An Object-Oriented Framework for the Design and the Implementation of Case-Based Reasoners

Michel Jaczynski - Brigitte Trousse

INRIA Sophia-Antipolis, Action AID
2004 route des lucioles - BP 93
06902 Sophia Antipolis Cedex, FRANCE
e-mail: {Michel.Jaczynski, Brigitte.Trousse }@sophia.inria.fr
http://www.inria.fr/aid/

Abstract

In this article, we emphasise the need of open software engineering tools for CBR and we present our object-oriented framework named CBR*Tools. This framework facilitates the development of new CBR applications mainly by reusing past designs and implementations, and by providing open components that can be customised to meet application requirements.

1 Introduction

Case-based reasoning has been studied and successfully used in many industrial and academic applications [AP94, AAB+95]. In order to address more and more complex problems, the challenge is now to formalise this kind of reasoning[1], to define application analysis methodologies [BWS97], and to provide a design and implementation assistance with software engineering tools. This paper focuses on the latter issue: we intend to design *open* software engineering tools (CBR tools) which *facilitate* the design and the development of case-based reasoners. We want to go beyond existing CBR shells [AAB+95] and their underlying libraries in terms of *reuse* and *flexibility*. Our goal is to design CBR tools that mainly allow:

- an easier development by reusing past designs and implementations,
- the integration of new components to enhance the tools from an application to another,
- and the extension of existing components and their collaborations to meet specific application requirements.

In this paper, we propose an *object-oriented framework* [FS97] for case-based reasoning, named CBR*Tools. A framework is composed of a set of abstract classes and defines the way objets collaborate [JF88]. A framework may also contain typical concrete classes as in class libraries, but a framework is much

[1] Some works have adopted an approach at the knowledge level [AP94b] and others are based on logics (terminological logics [Koe94], fuzzy logic [DEG+97]).

more than a library: it allows the reuse of both *design* and *code*. A framework is the state-of-the-art object-oriented approach to software reuse. Of course such a tool is hard to design and it raises other difficulties that have to be overcome to actually draw benefits [CHS+97]. CBR*Tools has been designed under the Rational Rose[2] visual modelling tool, and the reuse of the framework is facilitated since class diagrams can be graphically specialised with an automated generation of the appropriate code structure. Firstly, we will analyse our approach compared to existing case-based reasoning tools. Secondly, we will present the main design features of our framework, CBR*Tools, implemented in Java. Finally, we will conclude with the evaluation process undertaken in our research team and with future works.

2 Framework Approach Benefits for CBR

2.1 Limitation of Existing CBR tools

A CBR tool is a software that can be used to develop several applications that require case-based reasoning. The tool can be domain-independent or dedicated to an application domain or a type of problems (such as help-desk applications). CBR shells [AAB+95] are a kind of application generators with a sophisticated graphical user interface, where some parameters can be specified by the user to develop a new application. For example, you can specify the fields of cases, the domain knowledge, the weight vectors for the retrieval. CBR shells are a kind of tools that can usually be used by a non-programmer user, and the extension or the integration of new components in these tools are not possible.

CBR Application Programming Interfaces (APIs) provide a set of functions to manage CBR algorithms, and are intended to be used by a programmer. Shells usually provide APIs to embed the tools in an application with a specific user interface. CBR APIs can sometimes be extended by using a programming language (usually C or C++): new similarity measures or adaptation techniques can be added. However, the goal of this type of tools is not to provide generic or open components but only to customise the inputs and outputs of the system. Thus even if this API is sometimes called a library or a class library, it cannot be seen as a library of reusable components.

Most scientific papers give no details about extension and reuse issues. The FABEL PROTOTYPE [GVG+97] in the architectural design domain has solely addressed these issues: several different tools have been integrated for the retrieval and the reuse steps, and new tools can be connected to the system. This work has

[2] Rational Rose (http://www.rational.com) is a product of Rational Software Corporation.

emphasised the distribution of the tools, while we intend to design an integrated object-oriented model which could be distributed in a further step. Our approach leads to a more uniform tool at the design level which will certainly be easier to reuse.

2.2 The Challenge of Framework Design for CBR

The concept of object-oriented frameworks has been introduced in the late 80's and has been defined as « a set of classes that embodies an abstract design for solutions to a family of related problems, and supports reuses at a larger granularity than classes » [JF88]. Thus a framework is much more than a software library. A library defines a set of classes that may be reused independently or in very small groups. On the other hand, the goal of a framework is to capture a set of concepts related to a domain and the way they interact. In addition a framework is in control of a part of the program activity and calls specific application code by dynamic method binding. A framework can be viewed as an incomplete application where the user only has to specialise some classes to build the complete application.

The design of a framework is harder than a library of reusable components, and is essentially iterative: a high level of domain expertise is first needed, then a loop of evaluation and reuse on real problems is required to make the framework stable[3]. The design process is centred on the identification of *hot spots* [Sch96]. Hot spots are well defined features of the framework that can be customised for a specific application by specialisation (*white-box* hot spot) or composition (*black-box* hot spot). At the beginning a framework only defines white-box components and a more mature framework will also provide black-box components. The hot spots are usually created by using *design patterns* [GHJ+95] which provide typical design solutions and improve the framework documentation.

Thus designing a framework is a complex task with several issues that have to be overcome [CHS+97] and the framework requires a good documentation to be actually reused [FHL+97, Joh92]. However, the framework approach is very appealing. Frameworks allow the reuse of both code and design for a class of problems, giving the ability to non-expert to write complex applications quickly. Frameworks also allow the development of prototypes which could be extended further on by specialisation or composition. A framework is more difficult to understand than an application or a class library, but once this step is done, the framework can be applied in a wide spectrum of context, and can be enhanced by the integration of new components.

[3] Some say that a framework is never finished [CHS+97].

We propose to design a framework for case-based reasoning with the four following *axes of variability* (cf. [DMN+97]): *case representation*, *case storage*, *case indexing* and *reasoning steps*. These four axes represent the main ways applications may differ from one to another. It is important that a framework for CBR takes these axes into account in order to be reused in different contexts. These axes have to be handled as independently as possible to simplify the development and the maintenance of applications. These axes must lead to the definition of hot-spots that will support the development of CBR application by tailoring the framework.

2.3 Enhancing CBR Software Design Methodologies

As a CBR framework enables the reuse of a design represented by a set of abstract classes and collaboration schemes, this approach must be positioned in regards to CBR software design methodologies [BWS97] and CBR knowledge level models [AP94, AP94b]. We think that a framework is the right kind of tools to be used in conjunction with these methods or models. Firstly, a framework does not address any application analysis where case and indices, for instance, are modelled, but it brings interesting features such as: quick prototyping, good stability to requirement modifications and hot spots to tune the application. In addition, the reuse of a framework to design and implement a new application is based on a *delta analysis* [CHS+97]: we must identify which part of the framework can be directly reused and which part must be customised. This kind of analysis requires a good documentation, but it is appealing to save effort compared to an analysis done from scratch. Secondly, a CBR framework facilitates the transformation of the knowledge level models to the symbol level by providing abstract classes and collaboration models. These specific features of frameworks can be used to enhance a project management methodology for CBR.

3 CBR*Tools Framework Design

We have analysed the axes of variability defined in the previous section and we have first adopted an object-oriented representation of cases. Then we have designed hot spots in our framework CBR*Tools mainly based on: the delegation of reasoning steps, the separation of case storage and case indexing, the design of indexes as reusable components, and the design of adaptation patterns.

3.1 Delegation of Reasoning Steps

Case-based reasoning is usually divided into four steps [AP94]: *retrieve*, *reuse*, *revise*, *retain*. In CBR*Tools, we propose to delegate each reasoning step to a different object (cf. Fig. 1). This leads to the definition of four main hot-spots

using the *Strategy*[4] design pattern [GHJ+95]. Each class Retrieve, Reuse, Revise and Retain defines an abstract interface to a step of the reasoning while the Reasoner defines how to control the reasoning : starting, stopping and resuming. The step classes must be specialised to implement a specific reasoning. The Reasoner class is also a hot spot allowing the implementation of different reasoning control methods (sequential or multi-threaded for instance) and even the reasoning cycle can be modified (integrating a loop between the retrieval and reuse steps for instance, cf. discussion in [GVG+97] p. 185).

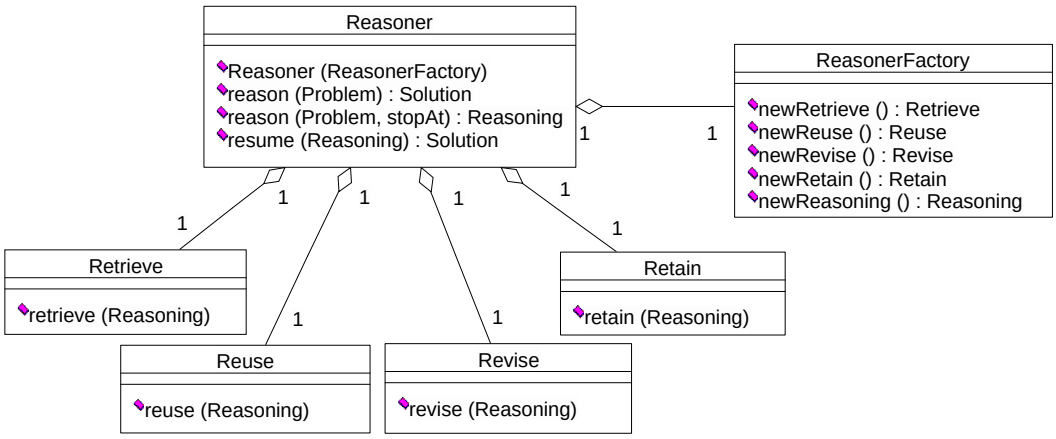

Fig. 1. Class diagram[5] of the Reasoner

In addition, to actually combine different step implementations, we have made explicit the execution context of each step with an object of the Reasoning class. This object stores the state of a reasoning and is updated and used by each step: this object must implement the interface required by each step and may provide additional adaptation code to combine steps. For example, if the *retrieve* step returns a list of retrieved cases as result but the *reuse* step does not handle multiple cases, the *reasoning* object must provide the method to get the best case from the list of retrieved cases to reuse only one case. Finally, in order to ensure that the reasoning step implementations and the reasoning object are consistent,

[4] The *Strategy* design pattern is based on the delegation of methods to an object that implements an abstract interface. This pattern allows the use of different encapsulated implementations of an algorithm.

[5] The class diagrams presented in this paper are simplified from the ones implemented in CBR*Tools. We have drawn the diagrams using the Unified Modelling Language (UML) defined by G. Booch, I. Jacobson and J. Rumbaugh (http://www.rational.com). UML has been recently submitted to the Object Management Group (OMG) as a proposal for a standard notation of object-oriented analysis and design techniques.

we provide the ReasonerFactory class using the *Abstract Factory*[6] design pattern [GHJ+95]. This flexible design allows:

- the independence of reasoning control and reasoning implementation,
- the reuse of different step implementations,
- the isolation of code required to combine steps,
- and the guarantee that steps can be combined (from the interface specification).

3.2 Separation of Case Storage and Case Indexing

Usually in CBR, we speak about the *organisation* of the case base meaning that the cases must be indexed so that the retrieval step can use a structure to have access to cases. In CBR*Tools, we propose to separate the case storage from the indexing structure because indexes[7] can be built without knowing how and where the cases are stored. Moreover, different indexes can be defined upon the same set of cases to allow the evaluation of different indexing techniques. Thus, we propose to define a Memory class composed of a CaseBase and an IndexBase (cf. Fig. 2). A CaseBase manages the low level access to cases given their ids and may even be specialised to have access to complex distributed cases [BWF95]. The IndexBase manages a set of indexes where each index gives access to cases from a set of indices.

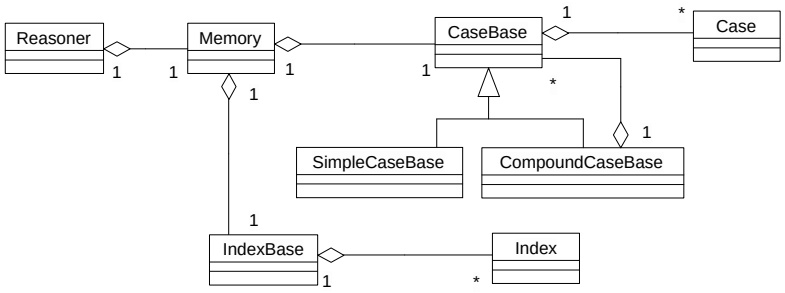

Fig. 2. Class diagram of the Memory

In CBR*Tools, we enhance the interoperability and the flexibility of a case base by applying the *Composite*[8] design pattern [GHJ+95] so that we can represent simple and compound case base. Compound case base can be used to store cases in different places (relational or object-oriented database servers, files) and case

[6] The *Abstract Factory* design pattern allows the creation of a family of related objects without knowing explicitly the real class of each object.
[7] *Indexes* are structures while *indices* are indicators used to retrieved cases through indexes.
[8] The *Composite* design pattern allows the management of composite objects and simple objects with the same interface.

base can have different properties (shared case base open in read-only, temporary case base). Due to our design, the reasoner still uses only one case base that encapsulates this heterogeneity, and indexes can be built without managing explicitly these differences.

3.3 Designing Indexes as Reusable Components

An index is a structure that enables the retrieval of cases based on a set of indices described in the problem. In CBR*Tools, an index is designed as an object, and following the *Composite* design pattern, these objects can be composed to build compound index. Our motivation is to be able to apply the appropriate indexing technique to each known sub-space of the case base, and to connect several indexing techniques to build sequences or alternatives. In CBR*Tools, we have implemented the hierarchy of indexes given in the Fig. 3.

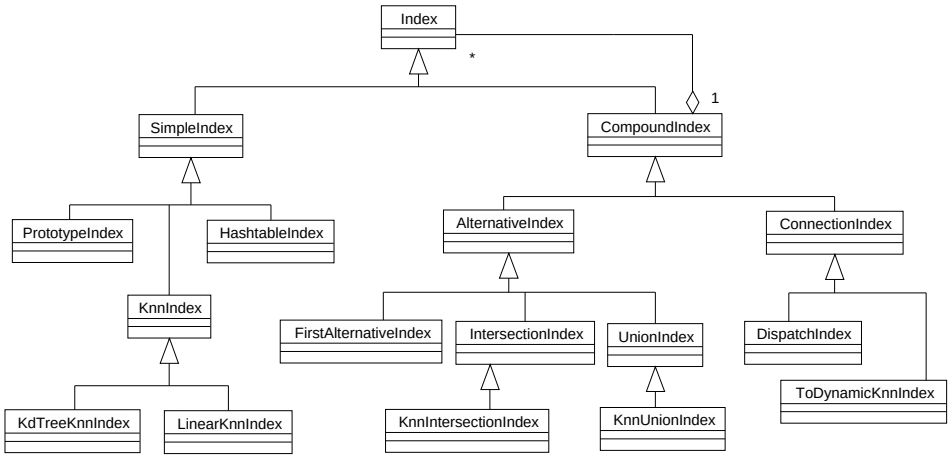

Fig. 3. Hierarchy of indexes in CBR*Tools

Three main types of indexes have been defined:

- *simple indexes* (subclasses of SimpleIndex): these indexes implement a single indexing technique such as standard linear *knn* algorithm, *kd-trees* [WAD94], prototypes filtering [JT94], or hash tables.
- *connection indexes* (subclasses of ConnectionIndex): these indexes are used to build sequences, where the retrieval result from an index is given as a parameter to the following one. For instance, with the ToDynamicKnnIndex, we can connect a PrototypeIndex to a LinearKnnIndex in order to apply a knn retrieval on the cases retrieved by the prototype index. The DispatchIndex is used to activate further index objects depending on the case subsets first retrieved.
- *alternative indexes* (subclasses of AlternativeIndex): alternatives are used to aggregate different ways of retrieval. We can make an union or an

intersection of cases retrieved by different indexes, or manipulate a set of indexes. For example, handling missing values in cases can be simplified by making the union of two alternatives: the first one based on a Kd-tree index that performs well on cases without missing values and the second one for cases with missing values using a linear knn index.

The composition of indexes is based on the following protocol: an index returns a set of cases and/or a set of case subsets. Depending on the input and output requirements of each index, indexes can be composed or not. The data returned by each index can be specialised to extended this protocol. Each index also encapsulates the update of its internal structure (adding or removing a case, rebuilding the index). Thus indexes are reusable components, and our model allows the design of complex retrieval strategies which have been intensively used in our applications developed with CBR*Tools [Jac97, JT97, JT97b].

3.4 A Library of Adaptation Patterns

Our aim is to provide a library of adaptation patterns in which one pattern can be selected and specialised for a given application. For example, we describe in this section the ActionListAdaptation pattern (cf. Fig. 4) motivated by BROADWAY [JT97b], our CBR navigation advisor for the Web. A case is considered to be a couple (situation, list of advised actions). A list of matching cases are retrieved for the current situation and the adaptation must build the final list of advised actions. The advised actions are first collected from the retrieved cases, and each action is analysed according to a set of features. Each feature represents one aspect of the interest of an action according to the matching with the current situation. The features are analysed, evaluated and compared, and finally the actions are combined to build the solution. This pattern can be specialised for an application and a set of features can be defined by deriving new classes from ActionFeature.

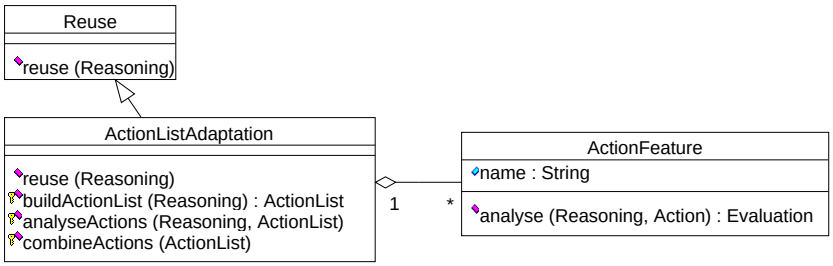

Fig. 4. Action list adaptation pattern

In BROADWAY, a case is mainly composed of the user's behavioural situation and a list of advised Web pages to visit next (advised actions). The pages advised by the retrieved cases are evaluated, selected and ordered to build the solution. We have defined eight page features such as the number of different cases advising the

same page and the best similarity of the cases that advise a given page. More features can be easily added.

4 Conclusion

We have presented CBR*Tools, our object-oriented framework for CBR and we have explained the main flexible designs adopted but our framework integrates much more flexibility such as an object-oriented representation of cases, a hierarchy of similarity measures and a hierarchy of weight vectors. The two main contributions of our work are:

- an object-oriented model that defines guidelines to design and build reusable CBR software,
- and an implementation in Java of that model with the object-oriented framework CBR*Tools which can be used through a visual modelling tool (Rational Rose).

This framework is the result of a one-year-project, and CBR*tools currently offers more than 220 classes. In order to evaluate the framework, we have developed typical CBR applications such as car insurance risk detection (also used in [AAB+95, WAD94]) : this simple implementation requires the specialisation of only 5 classes even with the use of a kd-tree index. We have also extended our framework with a sub-framework for the management of cases with time-extended situations [Jac97, JT97]. Two main systems have been then implemented: an application of case retrieval for plant nutrition control assistance [Jac97] and BROADWAY, a browsing advisor reusing past navigations of a group of users [JT97b]. The implementation of complex reasoning required for these applications is straightforward and requires the specialisation of about 20 small classes. The framework is currently used by several developers (integration of new indexes, distribution and persistence) but additional tools to facilitate its use are needed: we are studying the use of *hooks* [FHL+97] or *reuse patterns* [Joh92] which describe reuse scenarios.

References

[AAB+95] K. D. Althoff, E. Auriol, R. Barletta and M. Manago. *A Review of Industrial Case-Based Reasoning Tools*. AI Perspectives Report, AI Intelligence, 1995.

[AP94] A. Aamodt and E. Plaza. Case-Based Reasoning: Foundational Issues, Methodological Variations, and Systems. *AI Communications*, 7(1):36–59, March 1994.

[AP94b] E. Armengol and E. Plaza. A Knowledge Level Model of Knowledge-Based Reasoning. In S. Wess, K.D. Althoff and M. M. Richter, editors, *Topics in Case-Based Reasoning*, volume 837 of *Lecture Notes in AI*, pages 54–164, Springer, 1994.

[BWF95] M. Brown, I. Watson, and N. Filer. Separating the Cases from the Data: Towards More Flexible Case-Based Reasoning. In M. Veloso and A. Aamodt, editors, *Case-Based Reasoning Research and Development*, volume 1010 of *Lecture Notes in AI*, pages 157-168, Springer, 1995.

[BWS97] R. Bergmann, W. Wilke and J. Schumacher. Using Software Process Modeling for Building a Case-Based Reasoning Methodology: Basic Approach and Case Study. In D. B. Leake and E. Plaza, editors, *Case-based Reasoning Research and Development*, volume 1266 of *Lecture Notes in AI*, pages 509–518, Springer, 1997.

[CHS+97] W. Codenie, K. De Hondt, P. Steyaert and A. Vercammen. From Custom Applications to Domain-Specific Frameworks. *Communication of the ACM*, vol. 40, No 10, pages 71–77. October 1997.

[DEG+97] D. Dubois, F. Esteva, P. Garcia, L Godo, R. L. DeMontaras and H. Prade. Fuzzy Modelling of Case-Based Reasoning and Decision. In D.B. Leake and E.Plaza, editors, *Case-Based Reasoning Research and Development (ICCBR'97)*, volume 1266 of *Lecture Notes in AI*, pages 599–610. Springer, 1997.

[DMN+97] S. Demeyer, T.D. Meijler, O. Nierstrasz and P. Steyaert. Design Guidelines for 'Tailorable' Framework. *Communication of the ACM*, vol. 40, No 10, pages 60–64. October 1997.

[FHL+97] G. Froedlish, H. J. Hoover, L. Liu and P. Sorenson. Hooking into Object-Oriented Application Frameworks. In *Proceedings of ICSE'97*, Boston, pages 491–501, 1997.

[FS97] M. E. Fayad and D. C. Schmidt. Object-Oriented Application Frameworks. *Communication of the ACM*, vol. 40, No 10, pages 32–38. October 1997.

[GHJ+95] E. Gamma, R. Helm, R. Jonhson and J. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison Wesley, 1995.

[GVG+97] F. Gebhardt, A. Voss, W. Grather and B. Schmidt-Belz. *Reasoning with Complex Cases*. Kluwer Academic Publisher, 1997.

[Jac97] M. Jaczynski. A Framework for the Management of Past Experiences with Time-Extended Situations. In *Proceedings of the 6th International Conference on Information and Knowledge Management (CIKM'97)*, pages 32–39. Las Vegas, November 1997.

[JF88] R. E. Johnson and B. Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1(2):22–35. June 1988.

[Joh92] R. E. Jonhson. Documenting Frameworks using Patterns. In *Proceedings of OOPSLA'92*, ACM SIGPLAN Notices, 27(10): 63-76. October 1992.

[JT94] M. Jaczynski and B. Trousse. Fuzzy Logic for the Retrieval Step of a Case-Based Reasoner. In M. Keane, J.P. Haton, M. Manago, editors, *EWCBR-94 : Second European Workshop on Case-Based Reasoning*, pages 313-321, Chantilly, France, novembre 1994.

[JT97] M. Jaczynski and B. Trousse CBR*Tools: An Object-Oriented Library for Indexing Cases with Behavioural Situation. Research Report n°3215, INRIA, July 1997. (in French)

[JT97b] M. Jaczynski and B. Trousse. Broadway: A Browsing Advisor Reusing Past Navigations from a Group of Users. In *Proceedings of the 3rd UK Workshop on Case-Based Reasoning*, Manchester. September 1997.

[Koe94] J. Koehler. An Application of Terminological Logics to Case-Based Reasoning. In *Proceedings of the 5th International Conference of Principles of Knowledge Representation*, pages 351–362, Bonn, Germany, 1994.

[Sch96] H. A. Schmid. Design Patterns for Constructing the Hot Spots of a Manufacturing Framework. In *Journal of Object-Oriented Programming*, 9(3):25–37. June 1996.

[WAD94] S. Wess, K.D. Althoff, and G. Derwand. Using K-d Trees to Improve the Retrieval Step in Case-Based Reasoning. In S. Wess, K.D. Althoff and M. M. Richter, editors, *Topics in Case-Based Reasoning*, volume 837 of *Lecture Notes in AI*, pages 167–181, Springer, 1994.