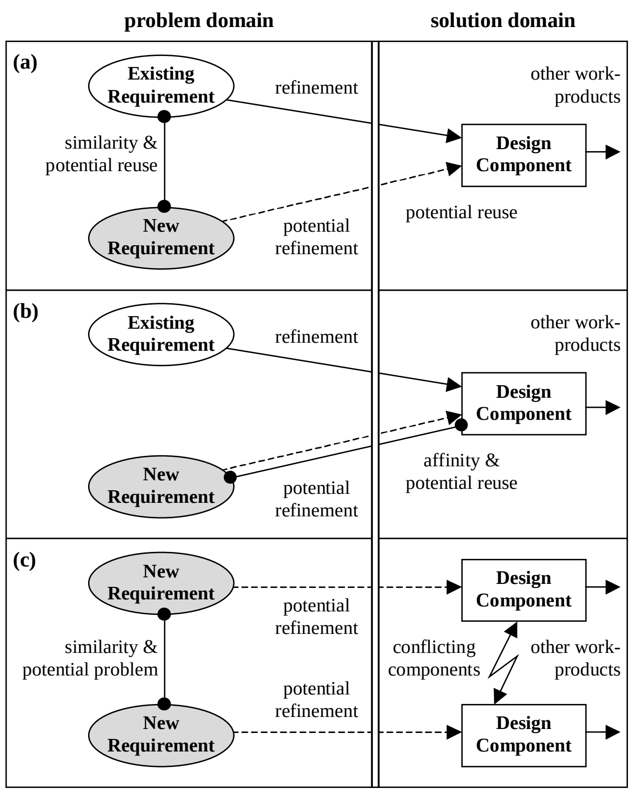

Figure 1: Identification of reuse in RARE

in the following sections.

3. Keywords or Facets

The main aim of RARE is to establish a collection of informal requirements text that could be reused from one project to another, and a collection of formal designs that could be used to refine them. Together, requirements and designs form a repository of reusable artefacts. For each requirement statement there may be a number of design artefacts perceived as useful in its refinement. IDIOM's role is to assist designers in selecting candidate artefacts that could be combined to form a single design, and in rejecting those artefacts found to be inappropriate in further refinement.

In this work, we seek 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 work, we focus on nominating lexically and syntactically interesting concepts to become members of a list of domain terms characteristic to each requirement.

Table 2 and Table 3 show a number of simple requirements statements drawn from the Dice Game (D1 to D7) and the Coin Game (C1 to C7) systems. Each statement can be characterised with a list of standardised domain terms, which were selected from their text. Some of the terms are specific to the domain of game playing, e.g. "dice" and "coin", "player", "roll" and "flip", "win" and "loose". Others are common to many application domains, e.g. "specify" and "depict", "the number of", "represent", "random", etc. The terms are ordered by their *relevance* to the meaning of their respective requirement.

Some researchers show that the terms characterising the requirements may be used to determine the *similarity* between requirements documents [5, 17, 19]. Finding the relevant classification terms is easy and the automation of the process is also straightforward [38]. In our example, the documents found to be similar to Dice and Coin games could also describe some gaming systems. The Dice Game requirements will match requirements documents that refer to "dice", "rolling", "players", "random", "winning" and "loosing". The Coin Game requirements, on the other hand, will match the documents that make statements about "coins", "heads" and "tails", "flipping", but also "players", "random", "winning" and "loosing". It is also clear that there exist similarities between Coin and Dice gaming systems as well, as they both make use of some common terms, e.g. "players", "random", "winning" and "loosing".

To facilitate such document matching, we could use many commercially available information retrieval systems. Such systems include full-text databases such as Knowledge Engineering Texpres[2] or askSam Professional[3], or text analysis programs such as Concordance[4] and SPSS TextSmart[5]. General-purpose document cataloguing systems such as dtSearch[6] and ConSearch (Readware Intelligence Warehouse)[7] can also be used very effectively. We could also utilise some of the web search engines, e.g. AltaVista Discovery[8] and Ultraseek Server[9]. Some document-classifying software are available for retrieval of both disk and web-based texts.[10]

[2] See http://www.ke.com.au/texpress/index.html.

[3] askSam Systems, askSam Professional, http://www.askSam.com/.

[4] R. J.C. Watt, Concordance, http://www.rjcw.freeserve.co.uk/.

[5] SPSS, TextSmart, http://www.spss.com/software/textsmart/.

[6] DT Software, Inc., dtSearch, http://www.dtsearch.com/.

[7] ConSearch, http://www.readware.com/.

[8] DEC AltaVista, Discovery, http://discovery.altavista.com/.

[9] Infoseek, Ultra Seek Server, http://software.infoseek.com/.

[10] For example, dtSearch and AltaVista Discovery.

Table 2: Dice-game requirements characterised with domain terms (weighed by relevance)

The Dice-Game System	Term list: Relevance:	Term 1 (relev 0.4)	Term 2 (relev 0.3)	Term 3 (relev 0.2)	Term 4 (relev 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 3: Coin-game requirements characterised with domain terms (weighed by relevance)

The Coin-Game System	Term list: Relevance:	Term 1 (relev 0.4)	Term 2 (relev 0.3)	Term 3 (relev 0.2)	Term 4 (relev 0.1)
C1. The system shall depict two coins that can be "flipped".		depict	coin	two	flip
C2. At each turn the player flips both coins.		flip	coin	player	both
C3. Each coin has two sides, the head and the tail.		coin	side	head	tail
C4. The coins are placed on their randomly selected sides.		place	coin	side	random
C5. Every time the coins are flipped, their face values are assigned in a random fashion.		assign	value	random	every
C6. If both coins have identical face values, the user shall win.		identical	face	win	if
C7. Otherwise the user looses.		loose	user	otherwise	

Use of document retrieval systems leads to a number of problems. The first problem is that such systems commonly use a document rather than its parts as a retrieval entity. In requirements reuse, where requirements documents tend to be very large, it is an individual requirement or a group of related requirement statements that are of interest to a reuser. Another problem is that a simplistic lexical matching of requirements stamements is doomed to fail, as it ignores the semantic similarity of words used in document indexing. A knowledge-based approach could be more successful here, as a large knowledge base of common-sense facts could assist in assessing the semantic similarity of related concepts. A simpler approach could rely on the use of a domain thesaurus able of cross-referencing similar terms.

Terms extracted from the requirements text of two gaming systems (see Table 2 and Table 3) clearly illustrate the potential semantic disparity between requirements documents.[11] Since both documents belong to the same domain of discourse, and both were produced according to the same requirements template, we could expect their characteristic terms to be similar across the documents, e.g. D5-C5 or D7-C7. However, some of the corresponding requirements have completely different term characterisation, e.g. D1-C1, D2-C2 or D4-C4. We could use of a thesaurus to deal with these disparities, e.g. by defining some of the terms as synonyms, e.g. "specify" and "depict", "dice" and "coin", "roll" and "flip", "assign" and "place". None of these "synonyms", however, make any sense lexically or semantically. This is because the equivalence between these concepts exist only in terms of a functional design of both games, whereby "dice" and "coin" are game's "instruments"; whereas "rolling" and "flipping" or "placing" are actions assigning a value to the "instrument".

To capture the functional aspects of requirements statements, we need to explicitly model the required system context, its function, the data manipulated and the constraining methods. The simplistic keyword-based approach to text classification may not be appropriate, in spite of its many advantages. Methods more suitable to the task of functional requirements classification are those which can identify and categorise different aspects of classified artefacts - these include the attribute-value, enumerative and faceted classification techniques [12].

[11] Such disparity in small requirements documents may not present any problems to an experienced analyst, who would immediately identify an opportunity to abstract the required system functions into a description of a more general problem. For large sets of requirements, tool-support becomes essential!

Table 4: Faceted classification of "Dice Game" requirements (with facet weights)

Dice Game Requirements	Function (weight 0.4)	Data (weight 0.3)	Method (weight 0.2)	Environ. (weight 0.1)
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 5: Faceted classification of "Coin Game" requirements (with facet weights)

Coin Game Requirements	Function (weight 0.4)	Data (weight 0.3)	Method (weight 0.2)	Environ. (weight 0.1)
C1. The system shall depict two coins that can be "flipped".	output	value	direct	user
C2. At each turn the player flips both coins.	assign	value	random	instrument
C3. Each coin has two sides, the head and the tail.	any	value	any	instrument
C4. The coins are placed on their randomly selected sides.	any	value	random	instrument
C5. Every time the coins are flipped, their face values are assigned in a random fashion.	assign	value	direct	any
C6. If both coins have identical face values, the user shall win.	end	boolean	choice	success
C7. Otherwise the user looses.	end	boolean	choice	failure

Table 6: Faceted classification of design artefacts (with facet weights)

Design Artefacts	Function (weight 0.4)	Data (weight 0.3)	Method (weight 0.2)	Environ. (weight 0.1)
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

Although these methods provide similar usability and effectiveness [16], we believe that the faceted classification is 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.

Table 4 and Table 5 show the faceted classification of requirements drawn from the Dice and Coin game examples. The classification scheme uses four facets, i.e.

Table 7: Domain-mapping thesaurus terms (with sense strengths)

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				

function, data, method and environment, which all define terms from a design domain. The example clearly illustrates that comparing requirements in terms of their functional facets values (from the design domain) could be more effective than by matching the keywords found in the body of these requirements (from the problem domain). According to the faceted classification, the requirements that were previously found to be dissimilar, i.e. D2-C2 and D4-C4, can now be determined to be very much alike. As faceted classification also allows fuzzy-matching of terms, so that requirements statements can also be compared for their semantic distance, even if their classifying terms are not identical, e.g. D1 with C1.

The advantage of the functionally based faceted classification is that requirements can be compared regardless of terminology used in their expression. The classifying terms can be drawn from the solution domain of discourse. This means that requirements can be classified in the same manner as design artefacts (see Table 6 on the previous page) and hence can be compared against them to facilitate the process of requirements refinement. By visual inspection,[12] we can see that faceted descriptors of requirements D2 and D4 match those of artefacts A8 and A9. Since requirements and designs are significantly different in their form and nature

[12] A critical reader may wish to calculate the similarity of these descriptors using the formulae defined in section 5. See also Cybulski and Reed [10] for the details of calculating affinities between requirements and designs.

their similarity is rooted in the design process, hence, we will refer to it as their *affinity* (or tendency to combine).

One of the disadvantages of this approach lies in the difficulty of constructing requirements descriptors, which are no longer based on the body of their text. Another, a harder problem, is in the allocation of facet values to the requirement descriptor, which may necessitate making certain design decisions, and is hence going beyond a simple classification process. Both of these issues imply the need for the manual classification of requirements by a skilled analyst or a designer, thus, incurring significant labour costs and demanding considerable time to complete the task - the problems commonly associated with faceted classification in general [32]. It should be noted, however, that simply changing the classification scheme would not eliminate our problems, which find their source in the cross-domain nature of the classification process!

4. Domain Mapping

To resolve our cross-domain classification problems, 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 designed as a repository of problem-domain concept descriptors *sensing* (associating or linking) facet values drawn from the solution domain. Table 7 (see previos 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 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", etc. 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.

The domain-mapping thesaurus is used to translate problem domain keywords into solution domain facets. Consider a single requirement for the dice game system (see Table 2), 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, which include "specify", "the number of", "dice" and "player". Each of these terms senses a few facet values from the solution domain (cf. Table 7), 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 a number of factors stored and available to the domain-mapping process, i.e.

- ♦ the relevance of terms extracted from the body of requirements text to the classification of this requirement (w_i); and,
- ♦ the strength of each term's sense ($s_{i,j}$) as defined by the domain-mapping thesaurus.

The two factors can be combined together to calculate the *facet value priority* for the classification which is calculated according to the following formula [9]:

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

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
$s_{i,j}$ - j-th sense strength of facet j in term i
$p_{i,j}$ - priority of a facet value $f_{i,j}$ for a facet j in term i
α - importance of the relevance factor (0.9)
β - importance of the sense strength (0.1), $\alpha + \beta = 1$.

The priorities are then used to rank all possible classifications of each requirement (see Table 8). 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

Table 8: Resolution of requirements classification terms (ranked by facet value priority)

Requirements (with selected keywords)	Function		Data		Method		Environment	
1. The system shall allow players to specify the number of dice to "roll". (specify, the number of, dice, player)	*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. (roll, dice, player, then)	*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. (represent, dice, number, each)	*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, value, dice, random)	*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, value, random, every)	*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. (total, even, win, if)	*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. (loose, user, otherwise)	*end* interact	0.40 0.31	*boolean*	0.20	*choice*	0.26	*failure* user	0.42 0.33

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.

5. Requirements Similarity

So far, we have developed all instruments necessary to compare requirements with design artefacts. We are now able to characterise requirements in terms of their problem domain attributes, to automatically translate these terms into a solution domain, and to compare such requirements using a faceted classification method. For completeness reasons we provide the reader with details of our faceted classification (a working model of the RARE method is also available as an Excel spreadsheet that can be obtained from the web).[13]

Our classification method defines four facets (for this small example), i.e. function (Figure 2), data (Figure 3), method (Figure 4) and environment (Figure 5). Each facet defines a collection of classification terms or values, and also defines a *conceptual distance* measure between facet values. A small conceptual distance value indicates the facet values to be very close, conversely, large distance represents the two facet values to be far apart.

During the development of the facet structure, facet values are represented as an associative network, or a connected weighted graph, of inter-related concepts. In this model, the measure of closeness between two concepts can be defined as an *associative distance*, i.e. the weight attached to each network link. Any two concepts *x* and *y* in a facet *f* would have an associative distance of $A_f(x, y) \in [0, 1]$, and $A_f(x,x) = 0$. For the concepts that are not directly associated, the length of a path leading from one concept to another would determine their proximity [36]. In such a case, the distance between two distant facet values could be defined as the length of the shortest path between the nodes in the facet value network, i.e.

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

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

where:

$D_f(x, y)$ - distance between "x" and "y" in facet "f"
$d_f(x, y)$ - normalised distance between "x" and "y" in "f"
$A_f(x, y)$ - associative distance between "x" and "y" in "f"
F_f - a set of "x_i" values in facet "f"

Inability to reach one facet value from another would be represented by the length of the longest possible path in a facet network.

[13] http://www.dis.unimelb.edu.au/staff/jacob/seminars/DiceGame.xls.

	Coin Requirements (New):	**C1**	**C2**	**C3**	**C4**	**C5**	**C6**	**C7**
	Dice Requirements (Reused):	The system shall depict two coins that can be "flipped".	At each turn the player flips both coins.	Each coin has two sides, the head and the tail.	The coins are placed on their randomly selected sides.	Every time the coins are flipped, their face values are assigned in a random fashion.	If both coins have identical face values, the user shall win.	Otherwise the user looses.
D1	The system shall allow players to specify the number of dice to "roll".	**0.592**	0.559	0.885	0.752	0.592	0.379	0.379
D2	The player shall then roll dice.	**0.506**	**1.000**	**1.000**	**1.000**	**0.927**	0.460	0.460
D3	Each die represents numbers from 1 to 6.	**0.605**	0.574	**1.000**	0.781	0.607	0.585	0.585
D4	The dice are assigned their values randomly.	0.512	**0.927**	**1.000**	**0.927**	**1.000**	0.475	0.475
D5	Every time the dice are rolled, their values are assigned in a random fashion.	0.513	**0.927**	**1.000**	**0.927**	**1.000**	0.496	0.496
D6	If the total of both dice is even, the user shall win.	0.355	0.537	0.817	0.715	0.592	0.305	0.301
D7	Otherwise the user looses.	0.484	0.460	0.817	0.766	0.496	**0.927**	**1.000**

Table 9: Requirements affinity and opportunity to reuse

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 2).

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 2: Conceptual distance matrix for the "function" facet

To counter the facet size differences, this distance is also normalised to the interval of *[0, 1]* by dividing the conceptual distance between facet values by the maximum path length within the facet.

Requirement descriptors are used as the basis for the construction of artefact vectors and queries. The match between a query and an artefact is defined in terms of their similarity/affinity, given as a weighted geometric distance metric, i.e.

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

where:

dist(q, a) - normalised distance between "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 "a" and "b" in "I"

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 3: "Data" facet

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 4: "Method" facet

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 5: "Environment" facet

Consider the two sets of user requirements for the previously discussed Dice game and a Coin game (see Table 9 on the previous page). Assume that dice-game requirements have already been refined and implemented, hence they have all been already classified (see Table 2 and Table 4). The coin-game requirements define a new system to be implemented (see Table 3), these requirements are mapped into design facets using a domain-mapping thesaurus (see Table 5). During the coin-game requirements affinity analysis, the new requirements are matched against all stored artefacts, which include both design and requirements artefacts. Knowing that the two sets of requirements match almost line-by-line (C1-D1, C2-D2, etc.), we would expect the best matches and identification of reuse opportunities to fall along the matrix diagonal. So there they are (see Table 9) - clear opportunity to reuse!

Requirement C3 does not have a clear match with any of the D1-D7, it matches the majority of artefacts in the repository (dashed line in Table 9). Since both its "function" and "method" facets are undefined (see Table 5), they result in small conceptual distances from every other facet value, hence, leading to the high affinity with every design artefact. This vagueness results from our selection of terms characteristic for requirements C3 ("side", "head", "tail"), which are not in the thesaurus. Such situations are easy to detect and may need manual correction (by defining a new domain term of by rephrasing the requirement). Similar problems may also occur due to the non-functional nature of a requirement, which may lead to certain facets being unfilled (also C3).

Another interesting phenomenon can be observed by studying the similarity of requirements D2, D4 and D5 vs. all other coin-game requirements. The results indicate that descriptors of these requirements overlap (or requirements are redundant). Detection of such cases can be achieved in a simple way, i.e. by comparing requirements of a single document one with another and determining their relative similarity (we will not conduct this analysis here due to the shortage of space).

Note also that due to the classification of D6, which emphasises its calculation aspect rather than game's exit condition, as is the case with D7, C6 unexpectedly matches D7. Requirements C6 and D7 both refer to the end of a game, though with different consequences for the player. These cases of "misclassification" could be dealt with by allowing multiple classification vectors per each requirement, in which case D6 could be classified with the use of two vectors, one to cover its calculation aspects and another its exit condition, thus improving the match!

6. Summary and Conclusions

Reuse of software requirements lead to the effective reuse of all software work-products derived from these requirements downstream the development process. Requirements reuse can, therefore, provide significant gains in developmental productivity and in the quality of the resulting software product.

In software development, the semantics of requirements can be either found in the knowledge of a domain or in the designs derived from these requirements. Either of these approaches could be used to determine requirements similarity. In this paper, we proposed a method of requirements classification that takes advantage of design-based semantics for requirements.

Our approach suggests combining keyword-based and faceted classifications of requirements and designs. The keywords are (efficiently) extracted from the body of requirements text, hence they represent the requirements characterisation in the problem domain. With the use of a domain-mapping thesaurus, keywords are then translated into design terms of a faceted classification. Facets are subsequently used to determine affinity between requirements and design artefacts, which can be used as a basis for assessing requirements similarity and for reuse-based refinement of requirements documents.

Cross-domain classification is an integral part of the RARE method of requirements engineering, proposed by the authors, and supported by IDIOM, a prototypic software tool. Our experimental studies (to be reported elsewhere) show that IDIOM offers some superiority over simple document/text classification and retrieval software, such as web search engines, which have been recently promoted by other researchers as suitable to facilitate software reuse. We have also conducted a number of RARE IDIOM useability experiments, which have drawn our attention to the features required of IDIOM, should the tool be considered for further commercial exploitation.

Overall, we are satisfied that RARE IDIOM classification method addresses the "boundary" problem of requirements reuse. The problem that cannot be dealt with by simply adopting or changing existing reuse methods. Domain-mapping is a new technique that could complement other approaches to classification and retrieval of requirements texts, and hence, enhancing their reusability.

7. References

1. Agresti, W.W. and F.E. McGarry (1988): "The Minnowbrook Workshop on Software Reuse: A summary report", in *Software Reuse: Emerging Technology*, W. Tracz (Editor). Computer Society Press: Washington, D.C. p. 33-40.
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. Allen, B.P. and S.D. Lee (1989): "A knowledge-based environment for the development of software parts composition systems". in *11th International Conference on Software Engineering*. Pittsburgh, Pennsylvania: IEEE Computer Society Press, p. 104-112.
4. Basili, V.R. (1990): "Viewing maintenance as reuse-oriented software development". *IEEE Software*: p. 19-25.
5. Bellinzona, R., M.G. Fugini, and B. Pernici (1995): "Reusing specifications in OO applications". *IEEE Software*. **12**(2): p. 65-75.
6. Borgida, A., S. Greenspan, and J. Mylopoulos (1985): "Knowledge representation as the basis for requirements specifications". *IEEE Computer*: p. 82-90.
7. Bubenko, J., C. Rolland, P. Loucopoulos, and V. DeAntonellis (1994): "Facilitating "Fuzzy to Formal" requirements modelling". in *The First International Conference on Requirements Engineering*. Colorado Springs, Colorado: IEEE Computer Society Press, p. 154-157.
8. Castano, S. and V. De Antonellis (1994): "The F3 Reuse Environment for Requirements Engineering". *ACM SIGSOFT Software Engineering Notes*. **19**(3): p. 62-65.
9. Cybulski, J. (1999): *Application of Software Reuse Methods to Requirements Elicitation from Informal Requirements Texts*, PhD Thesis Draft, La Trobe University: Bundoora.
10. Cybulski, J. and K. Reed (1999): "Automating Requirements Refinement with Cross-Domain Requirements Classification". in *Australian Conference on Requirements Engineering ACRE'99*. Sydney: Macquarie University.
11. Davis, A.M. (1998): "Predictions and farewells". *IEEE Software*. **15**(4): p. 6-9.
12. DoD (1995): *Software Reuse Initiative: Technology Roadmap, V2.2*, Report http://sw-eng.falls-church.va.us/reuseic/policy/Roadmap/Cover.html, Department of Defense.
13. Fowler, M. (1997): *Analysis Patterns: Reusable Object Models*. Menlo Park, California: Addison-Wesley.
14. Frakes, W. and S. Isoda (1994): "Sucess factors of systematic reuse". *IEEE Software*. **11**(5): p. 15-19.
15. Frakes, W., R. Prieto-Diaz, and C. Fox (1998): "DARE: domain analysis and reuse environment". *Annals of Software Engineering*. **5**: p. 125-141.
16. 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.
17. Fugini, M.G. and S. Faustle (1993): "Retrieval of reusable components in a development information system", in *Advances in Software Reuse: Selected Papers from the Second International Workshop on Software Reusability*, P.-D. Ruben and B.F. William (Editors). IEEE Computer Society Press: Los Alamitos, California. p. 89-98.
18. Garg, P.K. and W. Scacchi (1990): "Hypertext system to manage software life-cycle documents". *IEEE Software*. **7**(3): p. 90-98.

19. Girardi, M.R. and B. Ibrahim (1993): "A software reuse system based on natural language specifications". in *5th Int. Conf. on Computing and Information*. Sudbury, Ontario, Canada, p. 507-511.

20. Johnson, W.L. and D.R. Harris (1991): "Sharing and reuse of requirements knowledge". in *6th Annual Knowledge-Based Software Engineering Conference*. Syracuse, New York, USA: IEEE Computer Society Press, p. 57-66.

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

22. Kaiya, H., M. Saeki, and K. Ochimizu (1995): "Design of a hyper media tool to support requirements elicitation meetings". in *Seventh International Workshop on Computer-Aided Software Engineering*. Toronto, Ontario, Canada: IEEE Computer Society Press, Los Alamitos, California, p. 250-259.

23. Kang, K., S. Cohen, J. Hess, W. Novak, and S. Peterson (1990): *Feature-Oriented Domain Analysis (FODA) Feasibility Study*, Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie-Mello University.

24. Kang, K.C., S. Cohen, R. Holibaugh, J. Perry, and A.S. Peterson (1992): *A Reuse-Based Software Development Methodology*, Technical Report CMU/SEI-92-SR-4, Software Engineering Institute.

25. Lam, W. (1998): "A case study of requirements reuse through product families". *Annals of Software Engineering*. **5**: p. 253-277.

26. Lowry, M. and R. Duran (1989): "Knowledge-based software engineering", in *The Handbook of Artificial Intelligence*, A. Barr, P.R. Cohen, and E.A. Feigenbaum (Editors). Addison-Wesley Publishing Company, Inc.: Readings, Massachusetts. p. 241-322.

27. Lubars, M.D. (1988): "Wide-spectrum support for software reusability", in *Software Reuse: Emerging Technology*, W. Tracz (Editor). Computer Society Press: Washington, D.C. p. 275-281.

28. Lubars, M.D. (1991): "The ROSE-2 strategies for supporting high-level software design reuse", in *Automatic Software Design*, M.R. Lowry and R.D. McCartney (Editors). AAAI Press / The MIT Press: Menlo Park, California. p. 93-118.

29. Lubars, M.D. and M.T. Harandi (1989): "Addressing software reuse through knowledge-based design", in *Software Reusability: Concepts and Models*, T.J. Biggerstaff and A.J. Perlis (Editors). ACM Addison Wesley Publishing Company: New York, New York. p. 345-377.

30. Maiden, N. and A. Sutcliffe (1991): "Analogical matching for specification reuse". in *6th Annual Knowledge-Based Software Engineering Conference*. Syracuse, New York, USA: IEEE Computer Society Press, p. 108-116.

31. Matsumoto, Y. (1989): "Some experiences in promoting reusable software: presentation in higher abstract levels", in *Software Reusability: Concepts and Models*, T.J. Biggerstaff and A.J. Perlis (Editors). ACM Addison Wesley Publishing Company: New York, New York. p. 157-185.

32. 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.

33. Naka, T. (1987): "Pseudo Japanese specification tool". *Faset*. **1**: p. 29-32.

34. Poulin, J. (1993): "Integrated support for software reuse in computer-aided software engineering (CASE)". *ACM SIGSOFT Software Engineering Notes*. **18**(4): p. 75-82.

35. Prieto-Diaz, R. (1988): "Domain analysis for reusability", in *Software Reuse: Emerging Technology*, W. Tracz (Editor). IEEE Computer Society Press. p. 347-353.

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

37. 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.

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

39. Simos, M. (1995): "WISR 7 Working Group Report: Domain Model Representations Strategies: Towards a Comparative Framework". Andersen Center, St. Charles, Illinois, p. http://www.umcs.maine.edu/~ftp/wisr/wisr7/dawg-nps/dawg-nps.html.

40. Simos, M.A. (1991): "The growing of organon: a hybrid knowledge-based technology and methodology for software reuse", in *Domain Analysis and Software Systems Modeling*, R. Prieto-Diaz and G. Arango (Editors). IEEE Computer Society Press: Los Alamitos, California. p. 204-221.

41. Tamai, T. (1989): "Applying the knowledge engineering approach to software development", in *Japanese Perspectives in Software Engineering*, Y. Matsumoto and Y. Ohno (Editors). Addison-Wesley Publishing Company: Singapore. p. 207-227.

42. Wirsing, M., R. Hennicker, and R. Stabl (1989): "MENU - an example for the systematic reuse of specifications". in *2nd European Software Engineering Conference*. Coventry, England: Springer-Verlag, p. 20-41.

43. Yglesias, K.P. (1993): "Information reuse parallels software reuse". *IBM Systems Journal*. **32**(4): p. 615-620.

44. Zeroual, K. (1991): "KBRAS: a knowledge-based requirements acquisition system". in *6th Annual Knowledge-Based Software Engineering Conference*. Syracuse, New York, USA: IEEE Computer Society Press, p. 38-47.