Management and Reuse of Software Design Knowledge Using a CBR Approach

Paulo Gomes, Francisco C. Pereira, Paulo Paiva, Nuno Seco, Paulo Carreiro
José L. Ferreira and Carlos Bento
CISUC - Centro de Informática e Sistemas da Universidade de Coimbra
Departamento de Engenharia Informática - Polo II
Universidade de Coimbra - Portugal
pgomes@dei.uc.pt

Abstract

From the viewpoint of an organization that develops software, knowledge generated during software development is valuable and would be very useful if it could be stored for later reuse. In this paper we describe a computational system for software design knowledge management and reuse. Our approach is based on Case-Based Reasoning and WordNet. We explain how knowledge management is performed in our system, and how some tools that we have implemented help the system administrator do this. We also describe how the stored knowledge is reused. Finally we discuss the advantages and limitations of our approach.

1 Introduction

Generally knowledge generated in the software development process is not stored in a way that it can be reused later on other projects. If software development companies reuse knowledge associated with the development process, they will improve in at least two major aspects: development efficiency and software quality. Another main advantage of performing knowledge management and reuse of software development knowledge is that it could minimize the loss of valuable software engineers. Storing, managing and reusing the development knowledge would enable the company to have access some of the know-how of their development teams.

Software development has several phases [Boehm, 1988], from analysis to integration, passing by testing and design. Among these phases there is one that is the focus of this work: the design phase. During the design phase the structure and behavior of the system is specified. Generally design is a complex task an ill-defined task [Tong and Sriram, 1992], making it hard to model and to automate. The knowledge generated during the design phase, is going to be reused later on other projects by the software engineers. We are interested in studying the management of design knowledge in a software development company, involving several software designers.

Most of the software designers decisions are taken using their experience. The more experience a designer has, the better s/he can perform its job, in principle. Reasoning based on experience is a basic mechanism for designers, enabling them to reuse previous design solutions in well known problems or even in new projects, sometimes generating innovative or creative solutions. In artificial intelligence there is a sub area called Case-Based Reasoning (CBR, see [Kolodner, 1993; Aamodt and Plaza, 1994; Maher et al., 1995]) that uses experiences, in the form of cases, to perform reasoning. We think that CBR is a good candidate for building a design system that can act like an intelligent design assistant.

The observations presented before, lead us to the development of a computational system that could perform three tasks: storage of software design, management of this knowledge, and reuse of it. To achieve these goals, we propose a system based on CBR. The CBR framework is flexible enough to comply with different knowledge types and reasoning mechanisms, enabling the software designer to use whatever design assistant s/he wants to use. The case-base used in the CBR system is a key point in the system, allowing the system to store and reuse design knowledge. We also integrated a lexical resource - WordNet [Miller et al., 1990] - which enables several semantic operations like indexing software objects and computing semantic distances between concepts.

The next section describes the CBR methodology describing its main ideas. Section 3 describes the architecture of our system - REBUILDER. We then detail some key issues of REBUILDER: knowledge base (section 4), design knowledge management (section 5) and design knowledge reuse (section 6). Section 7 describes and example of REBUILDER's use, providing more information about the system. Finally section 8 discusses several important issues concerning design knowledge management in REBUILDER.

2 Case-Based Reasoning

Case-Based Reasoning can be viewed as a methodology for developing knowledge-based systems [Althoff, 2001] that makes use of experience for reasoning about problems. Its main idea is to reuse past experiences to solve new situations or problems.

A case is a central concept in CBR, and it represents a chunk of experience in a format that can be reused by a CBR system. Usually a case comprises three main parts: problem, solution, and outcome. The problem is a description of the situation that the case is representing. This can be, for example, the symptoms of a patient in a medical situation, or
a software system’s requirements, or any description that can characterize the situation being represented. The solution describes what was used to solve the situation described in the problem. For instance, in the medical domain it can be the treatments used to heal the patient, or in the software domain a design that complies with the system’s requirements. The outcome expresses the result of the application of the solution to the problem. This means that commonly there are two possible outcomes: success or failure. A success case represents a situation in which the solution worked well, while a failure case represents a situation where the solution did not work. There can be other parts of cases like the justification that relates problem with solution through causal relations [Bento et al., 1995].

Another important part of the CBR methodology is the case library. This is the place where all the cases are going to be stored and organized. Due to the high number of cases that the library can have, most of the CBR systems use indexing structures that enable fast retrieval of relevant cases from memory. So, most of the times a case library is more than just the place to store cases, but it defines how they are stored and how they can be accessed.

At an abstract level CBR can be described by the reasoning cycle depicted in figure 1 [Aamodt and Plaza, 1994]. The reasoning process starts with the problem description, which is then transformed into a target case (or query case). The problem is provided by a system user or by another system. The first phase in the CBR cycle is to retrieve from the case library the cases that are relevant for the target case. The relevancy of a case must be defined by the system, but the most common one is similarity of features. In the end of retrieval, the best retrieved case is returned and passes to the next phase along with the target case.

The reuse phase (also designated as adaptation phase) adapts the retrieved case to the target case, yielding a solved case (or new case). This process can be performed with several inference techniques, and many work has been done on the subject [Voss, 1996]. The next step for a CBR system is to revise the new case returning a tested and repaired case. This phase usually comprises two parts: verification and evaluation. While verification checks the new case consistency and coherence, the evaluation phase assesses the performance characteristics of the new case. Finally, the retain phase learns the solved case by storing it in the case library. This phase is more complex than it seems, because not all cases should be stored. If a new case is equal or very similar to a case already in the library, then it should not be stored because it brings nothing new to the system and it degrades the system’s performance. This last phase closes the CBR cycle by feeding the system with new experiences, making the system capable of learning.

3 REBUILDER

The approach carried out in REBUILDER uses CBR as the main reasoning framework, enabling the coordination of several different types of knowledge. This section, first gives an overview on the system’s architecture, and then describes the knowledge base used. As a CBR system, REBUILDER implements the reasoning cycle described in section 2, the details of each reasoning step are described later in this section.

Figure 2 shows the architecture of REBUILDER. It comprises four main modules: the UML editor, the knowledge base manager, the knowledge base (KB), and the CBR engine. It also shows the two different user types: software designers, and KB administrators. Software designers use REBUILDER as a CASE tool, and the reuse capabilities of the system. A KB administrator as the function of keeping the KB updated and consistent. The UML editor is the interface between REBUILDER and the software designer, while the KB manager is the interface between the KB administrator and the system.

The UML editor is the front-end of REBUILDER and the environment where the software designer develops designs. Apart from the usual editor commands to manipulate UML objects, the editor integrates new commands capable of reusing design knowledge. These commands are directly re-

---

1There are system that retrieve more than one case, depending on the system’s purpose.

2Unified Modelling Language [Rumbaugh et al., 1998].

3Computer Aided Software Engineering.
lated with the CBR engine capabilities and are divided into two main categories: KB actions, such as connect to KB and disconnect from KB; and cognitive actions, such as retrieve design, adapt design using analogy or design composition, verify design, evaluate design, and actions related with object classification.

The KB Manager module is used by the administrator to manage the KB, keeping it consistent and updated. This module comprises all the functionalities of the UML editor, and adds case-base management functions. These are used by the KB administrator to update and modify the KB. The list of available functions are: create KB, open KB, close KB, case library manager (which comprises functions to manipulate the cases in the case library, like adding new cases, removing cases, or changing the status of a case), activate learning (gives the KB administrator an analysis about the contents of the case library), and system settings.

The KB comprises four different parts: the case library, which stores the cases of previous software designs; an index memory used for efficient case retrieval; a data type taxonomy, which is an ontology of the data types used by the system; and WordNet [Miller et al., 1990], which is a general purpose ontology. This module is described in more detail in the next subsection.

The CBR Engine is the reasoning module of REBUILDER. As the name indicates, it uses the CBR paradigm to establish a reasoning framework. This module comprises five different parts: Retrieval, Design Composition, Analogy, Verification, and Learning. All these modules are detailed in section 6.

4 Knowledge Base

The KB comprises the WordNet ontology, a case library, the case indexes and the data type taxonomy. In REBUILDER a case describes a software design, which is represented in UML through the use of class diagrams. Figure 3 shows an example of a class diagram representing part of an educational system. Nodes are classes, with name, attributes and methods. Links represent relations between classes. Closedarrow relations represent generalizations, and no-arrow relations represent participative associations. Conceptually a case in REBUILDER comprises: a name used to identify the case within the case library; the main package, which is an object that comprises all the objects that describe the main class diagram; and the file name where the case is stored. Cases are stored using XML/XMI (eXtended Mark-up Language), since it is a widely used format for data exchange. UML class diagram objects considered in REBUILDER are: packages, classes, interfaces and relations. A package is a UML object used for grouping other UML objects. A class describes an entity in UML and it corresponds to a concept described by attributes at a structural level, and by methods at a behavioral level. Interfaces have only methods, since they describe an entity in UML and it corresponds to a concept described by attributes at a structural level, and by methods at a behavioral level. Interfaces have only methods, since they describe an entity in UML and it corresponds to a concept described by attributes at a structural level, and by methods at a behavioral level.

WordNet is used in REBUILDER as a common sense ontology. It uses a differential theory where concept meanings are represented by symbols that enable a theorist to distinguish among them. Symbols are words, and concept meanings are called synsets. A synset is a concept represented by one or more words. If more than one word can be used to represent a synset, then they are called synonyms. The same word can have more than one different meaning (polysemy). WordNet is built around the concept of synset. Basically it comprises a list of word synsets, and different semantic relations between synsets. The first part is a list of words, each one with a list of synsets that the word represents. The second part, is a set of semantic relations between synsets, like is-a relations, part-of relations, and other relations. REBUILDER uses the word synset list and four semantic relations: is-a, part-of, substance-of, and member-of. Synsets are classified in four different types: nouns, verbs, adjectives, and adverbs. REBUILDER uses synsets for categorization of software objects. Each object has a context synset which represents the object meaning. The object’s context synset can be used for computing object similarity (using the WordNet semantic relations), or it can be used as a case index, allowing the rapid access to objects with the same classification.

As cases can be large, they are stored in files, which makes case access slower then if they were in memory. To solve this problem we use case indexes. These provide a way to access the relevant case parts for retrieval without having to read all the case files from disk. Each object in a case is used as an index. REBUILDER uses the context synset of each object to index the case in WordNet. This way REBUILDER can retrieve a complete case, using the case root package, or it can retrieve only a subset of case objects, using the objects’ indexes. This allows REBUILDER to provide the user the possibility to retrieve not only packages, but also classes and interfaces. To illustrate this approach, suppose that the class diagram of Figure 3 represents Case1. Figure 4 presents part of the WordNet structure and some of the case indexes associated with Case1. As can be seen, WordNet relations are of the types is-a, part-of and member-of, while the index relation relates a case object (squared boxes) with a WordNet synset.
Institution
- a

School

Staff

Teacher

Classroom

Room

is
- a

Figure 4: A small example of the WordNet structure and case indexes.

(rounded boxes). For instance Case1 has one package called School (the one presented in Figure 3), which is indexed by synset School. It has also a class with the same name and categorization, indexed by the same synset, making also this class available for retrieval.

The data type taxonomy is a hierarchy of data types used in REBUILDER. Data types are used in the definition of attributes and parameters. The data taxonomy is used to compute the conceptual distance between two data types.

5 Design Knowledge Management

REBUILDER stores and manages design knowledge gathered from the software designers activity. This knowledge is stored in a central repository, which is managed by the KB administrator.

The knowledge that can be managed by the administrator are the design cases, which is performed through the KB manager module. The basic responsibilities of the administrator is to setup the system and to decide which cases should be in the case library. Another task the s/he has to perform is to revise new diagrams submitted to the case library by the software designers.

Deciding the contents of the case library is not an easy task, especially if the case-base has a large number of cases. In this situation the KB manager provides a set of case-based maintenance policies that s/he can use to determine which are the cases that should be deleted or added to the case library. These policies are:

Frequency Deletion Criteria [Minton, 1990] This criteria selects cases for deletion based on the usage frequency of cases.

Subsumption Criteria [Racine and Yang, 1997] The subsumption criteria defines that a case is redundant if: is equal to another case, Or is equivalent to another case, Or is subsumed by another case. In REBUILDER a case is considered subsumed when, there is another case with the same root package synset, and the package and objects structure is a substructure of the other case.

Footprint Deletion Criteria [Smyth and Keane, 1995] This criteria is based on two definitions: case coverage, or the set of problems that a case can solve; and reachability set of a case is the set of cases that can be used to provide solutions for a problem. Based on these two notions the case base is divided in three groups (initially in Smyth’s work were four, but one of the subgroups can not be distinguished in our approach). Pivotal cases represent an unique way to answer a specific query. Auxiliary cases are those which are completely subsumed by other cases in the case base. Spanning cases are cases between Pivotal and Auxiliary cases, which link together areas covered by other cases. When the case library has too many cases the recommended order of deletion is: auxiliary, spanning, and pivotal.

Footprint-Utility Deletion Criteria [Smyth and Keane, 1995] This criteria is the same as the Footprint Deletion Criteria, with the exception that when there is a draw the selection is based on the case usage - less used cases are chosen for deletion.

Coverage Criteria [Smyth and McKenna, 1998] The coverage criteria involves three factors: case-base size, case-base density, and case-base distribution. A new case is added to the case library if its inclusion in the case-base increases the case-base coverage/case-base number ratio.

Case-Addition Criteria [Zhu and Yang, 1999] The case-addition criteria involves the notion of case neighborhood, which in REBUILDER is defined as all the cases that have the same root package synset as the case being considered. This criteria uses the notion of benefit of a case in relation to a case set, which is based on the frequency function of cases. A new case is added to the case base if its benefit is positive.

Relative Coverage [Smyth and McKenna, 1999] The goal of this criteria is to maximize coverage while minimizing case-base size. The proposed technique for building case-bases is to use Condensed Nearest Neighbor on cases that have first been arranged in descending order of their relative coverage contributions.

Relative Performance Metric [Leake and Wilson, 2000] This criteria uses the notion of relative performance to decide if a case should be added to the case base or not. The relative performance of a case is based on the sum of the adaptation cost of the case being considered to all cases in its coverage set. If the relative performance of a new case being considered is higher than an established threshold, then the new case is added to the case library.

Competence-Guided Criteria [McKenna and Smyth, 2000] The competence-guided criteria extends previous works of Smyth, which use the notions of case competence based on case coverage and reachability. This criteria uses three ordering functions:

- Reach for Cover (RFC): which uses the size of the reachability set of a case. The RFC evaluation function implements this idea: the usefulness of a case, which is an inverse function of its reachability set size.
- Maximal Cover (MCOV): which uses the size of the coverage set of a case. Cases with large coverage sets can classify many target cases and as such, must make a significant contribution to classification competence.
• Relative Coverage (RC): which is defined in a previous criteria.

When a diagram is submitted by a software designer as a candidate to a new case to be added to the case library, the administrator has to check some items in the diagram. First the diagram must have synsets associated to the classes, interfaces and packages. This is essential for the diagram to be transformed in a case, and to be indexed and reused by the system. The diagram consistency and coherence must also be checked. REBUILDER provides a verification and evaluation mechanism that can help the administrator in two different ways: verification can identify certain syntax and semantic errors or less common items in the diagram and draw the administrator’s attention to them; evaluation assesses the design properties and characteristics presenting them as guidelines for the diagram’s evaluation, this is performed using several object-oriented metrics [Bär et al., 1999; Rosenberg and Hyatt, 1996]. Finally the administrator can use the case-based maintenance policies described before to help him to decide if the submitted diagram should be added to the case library or not.

The KB Manager module is used by the administrator to manage the KB, keeping it consistent and updated. This module comprises all the functionalities of the UML editor, and it adds case-base management functions to REBUILDER. These are used by the KB administrator to update and modify the KB. The list of available functions are:

KB Operations Create, open or close a KB.

Case Library Manager Opens the Case Library Manager, which comprises functions to manipulate the cases in the case library, like adding new cases, removing cases, or changing the status of a case.

Activate Learning Gives the KB administrator an analysis about the contents of the case library. REBUILDER uses several case-base maintenance policies to determine which cases should be added or removed from the case library.

Settings Adds extra configuration settings which are not present in the normal UML Editor version used by the software designers. It also enables the KB administrator to configure the reasoning mechanisms.

6 Design Knowledge Reuse

Reuse of UML class diagrams can be done in REBUILDER in three different ways: using retrieval, using analogy or using design composition. The retrieval mechanisms searches the WordNet structure looking for similar cases and then ranks and presents them to the designer. Analogy takes as input a UML class diagram and tries to complete it using analogical mapping and knowledge transference. Design composition generates a new class diagram through the integration of different pieces of cases, using the target diagram for guidance in this process. The next sub sections describe each one of these reuse mechanisms.

![Figure 5: The case retrieval algorithm used in REBUILDER.](image-url)

6.1 Retrieval

Retrieval in REBUILDER comprises two phases: retrieval of a set of relevant cases from the case library, and assessment of the similarity between the target problem and the retrieved cases.

The retrieval phase is based on the WordNet structure, which is used as an indexing structure. The retrieval algorithm uses the classifications of the target problem object as the initial search probe in WordNet. This algorithm is flexible enough to retrieve three different types of UML objects: packages, classes and interfaces, depending on the type of object selected as target problem. For example, if the designer selects a package as the target problem, the retrieval algorithm uses the package’s synset as the initial search probe. Then the algorithm checks if there are any packages indexed associated with the WordNet node of that synset. If there are enough indexes, the algorithm stops and returns them. Otherwise, it explores the synset nodes adjacent to the initial one, searching for package indexes until the number of found indexes reaches the number of objects that the user wants to be retrieved. 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), then the algorithm is described in figure 5.

The second step of retrieval is ranking the retrieved objects by similarity with the target object. Since there are three types of target objects (packages, classes and interfaces) we have developed a specific similarity metric for each type of objects (for more details on these metrics see [Gomes et al., 2002a]).

Package Similarity Metric This metric is based on four different aspects: the similarity between packages’ synsets,
similarity between packages’ dependencies, similarity between packages’ class diagrams, and similarity between the sub-packages (a recursive call to this metric). Basically this metric assesses structure similarity and semantic similarity of packages and its objects.

**Class Similarity Metric** The class similarity metric is based on three items: synset similarity of classes being compared, inter-class similarity comprising the assessment of relation similarity between classes, and intra-class similarity which evaluates the similarity between classes’ attribute and methods.

**Interface Similarity Metric** The interface similarity metric is the same as the class similarity, except in the intra-class similarity, which is based only in method similarity, since interfaces do not have attributes.

6.2 Analogy

The analogy mechanism relies in a set of cases given by the retrieval phase. From these cases the n best ones are selected for generating new diagrams. This number is chosen by the user and represents the number of alternative solutions required. Then, for each case, the analogy module tries to map each object in the target diagram to an object in the case diagram. In this process structural relations are used as constraints, transforming this process in a structural matching algorithm, in which node matching is based on semantic similarity of objects’ synsets. This semantic similarity is based on the WordNet distance between synsets. The result of this process is a mapping between target and case objects, which is then used for transferring the knowledge from the case diagram to a new diagram (which is a copy of the target diagram). Knowledge transference is performed in two steps: first attributes and classes are transferred from case classes/interfaces to the new diagram classes/interfaces, and then new objects and relations are transferred from the case diagram to the new diagram. The final result is a new class diagram based on the target diagram. For more details on the analogy process see [Gomes *et al.*, 2002b].

6.3 Design Composition

The design composition is a different way of generating new diagrams based on the merging of different case pieces. Design composition starts with a set of retrieved cases, which are then used to generate the new class diagrams. There are two different composition strategies: best case composition (BCC) and best complementary set of cases composition (BCSCC).

The main idea of the BCC strategy is based on selecting the best retrieved case, mapping it to the target diagram, and transferring the knowledge to a new diagram. If there are unmapped target objects, then this strategy tries to find matching objects in other retrieved cases. This goes on until all the target objects are mapped or no more retrieved cases can be used.

![Figure 6: The initial diagram used by the designer as target problem.](image)

The BCSCC strategy is based on the idea of complementary cases, regarding the target diagram. This strategy starts by mapping every retrieved case to the target diagram, yielding a mapping, which is then used for grouping the retrieved cases in sets. These sets are generated based on how well the cases of a set, map the target diagram. The preference is to have sets, whose cases merged, completely map the target diagram. Then, each one of these sets can give origin to a new diagram. As in analogy, only the n best sets originate new diagrams. Sets are ranked by the degree of mapping of the target diagram.

7 Example of Use

This section describes an example that illustrates how the system can be used by a software designer. This example is about the designer client of REBUILDER, showing how design knowledge is reused.

Suppose that a designer is starting the design of a information system for a high school. She has already the system’s analysis done, in the form of use cases. From these use cases, some initial entities are extracted by the designer and drawn in the REBUILDER system. Figure 6 shows the initial class diagram, representing one of the system’s modules (scheduling). This module is responsible for handling the information data about teachers, classes and rooms timetables.

One of the tools available to the designer is the retrieval of similar designs from the case library. The designer can retrieve three types of objects: packages (a complete class diagram, what we call a case), classes or interfaces. Imagine that she selects the package object and clicks on the retrieval command. REBUILDER retrieves the number of diagrams defined by the designer (in this situation three cases). Figure 7 presents part of one of the retrieved cases (the most similar one), notice that in the diagram’s name there is the similarity score with the target design.

The retrieved designs can help the designer exploring the design space, aiding the assessment of the different alternative designs. Or she can go further and use the adaptation mechanisms of REBUILDER to generate new designs. The adaptation mechanisms are: analogy, design composition and

---

4Dependencies are a type of UML relations that can exist between packages, expressing a package’s dependency on another package.

5It is a UML diagram type used for describing system requirements.
Suppose that the designer selects design composition, figure 8 shows part of a diagram generated by this mechanism. As can be seen, it used the most similar case (case in figure 7) to build this new case, and then completed it with the missing objects.

Generated diagrams can have some inconsistencies, which can be fixed using the verification module. For example, suppose that, in the generated diagram (figure 8) the relation between class `Teacher` and class `Timetable` is out of context. The verification module checks four different knowledge sources to assess the relation’s validity: WordNet, design cases and verification cases. In WordNet the system looks for a relation between the `Teacher` synset and the `Timetable` synset, if it is found then the relation is considered valid. Otherwise, the system searches the design cases for a similar relation (a relation of the same type between the synsets of classes `Teacher` and `Timetable`). If the algorithm fails to find it, the next step is looking in the verification cases. It searches for a verification case describing the validity of a similar relation. A verification case can have two outcomes: success or fail. This way, if the algorithm finds a similar verification case and the outcome is success, the relation is considered valid, otherwise is considered invalid. If in the end the relation is considered invalid, then the designer is asked for a judgement about the relation and a new verification case is generated and stored in the case library.

After verification, if the designer considers the diagram correct and ready for being stored in the KB, for later reuse, then she can submit the diagram to the KB administrator. This new diagram goes into a list of unconfirmed cases of the case library. The KB administrator has the task of examining these diagrams more carefully, deciding which are going to be transformed into cases, going to the list of confirmed cases (ready to be reused by REBUILDER), and which are going to the list of obsolete cases not being used by the system.

8 Discussion

This paper describes REBUILDER a CASE tool that has the ability to manage the design knowledge gathered during software development. REBUILDER is used as a platform for reuse of that knowledge. If we analyze deeper the implications of software reuse as done in REBUILDER, it can be seen that inexperience designers profit a lot from the reuse of design knowledge, inhibiting the making of mistakes of previous systems. From the point of view of experienced designers, the system can help them exploring different alternative designs. This exploration can be very useful in innovative or creative design (see [Gomes et al., 2002c]).

From the company’s viewpoint, REBUILDER can be a very useful tool for minimizing the loss of know-how of software engineers that leave the company. Part of their experience is kept in the organization in the form of design cases. For the software development process the management and reuse of design knowledge can lower the development time and increase the software quality.

A limitation of our approach is that the company should change the software development process in order to fully use all the advantages of REBUILDER. The organization should shape its development process with the goal of developing software through reuse. Of course, that REBUILDER can also be used in different ways, from just a CASE tool to a complete Knowledge Management tool integrated with a CASE tool. One aspect of this limitation is that the system should be managed by a KB administrator. This is a fundamental piece in the system, and the good functioning of the system depends on her/his performance. REBUILDER tries to help the administrator in her/his task automating several case-base maintenance policies, but the last decision is always up to the administrator.

There is another important limitation to our approach, which relates to WordNet. From our experience the concepts defined in WordNet (the synsets) are sufficient to index cases and objects when the designer is working at a conceptual level. Objects represent abstract concepts and do not necessarily represent classes to be implemented. A problem arises
when the designer goes into the implementation (or detailed) design, where the objects must represent classes and interfaces to be implemented. To address this problem we have to WordNet the Java class hierarchy. Several software specific concepts, in this case relating to Java language, where added to WordNet hierarchies along with is-a links connecting them. These concepts allow the indexing of many software specific objects, ones that are only defined in the detailed design level.

Research works integrating CBR and Knowledge Management, use CBR to fulfill the technology needs of some of the Knowledge Management tasks [Aha and Munoz-Avila, 1999]. In our work, CBR is more than that, it provides the basic framework for all the Knowledge Management process, from the knowledge acquisition (case learning), knowledge management (KB manager) to the reuse of this knowledge (reasoning engine). Despite this, we are aware that the scope of REBUILDER is limited to the software design task, which is a specific task compared with the all process of knowledge management in a software development company or organization.

Acknowledgments

This work was partially supported by POSI - Programa Operacional Sociedade de Informação de Fundação Portuguesa para a Ciência e Tecnologia and European Union FEDER, under contract POSI/33399/SRI/2000, by program PRAXIS XXI. REBUILDER homepage is http://rebuilder.dei.uc.pt. We would like to thanks the reviewers of this paper for their helpful comments.

References


