LASER: A Lexical Approach to Analogy in Software Reuse

Rushikesh Amin, Mel Ó Cinnéide and Tony Veale
Department of Computer Science,
University College Dublin,
Belfield, Dublin 4,
Ireland.
{rushikesh.amin,mel.ocinneide,tony.veale}@ucd.ie

Abstract

Software reuse is the process of creating a software system from existing software components, rather than creating it from scratch. With the increase in size and complexity of existing software repositories, the need to provide intelligent support to the programmer becomes more pressing. An analogy is a comparison of certain similarities between things which are otherwise unlike. This concept has shown to be valuable in developing UML-level reuse techniques. In the LASER project we apply lexically-driven Analogy at the code level, rather than at the UML-level, in order to retrieve matching components from a repository of existing components. Using the lexical ontology WordNet, we have conducted a case study to assess if class and method names in open source applications are used in a semantically meaningful way. Our results demonstrate that both hierarchical reuse and parallel reuse can be enhanced through the use of lexically-driven Analogy.

1. Introduction

Software reuse, in its broadest terms, has been viewed as the reapplication of knowledge from one software system to another [4]. Software reuse reduces development time and cost while improving quality. Most engineering disciplines are based on the reuse concept, from components to formulas and ideas themselves [9]. Nonetheless, in terms of reliability and maintenance effort, software engineering compares badly to other, more physical forms of engineering such as chip design, which ultimately employ the same logical concepts.

In general, a reuse environment comprises a repository of reusable components, together with a mechanism for their retrieval, adaptation and integration. One model of software reuse is the so-called 3C model, involving the notions of *concept*, *content*, and *context* [10]. In this model, the software engineer must specify the conceptual requirements of the component they are seeking. Components are described in terms of their content (data structure used, etc.). In the description of both content and concept, it may also be necessary to provide contextual information. Such an approach has the disadvantage that it involves considerable work in specifying this additional information. Also, the repository of reusable components will continue to grow, thus making the task of finding and choosing an appropriate component more difficult [6].

Analogy involves a structural comparison of two concepts that appear substantially different on the surface but which exhibit important causal or semantic symmetries. Computational models of analogy have shown themselves to be valuable in the development of UML level reuse techniques; examples include *ReBuilder* [2, 3]. Analogy typically has both a linguistic and a conceptual dimension, the latter communicated by the former. Words and their meanings thus play an important role in the effectiveness and comprehension of analogies. In this research we examine whether software artifacts like Java programs exhibit the same reliance on lexical expression for their meaning. It seems clear that good developers choose class, method and variable names that are lexically expressive about the goals of the code. If this is true, we can apply lexical analogy techniques at code level rather than UML-level in order to retrieve matching components from the reuse environment.

The object-oriented paradigm facilitates reuse of code by packaging the most reusable structure and behaviours into distinct classes. The programmer can extend the basic functionality of these classes and/or modify it to get the desired functionality. Our research currently focuses on software written in the Java programming language for a number of reasons. Firstly, Java provides an elegant means of encouraging class creation, extension, composition and abstraction. Secondly, various conventions for Java code suggest that the linguistic elements of Java programs, such as method, class and variable names, are more likely to cor-

respond to meaningful words or phrases in a natural language like English. As such, the innovative core of this research project is the belief that natural language techniques can be productively applied to artificial language constructs like software code to yield a meaningful conceptual representation. By focusing on the actual code level of a new software design rather than the abstract component level, we can exploit a rich vein of lexical-driven opportunities for reasoning about a developer's goal and requirements. While it is true that the general structure of software code does not conform to a natural-language grammar, most modern programmers do employ many of the principles of natural language expression in the naming of classes and methods.

In this paper we present our ongoing work on the application of Analogy to the software reuse domain. We have completed a preliminary study by parsing *JRefactory* [1], an open source refactory tool, to determine if class names and method names follow the natural language technique. We also researched how hierarchical reuse and parallel reuse can be enhanced using a lexically-driven analogical approach. As a basis for our work we are using WordNet, a broad coverage knowledge-base of the English lexicon. In the next section we introduce related work, while in section 3 a brief description of WordNet is given. Section 4 describes analogical reasoning as it is used in LASER. A case study is presented in section 5 and finally, directions for future research and conclusions are discussed in section 6.

2. Related Work

ReBuilder [2, 3], a software tool being developed in the AI Lab of the University of Coimbra, is innovative in the way it uses WordNet to index and retrieve software cases. ReBuilder allows analogical retrieval and mapping between UML descriptions of software systems, and uses analogical transfer to flesh out a new software design based on structural parallels with a pre-existing design. Developers explicitly associate software designs with one or more nodes in the taxonomy, so when the new development project is initiated, the taxonomy can be searched for similar precedents. However, the benefits of this reuse scheme can only be reaped by those developers who take the time to tell ReBuilder how to appropriately annotate and index their software for future retrieval. We also exploit WordNet, but at a much finer level of analysis. By prying into the internal lexical structure of software components, to analyse the names given to the methods and classes and relate these to concepts in the WordNet taxonomy, we create a whole spectrum of new possibilities.

CodeFinder, together with PEEL (Parse and Extract Emacs Lisp), is a software tool that supports the process of finding components for reuse [5]. Repositories are initially seeded semi-automatically with structure and index terms by PEEL. These retrieval structures are used by CodeFinder to allow the user to find semantically-related components. The initial retrieval structures are likely to be incomplete, but CodeFinder also enables the user to add new structure and index terms to the repository to improve future component retrieval. In this way, CodeFinder supports the user in finding reusable components in less-than-perfect repository structures.

3. WordNet

WordNet, a broad coverage knowledge base of the English lexicon from Princeton University [8], is the linguistic core underpinning LASER. Language is inherently ambiguous, especially at the lexical level, and so WordNet is built around the notion of a *synset*. A synset is an indirect means of denoting a concept or specific word sense, by providing a set of synonyms that can each denote that sense. For our initial experiments with LASER we exploit the large number of synsets defined by WordNet, as well as two semantic relations, *is-a* and *part-of*, that WordNet uses to connect these synsets. For example, `Student` *is-a* `Person` and `Classroom` is *part-of* `School`. WordNet will serve as the knowledge-base component in LASER, allowing the system to predict potential parent classes using lexico-conceptual knowledge. For example WordNet can be used to lookup all synsets containing the word `client` to find synonyms and hypernyms like `Customer` and `Person`, as well as neighbours like `Patient`. This will allow LASER to establish reuse connections between a new class called `Client` and existing classes called `Customer`, `Person` or `Patient`.

4. Analogical Reasoning in LASER

Analogy in its simplest form can be defined as a comparison between pairs and is widely used as a problem solving method. The "Structure Mapping" approach in ReBuilder views analogy as a suggestion of a class diagram based on the query diagram [2, 3]. As its name suggests, structure mapping does not consider the lexical labels assigned to elements in the each structure, assuming instead that the meaning of these elements derives not from names but from their structural relationships with other elements. However, in object-oriented code, class structures are usually annotated with meaningful lexical labels for a reason: to allow human comprehension and insight [7]. Analogical reasoning can be used in two of the most useful reuse techniques in object oriented programming, *hierarchical reuse* and *parallel reuse*. The first, *hierarchical reuse*, occurs when one class is an extension of another class. For example, consider a

context in which a programmer is about to extend a super class for a class called `Client`.

```
public class Client extends _
```

By using analogical reasoning we can implement different strategies to suggest new superclass for the class `Client`. Using WordNet as a knowledge base, we can suggest a superclass that has similar meaning as `Client`, for example `Person`, `Customer` etc.

Analogical reasoning also facilitates the second, and the most ambitious, form of reuse, *parallel reuse*, in which the system recognizes that a developer is about to implement a component that already exists in a similar form. For example, analogical reasoning can be used to determine that a new class, `Client`, is structurally similar to another class called `Patient`, from another application. This recognition of this similarity is both lexical and structural: lexical in the sense that `Client` and `Patient` are taxonomically similar terms in WordNet, occupying neighboring area of the taxonomy under a common parent class called `Person`; and structural in the sense that similarity can be measured by recognizing common class members (such as variables with lexically similar names and types) and by recognizing the lexically similar names of any superclasses.

A major use of analogical reasoning is to perform analogical transfer (also called *candidate inference*), in which a target structure is enriched with structure that is mapped from a more elaborate source. In most structure-mapping systems, transfer is performed using structural criteria only, which can lead to incongruous completion patterns. We intend to conceptually ground the transfer process by making it lexically-driven: a combination of lexical and structural evidence will be required to motivate any transfer of structure from a past design to the current development context.

In order to explain the potential of analogical reasoning in software reuse, we present an example of how such reasoning might be performed. Consider a context, in which a developer begins to create an application for a School, starting with the creation of a class called `Student`. This nascent basis may allow for an existing software design for a Hospital to be retrieved, based on the similarity (in lexical and structural terms) of the `Student` and `Patient` classes. Structure-mapping between the nascent elements of the school system and the existing case for a hospital might then suggest that the school system needs classes for `Desk, Room, Class, Teacher, Principal` and `TeachingAssistant`. LASER may create stubs for these classes automatically and add them to the current project specification. However, while there is lexical evidence for the utility of a `Room` class in the School domain, there is no such evidence for the necessity of a `Desk`, `Class`, `Nurse` or even `Doctor` class. Therefore, LASER may also suggest to the developer to create a `Room` class only.

The structural evidence for a `Nurse`, `Doctor` and `Bed` class is not entirely discarded, however: it is merely put to one side to await a more appropriate reuse opportunity. For example, should the developer begin to create a class for `Teacher`, lexical analysis using WordNet will reveal that this class corresponds to a known taxonomic subordinate of `Professional`, as does `Doctor`. At this point it becomes sensible for LASER to suggest to the developer that the `Doctor` class be reused. Since this is an example of *parallel reuse* (`Teachers` are like `Doctors`, but not a kind of `Doctor`), LASER can suggest two alternative strategies: first, a new generic class `Professional` can be created, and the bulk of the `Doctor` class can be lifted to this new parent class, so that it can be reused via class extension by the new class `Teacher`; or secondly, the structure of the `Doctor` class can be directly imported to provide a code skeleton around which to create the `Teacher` class. In a similar way, the class for `Bed` can be reused as the basis for a new class `Desk`, and if the conceptual structure is available to make such a suggestion, `Nurse` can be reused as the basis for `TeachingAssistant`.

5. Case Study

In this section we demonstrate that *hierarchical reuse* and *parallel reuse* in object oriented code can be enhanced through lexically-driven analogy. Our fundamental assumption is the belief that most programmers follow natural language conventions in naming classes and methods. As a pilot study, the *JRefactory* package [1], which contains 1259 classes, was parsed for the purpose of collecting and storing class and method names. We used WordNet to lexically ground class names and method names used by the programmer. The distinction between names that are linguistically sensible and those that are not is a fuzzy one: many names contain a linguistic root that can be extracted with some basic language heuristics. To calculate the accuracy of such a heuristic WordNet match with a particular class or method name, we initially decompose the name using the capitalisation convention used as standard in Java We have also included some common Java words in our experiment that are not part of standard WordNet database: *AST, int*, etc. An example of how we interpret a class name is as follow:

Class Name	Match after decomposition			Average match
MyBeerCase	My=100%	Beer=100%	Case=100%	100%
MyBeerPZK	My=100%	Beer=100%	PZK=0%	66.66%

Table 1: Average accuracy match for each class with WordNet

Once names have been decomposed following the capitalisation convention, we then compare each component word against the WordNet synset index and calculate an average accuracy for each name as follows:

$$P_{avg} = \sum_{i=1}^{N} (f_e/N) \quad (1)$$

Where P_{avg} is Accuracy of match per class, f_e is accuracy of match with WordNet for each Word and it could be either 0 or 100 and N=Number of Words.

For MyBeerCase it is (100+100+100)/3=100%.
For MyBeerPZK it is, (100+100+0)/3= 66.66%. The final result is calculated as follows.

$$R_{avg} = \sum_{i=1}^{N} (P_{avg}/N) \quad (2)$$

Where R_{avg} is the average accuracy of percentage match with WordNet.
P_{avg} is Accuracy of match per class and N is Total number of class,1259 in this case.
The following graphs depict the result of these techniques applied to *JRefactory* as a whole:

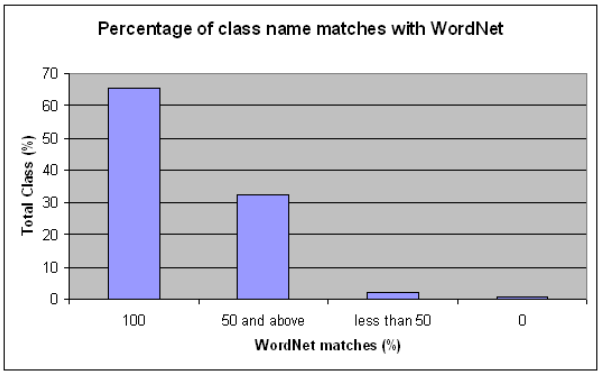

Figure 1. Class Name match with WordNet

The graph above displays the percentage of class names that correspond to WordNet lexical entries. Encouragingly, out of the total number of 1259 classes, an average match of 84.96% with WordNet entries was calculated. We group class name matches into four categories as shown in figure 1. 65.66%, of the classes had a 100% match with WordNet. This was significantly greater than the 0.6% of classes that had no match at all with WordNet.

Figure 2 shows the percentage of method names that correspond to WordNet entries. *JRefactory* contains 6966 methods, and of these, 74.33% match with WordNet entries. The vast majority (60.70%) achieves a 100% match

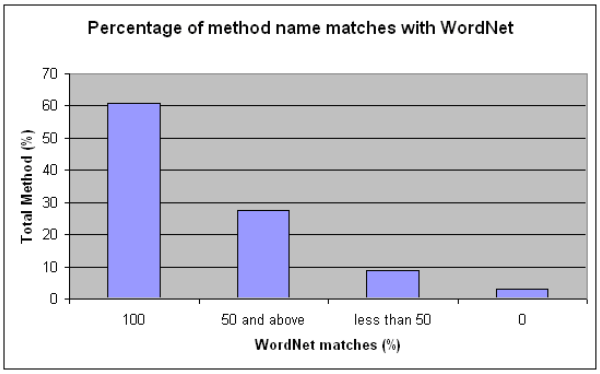

Figure 2. Method Name Match with WordNet

with WordNet; while very few (2.97%) has no discernable mapping to WordNet at all.

Although this is as yet only a pilot study, the results are very promising as regards the goals the LASER project. The vast majority of class names and their lexical components, and a strong majority of method name components, are amenable to conceptual annotation using linguistic techniques.

It is not enough that code-level names correspond to known words; these names must be used in ways consistent with their meaning so that the conceptual basis of the code can be discerned. Ideally, when subclass and its superclass can both be mapped to WordNet, we should expect the corresponding synsets to explicitly relate via an is-a relationship.

To test this expectation, we performed another experiment to consider the *hypernyms* of the lexical labels associated with class names. Hypernym is linguistic term for a word whose meaning includes the meanings of other words, as the meaning of `Transportation` includes the meaning of `Airplane`, `Train` and `Automobile`.

To conduct this experiment we parsed 1259 classes from the open source *JRefactory* [1] project. We completed a study to see if the subclass name and its superclass name have a *hypernym* relationship or not. We explore WordNet to conduct this experiment. Results are calculated as follows: Suppose subclass name is `MySchool` and superclass name is `Organization`. We developed a technique to decompose the names using the capitalization standard used in Java. We use WordNet to search for the *hypernym* relationship between linguistic-head of subclass name(`School` in this case) and those of superclass name(`Organization` in this case). If we find *hypernym* relationship then we associate it with 100% match otherwise with 0% match.In this case it is 100% match. Results are shown in Table 2.

	Total class	**Hypernym relationship** (%)	**No relationship** (%)
Classes	1259	84.51	15.49

Table 2: Hypernym Relationship between subclass and superclass

We focused only on subclasses and their associated superclass. We discovered 1259 superclasses in the *JRefactory* project; of these 84.51% had a *hypernym* relationship with their subclasses. Based on this impressive result, we can argue that a subclass and its superclass are lexically similar and if also, future experiment prove structural similarity, as we suspect, then this presents as with a real opportunity for future *parallel reuse*.

6. Conclusion and Future Work

Analogical Reasoning enables a software environment to provide automated support for software reuse. Lexically-driven analogy at the code level can be used for better understanding of the developer's conceptual goals. In this paper we presented ongoing work on the concept of lexically-driven analogy. We have shown that both *hierarchical* and *parallel reuse* can be enhanced using lexical similarities of class names.

Future work on LASER will be focused on the implementation of techniques for suggesting a super-class for the current class under development using lexically-driven analogy. In our next case study we will consider variables, calculating the *part-of* relationship between a variable name and its class name. LASER will also facilitate intelligent refactoring, based on lexico-conceptual understanding of the software being developed.

7. Acknowledgement

The authors would like to acknowledge the financial contribution of Faculty of Science, University College Dublin to this project.

References

[1] *JRefactory, An Open Source Refactoring Tool for Java*. http://jrefactory.sourceforge.net.

[2] P. Gomes, F. Pereira, P. Paiva, J. Ferreira, and C. Bento. Case retrieval of software designs using wordnet. *ECAI-2002, the Fifteenth European Conference on Artificial Intelligence*, 2002.

[3] P. Gomes, F. Pereira, P. Paiva, N. Seco, J. Ferreira, and C. Bento. Experiments on case-based retrieval of software designs. *ECCBR-2002, the 2002 European Conference on Case-Based Reasoning*, 2002.

[4] M. T. Harandi. The role of analogy in software reuse. *ACM symposium on applied computing states of the art and practice*, 1993.

[5] S. Henninger. An evolutionary approach to constructing effective software reuse repositories. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1997.

[6] C. W. Krueger. Software reuse. *ACM Computing Surveys (CSUR), Volume 24 Issue 2*, 1992.

[7] H. E. Letha and G. D. Carl. Automatically identifying reusable oo legacy code, computer. 1997.

[8] G. A. Miller. *WordNet,Cognitive Science Laboratory, Princeton University*.

[9] R. Prieto-Diaz. Status report- software reusability. *IEEE Software, v10 n3*, May 1993.

[10] W. Tracz and S. Edwards. Implementation working group report. *Reuse In Practice Workshop, Software Engineering Institute, Pitt, Pa*, 1989.