Case Retrieval of Software Designs using WordNet

Paulo Gomes[1], Francisco C. Pereira[1], Paulo Paiva[1], Nuno Seco[1], Paulo Carreiro[1], José L. Ferreira[1] and Carlos Bento[1]

Abstract. Software design is one of the most important phases in system development, due to crucial decisions that are made during this phase. The need for software being developed in less time puts a lot of pressure in the design phase. One way to solve this problem is to reuse previous design solutions. In software design reuse the retrieval of relevant designs is a key issue.

Case-Based Reasoning reuses past experiences to solve new problems, providing a reasoning framework for design reuse. But designing software involves reasoning at a more abstract level than coding software, thus a software design reuse tool must be able to work with a broad range of abstract concepts. A possible solution is the use of a common sense ontology, capable of providing this kind of knowledge, otherwise the system would have to demand a lot of knowledge from the designer.

This paper presents an approach to software design retrieval based on Case-Based Reasoning combined with a common sense ontology – WordNet. We describe the case retrieval algorithm, the case similarity metrics and experimental results.

1 THE PROBLEM

As one of the main software development phases, system design has been gaining more importance as the complexity level of software increases. This also drives software development teams to be more efficient. Software designers must find new ways to design software, trying to optimise development time, processing time, required memory, and other resources. Like architects, software designers frequently use their experience from the development of previous systems to design new ones. Most of the mature engineering fields make the reuse of components a development rule, but in software engineering the reuse of components and/or design ideas is not easy, given the conceptual complexity of software. Thus, intelligent tools that support the software design task need to be at the disposal of software designers. These tools must implement software reuse [1, 2] techniques, but they also have to go further, providing support for more complex reasoning abilities.

Most of the software reuse tools [3-5] support only the retrieval of software components (like classes, functions or specifications) from repositories. But reusing software involves also reusing successful past designs. This is not what commonly happens in software development, since it is a more complex and demanding task, usually there is only code reuse. Decisions made at the design level are more important than the coding decisions, which can only influence the implementation. This is another reason to reinforce the need for an intelligent software design tool. But several obstacles appear in the construction of such tools, for example, the design communication language is usually too abstract and informal to be computationally formalized. Thus an intelligent tool capable of working at design level must also be capable of using a language used by human designers.

There are several research works that explore retrieval and similarity mechanisms, for example González et. al. [6] presented a CBR approach to software reuse and design at the code level. The work developed is based on the reuse and design of object-oriented code. Using the object description they use two retrieval algorithms, a lexical retrieval using a natural language query, and a conceptual retrieval using an entity and slot similarity measures. Déjà vu [7] is a CBR system for code reuse and generation using hierarchical CBR. Like the case representation of González, Déjà Vu uses a hierarchical case representation, indexing cases using functional features. Althoff and Tautz [8] have a different approach to software reuse and design. Instead of reusing code, they reuse system requirements and associated software development knowledge. The RSL [9] is a software design system that allows the reuse of code and design knowledge. Component retrieval can be done using a natural-language query, or using attribute search. Component ranking is an interactive and iterative process between RSL and the user. Prieto-Díaz [3] approach to code reuse is based on a faceted classification of software components. Conceptual graphs are used to organize facets, and a conceptual closeness measure is used to compute similarity between facets. Borgo [10] uses WordNet as a linguistic ontology for retrieval of object oriented components. His system uses a graph structure to represent both the query and the components in memory. The retrieval mechanism uses a graph matching algorithm returning the identifiers of all components whose description is subsumed by the query.

2 OUR APPROACH

Reusing design knowledge is a form of using experience, which corresponds to the Case-Based Reasoning (CBR) definition [11-13]. In CBR, experiences are called cases and are stored in a case library for later use. These cases are indexed so that they are retrieved when relevant for a new problem or situation. There is a clear parallel with CBR and software reuse, which lead us to choose CBR as the main rationale for the reasoning paradigm in our approach to software reuse and design.

Being able to explore the huge design space for Object-Oriented software is not an easy task. A software design system capable of coping with this exploration task, must be able to

[1] CISUC – Centro de Informática e Sistemas da Universidade de Coimbra. Departamento de Engenharia Informática, Polo II – Universidade de Coimbra. 3030 Coimbra. Portugal.

handle a huge amount of different domain knowledge. There are two main solutions for this problem: the system is capable of acquiring this knowledge; or the system uses a big enough common sense knowledge base. The first approach demands that a lot of knowledge engineering work must be done, parallel with the normal functioning of the system. Besides this limitation, there is also the problem of the system needing knowledge that is not in the knowledge base. The second approach has also limitations, such as that is very hard to codify all the common sense knowledge in the world. But it as a positive aspect, it does not need a knowledge engineering effort to update the knowledge base. This is a main problem for a software design system, because it would be necessary at least a knowledge engineer to make the system work properly, or it would be needed a lot of strict knowledge base updating rules, that would have to be followed by the designers. Having into account these arguments, we selected to use the common sense knowledge base approach. There are some common sense ontologies built by the science community, from which we selected WordNet [14].

We are developing a CASE (Computer Aided Software Engineering) tool, REBUILDER, which uses CBR to reuse and design object oriented software. Our goal is to develop an intelligent system based on a repository of past designs and a general ontology, capable of supporting software design. We represent software models in Unified Modelling Language (UML) [15] providing the user with a intuitive and commonly used design language. UML is a graphical language used to describe and document object oriented software, and is a standard for most of the software development companies.

In the next section we present the architecture of REBUILDER, describing its modules. Then we focus in the case retrieval module, describing the retrieval algorithm, the object similarity metrics, and how the WordNet knowledge is used. Then we present experimental work and results observed with our system. Finally we present some conclusions.

3 REBUILDER

REBUILDER is more than a CASE tool in the sense that it can provide intelligent design support for the software developer. Used as an UML modelling system, REBUILDER can suggest relevant designs to the user, generate new design solutions, verify and evaluate designs, and learn new knowledge.

3.1 Architecture

REBUILDER comprises several modules: an UML Editor, the Knowledge Base Manager, the Knowledge Base (KB), and the CBR Engine.

The UML Editor is the interface for the software designer, and from her/his point of view is a normal UML Editor with special functionalities. The KB administrator has another interface to interact with the system – the KB Manager module. It allows the manipulation of the system's knowledge.

The KB comprises four different types of knowledge: cases, case indexes, data types relations, and WordNet knowledge. Cases are stored in the case library and are retrieved using case indexes. Cases represent software designs. The data type taxonomy is a hierarchy relating the data types used in REBUILDER (e.g. int, String, boolean, …), this taxonomy is used to compute the conceptual distance between two data types. WordNet is a lexical reference system used in REBUILDER as an common sense ontology for object categorization [14].

The CBR Engine is responsible for all the system's reasoning, and has five modules: retrieval, analogy, adaptation, verification, and learning. The retrieval module suggests cases similar to the query design. The retrieval is based on WordNet categorization and structural similarity. But retrieval does not modify the selected designs to adapt them to the query design. This is done by two modules: adaptation and analogy. The adaptation module uses software engineering methods (design patterns) and design composition to generate new solutions. Analogy is also used to generate new designs based on object mapping between the retrieved design and the query design. The verification module checks the UML design consistence and coherence, and it also evaluates designs. REBUILDER can learn new knowledge from the user interaction, or from generated designs. In the remainder of this paper we focus in the retrieval module.

3.2 Case Representation

Cases are represented as UML class diagrams [15], which represent the software design structure. Class diagrams can comprise three types of objects (packages, classes, and interfaces) and four kind of relations between them (associations, generalizations, realizations and dependencies). Class diagrams are very intuitive, and are a visual way of communication between software development members. A simple class diagram example is presented on Figure 1.

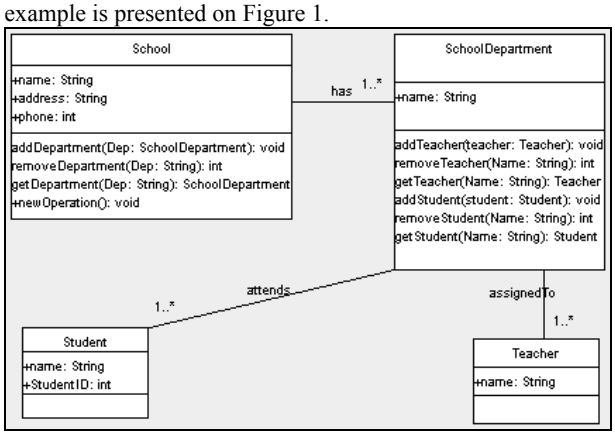

Figure 1. An example of an UML class diagram.

4 CASE RETRIEVAL

The case retrieval module retrieves three types of objects: packages (can be an entire design case or a case component), classes or interfaces, depending on the object selected when the retrieval command is selected by the user. In the next subsection we describe some basic notions about WordNet, which are needed to understand the retrieval mechanism. Then we explain our retrieval algorithm and similarity measures.

4.1 WordNet Ontology

WordNet is a common sense ontology that defines lexical concepts based on the notion of synset. A synset is defined as a

set of synonym words expressing the same concept. This implies that several words can be used to express the same synset (synonyms) and a word can have several meanings or synsets (polysemy).

WordNet comprises several semantic relations, which are relations between synsets. REBUILDER uses two types of relations: hyponyms (is-a), and meronyms (part-of, element-of or substance-of). Each object has a specific meaning corresponding to a specific synset, which we call context synset. The object's name is used to find its context synset, along with the other objects in the same class diagram. The object's class diagram is the context in which the object is referenced, so we use it to disambiguate the meaning of the object. To illustrate this, suppose that an object named *board* is created, this object can mean either a piece of lumber or a group of people assembled for some purpose. This name has two possible synsets, one for each meaning. But suppose that there are other objects in the same diagram, such as *boardMember* and *company*. These objects can be used to select the right synset for *board*, which is the one corresponding to the group of people.

4.2 Retrieval Algorithm

The retrieval algorithm is the same for all three types of objects (packages, classes and interfaces), and is based on the object classification using WordNet. Suppose that the *N* best objects are to be retrieved, *QObj* is the query object, and *ObjectList* is the universe of objects that can be retrieved (usually *ObjectList* comprises all the library cases). The algorithm is:

```
ObjsFound ← ∅
PSynset ← Get context synset of QObj
PSynsets ← {PSynset}
ObjsExplored ← ∅
WHILE (#ObsFound < N) AND (PSynsets ≠ ∅) DO
  Synset ← Remove first element of PSynsets
  ObjsExplored ← ObjsExplored + Synset
  SubSynsets ← Get Synset hyponyms (subordinates)
  SuperSynsets ← Get Synset hypernyms (super ordinates)
  SubSynsets ← SubSynsets – ObjsExplored – PSynsets
  SuperSynsets ← SuperSynsets – ObjsExplored – PSynsets
  PSynsets ← Add SubSynsets to the end of PSynsets
  PSynsets ← Add SuperSynsets to the end of PSynsets
  Objects ← Get all objects indexed by Synset
  Objects ← Objects ∩ ObjectList
  ObjsFound ← ObjsFound ∪ Objects
ENDWHILE
ObjsFound ← Rank ObjsFound by similarity
RETURN Select the first N elements from ObjsFound
```

Object retrieval has two distinct phases. First the WordNet *is-a* relations are used as an index structure to find relevant objects. Then a similarity metric is used to select only the best *N* objects. This process is a compromise between a first phase, which is inexpensive from the computational point of view, and a second phase more demanding of computational resources, but much more accurate in the object selection and ranking.

In the first phase the object's context synset works like an index. Starting by *QObj* context synset, the algorithm searches for objects indexed with the same synset. If there are not enough objects, the algorithm uses the hypernyms and hyponyms of this synset to look for objects, going in a spreading activation kind of algorithm. When it has found enough objects, it stops and ranks them using the similarity metric. In the next subsection we present the similarity metrics.

4.3 Similarity Metric

There are three similarity metrics: for classes, for interfaces, and for packages, the next subsections describe them.

4.3.1 Class Similarity

The class similarity metric is based on three components: categorization similarity, inter-class similarity, and intra-class similarity. The similarity between class C_1 and C_2, is:

$$S(C_1, C_2) = \begin{bmatrix} \omega_1 \cdot S(S_1, S_2) + \\ \omega_2 \cdot S(Ie_1, Ie_2) + \\ \omega_3 \cdot S(Ia_1, Ia_2) \end{bmatrix} \quad (1)$$

Where $S(S_1, S_2)$ is the categorization similarity computed as the distance, in *is-a* relations, between C_1 context synset (S_1) and C_2 context synset (S_2). $S(Ie_1, Ie_2)$ is the inter-class similarity based on the similarity between the diagram relations of C_1 and C_2. $S(Ia_1, Ia_2)$ is the intra-class similarity based on the similarity between attributes and methods of C_1 and C_2. w_1, w_2 and w_3 are constants. We use 0.6, 0.1, and 0.3 as the default values of these constants, based on experimental work.

4.3.2 Interface Similarity

The interface similarity metric is the same as the class metric with the difference that the intra-class similarity metric is only based on the method similarity, since interfaces do not have attributes.

4.3.3 Package Similarity

The package similarity between packages PK1 and PK2 is:

$$S(Pk_1, Pk_2) = \begin{bmatrix} \omega_1 \cdot S(SPs_1, SPs_2) + \omega_2 \cdot S(OBs_1, OBs_2) \\ + \omega_3 \cdot S(T_1, T_2) + \omega_4 \cdot S(D_1, D_2) \end{bmatrix} \quad (2)$$

This metric is based on four items: sub-package list similarity – $S(SPs_1, SPs_2)$, UML class diagram similarity – $S(OBs_1, OBs_2)$, type similarity – $S(T_1, T_2)$, and dependency list similarity – $S(D_1, D_2)$. These four items are combined in a weighted sum. The categorization similarity is the same as in the class similarity metric.

4.3.4 Object Categorization Similarity

The object type similarity is computed using the context synsets of the objects. The similarity between synset S_1 and S_2 is:

$$S(S_1, S_2) = \frac{1}{\ln(Min\{\forall Path(S_1, S_2)\} + 1) + 1} \quad (3)$$

Where *Min* is the function returning the smaller element of a list. $Path(S_1, S_2)$ is the WordNet path between synset S_1 and S_2, which returns the number of relations between the synsets.

4.3.5 Inter-Class Similarity

The inter-class similarity between two objects (classes or interfaces) is based on the matching of the relations in which both objects are involved. The similarity between objects O_1 and O_2 is:

$$S(O_1,O_2) = 2 \cdot \left(\begin{array}{l} \omega_1 \cdot \left(\frac{\sum_{i=1}^{n} S(R_{1i},R_{2i})}{n} \right) \\ -\omega_2 \cdot \frac{Unmatched(R_1)}{\#R_1} \\ -\omega_3 \cdot \frac{Unmatched(R_2)}{\#R_2} + \omega_2 + \omega_3 \end{array} \right) - 1 \quad (4)$$

Where R_i is the set of relations in object *i*, R_{ij} is the *j* element of R_i, *n* is the number of matched relations, $S(R_{1i}, R_{2i})$ is the relation similarity, and ω_1, ω_2 and ω_3 are constants, with $\sum\omega_i = 1$ (default values are: 0.5; 0.4; 0.1).

4.3.6 Intra-Class Similarity

The intra-class similarity between objects O_1 and O_2 is:

$$S(O_1,O_2) = \omega_1 \cdot S(As_1,As_2) + \omega_2 \cdot S(Ms_1,Ms_2) \quad (5)$$

Where $S(As_1, As_2)$ is the similarity between attributes, $S(Ms_1, Ms_2)$ is the similarity between methods, and ω_1 and ω_2 are constants, with $\sum\omega_i = 1$ (default values are: 0.6; 0.4).

4.3.7 Sub-Package List Similarity

The similarity between sub package lists SPs_1 and SPs_2 is:

$$S(SPs_1,SPs_2) = 2 \cdot \left(\begin{array}{l} \omega_1 \cdot \left(\frac{\sum_{i=1}^{n} S(SPs_{1i},SPs_{2i})}{n} \right) \\ -\omega_2 \cdot \frac{Unmatched(SPs_1)}{\#SPs_1} \\ -\omega_3 \cdot \frac{Unmatched(SPs_2)}{\#SPs_2} + \omega_2 + \omega_3 \end{array} \right) - 1 \quad (6)$$

Where ω_i are constants, and $\sum\omega_i = 1$ (default values are: 0.5; 0.4; 0.1), *n* is the number of sub packages matched, $S(Pk_{1i}, Pk_{2i})$ is the similarity between packages, Unmatched(Pk_i) is the number of unmatched packages in Pk_i, SPs_{ij} is the *j* element of SPs_i, and $\#Pk_i$ is the number of packages in Pk_i.

4.3.8 UML Class Diagram Similarity

The similarity between lists of UML objects OBs_1 and OBs_2 is:

$$S(OBs_1,OBs_2) = 2 \cdot \left(\begin{array}{l} \omega_1 \cdot \left(\frac{\sum_{i=1}^{n} S(OB_{1i},OB_{2i})}{n} \right) \\ -\omega_2 \cdot \frac{Unmatched(OBs_1)}{\#OBs_1} \\ -\omega_3 \cdot \frac{Unmatched(OBs_2)}{\#OBs_2} + \omega_2 + \omega_3 \end{array} \right) - 1 \quad (7)$$

Where ω_i are constants, and $\sum\omega_i = 1$ (default values are: 0.5; 0.4; 0.1), $\#OBs_i$ is the number of objects in OBs_i, Unmatched(OBs_i) is the number of objects unmapped in OBs_i, *n* is the number of objects matched, OB_{ij} is the *j* element of OB_i, and $S(OB_{1i},OB_{2i})$ is the object categorization similarity.

4.3.9 Dependency List Similarity

Dependencies is a UML relation type, and is commonly used to describe dependencies between packages. The similarity between dependency lists D_1 and D_2 is given by:

$$S(D_1,D_2) = \left[\begin{array}{l} \omega_1 \cdot \frac{\left| \#ID_1 - \#ID_2 \right|}{Max\{\#ID_1,\#ID_2\}} \\ +\omega_2 \cdot \frac{\left| \#OD_1 - \#OD_2 \right|}{Max\{\#OD_1,\#OD_2\}} \end{array} \right] \quad (8)$$

Where ω_1 and ω_2 are constants, and $\sum\omega_i = 1$ (default values are: 0.5; 0.5), ID are the input dependencies, and OD are the output dependencies.

5 EXPERIMENTS

This section describes the experimental tests developed to study the retrieval module of REBUILDER.

5.1 Experiments Design

The Knowledge Base used comprises a case library, a set of query problems, WordNet synsets, hypernyms, meronyms, and the data type taxonomy. From WordNet we use the noun synsets (78158) and semantic relations (97887).

The case library comprises 60 cases. Each case comprises a package, with 5 to 20 objects (the total number of objects is 586). Each object has up to 20 attributes, and up to 20 methods.

Three sets of package problems were specified, based on the cases. Incomplete set P20, with 25 problems, each problem is a case copy with 20% of its objects deleted, attributes and methods are also reduced by 20%. The other sets are the same problems but with 50% (P50) and 80% (P80) deleted.

The experimental runs had the goal to define the best weight configuration for the package retrieval, and also to compare categorization similarity with structural similarity and both. For each problem a best case is defined and a set of relevant cases were also defined before the runs were executed. These sets of cases are used to evaluate the accuracy of the algorithm. For each problem set the following weight configurations was used:

	w1	w2	w3	w4
Configuration 1 (C1)	0	1	0	0
Configuration 2 (C2)	0	0.75	0.25	0
Configuration 3 (C3)	0	0.5	0.5	0
Configuration 4 (C4)	0	0.25	0.75	0
Configuration 5 (C5)	0	0	1	0

Because each case has only one package, weights w1 and w4 are not used, leaving only weights w2 and w3 which concern class diagram similarity and package type similarity. For each problem run the best 20 retrieved cases were analysed. The data gathered was: best case is first (yes or no), best case is selected (yes or no), and percentage of the relevant cases retrieved.

5.2 Experimental Results

Table 1 shows the average values (in percentage) for the data gathered in the experimental runs of problem sets P20, P50 and

P80. As can be seen configurations C2 , C3 and C4 are identical, having a better performance then C1 or C5. C1 is more accurate than C5 in selection of the best case, but it is worst in the percentage of relevant cases retrieved. While C1 uses only class diagram similarity, thus being more precise in case similarity evaluation, C5 uses only type similarity which performs better in selecting cases within the same category.

	C1	C2	C3	C4	C5
Best Case First	81.33	85.33	88.00	88.00	61.33
Best Case Selected	93.33	93.33	93.33	93.33	93.33
Percentage of Relevant Cases	82.03	84.15	84.75	85.12	83.95

Table 1 - Experimental values gathered on the 75 problem runs.

The data in Figure 2 relates to the percentage of Best Case First by configuration and problem set. As it can be seen there is a clear trade-off between configurations and the size of the query. The increase in the weight of the type similarity improves the results until it stabilizes in C3 and C4. With an increase of accuracy when the size of the query also increases, which was expected. The use of only the type similarity decreases a lot the retrieval results.

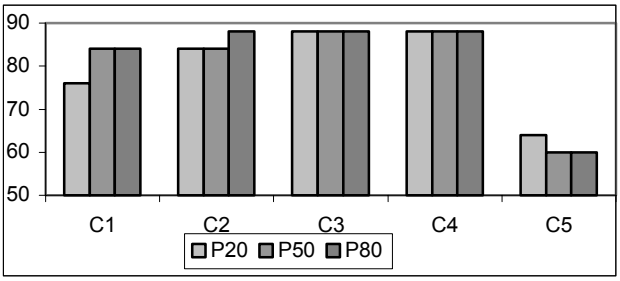

Figure 2 – Experimental values for the % of Best Case in First place.

Future work on experimentation will evaluate our approach with respect to its usefulness from the user view point, we intent to use the Nick et. al. [16] approach.

6 CONCLUSIONS

In this paper we present REBUILDER a CASE tool that uses a CBR framework with a general ontology to retrieve past UML models. Using a human-created design language like UML, allows REBUILDER to be a general software design tool. Also with the intent of being an easy to use tool, REBUILDER integrates a general ontology (WordNet), so that the designer can be understood by the machine, instead of having to explain himself to it.

Most of the CBR tools for software reuse and design are for code reuse, which is not the aim of REBUILDER. By working in software development at the design level, REBUILDER deals with more abstract and human-like issues. Though code reuse is also important, it is already a well explored area using the CBR framework. The only presented tool that uses CBR is also at a different development cycle, which is the software specification level. From the tools that do not use CBR, RSL and Borgo are the most similar tools to REBUILDER. While RSL uses a specific domain ontology, Borgo and REBUILDER use WordNet. The advantage of using WordNet is removing the knowledge engineering work from the user or from a system administrator. It also provides a much more wider coverage of domains, making the system domain independent. Using general ontologies has its limitations also, sometimes there is a lack of more specific or technical knowledge, but several mechanisms (like using machine learning) can overcome this limitation. The retrieval approach presented by Borgo is similar to the one presented here. But, while Borgo uses only the categorization similarity and structural similarity between software designs, REBUILDER uses intra-object similarity and package-structure similarity, allowing a more accurate retrieval accuracy, as seen by the experimental results.

ACKNOWLEDGEMENTS

This work was partially supported by POSI - Programa Operacional Sociedade de Informação of Portuguese Fundação para a Ciência e Tecnologia and European Union FEDER, under contract POSI/33399/SRI/2000, by program PRAXIS XXI.

REFERENCES

1. Meyer, B., *Reusability: The Case for Object-Oriented Design.* IEEE Software, 1987. **4**(2, March 1987): p. 50-64.
2. Coulange, B., *Software Reuse*. 1997, London: Springer-Verlag.
3. Prieto-Diaz, R., *Implementing Faceted Classification for Software Reuse.* Communications of the ACM, 1991(May).
4. Katalagarianos, P. and Y. Vassiliou, *On the reuse of software: a case-based approach employing a repository.* Automated Software Engineering, 1995. **2**: p. 55-86.
5. Fernández-Chamizo, C., *et al. Supporting Object Reuse through Case-Based Reasoning*. in *Third European Workshop on Case-Based Reasoning (EWCBR'96)*. 1996. Lausanne, Suisse: Springer-Verlag.
6. González, P.A. and C. Fernández. *A Knowledge-Based Approach to Support Software Reuse in Object-oriented Libraries*. in *9th International Conference on Software Engineering and Knowledge Engineering, SEKE'97*. 1997. Madrid, Spain: Knowledge Systems Institute, Illinois.
7. Smyth, B. and P. Cunningham. *Déjà Vu: A Hierarchical Case-Based Reasoning System for Software Design*. in *10th European Conference on Artificial Intelligence (ECAI'92)*. 1992. Vienna, Austria: John Wiley & Sons.
8. Tautz, C. and K.-D. Althoff. *Using Case-Based Reasoning for Reusing Software Knowledge*. in *International Conference on Case-Based Reasoning (ICCBR'97)*. 1997. Providence, RI, USA: Springer-Verlag.
9. Burton, B.A., *et al.*, *The Reusable Software Library.* IEEE Software, 1987. **4**(July 1987): p. 25-32.
10. Borgo, S., *et al. Using a Large Linguistic Ontology for Internet-Based Retrieval of Object-Oriented Components*. in *9th International Conference on Software Engineering and Knowledge Engineering, SEKE'97*. 1997. Madrid, Spain: Knowledge Systems Institute, Illinois.
11. Aamodt, A. and E. Plaza, *Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches.* AI Communications, 1994. **7**(1): p. 39-59.
12. Kolodner, J., *Case-Based Reasoning*. 1993: Morgan Kaufman.
13. Maher, M.L., M. Balachandran, and D. Zhang, *Case-Based Reasoning in Design*. 1995: Lawrence Erlbaum Associates.
14. Miller, G., *et al.*, *Introduction to WordNet: an on-line lexical database.* International Journal of Lexicography, 1990. **3**(4): p. 235 - 244.
15. Rumbaugh, J., I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*. 1998, Reading, MA: Addison-Wesley.
16. Nick, M., K.-D. Althoff, and C. Tautz. *Facilitating the Practical Evaluation of Organizational Memories Using the Goal-Question-Metric Technique, KAW'99*. in *Twelfth Workshop on Knowledge Acquisition, Modeling and Management*. 1999. Banff.