Applying Case-Based Reasoning to Code Understanding and Generation

Andrew Broad[1] and Nick Filer
Department of Computer Science, University of Manchester,
Manchester, England
{broada/nfiler}@cs.man.ac.uk
http://www.cs.man.ac.uk/~broada/

Abstract

This paper briefly reviews the applicability of case-based reasoning (CBR) to code understanding and generation. The paper suggests that case-based techniques are already common in code understanding and generation but are seldom labelled as CBR. Some examples of 'explicit' and 'covert' CBR are briefly examined. It is suggested that extending the existing code understanding and generation methods to make more complete use of CBR may usefully increase the capability of these methods.

1 Introduction

This paper briefly reviews the applicability of case-based reasoning (CBR) to code understanding (and its inverse, code generation). This topic is of interest because it suggests a practical route towards much more powerful software engineering technology where, for example, reuse and interoperability might be managed directly, by systems rather than by human operators.

The rest of this paper is organised as follows:

- Section 2 introduces code understanding and code generation, and discusses the motivations for them.
- Section 3 discusses the benefits of CBR, and why it is appropriate to apply CBR to code understanding and generation.
- Section 4 reviews some of the existing approaches to code understanding and generation from a CBR perspective - both those which claim to take a CBR approach, and those that don't but have some affinity with CBR.

1. This work was supported by the Engineering and Physical Sciences Research Council (EPSRC).

- Section 5 mentions the Authors' current research on constraint understanding, and points out some future directions for code understanding and generation.

2 Code Understanding and Code Generation

A theme of this paper is *code understanding*. In this context, *code* refers to any code written in a formal language. This often means program code, and so code understanding is often called *program understanding* in the literature [1]. However, there are other kinds of code than programs, such as:
- inexecutable formal specifications (e.g. VDM (Appendix B of [2]) and Z [3]);
- SQL queries [4];
- information models (written in, for example, EXPRESS [5] or UML [6]).

To understand code means to extract knowledge from it. The knowledge extracted from code is usually at a higher level of abstraction than the code itself.[2] Usually, the direction of code understanding is from a lower to a higher level of abstraction. Another common name for code understanding is *reverse engineering* [7].

Code understanding has a very important inverse process, *code generation* (or *forward engineering*). This means generating code from knowledge that is at a higher level of abstraction than the code itself. In the extreme, this could mean *automatic programming* [8] - the automatic generation of executable programs from specifications. Again, though, it could mean the generation of code other than program code. By analogy to code understanding, it will generally be assumed that the direction of code generation is from a higher to a lower level of abstraction.

2.1 Motivations for Code Understanding and Code Generation

The motivations for code understanding often relate to *software maintenance*. Software evolves, and in order to modify software to meet new requirements, it is clearly necessary to understand the code. These new requirements may be for extra functionality, improved performance, improved usability, or simply the removal of bugs. It may also be necessary to upgrade software to run on different platforms, to be written in another (e.g. more modern) language, or to use different underlying tools.

A common problem mentioned in the literature [9,10] is that of *legacy systems*. A legacy system is an old piece of software, written in a language such as COBOL which typically runs on an outdated platform. Legacy systems often need to be reimplemented in more modern languages to run on modern platforms. Also, it is not uncommon for vital knowledge to be embedded in legacy code without having been satisfactorily documented elsewhere.

2. Although pedantically it could be argued that compilers extract knowledge that is at a *lower* level of abstraction than the code itself!

The process of *reengineering* consists of a reverse engineering step, followed by a forward engineering step [7]. Reengineering can be used for such tasks as reimplementing legacy systems, restructuring existing code, and translating code from one language to another. Complex reengineering tasks may not be accomplishable by direct transformation of the code; rather, they may require the utilisation of higher-level knowledge about the code. For example, rewriting procedural code in an object-oriented style requires knowledge of the abstractions represented in the code, which can be extracted from the source code using reverse engineering techniques [1].

Software reuse [11] is a very important issue in modern software engineering.

- On the one hand, there are demands for software of increasing size and complexity, which is expected to be written in a tight time-schedule. Software is consequently often not delivered on time, does not meet the clients' requirements, is not adaptable to changing circumstances, and has errors that are not discovered until after the software is released - an affliction commonly known as the *software crisis* (Chapter 1 of [2]), although it has existed since the 1960s!
- On the other hand, there is a rich legacy of existing software components from the history of programming, in which commonly-recurring software problems are solved again and again. The idea is that if these components could be reused in new code, it should improve programmer productivity and save having to 'reinvent the wheel'.

  Unfortunately, software reuse has not been practised as widely as those who preach it may hope. Some of the reasons for this are:
  - The components that would be desirable to reuse are often not in a form where they can be plugged in without modification. They may not be conveniently partitioned into subroutines that can just be called, or objects that encapsulate state and behaviour. Their functionality may be similar but not the same as what is needed in the new system, requiring them to have to be adapted.
  - Often, components from legacy systems are written in languages which are no longer supported on modern platforms.
  - Human programmers tend to be reluctant to modify code that they didn't write in the first place, either because they don't understand it properly, or because they think it would be unfaithful to modify it.

Reuse reengineering [12] is one attempt to address the problem of software reuse. The idea is to reengineer code to make it more reusable, using techniques such as:

- *Program slicing* [13], which extracts a subset of (not necessarily contiguous) statements from a program that meet some criterion (such as contributing to the computation of the output data). Those statements which are not relevant to the slicing criterion are left out of the slice.
- Identifying objects in procedural code, so that it can be rewritten as object-oriented code which encapsulates state and behaviour and is thus more

conducive to reuse [14].

The code produced by such techniques is then packaged as reusable components for the future.

The philosophy underlying reuse has much in common with CBR, which will be introduced in the next section and suggested as an appropriate approach to code understanding and generation.

3 The Applicability of Case-Based Reasoning

3.1 CBR and Its Benefits in General

CBR [15] is the enterprise of solving new problems by analogy with old ones, rather than solving all problems from first principles. Essentially, a *case* consists of an old problem and its solution. Cases are stored in a *case library* (or *case base*). To solve a new problem, a case-based reasoner retrieves a case in which the old problem is usefully similar to the new problem, so that the old solution can be adapted to solve the new problem. The new problem and its solution may then be stored in memory as a case, which can be used in turn to solve future problems.

In general, CBR offers the following benefits (Section 1.5.2 of [15]):

- It should be more efficient to solve new problems by adapting old solutions than solving them from scratch. A case can be a large, contextualised chunk of knowledge, which can achieve in a single inference step that which would take a rule-based system many small inference steps to achieve. A case provides the glue which holds together the inferences used to derive a solution, keeping them from slipping into a search space that may be exponential or worse.
- Cases can suggest solutions to problems in weak-theory domains that cannot be completely formalised by a hard-wired algorithm, a set of rules or a general model, but for which there are many experiences of solving similar problems (e.g. law, medicine).
- In addition to cases that suggest solutions to problems, there can be cases that warn of the potential for failure, and cases that suggest how to repair failures. This enables the reasoner to anticipate failures before they occur and recover from them when they do.
- CBR alleviates the *knowledge-elicitation bottleneck* (Section 9.3.2 of [16]) - in complex domains, it is often easier to elicit knowledge in the form of cases than to try to elicit a complete set of rules.
- CBR is good for prototyping - experience has shown that a CBR system is often easier to build and maintain than the equivalent rule-based system (Section 3.4.1 of [15]).

Above all, case-based reasoners can learn from experience (by acquiring new cases), which occurs as a natural by-product of reasoning (Section 1.1 of [15]). This enables a CBR system to become more efficient and more competent as it gains in experience, as opposed to traditional expert systems which are brittle because they cannot improve beyond their particular plateaux of expertise (Section 1.2.2 of [17]).

3.2 Why CBR Should Be Applied to Code Understanding and Generation

Automatic programming in its full generality is an extremely difficult task to automate. Rich and Waters [8] suggest that the ultimate goal of automatic programming has three features:

- that it is fully automated;
- that it is general-purpose;
- that it takes a brief and informal requirements specification (written in natural language) as its input.

They suggest that it isn't possible to achieve all three (at least not with the current state-of-the-art in artificial intelligence), but that in practice, at least one of them will have to be compromised to achieve the others - for example:

- *Semi-automatic programming*: the system interacts with a human (as in the Programmer's Apprentice [18]).
- *Limit automatic programming to a restricted domain* (as in the Déjà Vu system [19,20]), rather than expecting the system to solve any given programming problem in any given domain.
- *Lower the level of abstraction of the input specification*. An extreme example of this is that compilers were considered as automatic programming systems in the 1950s! A more modern approach would be to use a so-called *very-high-level language*, but this tends to be painfully formal, and could even be more difficult and unpleasant than conventional high-level programming (for example, KIDS [21] is a clever system, but a great deal of sophistication on the part of the user is necessary to write the specifications it works from).

However, a premise of our research is the belief that we could move closer to the goal of general-purpose, fully automatic programming from informal requirements specifications by applying CBR to code generation:

- The main feature of automatic programming which suggests that CBR is more appropriate than non-CBR technologies is that it is theory-weak. It is very difficult to formalise how specifications should be transformed into programs, and this is truer, the higher the level of abstraction the specification is and the more informal it is. There are certainly plenty of cases from the history of programming, though.
- If CBR could be used to overcome this theory-weakness, it should also be possible to automate programming more fully. For example, automatic programming systems such as KIDS [21] have a repertoire of mechanical transformations that they can execute to transform a specification step-by-step into an executable program. However, these systems have to be told by humans which transformations to apply to the specification. This implies that there are no hard-and-fast rules about which transformations should be applied, but there are certainly cases that suggest which ones to try.
- Achieving general-purpose automatic programming is still going to be a major difficulty, because it would require the system to have vast amounts of

knowledge about all domains [22]. That implies having a prohibitively large case library. Analogical reasoning [23] should be able to help, because it would allow cases to be used analogously across different domains.

After arguing for the applicability of CBR to code generation, it is natural to consider applying CBR to the inverse problem, code understanding. For either process, the cases associate fragments of implementation code with fragments of specification code or higher-level knowledge. The distinction between implementation and specification is not absolute, but lies along a continuum between two extremes - a case effectively associates two points in this interval such that one is closer to implementation and the other is closer to specification.

It may be possible to take a bidirectional CBR approach, in which the same case library could be used either for code generation or for code understanding - either the specification side or the implementation side of cases could be matched to the input, and the other side extracted as output. This suggests that it would be equally possible, in principle, to extract a 'specification' from code as to generate the code from that specification.

It may be that code understanding would be more difficult in practice than code generation, because the implementation code that has to be (partially) matched is likely to be much bigger and more complicated than the corresponding specifications. Code understanding is an NP-complete problem [24], but CBR may be able to avoid the combinatorial explosion this threatens by matching larger chunks of code and extracting (with adaptations) more higher-level knowledge at once than if lots of little pieces of code had to be matched.

Another difficulty with code understanding is that the mapping from specification to implementation often involves the loss of some information which is specific to the specification, and the gain of some information which is specific to the implementation. The lost information cannot be recovered from the source code alone - the reverse-engineering process would have to take other artifacts into account.

Chin and Quilici [25] complain that plan libraries can never be guaranteed to be complete, and argue that this means code understanding can only be partially automated. Their problem is that their plans cover only normative code, and cannot deal with idiosyncrasies. This approach seems to be inflexible and prone to over-specialising to its examples. It cannot automatically learn new plans from the code it is given. However, CBR is well suited to covering idiosyncrasies because it can acquire new cases that are exceptions to the norm (c.f. Section 3.4.2 of [15]). CBR's ability to learn from experience means that case libraries should become more complete over time, thus increasing the extent to which tasks such as code understanding and generation can be automated.

4 Existing Approaches to Code Understanding and Generation

Given that there appears to be an overlap between CBR and automatic programming, this section reviews some of the literature in the field of code understanding and code generation. There have been few explicit attempts to apply CBR in this field. However, the spirit of CBR is latent in much work in this field, although the authors seldom claim any connection with CBR. There is thus a distinction between *explicit CBR* and *covert CBR*.

4.1 Explicit CBR Approaches

Most of the explicit CBR in code generation concerns software reuse [26-30]. These researchers have realised that the retrieval and adaptation needed for reuse correspond naturally to CBR retrieval and adaptation, although not all of them automate the adaptation.

Other examples of explicit CBR being applied in this field include the Déjà Vu system [19,20], which applies CBR to automatic programming in the restricted but non-trivial domain of moving robots to carry components around a plant. Déjà Vu has two interesting key features:

- *Hierarchical CBR* [19]. The process of adapting a solution (i.e. a program) includes retrieving and adapting cases to solve subproblems - i.e. CBR is being applied recursively. Hierarchical CBR is particularly appropriate to programming because programs tend to be hierarchical - the instructions in one routine can be thought of as specifications for subroutines, and cases for subroutines can be retrieved dynamically rather than hard-wired into the case for a routine.
- *Adaptation-guided retrieval* [20]. The retrieval process uses knowledge of the available adaptation methods so that it retrieves a case that can definitely be adapted - indeed, that is the easiest case to adapt. This is in contrast to traditional CBR systems which retrieve cases on the basis of semantic similarity rather than adaptability (or its superclass, *utility* - the most adaptable case won't necessarily yield the best solution).

Mostow and Fisher [31] describe a *derivational replay* approach to algorithm design, in which the process of deriving an old algorithm from an old specification is replayed to derive a new algorithm from a new specification (this is better than directly adapting the old algorithm to meet the new specification, because there are major structural differences between the two algorithms, but the processes of deriving them from their specifications are much more similar).

Melis and Schairer [32] describe an application of CBR to software verification. In proving the correctness of a piece of software, it is common for similar subproofs to recur, so it is more efficient to remember and reuse a similar subproof (again, using derivational replay) than to do every subproof from scratch.

4.2 Covert CBR Approaches

Many of the approaches to code understanding, code generation and related topics reported in the literature have much in common with CBR, but are not claimed to be CBR approaches. Our term for this is *covert CBR*.

An important class of code understanding techniques is based on *clichés* [33,34]. Clichés are commonly-used computational structures. There are *data-structure clichés* such as stacks, queues and hash tables, and *algorithmic clichés* such as sorting or binary search.

A system that uses clichés for code understanding has a *cliché library*, which contains *overlays*. Each overlay associates an *implementation cliché* with a *specification cliché*, and encodes the correspondences between the two. Implementation clichés can comprise both fragments of concrete code and intermediate clichés (from which further, more abstract clichés may be extracted). Cliché-based code understanding algorithms work by matching actual fragments of code (and clichés already extracted) to the implementation clichés of overlays, and extracting an instance of the specification cliché of each matching overlay. In this way, a hierarchy of clichés representing the design of the code is built up.

Code understanding can be bottom-up or top-down:

- In the *bottom-up* approach to code understanding, clichés are extracted first from the code itself, then further clichés are extracted from those clichés, and so on. This approach can be prone to combinatorial explosion if every fragment of code and every cliché extracted has to be matched to every overlay in the cliché library, although this problem can be alleviated using indexing [1]. It could also be alleviated by extracting clichés selectively (according to the purpose for which they are being extracted).
- In the *top-down* approach to code understanding, the system starts with a cliché to try to extract. It instantiates the clichés it needs to validly extract the target cliché, and so on until it gets down to the level of code fragments that it can match to the source code. Top-down extraction is useful when the system can indeed hypothesise clichés to extract.

Alex Quilici takes a similar cliché-based approach to code understanding [1,35,25], although both clichés and overlays are 'plans' in his vocabulary. His particular purpose was to extract object-oriented design knowledge from C programs, to aid their translation to C++. His method is characterised by being a hybrid bottom-up/top-down approach. Whereas Wills' approach [33,34] was wholly bottom-up, Quilici's approach is predominantly bottom-up, but there are so-called *implied plans* which are recognised top-down. For example, if a plan for calculating the distance between two points using Pythagoras' Theorem is recognised, it is known to imply the existence of two points. Recognising the points bottom-up would be inefficient if the system had to consider every pair of numerical variables in the code as potentially representing a point! Pythagoras' Theorem, on the other hand, is characterised by a very distinctive equation which serves as a beacon for the bottom-up recognition algorithm to spot.

Clichés are naturally also useful for code generation, as overlays suggest how given specification clichés can be implemented. Clichés are used in this way in the Programmer's Apprentice [36,37], a semi-automatic programming environment. The underlying philosophy is the same as that of CBR, but the user has to tell the system which cliché to use and how to adapt it - the system is merely carrying out mechanical transformations, whereas a CBR system should be able to decide for itself which cliché to use and what transformations to apply.

The cliché-based approaches to code understanding and generation are clearly in the spirit of CBR, and share the same underlying philosophies. The key difference is that clichés do not admit of any partial matching and adaptation, and thus do not have the flexibility or efficiency[3] that a CBR approach could offer. Clichés also tend to be small chunks of knowledge, whereas cases would tend to be large chunks - several clichés could be extracted all at once. That is why Quilici calls his work a *memory-based approach* [1] instead of a case-based approach (c.f. [38]).

The field of software reuse is strongly related to CBR (c.f. Section 2.1) - the terms *retrieval* and *adaptation* are part of the reuse community's vocabulary (Section V of [11]), but the term *case-based reasoning* is not in their mainstream vocabulary. As well as the explicit CBR approaches to reuse mentioned in Section 4.1, there is a lot of covert CBR (e.g. [39]).

Design patterns [40] share the philosophy which underlies CBR - they describe solutions to recurring software problems which conform to the principles of 'good' object-oriented programming (flexibility, elegance and reusability). However, in their current avatar, design patterns are intended as guidelines for human programmers rather than automatic programmers, although it is quite easy to conceive of automatically generating particular implementations of design patterns or recognising them in existing code.

Matsuura et al. [41] describe a system called EVA which derivationally replays the process of deriving a program from a specification when the specification changes. It is thus doing adaptation without needing a retrieval phase first (*evolution* as opposed to *reuse*). It is conceivable that EVA could be extended to be a reuse tool by augmenting it with a retrieval phase, and it would not then be unreasonable to call it a case-based automatic programming system.

Maiden and Sutcliffe [42] describe an *analogical reasoning* approach to specification reuse, i.e. developing new specifications by analogy with old ones. Analogical reasoning [23] is like CBR except that the 'case' which is used to solve the new problem can be from a different domain (e.g. a heat flow problem could be solved by analogy with a current electricity problem, and a situation involving planets orbiting a star could be understood by analogy with a situation involving electrons orbiting the nucleus of an atom). Analogical reasoning requires a process of *analogical mapping* to determine the correspondences between two analogous situations. Keane [43] suggests that much adaptation is latent in this analogical

---

3. The efficiency of CBR is that it should be able to recognise larger chunks at once and avoid combinatorial explosion. This should outweigh the cost of retrieval, partial matching and adaptation.

mapping process.

5 Future Directions

Our research focuses on understanding the constraints in information models [44]. This is a special case of code understanding (information models are written in a formal language, namely EXPRESS [5], which is a kind of code). The project aims to take a CBR approach to extracting higher-level knowledge about constraints from the EXPRESS code. The constraint-understanding system is also being extended to other kinds of code. Many other kinds of code are being considered for this extension, such as:

- other information modelling languages (e.g. UML [6]);
- imperative program code (e.g. Java [45] or C/C++ [46]);
- machine code;
- functional languages (e.g. SML [47] or LISP [48]);
- logic languages (e.g. Prolog [49]);
- formal specification languages (e.g. VDM (Appendix B of [2]) or Z [3]);
- database query languages (e.g. SQL [4]);
- scanner/parser generators (e.g. lex and yacc [50] or JavaCC [51]).

There are several motivations for constraint understanding, including:

- Supporting human understanding of constraints;
- Generating code to check constraints;
- Transferring constraints for schema-to-schema mapping, so constraints from the target schema can be checked on instances in the source repository before they are mapped to the target repository (Section 1.3.2 of [44]).

A major underlying motivation behind the constraint-understanding system is *comparative constraint understanding*, or, more generally, *comparative code understanding*. Whereas the current system just builds an understanding of the constraints on a single input EXPRESS model, the aim of a comparative constraint-understanding system is to understand the comparative semantics of the constraints on two models. More generally, comparative code understanding will involve comparing the semantics of two pieces of code (not necessarily written in the same language).

A vital prerequisite to comparative code understanding is how to represent the correspondences between two pieces of code to compare, e.g. how to represent the structural correspondences between two models. It is clearly necessary to know the correspondences between constraints in two models before the semantic equivalence between two corresponding constraints can be assessed. The obvious solution would be to represent the correspondences between two pieces of code in a formal, precise mapping language.

A more advanced research issue is whether a formal, precise specification of the correspondences between two pieces of code is really necessary, as humans can quite easily see the correspondences between them without needing such a specification. It would be very tedious to have to write specifications of correspondences for any code beyond small toy examples in any case. A computer might also be able to deduce correspondences by exploiting knowledge such as the meaning of names, expecting to find corresponding substructures in the other code, and propagating what it already knows about the correspondences to find other correspondences (e.g. inferring that the types of two attributes correspond when those attributes are known to correspond).

Another significant research area is the *discriminate* extraction of knowledge from code. The current constraint-understanding system indiscriminately extracts all the higher-level knowledge that it knows how to extract from the input model. Indeed, the code understanding techniques reported in the literature also seem to extract knowledge indiscriminately. Some of this knowledge may not be useful for the particular purpose which knowledge is being extracted from the code for, and it would be desirable for a code understanding system to avoid wasting its time extracting knowledge that won't be needed.

One way to avoid extracting unnecessary knowledge from code would be to let extraction be driven by the purpose for which the code is being understood. Again, a particular interest is how extraction could be driven by the need for a comparative understanding of the semantics of the code. For example, once some higher-level constraints have been extracted bottom-up from one model, the comparative constraint-understanding system will need to check whether they are implied by the constraints in the other model. So it could use the higher-level constraints extracted from the first model as hypotheses, and attempt to extract them top-down from the second model. This would avoid considering any higher-level constraints from one model which have no equivalents in the other model, as well as facilitating recording the correspondences between the constraints in the two models.

In general, deciding whether to bother extracting some particular knowledge from a piece of code entails some form of *meta-level reasoning*. Meta-level reasoning is at a level above base-level reasoning, controlling it (but not performing the base-level reasoning itself). Of course, for meta-level reasoning to be worthwhile, the gain in efficiency must outweigh the cost of doing the meta-level reasoning!

Finally, the code generation side is an exciting research topic for the future. Code generation as well as code understanding could be achieved using a bidirectional CBR approach, in which cases could be used both to understand code (by matching the implementation parts of cases to the input code and extracting the specification parts), and to generate code (by matching the specification parts of cases to input specifications and using the implementation parts to generate the output code). The code generation side could range from simple, restricted tasks such as transferring constraints for schema-to-schema mapping and generating the mapping program (see Section 1.3.2 of [44]), to a more general-purpose automatic programming system.

References

1. Quilici A. A memory-based approach to recognizing programming plans. *Communications of the ACM* 1994; 37(5):84-93
2. van Vliet JC. *Software engineering: Principles and perspective*. John Wiley & Sons, Chichester, 1993
3. Spivey JM. *The Z notation*. Prentice Hall, New York, 1994
4. Elmasri R, Navathe SB. *Fundamentals of database systems*, 2nd edition. The Benjamin/Cummings Publishing Company, Inc., Redwood City, 1994. Chapter 7
5. ISO TC184/SC4. Industrial automation systems and integration - Product data representation and exchange - Part 11: Description methods: The EXPRESS language reference manual. ISO standard, reference no. ISO 10303-11. ISO, Switzerland, 1994
6. Rational Software Corporation. *UML Resource Center: Unified Modeling Language, Standard Software Notation*. `http://www.rational.com/uml/index.jtmpl`
7. Chikofsky EJ, Cross JH II. Reverse engineering and design recovery: A taxonomy. *IEEE Software* 1990; 7(1):13-17
8. Rich C, Waters RC. Automatic programming: Myths and prospects. *IEEE Computer* 1988; 21(8):40-51
9. Ning JQ, Engberts A, Kozaczynski W. Recovering reusable components from legacy systems by program segmentation. In: Waters RC, Chikofsky EJ (eds) Proceedings of the 1st IEEE Working Conference on Reverse Engineering, Baltimore, 21st-23rd May 1993. IEEE Computer Society Press, 1993, pp 64-72
10. Ning JQ, Engberts A, Kozaczynski W. Automated support for legacy code understanding. *Communications of the ACM* 1994; 37(5):50-57
11. Mili H, Mili F, Mili A. Reusing software: Issues and research directions. *IEEE Transactions on Software Engineering* 1995; 21(6):528-562
12. Cimitile A, De Lucia A, Munro M. An overview of structural and specification driven candidature criteria for reuse reengineering processes. Department of Computer Science, University of Durham, 1995, technical report 7/95. `http://www.dur.ac.uk/CSM/projects/RE2/ps/TR7dur.ps`
13. Cimitile A, De Lucia A, Munro M. Identifying reusable functions using specification driven program slicing: A case study. In: Proceedings of the IEEE International Conference on Software Maintenance, Nice, October 1995, pp 124-133. `http://www.dur.ac.uk/CSM/projects/RE2/ps/ICSM95.ps`
14. Canfora G, Cimitile A, Munro M. An improved algorithm for identifying objects in code. *Software - Practice & Experience* 1996; 26(1):25-48
15. Kolodner JL. *Case-based reasoning*. Morgan Kaufmann Publishers, Inc., San Mateo, 1993
16. Watson ID. *Applying case-based reasoning: Techniques for enterprise systems*. Morgan Kaufmann Publishers, Inc., San Francisco, 1997
17. Brown MG. A memory model for case retrieval by activation passing. PhD thesis, University of Manchester, Manchester, 1993, technical report UMCS-94-2-1. `ftp://ftp.cs.man.ac.uk/pub/TR/UMCS-94-2-1.ps.Z`
18. Rich C, Shrobe HE. Design of a Programmer's Apprentice. In: Winston PH, Brown RH (eds) *Artificial intelligence: An MIT perspective*, vol 1: Expert problem solving, natural language understanding, intelligent computer coaches, representation and learning. MIT Press, Cambridge, MA, 1979, pp 137-173

19. Smyth B, Cunningham P. Déjà Vu: A hierarchical case-based reasoning system for software design. In: Neumann B (ed) Proceedings of the 10th European Conference on Artificial Intelligence (ECAI 92), Vienna, 3rd-7th August 1992. John Wiley & Sons Ltd, Chichester, 1992, pp 587-589
20. Smyth B, Keane MT. Adaptation-guided retrieval: Questioning the similarity assumption in reasoning. *Artificial Intelligence* 1998; 102(2):249-293
21. Smith DR. KIDS: A semiautomatic program development system. *IEEE Transactions on Software Engineering* 1990; 16(9):1024-1043
22. Barstow DR. Domain-specific automatic programming. *IEEE Transactions on Software Engineering* 1985; 11(11):1321-1336
23. Keane MT. *Analogical problem solving*. Ellis Horwood Limited, Chichester, 1988
24. Woods S, Yang Q. The program understanding problem: Analysis and a heuristic approach. In: Proceedings of the 18th International Conference on Software Engineering (ICSE-96), Berlin, 1996, pp 6-15
25. Chin DN, Quilici A. DECODE: A cooperative program understanding environment. *Journal of Software Maintenance: Research and Practice* 1996; 8(1):3-34
26. Fernández-Chamizo C, González-Calero PA, Gómez-Albarrán M, Hernández-Yáñez L. Supporting object reuse through case-based reasoning. In: Smith I, Faltings B (eds) *Advances in case-based reasoning*: Proceedings of the 3rd European Workshop (EWCBR-96), Lausanne, November 1996. Springer-Verlag, Berlin, 1996, pp 135-149 (Lecture notes in computer science no. 1168)
27. Henninger S. Accelerating successful reuse through the domain lifecycle. In: Proceedings of the 7th Annual Workshop in Software Reuse (WISR 7), St. Charles, 28th-30th August 1995
28. Maguire P, Szegfue R, Shankararaman V, Morss L. Application of case-based reasoning (CBR) to software reuse. In: Watson ID (ed) *Progress in case-based reasoning*: Proceedings of the 1st United Kingdom Workshop, Salford, January 1995. Springer-Verlag, Berlin, 1995, pp 166-174 (Lecture notes in computer science no. 1020)
29. Penix J, Alexander P. Component reuse and adaptation at the specification level. In: Proceedings of the 8th Annual Workshop on Institutionalizing Software Reuse (WISR 8), Ohio State University, Columbus, 23rd-26th March 1997. `http://www.umcs.maine.edu/~ftp/wisr/wisr8/papers/penix/penix.html`
30. Tessem B, Whitehurst RA, Powell CL. Retrieval of Java classes for case-based reuse. In: Smyth B, Cunningham P (eds) *Advances in case-based reasoning*: Proceedings of the 4th European Workshop (EWCBR-98), Dublin, 23rd-25th September 1998. Springer-Verlag, Berlin, 1998, pp 148-159 (Lecture notes in computer science no. 1488)
31. Mostow J, Fisher G. Replaying transformational derivations of heuristic search algorithms in DIOGENES. In: Proceedings of the DARPA Case-Based Reasoning Workshop, Pensacola Beach, 31st May - 2nd June 1989. Morgan Kaufmann Publishers, Inc., San Mateo, 1989, pp 94-99
32. Melis E, Schairer A. Similarities and reuse of proofs in formal software verification. In: Smyth B, Cunningham P (eds) *Advances in case-based reasoning*: Proceedings of the 4th European Workshop (EWCBR-98), Dublin, 23rd-25th September 1998. Springer-Verlag, Berlin, 1998, pp 76-87 (Lecture notes in computer science no. 1488)
33. Rich C, Wills LM. Recognizing a program's design: A graph-parsing approach. *IEEE Software* 1990; 7(1):82-89
34. Wills LM. Flexible control for program recognition. Pages 134-143 in Waters RC, Chikofsky EJ (eds) Proceedings of the 1st IEEE Working Conference on Reverse Engineering, Baltimore, 21st-23rd May 1993. IEEE Computer Society Press, 1993

35. Quilici A. A hybrid approach to recognizing programming plans. In: Waters RC, Chikofsky EJ (eds) Proceedings of the 1st IEEE Working Conference on Reverse Engineering, Baltimore, 21st-23rd May 1993. IEEE Computer Society Press, 1993, pp 126-133. `http://www-ee.eng.hawaii.edu/~alex/Research/Postscript/Papers/wcre93.ps`
36. Waters RC. The Programmer's Apprentice: A session with KBEmacs. *IEEE Transactions on Software Engineering* 1985; 11(11): 1296-1320
37. Rich C, Waters RC. The Programmer's Apprentice: A research overview. *IEEE Computer* 1988; 21(11):10-25
38. Aamodt A, Plaza E. Case-based reasoning: Foundational issues, methodological variations, and system approaches. *AICom - Artificial Intelligence Communications* 1994; 7(1):39-59. `http://www.iiia.csic.es/People/enric/AICom.html`
39. Jeng J-J, Cheng BHC. A formal approach to reusing more general components. In: Proceedings of the 9th IEEE Knowledge-Based Software Engineering Conference, Monterey, September 1994, pp 90-97. `ftp://ftp.cps.msu.edu/pub/serg/reuse/kbse94-reuse.ps.Z`
40. Gamma E, Helm R, Johnson R, Vlissides J. *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley Longman, Inc., Reading, 1995
41. Matsuura S, Kuruma H, Honiden S. EVA: A flexible programming method for evolving systems. *IEEE Transactions on Software Engineering* 1997; 23(5):296-313
42. Maiden NA, Sutcliffe AG. Exploiting reusable specifications through analogy. *Communications of the ACM* 1992; 35(4):55-64
43. Keane MT. Analogical asides on case-based reasoning. In: Wess S, Althoff K-D, Richter MM (eds) *Topics in Case-Based Reasoning,* Selected Papers from the First European Workshop (EWCBR-93), Kaiserlautern, November 1993. Springer-Verlag, Berlin, 1994, pp 21-32 (Lecture notes in computer science no. 837)
44. Broad AP. The application of case-based reasoning to the understanding of constraints on information models. MPhil thesis, University of Manchester, Manchester, 1999. `http://www.cs.man.ac.uk/~broada/cs/mphil/thesis/`
45. Grand M. *Java language reference*. O'Reilly & Associates, Inc., Sebastopol, 1997
46. Stroustrup B. *The C++ programming language*, 2nd edition. Addison-Wesley Publishing Company, Inc., Reading, 1991
47. Myers C, Clack C, Poon E. *Programming with Standard ML*. Prentice Hall, New York, 1993
48. Wilensky R. *Common LISPcraft*. Norton Press, New York, 1986
49. Clocksin WF, Mellish CS. *Programming in Prolog*, 4th edition. Springer-Verlag, Berlin, 1994
50. Levine JR, Mason T, Brown D. *Lex & yacc*, 2nd edition. O'Reilly & Associates, Inc., Sebastopol, 1992
51. Sun Microsystems, Inc. *Java Compiler Compiler (JavaCC) - The Java Parser Generator*. `http://www.sun.com/suntest/products/JavaCC/`