# A CBR Approach to Text to Class Diagram Translation

António Oliveira, Nuno Seco and Paulo Gomes

AILab, Centro de Informática e Sistemas da Universidade de Coimbra
apsimoes@student.dei.uc.pt, nseco@dei.uc.pt, pgomes@dei.uc.pt

**Abstract.** The help provided by CASE tools in the development of software systems is very important. These tools are evolving by integrating new ways of making the job of the software engineer easier. We are developing an intelligent CASE tool that integrates a module that translates natural language text into a UML class diagram. This translation is a complex problem and it depends on the user and the vocabulary used. In this paper, we present an approach based on Case-Based Reasoning to translate natural language requirements to class diagrams. Our approach enables the system to adapt to the user vocabulary and the way that s/he models software systems.

## 1 Introduction

As software systems become bigger and more complex, researchers try to find ways to increase the productivity and efficiency of software development. Knowledge generated during the software development process can be a valuable asset for a software company. But in order to take advantage of this knowledge, the company must reuse this knowledge. This can be achieved through the use of knowledge management tools integrated in CASE tools. We are developing a CASE tool named REBUILDER UML that integrates a module for translation of natural language text into an UML class diagram [1]. This module is called REBUILDER TextToDiagram and uses an approach based on Case-Based Reasoning (CBR [2, 3]) and Natural Language Processing (NLP [4]).

Case-Based Reasoning (CBR) can be viewed as a methodology for developing knowledge-based systems that uses experience for reasoning about problems [5]. The main idea of CBR is to reuse past experiences to solve new situations or problems. The main objective of our work is to produce UML class diagrams from natural language using NLP techniques and a CBR approach. Our system deals, essentially, with morphological, syntactic and semantic (and a bit of discourse) issues of requirements text. This paper describes our approach, illustrating it with examples.

The next section describes REBUILDER UML, the CASE tool in which our approach is integrated, and NOESIS [6] a previous translation module that we developed and in which this work is based on. Section 3 presents the proposed approach and the developed module. Section 4 presents works that are related with our approach and makes some final remarks.

## 2 REBUILDER UML and NOESIS

REBUILDER UML is implemented as a plug-in for Enterprise Architect (EA www.sparxsystems.com.au), a commercial CASE tool for UML modeling. The plug-in comprises three main modules (see Figure 1): the knowledge base (KB), the CBR engine, and the KB manager. The KB is the repository of knowledge that is going to be reused by the CBR engine. The main goal of the system is to reuse UML class diagrams, which are stored as cases in the case library and reused by the CBR engine. The knowledge base manager enables all the knowledge stored in the system to be maintained.

There are two types of users in REBUILDER UML: software engineers and the system administrator. A software engineer uses the CASE tool to model a software system in development, and s/he can use REBUILDER UML commands to reuse old diagrams. These diagrams result from previous systems developed by her/him, or by the development team in which s/he is integrated. The other user type is the system administrator, which has the aim of keeping the KB fine tuned and updated. Since each software engineer has a copy of the central KB, the system administrator is responsible for making new releases of the KB and installing it in the systems of the development team (or teams). Thus, the role of the administrator is crucial for the system to be used properly by several users, enabling the sharing of knowledge among them. Despite this, the system can also be used in a stand alone fashion, acting as a knowledge repository with intelligent reuse tools for a single user. In this setup, the user is at the same time playing both roles: reusing and maintaining knowledge.

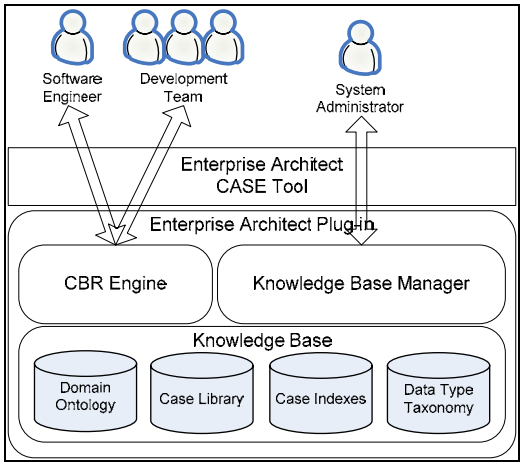


**Figure 1. The architecture of REBUILDER UML, based as a plug-in for Enterprise Architect ™.**

The integration with EA is made by a plug-in, enabling REBUILDER UML to have access to the data model of EA, and also to its model repository. Visually, the user interacts with REBUILDER UML through the main menu of EA. The user can

search, browse, retrieve and reuse past designs. Maintenance operations are also available for the administrator.

NOESIS [6] was the first prototype developed within the scope of REBUILDER with the goal of helping the designer formalize, in the form of UML class diagrams, the initial software modeling step. In other words, NOESIS produced an initial UML diagram that could be submitted to the other reasoning modules that would then complete and elaborate the initial model. The input to NOESIS is a text describing the structural requirements of the desired software (see [6] for examples). These texts are simplified in terms of language complexity, which attenuates the difficulty of understanding them. CBR was used to perform semantic analysis of the input text.

NOESIS did not handle some of the most important aspects of UML class modeling such as class attributes and methods. We hope to ameliorate our previous work and attain these lacking aspects with this new implementation.

## 3 REBUILDER TextToDiagram

REBUILDER TextToDiagram comprises a morphological analyzer that outputs tagged text, a syntactic analyzer that outputs chunks and a semantic analyzer that outputs UML class diagrams, as can be seen in Figure 2. It uses the OpenNLP tool [7] to do the morphological and syntactic analysis. The knowledge layer comprises three parts: a domain ontology used to compute semantic distances between concepts; case indexes, which are links between cases and the ontology; and the case base, comprising all the cases needed for the system to reason. The ontology comprises concepts and relations between concepts. Cases comprise a text, a diagram and mappings between the text and the diagram objects and relations. A case index is an association between a word in a case and an ontology concept.

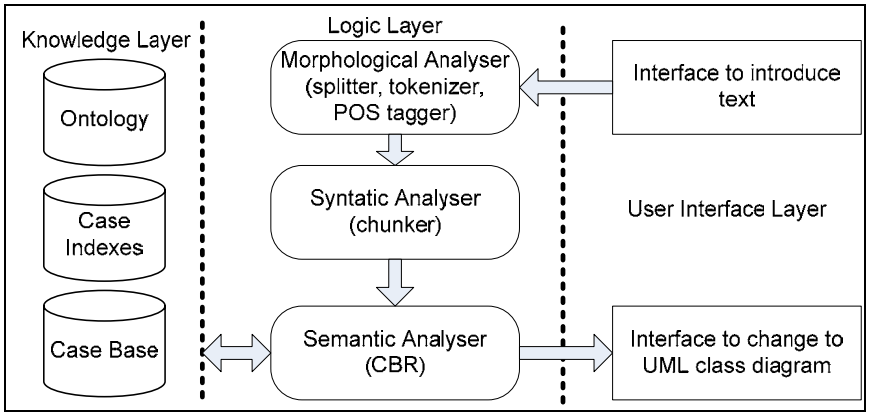


**Figure 2. REBUILDER TextToDiagram architecture.**

### 3.1 Morphological Analyzer

The Morphological analyzer receives a text where the user expresses system requirements. It has essentially three phases: splitting, tokenization and POS tagging. During the splitting phase, it identifies all the phrases of input text (see Figure 3).

```
Vendors may be sales employees or companies.
Sales employees receive a basic wage and a commission, whereas companies only receive a commission.
```

**Figure 3. Splitting of the input text.**

Then, the tokenizer breaks the identified phrases into tokens (see Figure 4). Finally, the POS tagger identifies all the grammatical categories (using tags for example: NN for noun, NNS for plural nouns, VB for verbs, and so on) of the tokens identified as can be seen in Figure 5.

```
Vendors | may | be | sales | employees | or | companies | .
Sales | employees | receive | a | basic | wage | and | a | commission | , | whereas | companies | only | receive | a | commission | .
```

**Figure 4. Text tokenization.**

```
Vendors/NNS may/MD be/VB sales/JJ employees/NNS or/CC companies/NNS ./.
Sales/NNS employees/NNS receive/VBP a/DT basic/JJ wage/NN and/CC a/DT commission/NN ,/, whereas/IN companies/NNS only/RB receive/VBP a/DT commission/NN ./.
```

**Figure 5. Text POS tagging.**

### 3.2 Syntactic Analyzer

The Syntactic analyzer receives all tags from the previous module and defines all the chunks. These chunks are syntactic units useful when looking for units of meaning larger than the individual words, identified in the text using shallow parsing. Figure 6 presents the obtained chunks. The chunks produced by the syntactic analyzer are then passed to the semantic analyzer to produce the corresponding class diagram.

```
[NP Vendors/NNS ] [VP may/MD be/VB ] [NP sales/JJ employees/NNS or/CC companies/NNS ] ./.
[NP Sales/NNS employees/NNS ] [VP receive/VBP ] [NP a/DT basic/JJ wage/NN ] and/CC [NP a/DT commission/NN ] ,/, [PP whereas/IN ] [NP companies/NNS ] [ADVP only/RB ] [VP receive/VBP ] [NP a/DT commission/NN ] ./.
```

**Figure 6. Text Chunking.**

### 3.3 Semantic Analyzer

The Semantic analyser receives the chunks and produces the corresponding UML class diagram using Case Based Reasoning (CBR). Each case (see Figure 7 for an example) has a problem description (chunks and a text ID), the solution description (the UML class diagram) and the mappings between the problem terms and the solution objects (from nouns to classes, verbs to relations or methods and so on). A case is stored in a XMI file (XMI stands for XML Metadata Interchange, which is a proposed use of XML intended to provide a standard way for programmers and other users to exchange information about metadata).

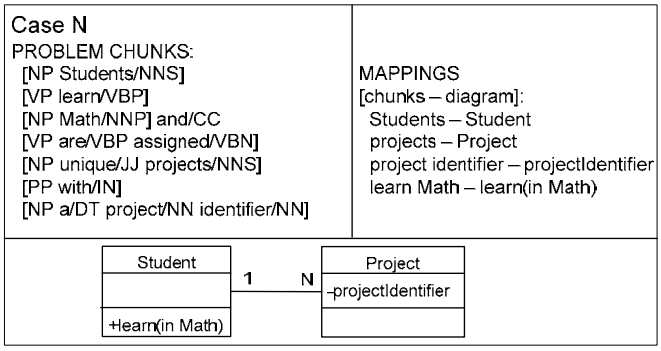


**Figure 7. Example of a case.**

There are two structures used to store cases indexes: an ontology and a syntactic tree. The ontology reflects the semantic similarity (and relatedness) using relations of specialization and generalization between concepts. There are indexes in the ontology that establish a relation between concepts and cases. The indexes are links between words in the case problem description and the corresponding ontology concept. This association is straightforward since the domain ontology is intended not to have ambiguities. There are also relations between chunks and cases in the syntactic tree.

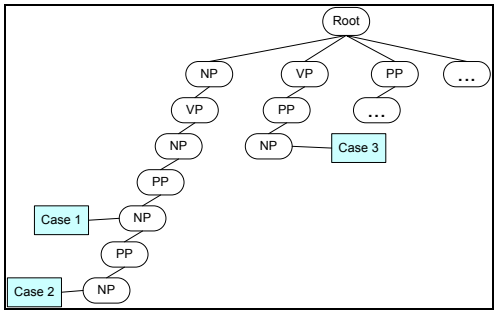


**Figure 8. Example of the syntactic tree.**

The syntactic tree (see Figure 8) reflects the syntactic similarity, storing cases in specific nodes of chunks. For instance the phrase "*A bank has many clients with several accounts*" would have the chunks [**NP** A/DT bank/NN ] [**VP** has/VBZ ] [**NP** many/JJ customers/NNS ] [**PP** with/IN ] [**NP** several/JJ accounts/NNS], so the most similar case with this one stored in the tree is the case 1.

The semantic analyzer architecture was developed according to the CBR main steps, as we can see in the Figure 9. The first step is case retrieval, which uses the index structures to select a group of similar cases. Then, from these cases, the system selects the best case using similarity metrics (case selection**)**. With the best case selected, the system reuses it using heuristic rules (**case adaptation**) by mapping the chunks of the solution with chunks of the problem. The last process stores the process obtained in the case base, if the similarity (to other cases in the case base) is beyond a specified threshold.

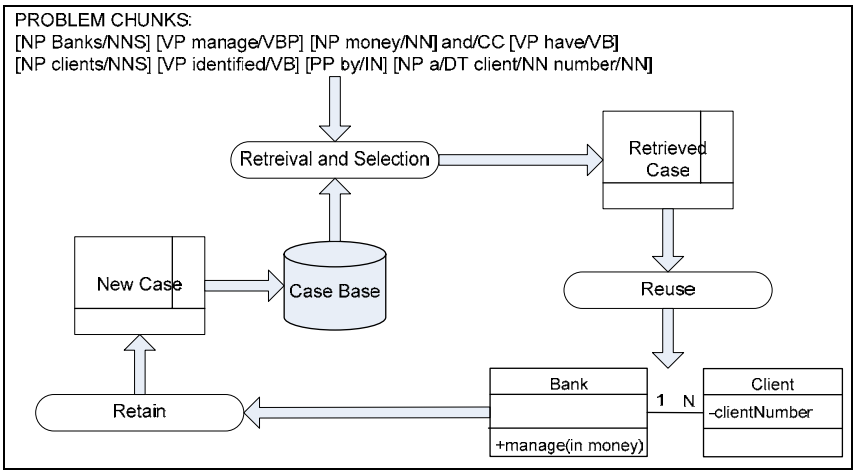


**Figure 9. Semantic analyser architecture.**

## Case Retrieval

The retrieval process comprises four algorithms that use the index structures: Syntactic, Semantic Filter, Semantic and Conjunctive. All algorithms start by finding a list of entry points into the index structures. Verbs and names, which are associated with ontology concepts, are used as the entry points in the ontology. In the syntactic index tree, the entry point is the node with the same chunks as our target sentence. If it does not exist, the algorithm creates a list of entry points (above a specified threshold) containing nodes that represents the generalization and specialization of the target sentence. The core of the retrieval algorithms is presented in Figure 10.

```
 indices syntacticRetrieval(syntacticNodes)
 for each element in syntacticNodes
frontier += initialFrontier
 end for
 while length(frontier) > 0
for each element in frontier
   indices += obtainSyntacticIndeces(element)
   auxFrontier += specializations(element) + generalizations(element)
end for
 end while
 return indices
```

**Figure 10. Core of the retrieval algorithm.**

The **syntactic retrieval** algorithm uses only the syntactic index tree. The **semantic filter retrieval** algorithm uses the syntactic index tree and the ontology to constrain the search to a specific semantic distance threshold. The **semantic filter** algorithm uses the ontology to obtain concept references to names of the target sentence and uses these concepts as the entry points. The **conjunctive retrieval** algorithm uses both the syntactic index tree (as in the syntactic retrieval) and the ontology (as in the semantic filter) to retrieve indexes and in the end intersects the indices returned from each structure.

**Case Selection**

The selection process can chose the most similar case using seven algorithms with different similarity metrics: syntactic, semantic, discourse, contextual, morphological, Levenshtein distance and n-gram similarity. We use the chunks, tags and tokens as comparison units in these algorithms.

The **syntactic similarity** metric verifies the number of common nodes between problem chunks and solution chunks. This metric is computed using the following formula (where $T$ – target; $S$ – Source, $I$ – number of intersection nodes, *nodes* – number of nodes):

$$\frac{1}{2} \times \left( \frac{I(T,S)}{nodes(T)} + \frac{I(T,S)}{nodes(S)} \right)$$

The **semantic similarity** metric verifies the semantic distance between the verbs and nouns of the problem and solution with this formula (*MSL* – Maximum Search Length allowed by the algorithm, constraining the search space):

$$\frac{1}{2 * MSL} \times \left( \frac{dist(T,S)}{nodes(T)} + \frac{I(T,S)}{nodes(S)} \right)$$

The **contextual similarity** metric verifies the number of times that specific cases from a specific text were used. If the case was already used the algorithm returns 1 else it returns 0.

The **discourse similarity** verifies the position of each sentence of both the problem and the solution in the original text:

$$\frac{1}{\text{sentenceNumber(T)} - \text{sentenceNumber(S)}^2 + 1}$$

The **morphological similarity** metric verifies the number of common tags between the phrase of the problem and the phrase of the solution, the formula is:

$$\frac{I(T,S)}{\max(nodes(T), nodes(S)}$$

The **Levenshtein distance** metric [8] is used between two phrases (chunks or tags) and is given by the minimum number of operations needed to transform one phrase into the other, where an operation is an insertion, deletion, or substitution of a single string.

The **n-gram similarity** metric [9] is used to compare n-grams of two phrases (chunks, tags or tokens):

$$\frac{I(nGramsT, nGramsS)}{nGrams(T)} + \frac{I(nGramsT, nGramsS)}{nGrams(S)}$$

Each previous equation is normalized to the interval [0...1]. Then the system computes a weighted sum to evaluate the overall similarity between the problem and the solution. That sum is given by:

$$\begin{aligned}\text{similarity(source, target)} &= \text{w1} * \text{syntactic similarity} + \text{w2} * \text{semantic similarity} \\ &+ \text{w3} * \text{contextual similarity} + \text{w4} * \text{discursive similarity} + \text{w5} * \text{morphological similarity} \\ &+ \text{w6} * \text{Levenshtein similarity} + \text{w7} * \text{n - grams similarity}\end{aligned}$$

The weights w1 through w7 are associated with each metric, the sum of the weights is equal to 1. In order to define a weight configuration, we are going to use genetic algorithms. An individual is a set of weights and the evaluation function is a set of test cases which are presented to the system as problems. We then compare the generated diagram with the pre defined diagram (established by software engineers). The idea of weight optimization is to minimize the distance between diagrams.

**Case Adaptation**

The adaptation process uses two algorithms: one to adapt the solution chosen for each phase and other to join all diagrams into one (remember that each diagram represents only one phrase of the original text). After the process of mapping all sentences of the target text with similar source sentences, we load the specific diagram. The selected index and names of the source diagram are replaced with the nouns encountered in the target sentence. Other heuristics are used to perform the adaptation, for example: after a prepositional phrase usually comes an attribute of the previous noun phrase; the first name of the phrase is always an entity; emergence and action verbs usually

represent entities methods; state verbs represent binary attributes; attributes can be identified by values expressed in attributive adjectives among others. An example of this process can be seen in the Figure 11.

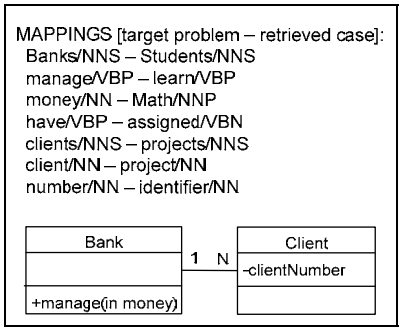


**Figure 11. Mappings of the adaptation process.**

## 4 Conclusions and Related Work

There are some research groups investigating ways of building conceptual models from software requirements expressed in natural language, we now provide a brief overview of some of these efforts and some concluding remarks.

Illieva and Ormandjieva [10] use a methodology that can be split into four phases. The first phase of linguistic analysis does the POS tagging of the text followed by chunking (shallow parsing), the second phase of tabular representation identifies the subject, the verb and the object of the various sentences, the third phase of semantic net representation does a translation of subjects and objects to entities, and verbs and prepositions to relations among these entities, finally the fourth phase produce a class diagram representation according to some rules from the semantic net.

In NL-OOPS [11] it is used the PLN system LOLITA, which does morphological, syntactic, semantic and pragmatic analysis of the input text and stores the result in a semantic net. NL-OOPS extracts knowledge from this semantic net: static nodes are mapped to classes, associations and attributes, dynamic nodes are mapped to methods.

Research groups from Birmingham and Indianapolis [12] developed an approach that uses formal verification of requirements and represent them in XML. Afterwards the requirements are specified according to a two level grammar and finally this specification can be mapped to JAVA, XMI/UML or VDM++.

Li, Dewar and Pooley [13] developed a methodology that does POS tagging, followed by a simplification step that transforms the original text into a text with the triples: subject, verb and object (SVO) and finally this simplified text is mapped to class diagrams using predefined rules.

This paper presents an approach to the translation of natural language text into a class diagram. This approach is based on CBR and is supported by an ontology. The

main advantage of our approach is the flexibility that it possesses in relation to its adaptation to the user way of modeling software systems, and to the vocabulary used. It also enables the system to learn new knowledge, thus covering other regions of the search translation space. A positive aspect of this approach is the knowledge sharing aspect, where new system users can benefit from an already developed case base.

REBUILDER TextToDiagram runs on an ordinary PC and is developed in C#. The ontology used is a domain ontology that can be pre defined, or can be learned during user usage of the system. The main focus of the ontology is to define an semantically organize every concept in the case base. These concepts are then used for similarity assessment and case retrieval. Our system is going to be evaluated by software engineers, where we are going to measure the time spent by software designers to develop a class diagram from text, with and without the system.

## References


1. Rumbaugh, J., I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*. 1998, Reading, MA: Addison-Wesley.
2. Aamodt, A. and E. Plaza, *Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches.* AI Communications, 1994. **7**(1): p. 39-59.
3. Kolodner, J., *Case-Based Reasoning*. 1993: Morgan Kaufman.
4. Jurafsky, D. and J. Martin, *Speech and Language Processing*. 2000, New Jersey: Prentice-Hall.
5. Althoff, K.-D., *Case-based reasoning*, in *Handbook on Software Engineering and Knowledge Engineering*, S.K. Chang, Editor. 2001, World Scientific. p. 549-588.
6. Seco, N., P. Gomes, and F.C. Pereira. *Using CBR for Semantic Analysis of Software Specifications*. in *7th European Conference on Case-Based Reasoning (ECCBR 2004)*. 2004. Madrid, Spain.
7. Project, C., *OpenNLP, Statistical parsing of English sentences*. 2005.
8. Day, W.H.E., *Properties of Levenshtein metrics on sequences.* Bull. Math. Biol., 1984. **46**.
9. Kondrak, G. *N-Gram Similarity and Distance*. in *String Processing and Information Retrieval (SPIRE 2005)*. 2005.
10. Ilieva, M.G. and O. Ormandjieva. *Automatic Transition of Natural Language Software Requirements Specification into Formal Presentation*. in *NLDB*. 2005.
11. Mich, L., *NL-OOPS: From Natural Language to Object-Oriented Requirements using the Natural Language Processing System LOLITA.* Natural language engineering, 1996. **2**(2): p. 161-187.
12. Bryant, B., et al. *From natural language requirements to executable models of software components*. in *The Monterey Workshop on software engineering for embedded systems: from requirements to implementation*. 2003.
13. Li, K., R.G. Dewar, and R.J. Pooley, *Object-Oriented Analysis Using Natural Language Processing*. 2005.