Journal of Universal Computer Science, vol. 9, no. 7 (2003), 627-640
submitted: 3/2/03, accepted: 21/7/03, appeared: 28/7/03 © J.UCS

Case-Based Reuse of Software Examplets

Markus Grabert
(University College Cork, Ireland
m.grabert@cs.ucc.ie)

Derek Bridge
(University College Cork, Ireland
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.

Key Words: software reuse, case-based reasoning, retrieval
Category: D.2.13, I.2.4

1 Introduction

It has long been an aspiration of the software industry that software development should proceed, at least in part, by a process of reuse of components. The anticipated benefits are improvements in programmer productivity and in software quality. A variety of innovations may be slowly making this aspiration into a reality. These include: the encapsulation and information hiding afforded by object-oriented programming languages; extensive libraries of components (especially of class definitions for object-oriented programming); component-based software; design patterns; and software frameworks.

Compositional software reuse consists of processes such as: identifying reusable components; describing the components; retrieving reusable components; adapting retrieved components to specific needs; and integrating components into the software being developed [Smolárová and Návrat 1997]. These are difficult processes, made more difficult by the high volume of reusable components with which a software developer must ideally be acquainted.

Over the last 15 years, researchers have been looking at ways of providing software support to programmers engaged in software reuse ([see Section 5]). Their efforts have mostly been concerned with the retrieval of reusable components (especially source code) from repositories. Search engines can scan repositories much more quickly than the human programmer can. The challenge, of course, is to equip the search engine with ways of recognising which of the items it visits have the potential to fulfil the user's needs. Processes other than retrieval have largely not been the subject of these research efforts. It is left to the human programmer to, for example, adapt the retrieved components and integrate them into her software.

The research we report in this paper is likewise 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 [Aamodt and Plaza 1994], retrieve-reuse-revise-retain, has obvious parallels with the processes involved in software reuse [Tautz and Althoff 1997].

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 libraries of reusable components (e.g. class definitions). These libraries are large. In the case of Java, for example, the standard class library (SDK 1.4) alone contains approximately 3000 class definitions and Java interfaces.

We want to make it easier for programmers to make use of the resources contained in these libraries. This may be especially helpful for novice programmers, whose familiarity with the contents of even standard libraries may be low. However, experienced programmers sometimes find themselves in the position of novices: when the software they are developing requires knowledge of technologies with which they are less familiar.

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." [Kolodner 1993].

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 [Fig. 1].

Examplets are widely available, both in printed form and on the World Wide Web, e.g. [Chan 1999]. They capture HOWTO knowledge; each might also be

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

thought of as a kind of FAQ. Each is hand-crafted, which tends to ensure that it addresses programmer needs. The effort of crafting examplets is borne by library authors and others interested in promoting use of the library.

Examplets facilitate reuse at two levels. On the one hand, they direct the attention of a programmer to the facilities provided within a library, which encourages reuse of those facilities. (The provision of hyperlinks within examplets to the library API can increase the likelihood of this.) On the other hand, they show a typical usage pattern, involving the co-ordinated use of multiple library facilities. Programmers may be able to adapt the usage patterns expressed in these stretches of source code to their own 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, due to, e.g., lack of familiarity with the language facilities or forgetfulness, they are uncertain how to solve. One programmer, for example, might not know how to define and export a remote object; another might need to be reminded how to open a text file for reading.

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].

In allowing the user to provide in his query a statement of what he is looking for (the query goal text), our system is no different from numerous other search engines, including ones that have been built to support software reuse. However, we had a suspicion, borne out by the results of experiments ([Section 4]), that matching the goal texts in the query and examplets would not alone give especially good results.

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 explicitly highlight a section of source code, e.g. the context that surrounds the part of the code that she does not yet know how to write.

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` (`http://www.htdig.org/`). 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. The thesaurus we use is based on data extracted from WordWeb (`http://wordweb.info/`), a free cut-down version of WordWeb Pro.

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 have to store each examplet's source code, if only so that it can later be displayed to the user. To support retrieval, we could have chosen to treat the source code as raw text and built an index to it using `ht://Dig`. We did not think this was appropriate for several reasons:

- Programming language keywords recur and so are likely to have low predictive power.
- Identifiers in examplets tend to be short and relatively non-descriptive. For example, a variable that references a button might be called simply `b`. Examplet authors can justify this practice because examplets are often short and are not situated in the context of a larger software system. But, the non-descriptiveness of these identifiers reduces the likelihood of true hits.
- Even when identifiers are meaningful, the user and the examplet authors may use a variety of idiosyncratic naming schemes. For example, a variable that holds a unique, numeric student identifier might be called any of `studentId`, `studentNum`, `studNum`, `stdntNo`, etc. These variations reduce the likelihood of true hits.
- A purely textual approach ignores the potentially valuable structural information conveyed by source code (e.g. class membership, message sending, subclassing, etc.).

We decided, in our system, to extract some structure from each snippet of source code and use this, rather than the raw source code, for retrieval.

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, ellipsis is common in examplets: the 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.

Our approach is to use a parser, and to build the net from parse trees. We used the ANTLR translator generator (`http://www.antlr.org/`), 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, and this parse tree 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*.

– A case node is constructed for each examplet.

– Class nodes and interface nodes are created for each unique class identifier or Java interface identifier, wherever it is encountered.

– Method nodes are created for each unique method identifier (whether encountered in method headers when defining a method or in blocks of code when invoking a method). A method identifier does not qualify for a new node if and only if there already exists a method node for the same method name, the same signature (including return type) within the same class or interface.

– Finally, variable nodes are created for each declaration of an instance variable or class variable. (We ignore formal parameters and local variables, and we consider only variable declarations, not variable accesses.)

Our net contains five kinds of relationship (although their semantics currently plays no part in the retrieval): *relevance*, *subclass*, *implements*, *member* and *invokes*.

– The net contains a relevance-arc between a case node for a particular examplet and each of the class, method and variable nodes that would be created from its source code.

– Where the source code declares that one class or interface extends another or a class implements an interface, the corresponding case nodes are linked with subclass- or implements-arcs, as appropriate.

– Class and interface nodes are linked by member-arcs to the nodes for their members. Possible members are: inner classes; variables declared in the class or interface; and methods defined within the class or interface.

– When a method body contains a statement that invokes another method, two kinds of arc are created, if possible. Firstly, there will be an invokes-arc between the two method nodes (the client method and the method being invoked). Secondly, an attempt will be made to link the node for the invoked method to its class or interface node using a member-arc, if such an arc does not already exist. If the method is invoked implicitly or explicitly by sending a message to `this`, the method is linked to the containing class node. If the method is invoked by sending a message to a variable that contains a reference to some object, then the type of that variable is determined from the source code, if possible, and this gives the class or interface node to which this method is linked. (Note that this has at least two limitations. Firstly, the code may be incomplete, so the variable declaration may not be present in the snippet, in which case no arc can be created. Secondly, Java's dynamic method binding means that the type of the variable in the source code may not fully determine the method's class.) Finally a class method can be invoked by prefixing the call with the class name, enabling the method and its class to be straightforwardly linked.

The arcs are given weights, initialised to 1 on creation. The weights are increased (currently by a factor of 1.2) for each time that the relationship is repeated in the source code (e.g. if a method body contains more than one invocation of some other method).

With these rules, our net is a good, pragmatic approximation of the source code structure. Due in particular to the possibility of incompleteness or ill-formedness in the source code, it may not be wholly faithful to the intended semantics of the code. Furthermore, our current implementation ignores, for example, the namespaces given by Java packages and the role of Java `import` statements. This can mean that equal identifiers from different packages might be incorrectly represented by a single node in the net. But we believe the quality of analysis that we get is good enough for the kind of retrieval that our system supports.

A fragment of the net, corresponding to the examplet in [Fig. 1], is shown in [Fig. 2]. (Nodes for the `String` and `Exception` classes have been omitted in the interests of compactness.)

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.

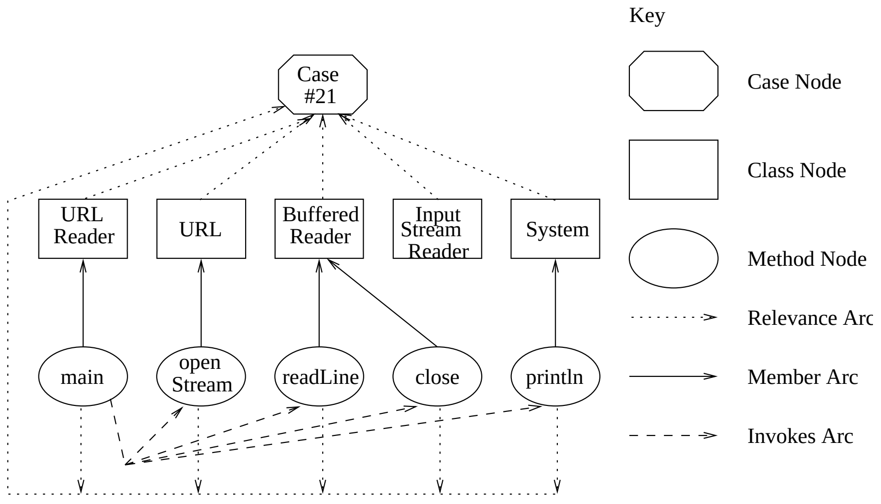

Figure 2: Semantic Net Fragment

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. 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 (including `http://java.sun.com/docs/books/tutorial/` and `http://examples.oreilly.com/jenut2/2nd\_edition/`) 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 write our own goal texts, and we use only these in the experiments.

Our experimental methodology is that of an *ablation study* and we use the *leave-one-in* methodology [Aha and Breslow 1997]. 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, and we measure the proportion of times this happens.

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 [Fig. 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.

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.

[Fig. 3] and [Fig. 4] show our results. In particular, [Fig. 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

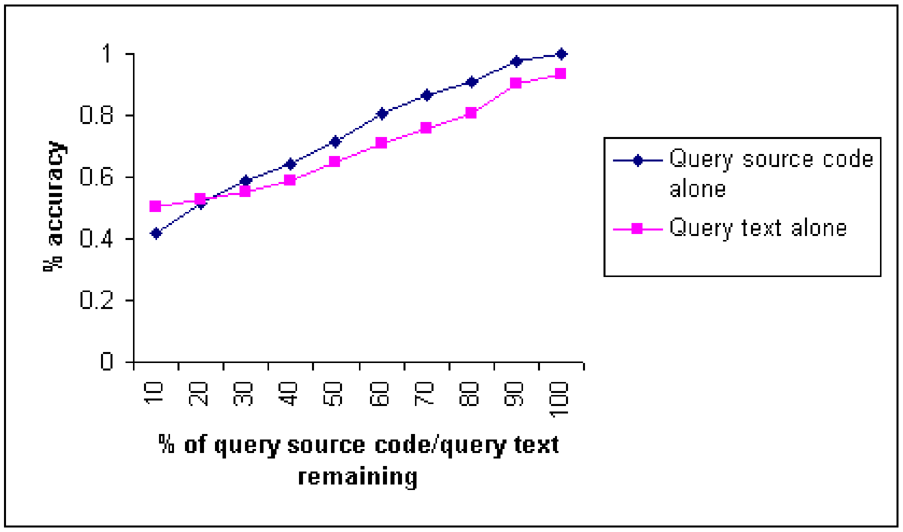

Figure 3: Accuracy for query source code/query text alone

result in query source code that is still structurally quite similar to the original examplet, especially at lower levels of ablation.

The results in [Fig. 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. An optimised and compiled version of the system would run much faster. It might even be possible to obtain a negligible response time if we were to redesign the system to work in an incremental 'any-time' fashion as a background activity.

5 Related Work

The literature reports numerous systems that have been built to support software reuse. Approaches vary widely. There are those based purely on textual retrieval. For example, in [Maarek et al 1994], software documentation (comments

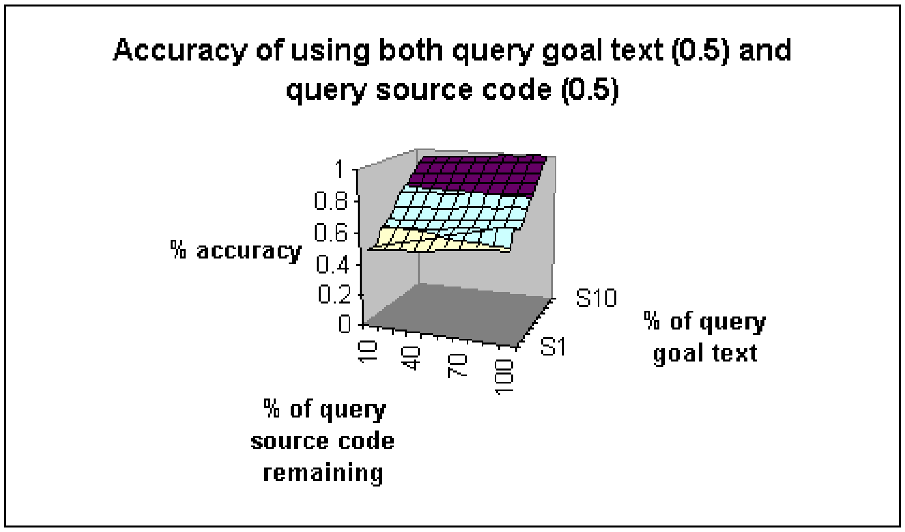

Figure 4: Combined Results

and manuals) are indexed (having regard for lexical affinities and statistical distributions) to allow the use of standard IR techniques.

In [Prieto-Díaz and Freeman 1987], software components themselves are described using sextuples of facets (features) whose values are drawn from expert-defined controlled vocabularies. They are classified by these facets, and the classes are assigned, by the experts, into a conceptual distance graph. User queries also take the form of sextuples. A similar, but perhaps more flexible approach, is reported in [Ostertag et al 1992]. In neither the IR approaches, nor these classification approaches, is there any real representation of the content of the code itself.

The LaSSIE system [Devanbu et al 1991] uses a system of frames to represent a large software system. There is an emphasis on representing the system's actions. The knowledge base is produced manually, which is an intensive task. User queries can also be expressed as frames with unfilled slots or in natural language. We think it an advantage of our approach, by contrast, that the semantic net is produced wholly automatically, and that queries can comprise code as well as text.

By far the greatest amount of related work uses CBR (focusing on case retrieval). An ambitious CBR system, for example, is proposed in [Fernández-Chamizo et al. 1996]. The system design combines text retrieval on component documentation with similarity-based retrieval on a case base of software components represented in LOOM. The cases represent classes, methods and what are described as ‘cookbook recipes’. Cookbook recipes may well correspond to what we are calling examplets. The LOOM representation captures much the same kind of structural information that we extract from our parse trees. However, certain components of their representation, especially those concerned with case justification, cannot be created automatically.

In [Tessem et al 1998], information about a repository of Java class definitions is extracted using Java’s reflection facilities, and this information is used to index the repository. In addition, knowledge engineers can encode information about abstract data types (ADTs). Any class whose indexes have a high degree of similarity with the features of the ADT will be indexed by that ADT. A user’s query is a possibly incomplete class interface. A potential weakness is that any user who can specify her query sufficiently in this way is probably knowledgeable enough to know which class definitions in the repository are relevant and so may find the system of limited value.

One of the more concerted efforts has been conducted by Gomes and others at the University of Coimbra in Portugal. In the earlier work [Gomes and Bento 1999] [Gomes and Bento 2000] 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 [Gomes et al 2001] [Gomes et al. 2002a] [Gomes et al. 2002b], 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.

CBR has also been used at a corporate level to support organisation learning in software development projects [Althoff et al. 1998] [Jedlitschka et al. 2001]. This work uses CBR to give a concrete realisation of the idea of an Experience Factory [Basili et al. 1994]. This work obviously addresses somewhat broader goals than our own.

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.

Acknowledgements

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

References

[Aha and Breslow 1997] 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.

[Althoff et al. 1998] Althoff, K.-D., Birk, A., von Wangenheim, C.G. & Tautz, C.: CBR for Experimental Software Engineering, in M.Lenz, B.Bartsch-Spörl, H.-D.Burkhard & S.Wess (eds.), *Case-Based Reasoning Technology: From Foundations to Applications*, LNAI 1400, Springer, pp.235–254, 1998.

[Aamodt and Plaza 1994] 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.

[Basili et al. 1994] Basili, V.R., G. Caldiera & H.D. Rombach: Experience Factory, in J.J. Marciniak, *Encyclopedia of Software Engineering*, vol.1, pp.469–476, Wiley, 1994.

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

[Devanbu et al 1991] Devanbu, P., Brachman, R.J., Selfridge, P.G. & Ballard, B.W.: LaSSIE: A Knowledge-Based Software Information System, *Communications of the ACM*, vol.34(5), pp.34–49, 1991.

[Fernández-Chamizo et al. 1996] Fernández-Chamizo, C., González-Calero, P.A., Gómez-Albarrán, M. & Hernández-Yáñez, L.: Supporting Object Reuse Through Case-Based Reasoning, in I.Smith & B.Faltings (eds.), *Procs. of the Third European Workshop on Case-Based Reasoning*, LNAI 1168, Springer, pp.135–149, 1996.

[Gomes and Bento 1999] 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.

[Gomes and Bento 2000] 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.

[Gomes et al 2001] 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.

[Gomes et al. 2002a] 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.

[Gomes et al. 2002b] 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.

[Jedlitschka et al. 2001] Jedlitschka, A., Althoff, K.-D., Decker, B., Hartkopf, S. & Nick, M.: Corporate Information Network (COIN): The Fraunhofer IESE Experience Factory, in R.Weber & C.G.von Wangenheim (eds.), *Procs. of the Workshop Programme of the Fourth International Conference on Case-Based Reasoning*, pp.9–12, 2001.

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

[Maarek et al 1994] Maarek, Y., Berry, D.M. & Kaiser, G.E.: GURU: Information Retrieval for Reuse, in P.Hall (ed.), *Landmark Contributions in Software Reuse and Reverse Engineering*, Unicom Seminars, 1994

[Ostertag et al 1992] Ostertag, E., Hendler, J., Prieto-Díaz, R. & Braun, C.: Computing Similarity in a Reuse Library System: An AI-Based Approach, *ACM Transactions of Software Engineering and Methodology*, vol.1(3), pp.205–228, 1992.

[Prieto-Díaz and Freeman 1987] Prieto-Díaz, R. & Freeman, P.: Classifying Software for Reusability, *IEEE Software*, vol.4(1), pp.6–16, 1987.

[Smolárová and Návrat 1997] Smolárová, M. & Návrat, P.: Software Reuse: Principles, Patterns, Prospects, *Journal of Computing and Information Technology*, vol.5(1), pp.33–49, 1997.

[Tautz and Althoff 1997] 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.

[Tessem et al 1998] Tessem, B., Whitehurst, A. & Powell, C.L.: Retrieval of Java Classes for Case-Based Reuse, in B.Smyth & P.Cunningham (eds.), *Procs. of the Fourth European Workshop on Case-Based Reasoning*, LNAI 1488, pp.148–159, Springer, 1998.