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

José Luis 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. Hyper-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 dynamic 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 powerful 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 absorption, 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:

\[
\begin{align*}
\text{VL} &= 1/16 \\
\text{L} &= 1/8 \\
\text{M} &= 1/4 \\
\text{H} &= 1/2 \\
\text{VH} &= 1
\end{align*}
\]

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

A sample FD with three features could be the following:

\[
\begin{align*}
\text{FD}: \\
\text{open-file,L} \\
\text{sort-file,VH} \\
\text{close-file,L}
\end{align*}
\]

Thus, our repository will be filled with this kind of components, functional descriptions as triplets \(<\text{action-object,importance}>\). 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 synonyms 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:

\[
C = (\text{sim}(C,R_1), \text{sim}(C,R_2), \ldots, \text{sim}(C,R_n))
\]

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 \(f(C_1) \times f(C_2)\) matrix, if \(<\text{action-object,}> = 1\) 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 \(<\text{action-object}>\). This importance is computed as \(\text{min}(1, \text{importance of } f(C_1) / \text{importance of } f(C_2))\). If \(\text{EQ}[i,j]=0\) then \(\text{IMP}[i,j]=0\), because features are not comparables. IMP is \(f(C_1) \times f(C_2)\).
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:

<table>
<thead>
<tr>
<th>feature number</th>
<th>(C_1)</th>
<th>(C_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(&lt;\text{top-stack,M}&gt;)</td>
<td>(&lt;\text{size-array,M}&gt;)</td>
</tr>
<tr>
<td>2</td>
<td>(&lt;\text{push-element,VH}&gt;)</td>
<td>(&lt;\text{size-queue,L}&gt;)</td>
</tr>
<tr>
<td>3</td>
<td>(&lt;\text{top-stack,VL}&gt;)</td>
<td></td>
</tr>
</tbody>
</table>

According to the table above, we will have:

\[
\begin{align*}
\text{EQ} &= [2\times 3] \\
\text{IMP} &= [3\times 2] \\
\text{SAT} &= [2\times 2] \\
\text{SIM} &= [2\times 1]
\end{align*}
\]

\[
\text{IMP}[3,1] = \text{min}(1, \text{VL/M}) = \text{min}(1, (1/16)/(1/4)) = \text{min}(1, 1/4) = 0.25
\]

\[
\text{IMP} = \begin{bmatrix}
0 & 0 \\
0 & 0
\end{bmatrix}
\]

\[
\text{IMP} = \begin{bmatrix}
0.25 & 0 \\
0 & 0.25
\end{bmatrix}
\]

\[
I &= \begin{bmatrix}
0.2 \\
0.8
\end{bmatrix}
\]

\[
\text{SAT} = \text{EQ} \times \text{IMP}, \text{ so } \text{SAT}[1,1] = 0.25 \text{ and 0 all others}
\]
\[ SAT = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0.25 & 0 \end{pmatrix} X \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0.25 & 0 \end{pmatrix} = \begin{pmatrix} 0.25 & 0 \\ 0 & 0 \end{pmatrix} \]

Finally, \( SIM = SAT \times IN. \) 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. \( \text{dim} \) \( n \) of dimensions = \( n \) of reference components (*initial value = 0*)
2. \( \text{numvis} \) \( n \) of finders (*initial value = 0*)
3. \( n \) \( n \) of components in the repository (*initial value = 0*)
4. \( \text{lev} \) list of components assigned to a finder (*store the value \( n \)*)
5. \( \text{lev} \) list of finders (*list of \( \text{lev} \)*)

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 \( \text{numcar} \) do
   1.1.1.1. \( \text{dim} = \text{dim} + 1; \)
   1.1.1.2. \( \text{R}_{\text{dim}} = c_w \)
   2. for \( i = 1 \) to \( \text{dim} \) do:
   2.1. compute \( \text{sim}(C_n, R_i) \) as explained before
   3. \( C_w = (\text{sim}(C_n, R_1), \text{sim}(C_n, R_2), \ldots, \text{sim}(C_n, R_{\text{lev}})) \)
   4. for \( i = 1 \) to \( \text{numvis} \) do:
   4.1. compute distance from \( C_w \) to \( V_{\text{numvis}} \) (*the center of \( V_{\text{numvis}} *\)
   4.2. if \( |1 - \text{sim}(C_w, C_{V_{\text{numvis}}})| \leq U \) then
   4.3. else possible(\( \text{numvis} \)) = 0
   5. if not empty possible, then
   5.1. sort possible in ascending order
   5.2. choose the finder closest to \( C_w \)
   6. else (*build a new finder*):
   6.1. \( \text{numvis} = \text{numvis} + 1 \)
   6.2. \( C_{V_{\text{numvis}}} = \text{C}_w \)
   6.3. \( \text{lev} = \text{lev} + 1 \) (*lev number \( \text{numvis} *\)
   7. add \( n \) to \( \text{lev} \) (*assing \( C_w \) to \( V_{\text{numvis}} \).
   8. verify if \( V_{\text{numvis}} \), can "see" another components, if it can, then:
   8.1. for each component \( C \) seen by \( V_{\text{numvis}} \) do:
   8.1.1. assing \( C \) to \( V_{\text{numvis}} \)
   8.1.2. move \( (V_{\text{numvis}}) \)
   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 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 scheme diagram]

**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


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


