Case-Based Reuse of Software Examplets*

Markus Grabert & Derek Bridge
University College Cork
Ireland
m.grabert/d.bridge@cs.ucc.ie

Abstract: We present a software tool for examplet reuse. We define examplets to be goal-directed snippets of source code, often written for tutorial purposes, that show how to use program library facilities to achieve some task. Our tool allows users to specify both their goal (in free text) and their 'situation' (the source code on which they are working). The system combines text retrieval and spreading activation through a semantic net representation of the source code.

1 Introduction

The research we report in this paper is concerned with retrieval of reusable components. Like a lot of the research into software-supported reuse, we draw ideas from Case-Based Reasoning (CBR). The CBR-cycle [AP94], retrieve-reuse-revise-retain, has obvious parallels with the processes involved in software reuse [TA97].

In Section 2, we describe examplets, which are the reusable components that our system stores and retrieves. Section 3 describes the architecture and operation of our system for examplet retrieval, explaining both the text retrieval and semantic net retrieval. In Section 4, we present the results of some experiments with the system. We describe related research in Section 5.

2 Examplets

Modern programming languages, especially object-oriented languages, make use of large libraries of reusable components (e.g. class definitions). We want to make it easier for programmers to make use of the resources contained in these libraries.

In many CBR systems for software reuse, each class definition in the library is treated as a case. But cases are supposed to have characteristics that class definitions in a library do not. "A case is a contextualized piece of knowledge representing an experience that teaches a lesson fundamental to achieving the goals of the reasoner." [Ko93].

*This research was funded in part by grant ST/2000/092 from Enterprise Ireland.

Examplet Goal Text

How to read directly from a URL using BufferedReader

Examplet Source Code

```
import java.net.*;
import java.io.*;
public class URLReader
{  public static void main(String[] args) throws Exception
   {  URL yahoo = new URL("http://www.yahoo.com/");
      BufferedReader in = new BufferedReader(
         new InputStreamReader(yahoo.openStream()));
      String inputLine;
      while ((inputLine = in.readLine()) != null)
         System.out.println(inputLine);
      in.close();
   }
}
```

Figure 1: An Examplet

The cases in our case base live up to the definition given in the previous paragraph. Each of our cases contains a representation of what we call an *examplet*. An examplet has two parts. One part is a snippet of source code, in our case in Java. This snippet shows how to accomplish a task in Java using library components. Crucially then, it shows library components *in use*. Each examplet is goal-directed, and so the other part of an examplet is a statement of the goal in free text. One of our smaller examplets is shown in Figure 1.

Examplets are widely available, e.g. [Ch99]. They capture HOWTO knowledge; each might also be thought of as a kind of FAQ. Each is hand-crafted, which tends to ensure that it addresses programmer needs.

3 A Software Tool that Recommends Examplets

3.1 Overview

The system that we have developed helps programmers to solve common problems by recommending the HOWTO knowledge embodied in a case base of examplets. We expect programmers who use such a system to be actively writing their program, and then to find that they have some quite specific goal which they are uncertain how to solve.

As we have seen, each examplet contains a free-text statement of the problem that it solves, the *examplet goal text*. The user will express her goal, the *query goal text*, also in free-text. Standard text retrieval techniques can be used to retrieve relevant examplets. We describe

the design of this part of our system in a little more detail in Section 3.2.

But, if the programmer is actively writing her program, then she can tell us, not only what she is looking for, but also what she has already. In addition to a goal text, her query can contain some or all of the source code that she has written already. By default, this source code would be the class definition that the user is currently editing; but a user might instead explicitly highlight a section of source code.

So in addition to doing text retrieval on goal texts, our system will attempt to match *query source code* with *examplet source code* (the snippets of code in the examplets). This matching is done using spreading activation in a semantic net. It is described in more detail in Section 3.3. We believe that this makes our system more faithful to strong conceptions of CBR. The user's problem (query) is described by both a goal and a 'situation'.

3.2 Text Retrieval for the User's Goal

For text retrieval, we are using a modified version of `ht://Dig`[1]. This is an open-source search engine, written in C/C++, designed for use with Web sites.

Given a set of cases, one per examplet, we use `ht://Dig` to produce an inverted index to the goal texts. Index entries are produced using word stemming and exclude a list of stop words.

For retrieval, we provide `ht://Dig` with a thesaurus.[2]

The query goal text, after word stemming and the removal of words from the stop list, is treated purely conjunctively. Cases are scored by counting how many word stems or their synonyms in the query match word stems in the cases.

3.3 Semantic Net Retrieval for the User's Situation

We decided to express essential aspects of the structure of each snippet of code using a semantic net. We placed two requirements on the process of constructing and activating the net from code snippets:

- It should be wholly automatic. This allows the easy incorporation of new examplets into the case base.
- It should be as robust as possible in the face of incompleteness or ill-formedness in the source code. This is needed for two reasons. Firstly, the examplet author may elide code that is unimportant to the lesson conveyed by the examplet. Secondly, since the query source code is still under development, it will typically be incomplete and may not yet compile.

[1] http://www.htdig.org/

[2] We based this on data extracted from WordWeb (`http://wordweb.info/`), a free cut-down version of WordWeb Pro.

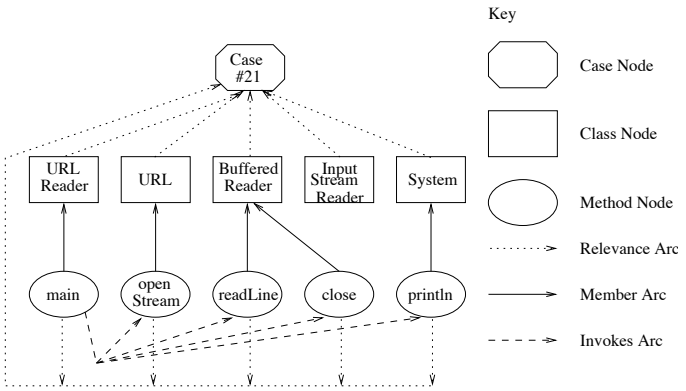

Figure 2: Semantic Net Fragment

Our approach is to use a parser, and to build the net from parse trees. We used the ANTLR translator generator[3], which comes with a Java grammar. We modified the parser that ANTLR generated so that, even in the face of compiler-errors, it would still output a parse tree, which would contain as much of the source code's token stream as possible.

Our net is constructed by walking the parse tree. It contains five kinds of node: *case*, *class*, *interface*, *method* and *variable*. And it contains five kinds of relationship (although their semantics currently plays no part in the retrieval): *relevance*, *subclass*, *implements*, *member* and *invokes*. A fragment of the net, corresponding to the examplet in Figure 1, is shown in Figure 2. (Nodes for the `String` and `Exception` classes have been omitted in the interests of compactness.)[4]

The source code in the examplets is used to *construct* the net. The query source code, by contrast, is used to *activate* the net. The query source code is parsed and the parse tree is walked in search of identifiers. For each class identifier, all class nodes for that identifier are activated. For each class variable or instance variable declaration, all variable nodes for the same identifier and type are activated. For each method identifier, all method nodes for the same identifier and signature (including return type) are activated.

In fact, this initial activation does not exclusively use identifier equality. We use an inexact string matching algorithm to compare identifiers in the query source code with node labels in the semantic net. The initial activation is multiplied by the degree of similarity, [0,1]. The current implementation of inexact string matching is simplistic: it is computed as the size of any common prefix divided by the length of the identifier in the query source code.

The search for relevant case nodes (examplets) is implemented by spreading activation through the net. At each time point, each node spreads a proportion of the activation that it received at the previous time point to all of its immediate neighbours. We spread only a proportion (presently 0.7) to simulate the idea that activation decays the further it travels.

[3] http://www.antlr.org/

[4] We give a detailed description of the way we construct the net in the longer version of this paper that appears in the on-line proceedings.

This also forms the basis of a stopping criterion (see below). The amount of activation spread down a particular arc is further modified by multiplying by the arc weight.

A node does not spread any of its new activation if the amount of that activation is less than a threshold amount (presently 0.1). When no node is in a position to spread any activation or when a maximum number of time points has elapsed (currently 150), the spreading activation terminates.

Those case nodes that have received the highest total activation are retrieved.

4 Experimental Results

We collected 40 examplets from the Web. They came from several different sources [5] which reduces the dependence of our results on any one style of examplet. Each examplet comprises between 10 and 120 lines of text.

As well as a snippet of source code, each examplet must have a goal text. Unfortunately, we found that the textual descriptions associated with the original examplets to be unsuitable. Too often, the descriptions were insufficiently goal-oriented. Rather than describing the problem that the examplet solves, they focused on how the lines of code contribute to the solution. We decided, therefore, to replace these by our own goal texts.

Our experimental methodology is that of an *ablation study* and we use the *leave-one-in* methodology [AB97]. Each case in the case base is selected in turn (with replacement); a query is created from the selected case (in the manner described below); and the query is evaluated against the full case base. The query is successful if the case from which it was created is among the top 5 retrieved cases.

We will explain first how we create the query goal text, and then how we create the query source code. We asked three experienced Java programmers to look independently at different subsets of the 40 examplets in our case base. They saw only the source code. For each examplet that they looked at, we asked them to write their own sentence describing the problem to which the examplet would be the solution. By this means, we obtained two query goal texts per case. Here are the goal texts we obtained for the examplet shown in Figure 1:

> *"How to copy from a URL to an output stream"*
> *"How to read from an URL using a BufferedReader"*

In the experiments, when constructing the query, one of the two query goal texts is chosen at random. Stopwords are removed and word-stemming is applied to the chosen goal text. Then a proportion of the text is deleted at random. The remainder is submitted to `ht://Dig`. Our approach loosely simulates users whose query goal texts might be quite fragmentary, perhaps comprising only one or two keywords.

[5] The sources include: `http://java.sun.com/docs/books/tutorial/` and `http://examples.oreilly.com/jenut2/2nd_edition/`

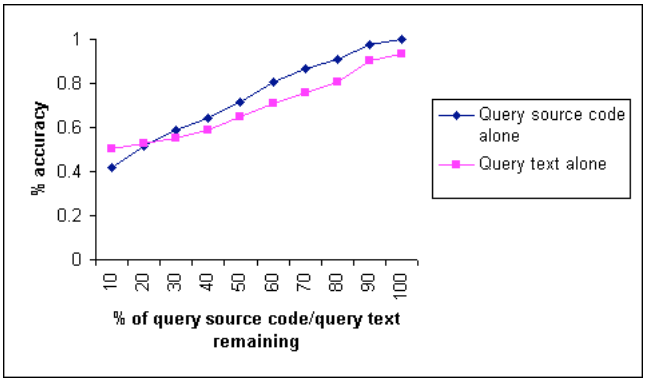

Figure 3: Accuracy for query source code/query text alone

The other part of a query is the query source code, which is used to activate the semantic net. We needed to simulate the idea that the user is working on some class definition when he submits his query. His class definition may therefore be incomplete and even ill-formed. So we delete a randomly-chosen proportion of the nodes in the parse tree and we use the remainder to activate the net.

As we have described, query creation for a given case involves random deletion of portions of the goal texts and source code. This places a requirement that we use cross-validation to ensure we do not report results from unduly favourable or unfavourable random selections. In our experiments, we use 100-fold cross-validation.

Figures 3 and 4 show our results. In particular, Figure 3 plots the retrieval accuracy for each retrieval mode separately. We see that the more query source code or query goal text that is supplied (i.e. the less that gets ablated) the higher the retrieval accuracy. Source code retrieval has marginally the poorer performance when there is most ablation, but it climbs slightly more steeply, and achieves 100% retrieval accuracy, which goal text retrieval does not do. However, our experimental results for source code retrieval may be better than they would be in practice: random ablation of an examplet's source code will result in query source code that is still structurally quite similar to the original examplet, especially at lower levels of ablation.

The results in Figure 4 are obtained by combining the retrieval scores from the two forms of retrieval using a weighted average, where the two forms of retrieval are weighted equally (both 0.5). Of course, this does not guarantee that the two forms of retrieval are being treated equally, since the normalisation of the scores may be imperfect. We have tried other weighting schemes (not shown in this paper); the results are not much different.

For our 40 examplets, the semantic net contains approximately 340 nodes and 480 arcs. The system is written in Java. Running the Java 1.3 interpreter on a 1GHz Pentium3 with 256MB RAM, it takes approximately 10 seconds on average to run a single query, of which slightly over half is the time to run our modified parser.

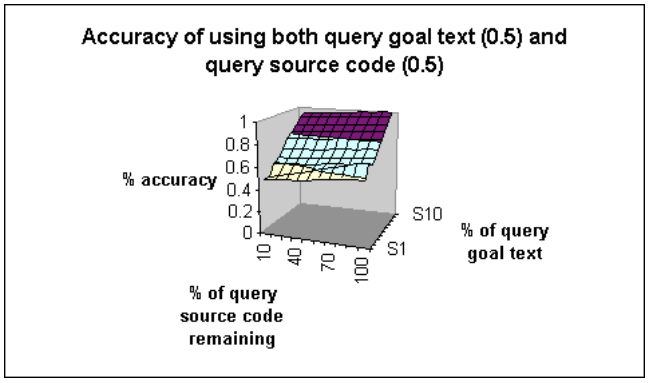

Figure 4: Combined Results

5 Related Work

The literature reports numerous systems that have been built to support software reuse. Approaches vary widely.[6]

One of the more concerted efforts has been conducted by Gomes and others at the University of Coimbra in Portugal. In the earlier work [GB99] [GB00] the emphasis was on a quite deep representation of software components. Specifically, they used what they called a Function-Behaviour Case Representation, attempting to express both the 'what' and the 'how' of the component. Attention, however, was confined to cases written in VHDL, a simple hardware description language.

In later work [Go01] [Go02a] [Go02b], their attention has moved to software design. Cases represent designs and design patterns expressed as class diagrams in the Unified Modeling Language (UML). Similarity-based retrieval exploits the identifiers (class, attribute and method names) and the structural relations in the UML diagrams. Semantic relations between identifiers can be found by using WordNet. Once candidate cases have been retrieved in this fashion, a heuristically-guided structural mapping algorithm sets up correspondances between the user's partial design and the retrieved cases. The work is unusual in providing some support for automatic adaptation of the user's design: the system has procedural knowledge that enables it to attempt to apply a retrieved design to the user's design.

[6]The longer version of this paper, which appears in the on-line proceedings, contains a more comprehensive review of the literature.

6 Conclusions

We have presented a tool for retrieval of software examplets. The user can specify both her goal (as text) and her current situation (the code that she has been writing). The system uses textual retrieval and spreading activation in a semantic net to achieve promising results.

In future work, we wish to take a broader view, supporting design-oriented activities as well as coding ones. We would expect, however, to continue to pursue the idea of retrieval based on both user goal and situation.

References

[AB97] Aha, D.W. & Breslow, L.A.: Refining Conversational Case Libraries, in D.B.Leake & E.Plaza (eds.), *Procs. of the Second International Conference on Case-Based Reasoning*, LNAI 1266, pp.267–278, Springer, 1997.

[AP94] Aamodt, A. & Plaza, P.: Case-Based Reasoning: Foundational Issues, Methodological Variants, and System Approaches, *Artificial Intelligence Communications*, vol.7(1), pp.39–59, IOS Press, 1994.

[Ch99] Chan, P.: *The Java Developers Almanac 1999*, Addison-Wesley, 1999

[GB99] Gomes, P. & Bento, C.: Automatic Conversion of VHDL Programs into Cases, in S.Schmitt & I.Vollrath (eds.), *Procs. of the Workshop Programme at the Third International Conference on Case-Based Reasoning*, 1999.

[GB00] Gomes, P. & Bento, C: Learning User Preferences in Case-Based Reuse, in E.Blanzieri & L.Portinale (eds.), *Procs. of the European Workshop on Case-Based Reasoning*, LNAI 1898, Springer, pp.112–123, 2000.

[Go01] Gomes, P., Pereira, F.C., Bento, C. & Ferriera, J.L.: Using Analogical Reasoning to Promote Creativity in Software Reuse, in R.Weber & C.G.von Wangenheim (eds.), *Procs. of the Workshop Programme of the Fourth International Conference on Case-Based Reasoning*, pp.152–158, 2001.

[Go02a] Gomes, P., Pereira, F.C., Paiva, P., Seco, N., Carreiro, P., Ferriera, J.L. & Bento, C.: Case Retrieval of Software Designs using WordNet, in F.van Harmelen (ed.), *Procs. of the 15th European Conference on Artificial Intelligence*, pp.245–249, 2002.

[Go02b] Gomes, P., Pereira, F.C., Paiva, P., Seco, N., Carreiro, P., Ferriera, J.L. & Bento, C.: Using CBR for Automation of Software Design Patterns, in S.Craw & A.Preece (eds.), *Procs. of the Sixth European Workshop on Case-Based Reasoning*, LNAI 2416, Springer, pp.534–548, 2002.

[Ko93] Kolodner, J.: *Case-Based Reasoning*, Morgan-Kaufmann, 1993.

[TA97] Tautz, C. & Althoff, K.-D.: Using Case-Based Reasoning for Reusing Software Knowledge, in D.B.Leake & E.Plaza (eds.), *Procs. of the Second International Conference in Case-Based Reasoning*, LNAI 1266, Springer, pp.156–165, 1997.