A New Way to Classify and Retrieve Reusable Components: the Finder Metaphor

José Luís Barros Justo [jbarros@uvigo.es]
Antonio Domínguez Iglesias [antonio@sig.uvigo.es]
LSI Dept., University of Vigo
Pontevedra, 36002. Spain

ABSTRACT

Our research group ISRI (Information Systems Reuse on Internet) is working in a new classification scheme for reusable software (assets) based on the idea of hyper-spherical finders. The Finder metaphor deals with a multidimensional space where the components are positioned according to its functional description, and vision fields, hiper-spheres, which can move in space and *see* the components. Our approach has some advantages over other techniques and methodologies, for example: we start with an empty space (the universe) and automatically fill it with reusable assets, which describe the application domain, but the space itself is dynamically generated as components are inserted; furthermore, we can build different finders, alter its radius or its kind of movement, without the need of reclassifying the repository, allowing a dynamical classification. We developed and tested a simple but practical method to compute similarities, allowing to locate assets in this kind of universe, generating finders and modifying them, querying the universe and retrieving similar components. The tested values of recall and precision were similar or better than other known methods but ours is by far easier to implement and maintain. An extensive list of references about reuse and classification/retrieval problem, where interested readers can investigate more deeply, is offered.

Keywords: reusable components, repository organization, classification, retrieval.

INTRODUCTION

Many papers were made about reuse, reusable components and repositories organization . All these works underline the problems related with human intervention, huge amount of information, classification and retrieval. Perhaps the main drawback of actual solutions lies in the need of human intervention, automatic support is considered crucial but none of the approach offers it. So we address this problem considering the following two challenges:

1. Automatic generation of the classification space
2. A powerfull but easy to implement retrieval technique

Information Retrieval (IR) techniques claim that they can automatically extract information about components (often textual documents) and classify them, but assets from the software development life cycle hardly ever resemble text documents in style or content . Knowledge-based systems (KBS) need a previous (human-dependent) acquisition phase in order to classify components. So we need to address this problem from another perspective, with techniques that allow the automatic inclusion of assets in the repository, automatic generation of a classification space, easy processes to reclassify or reorganize the repository, adaptable searching mechanisms, and so on.

THE FINDER METAPHOR

A finder is a multidimensional sphere with a prefixed radius, which can contain components. To be capable of doing this the finder needs to be mobile, their movement across the space allows different collections of components to be captured and grouped according to their similarities. A finder has a center that represents the average similarity of all the components which are seen by it, the first time this center is located at the same position of the first component but, as new components are added to the space and were captured by the finder, the center moves according to attraction rules. This dynamical behaviour of the finder guarantees its representability of common attributes of components that belong to it. When a finder center moves across the space it can capture other existing components, we call that absortion, and can leave others, leaving, causing new movements. This process iteratively repeats itself until no new components are captured or left.

Representation of components

We propose a representation technique similar to those in [18] [21]. In our method each component is represented by a functional description (FD) consisting in a set of features which in turn were made of triplets: <action-object, importance> where:

- action reminds us the functionality of the component (what it does)
- object means where the *action* applies
- importance gives an idea of to what extent this couple *action-object* represents the component with respect to the global set of features, in other words, the relative importance of this *action-object* within its FD.

We can set an importance scale made of five values: VL (Very Low), L (Low), M (Medium), H (High) and VH (Very High). In this way we have a balanced scale with two values in each arm and a middle one as an average. In order to compute similarities based in this scale we use a mapping function between letters and importance values:

VL = 1/16 L = 1/8 M = ¼ H = ½ VH = 1

which really means fuzzy values for this attribute (importance).

A sample FD with three features could be the following:

FD:
open-file,L
sort-file, VH
close-file,L

Thus, our repository will be filled with this kind of *components*, functional descriptions as triplets *<action-object,importance>*. We will have meta-information about the component, for example: author, where the component is really located, language, adaptation guidelines and so on.

Component classification
Our classification scheme is based in a positional approach, components are located in a multidimensional vector space according to its functionality. In fact, we do not work with real components, instead we use a description of the components (FD), actually a functional description in a simple representation language, with restricted vocabulary, and the help of a thesaurus, allowing an easy management of sinonyms and homonyms. So, we can easily modify a description without the need of manipulating the real component, which remains untouched. Building functional descriptions is a labor of the repository administrator, who is also responsible for components insertion, modification and/or deletion. When a new component is to be inserted in the repository its functional description is compared with functional descriptions of other special elements named reference components, and a similarity value is computed for each reference component, then we assign the following position to the new inserted component:

$$C = (sim(C,R_1), sim(C,R_2), ..., sim(C,R_n))$$

Where n is the number of reference components.
Let's suppose we need to know how similar two components C_1 and C_2 are, then we will need to build the following matrices:

1. the EQ (Equivalence) matrix expresses the degree of compatibility between the i-th feature of C_1 and the j-th feature of C_2. So EQ will be an $f\text{-}C_1$ x $f\text{-}C_2$ matrix. If *<action-object$_i$>* = <action-object$_j$> then EQ[i,j]=1, 0 otherwise.
2. the IMP (IMPortance) matrix shows the degree of satisfaction that FD of C_1 is compatible (or can be replaced) with the FD of C_2. It shows the importance between the i-th feature of C_1 and j-th of C_2, remember that EQ do not consider *importance* just <action-object>. This *importance* is computed as *min*(1, *importance of* $f\text{-}C_2$ / *importance of* $f\text{-}C_1$). If EQ[i,j]=0 then IMP[j,i]=0, because features are not comparables. IMP is $f\text{-}C_2$ x $f\text{-}C_1$.
3. the SAT (SATisfaction) matrix combines EQ and IMP, and is computed as EQ x IMP.
4. the I (Importance) matrix holds the normalized values of importance of C_1.
5. the SIM (SIMilarity) matrix, finally, is the product SAT x I. Then, entry SIM[k] represents a weighted satisfaction index for feature k of C_1 with respect to C_2.

A working example:

feature number	C_1	C_2
1	<top-stack, M>	<size-array,M>
2	<push-element,VH>	<size-queue,L>
3		<top-stack,VL>

According to the table above, we will have:

EQ = [2x3] (C1 x C2)
IMP = [3x2] (C2 x C1)
I = [2x1] (C1 x 1)
SAT = [2x2] (EQ x IMP → 2x3 X 3x2)

$$EQ = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}$$

SIM = [2x1] (SAT x I → 2x2 X 2x1)

IMP[3,1] = min (1, VL/M) = min (1, (1/16)/(1/4)) = min (1, ¼) = 0,25

$$IMP = \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0{,}25 & 0 \end{pmatrix}$$

I[1] = M / (M + VH) = (1/4) / (5/4) = 1/5 = 0,2
I[2] = VH / (M + VH) = 1 / (5/4) = 4/5 = 0,8

$$I = \begin{pmatrix} 0{,}2 \\ 0{,}8 \end{pmatrix}$$

SAT = EQ x IMP, so SAT[1,1] =0,25 and 0 all others

$$SAT = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix} X \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0.25 & 0 \end{pmatrix} = \begin{pmatrix} 0.25 & 0 \\ 0 & 0 \end{pmatrix}$$

Finally, SIM = SAT x IN, so

$$SIM = \begin{pmatrix} 0,25 & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} 0,2 \\ 0,8 \end{pmatrix} = \begin{pmatrix} 0,05 \\ 0 \end{pmatrix}$$

The similarity between C_1 and C_2 is therefore 0.05, and it is obtained summing up all the elements of SIM.

Component insertion

Once we have the position of C then we need to assign it to an existing finder or create a new one. The process is very simple:

- compute distance from C to centers of existing finders (distance is defined as |1-sim|)
- if there exist centers which distance is less that a prefixed threshold then C will belong to the nearest center, else a new finder is created with center coincident with C.
- if C was assigned to an existing finder then, move this finder center
- while this movement implies *capturing* or *leaving* components move it again

However, the algorithm is a little bit more complicated. Suppose a threshold value *U* and the following variables:

1. *dim* n° of dimensions = n° of reference components (*initial value = 0*)
2. *numvis* n° of finders (*initial value = 0*)
3. *n* n° of components in the repository (*initial value = 0*)
4. *lcv* list of components assigned to a finder (*store the value *n**)
5. *lv* list of finders (*list of *lcv**)

then, the algorithm to insert components will be:

```
1.      $n = n + 1$; verify if $C_n$ features are already
        reference components.
        1.1.    No, then (*build reference components*)
                1.1.1.  for $i = 1$ to numcar do
                        1.1.1.1. dim = dim+1; $R_{dim} = c_{ni}$
2.      for $i = 1$ to dim do:
        2.1.    compute $sim(C_n, R_i)$, as explained before
3.      $C_n$ = ($sim(C_n, R_1)$, $sim(C_n, R_2)$,...., $sim(C_n, R_{dim})$)
4.      for $i = 1$ to numvis do:
        4.1.    compute distance from $C_n$ to
                $v_{numvis}$ (*the center of $V_{numvis}$)
        4.2.    if $|1 - sim(C_n, cv_{numvis})| \leq U$ then
                possible[numvis] := 1-sim
        4.3.    else possible[numvis:=0
5.      if not empty possible, then
        5.1.    sort possible in ascending order
        5.2.    choose the finder closest to $C_n$.
6.      else: (* build a new finder*)
        6.1.    $numvis = numvis + 1$
        6.2.    $cv_{numvis} = C_n$
        6.3.    new lcv (*lcv number numvis*)
7.      add $n$ to lcv (*assing $C_n$ to $V_{numvis}$ finder*)
8.      verify if $V_{numvis}$ can "see" another
        components, if it can, then:
        8.1.    for each component C seen by $V_{numvis}$ do:
                8.1.1.  assing C to $V_{numvis}$
                        (*update lcv *)
                8.1.2.  move ($V_{numvis}$)
9.      end
```

Component retrieval

Retrieval of components follows a simple way, we treat a query as a functional description, then we try to *insert* this new component as if it were a normal component. The process is exactly the same as insertion but, when a finder is selected to cover this component instead of inserting it the tool retrieves all the components that belong to that finder. If no finder was selected in the previous process then the tool issues a warning message: "No similar components were found, please try another query".

CONCLUSIONS

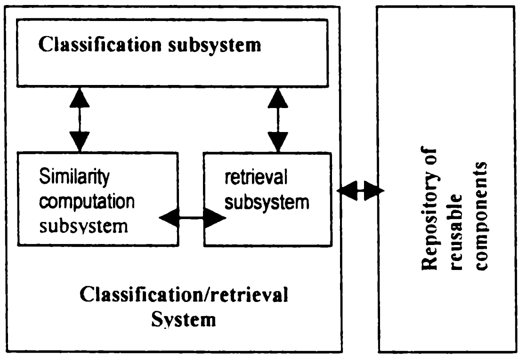
Recall and precision values tested were similar or better than other known methods . One key aspect of our approach is the easy implementation of a prototype, this means that algorithms to insert components, automatic generation of reference components (space), computation of similarities and creation of finders were very simple. Movement of finders were, perhaps, the most difficult part of the algorithm, due to the consideration of multiple situations which can happen, such as capturing of new components or leaving of existing ones. Another key functionality of our approach is the capability of modifying the finders width, the repository administrator can adjust the threshold value (distance from finder center to farthest components) to generate a new repository configuration, he or she can do that to guarantee a good retrieval behaviour, to obtain more recall the radius is enlarged, to get more precision is shortened.

When making queries to the repository, the reuser has to construct a query as a FD. We believe that it would be quite easy for the reuser to make a query since the FDs are simple and easy to construct, and the reuser need not know exact figures to insert as importances. The simple mapping of the importances into abbreviations such as H (for high) or L (low) to represent functionalities of various features within

FDs would be easier than to express queries in terms of reuse metrics, where sometimes, more exactness is required in the absence of a simple and non-complicated mapping system.

In our scheme the repository is organized automatically, and places no constraints in the event that more components are added to the repository. This is important since, software development is a dynamic process which requires many changes in the systems developed throughout the software lifecycle.

A definite advantage with our classification schemes is that they are implemented apart from the similarity computation method, that is, there is very low coupling between these two subsystems in their implementation. This implies that one could easily replace the current similarity computation method with another similarity computation method, and in this case, then our classification scheme would still function with minor modifications required to the classification subsystem, see figure below.


Classification subsystem
Similarity computation subsystem
retrieval subsystem
Classification/retrieval System
Repository of reusable components

FURTHER ACTIVITIES

More research is needed in knowledge representation techniques to guarantee an easy comprehension by the reuser. These techniques must allow automatic extraction from diverse information sources such as experts, documentation, source code, other repositories, and so on. We also need to combine present search tools such as spiders and intelligent agents, user profiles, visual querying languages, hypertext navigation, search histories and user responses (feedback). The fact that many distributed repositories can be interconnected also implies the need for a common interface and links between different abstraction levels of components.

REFERENCES

[1] Abran, A., Maya, M., Measurement of Functional Reuse, WISR8, Ohio State University, Columbus, Ohio, USA, March 23-26, 1997

[2] Agafonov, V.N., Reuse of General Specification Notions and Specification Languages, WISR8, Ohio State University, Columbus, Ohio, USA, March 23-26, 1997

[3] M. D'Alessandro, P. Iachini, and A. Martelli. The generic reusable component: an approach to reuse hierarchical OO designs. In Proceedings of the 2nd International Workshop on Software Reusability, March 24-26, Lucca, Italy, pages 39--46, 1993.

[4] Allemang, D., Liver, B., Functional Representation for Reusable Components, WISR7, Andersen Consulting Center, St. Charles, Illinois, USA, August, 1995

[5] Atkinson, S., A Unifying Model for Retrieval from Reusable Software Libraries, TR95-41, CS Dept., University of Queensland, Australia, December, 1995

[6] Atkinson, S., Examining Behavioural Retrieval, WISR8, Ohio State University, Columbus, Ohio, USA, March 23-26, 1997

[7] S. Arnold and S. Stepoway. The REUSE system: Cataloging and retrieval of reusable software. In Proceedings of COMPCON '87, pages 376--379, 1987.

[8] Börstler, J., Feature-Oriented Classification for Software Reuse, Proc. of SEKE'95, Rockville, MD, USA, pp.: 204-211., June 22-24, 1995

[9] Browne, S., Dongarra, J., Hohn, K., Niesen, T., Software Repository Interoperability, TR UT-CS-96-329, July, 1996

[10] Browne, S.V., Moore, J.W., Reuse Library Interoperability and the World Wide Web, SSR97, January 2, 1997

[11] Burton, B.A., Aragón, R.W., Bailey, S.A.,Koehler, K.D., Mayes, L.A., The Reusable Software Library, IEEE Software, Vol. 4, N° 4. pp.: 25-33, July, 1987

[12] Cheng, B.H.C., Jeng, J-J., Reusing Analogous Components, Technical Report MSU-CPS-94-28. April 1994 (revised June 1995), June, 1995

[13] W.G. Cho, Y.W. Kim, and J.H. Kim. CLIS: A software reuse library system with a knowledge based information retrieval model. In Proceedings of the Pacific Rim International Conference on AI, pages 402--407, 1990.

[14] Ciaccia, P., Patella, M., Zezula, P.. Processing Complex Similarity Queries with Distance-based Access Methods, Sept. 15, 1997

[15] Cybulski, J.L., Neal, R.D., Kram, A., Allen, J.C., Reuse of Early Life-Cycle Artefacts: Workproducts, Methods and Tools, Annals of Software Engineering, Vol 5., 1998

[16] Damiani, E., Fugini, M.G., Automatic Thesaurus Construction Supporting Fuzzy Retrieval of Reusable Components, Proc. of the 1995 ACM Symp. on Applied Computing, pp. 542-547, 26-28 Feb. 1995

[17] Damiani, E., Fugini, M.G., Bellettini, C., A Hierarchy-Aware Approach to Faceted Classification of Object-Oriented Libraries, Politecnico di Milano, Dipartimento di Elettronica e Informazione. Raporto interno 001-97, Gennaio, 1997

[18] S. Faustle and M.G. Fugini. Retrieval of reusable components using functional similarity. Technical Report ITHACA.POLIMI.E.6.92, University of Pavia, 1992.

[19] Finnigan, P.J., Holt, R.C., Kalas, I., Kerr, S., Kontogiannis, K., et all., The Software Bookshelf, IBM Systems Journal, Vol. 36, N° 4, 1997

[20] Frakes, W.B., Pole, T. P., An Empirical Study of Representation Methods for Reusable Software Components, EEE Trans. on Software Engineering, Vol. 20, N° 8. pp.: 617-630, August, 1994

[21] M. Fugini and S. Faustle. Retrieval of reusable components in a development information system. In Proceedings of 3rd International Workshop on Software reusability IWSR-3, pages 89--98, 1993

[22] Henninger, S., Supporting the Construction and Evolution of Component Repositories, Proc. of Int. Conf. On Software Engineering, Berlin, FRG, March, 1996

[23] Henninger, S., An Evolutionary Approach to Constructing Effective Software Reuse Repositories, ACM Trans. On Software Engineering and Methodology, Vol. 6, N° 2, pp.: 111-140, April, 1997

[24] Isakowitz, T., Kauffman, R.J., Supporting Search for Reusable Software Objects, IEEE TSE, Vol. 22, N° 6, pp. 407-423, June, 1996

[25] Jeng, J-J., Gheng, B.H.C., Using Formal Methods to Construct a Software Component Library, Lecture Notes in Computer Science, Vol. 717, pp.: 397-417, Sept. 1993

[26] Jilani, L.L., Desharnais, J., Frappier, M., Mili, R., Mili, A., Retrieving Software Components That Minimize Adaptation Effort, Automated Software Engineering, ASE'97, 12th IEEE Intl. Conf., Nevada, USA., November 1-5, 1997

[27] Jilani, L.L., Mili, R., Mili, A., Approximate Component Retrieval: An Academic Exercise or a Practical Concern?, WISR8, Ohio State University, Columbus, Ohio, USA, March 23-26, 1997

[28] Jurisica, I., A Similarity-Based Retrieval Tool for Software Repositories, 3rd Workshop on AI and Software Engineering, IJCAI-95, Montreal, Quebec, Canada, April 20-21, 1995

[29] Jurisica, I., Glasgow, J., Case-Based Classification Using Similarity-Based Retrieval, 8th IEEE Intl. Conf. On Tools with AI. Toulouse, France., November 16-19, 1996

[30] Jurisica, I., Similarity-Based retrieval for Diverse Bookshelf Software Repository Users, IBM CASCON Conference, Toronto, Canada, November 10-13, 1997

[31] Lim, W.C., Applying Cluster Analysis to Software Reuse, WISR7, Andersen Consulting Center, St. Charles, Illinois, USA, August, 1995

[32] Maarek, Y.S., Berry, D.M., Kaiser, G.E., An Information Retrieval Approach For Automatically Constructing Software Libraries, EEE Trans. on Software Engineering, Vol. 17, N° 8, pp.: 800-813, August, 1991

[33] Meyer, B., Lessons from the Design of the Eiffel Libraries, Communications of the ACM, Vol. 33, N° 9, pp.:68-88, Sept., 1990

[34] Mili, R., Mili, A., Mittermeir, R.T., Storing and Retrieving Software Components: A Refinement Based System, EEE Trans. on Software Engineering, Vol. 23, N° 7, pp.: 445-460, July, 1997

[35] Nato Communications and Information Systems Agency, NATO Standard for Management of a Reusable Software Component Library, 1991

[36] Noll, J., Scacchi, W., Integrating Diverse Information Repositories: A Distributed Hypertext Approach, IEEE Computer, Vol. 24, N° 12, pp.: 38-45, December, 1991

[37] E. Ostertag, J. Hendler, R. Prieto-Diaz, and C. Braun. Computing similarity in a reuse library system: an AI-based approach. ACM Transactions on Software Engineering and Methodology, pages 205--228, July 1992.

[38] Pathak, P., A Simulation Model of Document Information Retrieval System with Relevance Feedback, http://www.umich.edu/~praveen/research/irmodel.html

[39] Poulin, J.S., Iglesias, K.P., Experiences with a Faceted Classification Scheme in a Large Reusable Software Library (RSL), Proc. of COMPSAC'93, Phoenix-AZ, pp. 90-99., Nov. 3-5, 1993

[40] Poulin, J.S., Werkman, K.J., Software Reuse Libraries with Mosaic, Proc. of 2nd World Wide Web Conference'94: Mosaic and the Web, 1994

[41] Prieto-Diaz, R., Implementing Faceted Classification for Software Reuse, Communications of the ACM, Vol. 34, N° 5, pp.: 88-97, May, 1991

[42] Ribeiro, A.N., Martins, F.M., A Fuzzy Query Language for a Software Reuse Environment, WISR7, Andersen Consulting Center, St. Charles, Illinois, USA, August, 1995

[43] RIG: Reuse Library Interoperability Group, Standard Reuse Library Basic Interoperability Data Model (BIDM), IEEE Standards Department, January, 1995

[44] G. Salton, J. Allan, and C. Buckley. Automatic structuring and retrieval of large text files. Communications of the ACM, 37(2), February 1994.

[45] G. Sindre, E. Karlsson, and T. Staalhane. Organizing large libraries of reusable components: the REBOOT approach. Journal of Software Engineering and Knowledge Engineering, April 1992.

[46] G. Sindre, E. Karlsson, and T. Staalhane. A method for software reuse through large component libraries. In Proceedings of the International Conference on Computing and Information, pages 464--468, 1993.

[47] Sindre, G., Sorumgärd, S., Terminology Evolution in Component Libraries, Proc. of Terminology and Knowledge Engineering (TKE'93), Cologne-Germany., August 25-27, 1993

[48] Sivert, L., Sindre, G., Stokke, F., Experiences in Reusable Components Classification, Proc. of 2nd International Workshop on Software Reuse (IWSR-2), Lucca-Italy, March 24-26, 1993

[49] Spanoudakis, G., Constantopoulos, P., Similarity for Analogical Software Reuse: A Computational Model, ECAI'94. Edited by A. Cohn, Published by John Wiley & Sons, Ltd., 1994

[50] Spanoudakis, G., Constantopoulos, P., Measuring Similarity Between Software Artifacts, SEKE '94, Jurmala, Latvia, pp. 387-394, June 1994

[51] Terry, R.H., Price, M., Welton, L., Standardized Software Classification in the World Wide Web, WISR7, Andersen Consulting Center, St. Charles, Illinois, USA, August, 1995

[52] Veerasamy, A., Hudson, S., Navathe, S., Visual Interface for Textual Information Retrieval Systems, Proc. of the Third IFIP 2.6 Working Conference on Visual Database Systems, 1995