OO Design & Implementation of a Flexible Software Architecture for Decision Support Systems

Wolfgang Pree, Fabio Beckenkamp, Sergio Viademonte
Applied Computer Science
University of Constance
Universit‰tsstr. 10
D-78434 Constance, Germany
Ph.: +49.7531.88.44.33, Fax: +49.7531.88.35.77
{Wolfgang.Pree, Fabio.Beckenkamp, Sergio.Viademonte}@uni-konstanz.de

Abstract. Many implementations of decision support systems suffer from a lack of flexibility, that is, they are built for a specific application domain. For different application domains, large portions of the particular decision support system have to be reimplemented from scratch. As object-orientation allows the construction of flexible software architectures, this paradigm was applied in the realm of building decision support systems. The paper represents an experience report, which first outlines the conventional implementation of a decision support system and the problems that were encountered when the system was adapted to different application domains. The paper goes on to discuss the concepts of object-oriented components and frameworks and how these concepts were applied in particular in the construction of an object-oriented decision support system that deserves the attribute *generic*.

Keywords: decision support systems, hybrid intelligent systems, neural networks, software architecture, design patterns, frameworks, componentware, Java.

1 Introduction

One characteristic of computer-based decision support systems [Bonczek, 1981] is that they deal with complex, unstructured real-world tasks. The construction of computer-based decision support systems is regarded as typical domain of knowledge engineering and artificial intelligence. An adequate decision support system should

- have sufficient knowledge about the problem domain
- be able to learn
- have logical, deductive and inductive reasoning capability
- be able to apply known solutions to analogous new ones
- be able to draw conclusions

Expert systems represent a well-known example of this kind of system. In order to overcome several problems of expert systems, such as difficulties in building up a huge consistent knowledge base, so-called hybrid systems were built. They try to integrate various single AI technologies, in particular, expert systems, artificial neural networks, fuzzy logic, genetic algorithms and case-based reasoning.

Artificial neural networks support knowledge acquisition and representation. Fuzzy logic [Kosko, 1992] is useful to model imprecise linguistic variables, such as predicates and quantifiers (expressions like high, short, etc). Genetic algorithms [Lawrence, 1991] excel in the ability to do deductive learning. Case-based reasoning remembers previous problems and applies this knowledge to solve or evaluate new problems.

One difficulty in implementing hybrid systems is how to smoothely integrate the various single AI technologies. Besides this, a hybrid system should be flexible enough to solve problems in several application domains. The former aspect is discussed, for example, in [Medsker and Bailey, 1992]. This paper focuses on the latter aspect.

2 Hycones—an example of a hybrid system

Hycones (short for: Hybrid Connectionist Expert System) is a sample hybrid system that is especially designed for classification decision problems [Machado and Denis, 1991; Re·tegui, 1993]. The core technology is artificial neural networks (ANNs). Hycones generates ANNs based on various parameters that have to be specified. The generation of ANNs can be improved by adding appropriate expert rules. This section starts with a description of the principal features of Hycones. Based on this overview the problems regarding its flexibility are outlined. These problems were encountered when Hycones was applied to different domains [Rosa, 1994; Re·tegui, 1993].

2.1 Hycones as generator of decision support systems

Hycones was, among other domains, applied in the realm of a mail order reseller: Customers order goods, typically by sending in an order form. Based on the information on an order form (eg, value of ordered goods, home address, age) the reseller has to decide whether the customer receives the ordered articles or not, as the customer might not be able to pay.

In order to implement a decision support system, the mail order reseller provided numerous customer data from recent years that had the information whether the customer turned out to be liquid or not associated with each customer record.

The first step in using Hycones is to specify the input nodes and output nodes of the ANNs that are generated. In case of the customer checking system the input nodes correspond to the information on the order form. Hycones offers different data types (eg, boolean values, fuzzy value ranges) to specify the input nodes.

The output nodes (called hypotheses) corresponds to the desired decision support. In case of the customer checking system the information whether to send the goods to the customer or not would become the output node.

Additional expert knowledge can be modeled in expert rules [Le„o and Rocha, 1990]. For example, rules describing typical attributes of untrustworthy customers could be specified for the mail order decision support system.

Based on the information sketched above, Hycones generates numerous ANN topologies depending on some additional parameters (various thresholds, etc.). Figure 1 schematically illustrates this feature of Hycones.

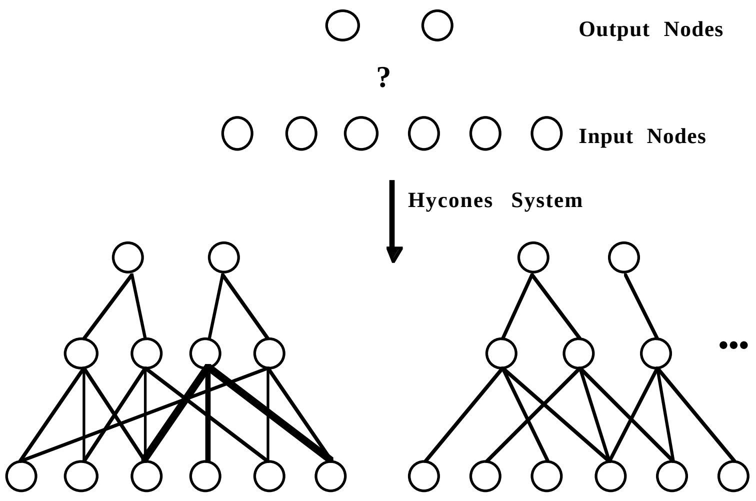

Figure 1 Hycones as ANN generator.

Learning

We discern inductive and deductive learning mechanisms. Inductive learning is performed through the training of the generated ANNs using a punishment and reward algorithm and an incremental learning algorithm [Machado and Rocha, 1989]. Inductive learning allows automatic knowledge acquisition and incremental learning.

Deductive learning can be implemented through genetic algorithms. This might imply further modifications in the topology of the ANNs, creating or restoring connections between neurons. The deductive learning is not implemented in the current version of Hycones.

Inference
Once the generated ANNs are trained, Hycones pursues the following strategy to come up with a decision for a specific case (eg, a customer): All ANNs evaluate the case. Each ANN calculates a confidence value for its hypotheses. The inference mechanism finds the winner-ANN and returns the corresponding result.

2.2 Adaptation problems

Despite the intention of Hycones to be a reusable generator of decision support systems, the Hycones implementation had to be changed fundamentally for each application domain over the recent years. In other words, the Hycones system had to be implemented almost from scratch for each new application domain. What are the reasons for this unsatisfying situation?

Limits of hardware & software resources
The first Hycones version was implemented in CLOS. CLOS simplified the implementation of core parts of Hycones, but the execution time turned out to be insufficient for the domain problems at hand.

In subsequent versions of Hycones, parts of the system were even implemented on different platforms to overcome performance problems and memory limits. For example, the ANN training algorithm was implemented in C on an Unix workstation as target platform. C was chosen to gain execution speed. Unix workstations can more easily be extended than desktop PCs regarding main memory. (The discussed versions of Hycones do not allow straight-forward parallelization of training and testing so that a network of computers could be used. So the training and testing has to be done on one computer.)

Other parts of Hycones were implemented on PCs and use Borland Delphi for building the GUI and the Borland Database Engine as database. (The specified input and output nodes as well as the generated ANN topologies were stored in database tables.) Some other parts were also implemented separately such as an interactive tool for modeling the domain.

The fact that Hycones became a hybrid system also regarding its implementation, implied tedious data shifting between different computing platforms. The parameters comprising the specification for the ANN generation were entered on PCs, but the ANN training and testing was done on Unix workstations. Finally, if the customer prefered to work with the decision support system on PCs the generated and trained ANNs had to be transfered back from the Unix platform to the PC environment.

Conversion of data
Companies which want to apply Hycones have to provide data for ANN training and testing. Of course, various different ways of dealing with these data have to be considered. For example, some companies prefer the ASCII-format, others relational database tables. Though this seems to be only a minor issue and a small part of the overall Hycones system, experience has proven that lots of work had to be done for converting training and test data.

For example, the mail order reseller changed the format of the ASCII-data files in different data sets. Thus the conversion routine that was implemented in C had to be adapted frequently.

Complex conceptual modeling
This issue is also related to performance problems: Hycones mananges the numerous different ANN topologies by storing the information of all the differences of the ANNs, in particular, the connections and their corresponding weights, in a database. Roughly speaking, one record represents the connections and weights of one ANN. Overall this forms a pretty complex conceptual model. It turned out that the way the information about all generated ANNs is stored in database tables had to be changed several times to optimize for the database system in use. These changes were not only tedious but also error-prone.

3 OO Redesign and Java-Implementation

In order to overcome the problems sketched above, Hycones was redesigned based on the object-oriented paradigm and implemented in Java (Sun, 1997). As object-oriented framework concepts were applied in particular, this section presents a general overview of frameworks. The application of framework concepts in the redesign of Hycones demonstrates how problems of the conventional Hycones implementation could be solved.

3.1 Software components

First of all let us clarify the term *(software) component*. Though some authors, for example, Nierstrasz and Dami (1995), view almost all programming language constructs (ranging from macros to classes and modules) as candidates for building components we use a more rigid definition: A component is simply a data capsule. Thus information hiding becomes the core construction principle underlying components. Parnas (1972, p.1056) defines information hiding as follows: “A module is characterized by its knowledge of a design decision which it hides from all others. Its interface was chosen to reveal as little as possible about its inner working.” Several ways of bringing information hiding down to earth have been proposed: In module-oriented languages such as Modula-2 and Ada components are called modules. In object-oriented languages such as Smalltalk, C++ and Java, components are instances of classes. A class represents an abstract data type (ADT). Analogous to modules, a class offers an interface and hides its realization. In contrast to a module, a class serves as a component factory by allowing the instantiation of any number of objects.

In order to overcome the reuse problems of module-oriented languages, object-oriented languages introduce language constructs to achieve *delta changes* (programming by difference) without having to touch the source code of original modules/classes. Inheritance is central to this solution: A subclass defines the delta by which a class differs from its superclass. In other words, inheritance allows ADT adaptations without having to edit source code or give up compatibility. To sum up, object-oriented languages improve the module concept. They allow a straightforward definition of ADTs and provide language constructs for their extension and modification.

As a consequence, many adopters of object technology expect that the usage of an object-oriented language alone yields significant improvements in software flexibility and thus reusability. This leads to a quite naive application of object technology: Reuse is then usually viewed as process of picking single components out of a huge set of components and putting them together. Thus, single component reuse means that programmers build the overall software system architecture on their own. They have to locate the appropriate components and define their interaction. Studies such as the one conducted by Barry Boehm at NASA (Boehm, 1994) have corroborated programmer's gut feeling that single-component reuse is almost as expensive as development from scratch. An alternative is framework reuse as discussed below. Though a framework might be viewed as coarse-grained component, it differs from a large single component regarding its flexibility.

In general, more flexibility implies more programming effort by the programmers who reuse the software components. The ordering of white-box and black-box frameworks below reflects decreasing levels of flexibility and thus increasing ease of reuse. But in contrast to monolithic coarse-grained single components there is still more flexibility left. Black-box frameworks represent the most rigid albeit the easiest way of reusing components, thus forming the backbone of pure plug-and-play componentware.

3.2 Framework concepts

Instead of reusing single components, most successful object-oriented projects do framework development and reuse. A framework is simply a collection of several single components with predefined cooperations between them. A framework accomplishes a certain task. Some of these single components are designed to be replaceable, typically corresponding to abstract classes in the framework's class hierarchy. We call the points of predefined refinement hot spots (Pree, 1995, 1996). Figure 2 shows these framework characteristics with the hot spots in gray color.

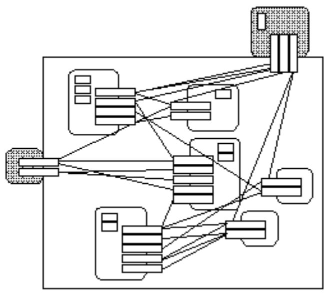

Figure 2 Framework with hot spots.

A framework deserves the attribute well-designed if it offers the domain-specific hot spots to achieve the desired flexibility via adaptation of these hot spots. Well-designed frameworks also predefine most of the overall architecture, i.e., the composition and interaction of its components. The lines connecting method interfaces in Figure 2 express this glue between the components. Applications built on top of a framework reuse not only source code but architecture design, which we consider as the most important characteristic of frameworks.

Note that the framework concept is quite independent of the way how components are implemented. Frameworks just require that components can be replaced by more specific ones that are compatible to the original placeholders. Of course, object-oriented languages support specialization in a straightforward manner through inheritance. The discussion of white-box and black-box frameworks below assumes that an object-oriented language is used for framework development.

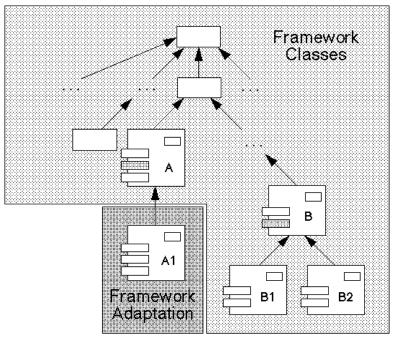

Figure 3 Sample framework class hierarchy.

White-box frameworks

White-box frameworks consist of several incomplete (abstract) classes, i.e., classes that contain methods without meaningful default implementations. Class A in the sample framework class hierarchy depicted in Figure 3 illustrates this characteristic of a white-box framework. The abstract method of class A that has to be overridden in a subclass is drawn in gray.

In order to modify the behavior of white-box frameworks, programmers apply inheritance to override methods in subclasses of framework classes. Such method (re)definitions are analogous to providing specific functions in procedure/function libraries with a callback style of programming: the application architecture resides in the framework. Programmers adapt the framework by overriding the hook methods called out from other methods in the framework. The necessity to override methods implies that programmers have to understand the framework's design and implementation, at least to a certain degree.

Black-box frameworks

Black-box frameworks offer ready-made components for adaptations. Modifications are accomplished by *composition*, not by programming. In the framework class hierarchy in Figure 3, class B already has two subclasses B1 and B2 that provide default implementations of B's abstract method. Supposed that the framework components interact as depicted in Figure 4(a), a programmer adapts this framework, for example, by instantiating classes A1 and B2 and plugging in the corresponding objects (see Figure 4(b)). In the case of class B, the framework provides ready-to-use subclasses; in the case of class A the programmer has to subclass A first.

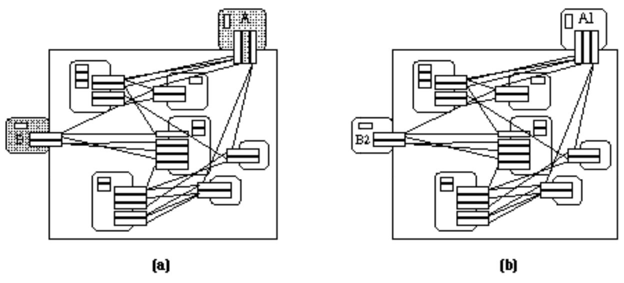

Figure 4 Framework (a) before and (b) after specialization by composition.

Available frameworks are neither pure white-box nor pure black-box frameworks. If the framework is heavily reused, numerous specializations will suggest which black-box defaults could be offered instead of providing a white-box interface. So frameworks will evolve more and more into black-box frameworks when they mature.

Pros and cons of frameworks

Besides the fact that reuse of architecture design amounts to a standardization of the application structure, frameworks offer further advantages. Adapting a framework to produce a specific application implies a significant reduction in the size of the source code that has to be written by the programmer who adapts a framework. Mature frameworks allow a reduction of up to 90% (Weinand et al., 1989) compared to software written with the support of a conventional function library.

More good news is that framework-centered software development is not restricted to specific domains, such as graphic user interfaces. Actually, frameworks are well-suited for almost any commercial and technical domain where the real world is simulated in a broad sense. To name just a few, decision support systems, process control systems, reservation systems, and banking software belong to this category.

The bad news is that framework development requires an enormous development effort. Many problems result from complicated interaction scenarios between stateful, partially defined components. The costs for developing a framework are significantly higher compared to the development costs of a specific application. So frameworks represent a long-term investment that pays off only if similar applications are developed again and again in a domain.

Furthermore, tools and methods assisting in framework development are almost non-existent or in their infancy. Framework technology itself is not yet mature. For example, it remains unclear how frameworks designed by different teams with separate control flows can interoperate. The fragile base-class problem (Lewis et al., 1995) might overthrow fundamental framework design decisions: Changes in base classes of a framework can fracture numerous classes inheriting from them.

Finally, framwork development and reuse is at odds with the current project culture that tries to optimize the development of specific software solutions instead of generic ones. As this paper focuses on technical aspects, we refer to the excellent discussion of organizational issues by Goldberg and Rubin (1995).

Despite the mentioned problems with the state-of-the-art in framework technology, (black-box) frameworks form the enabling technology of plug-and-play software, where most adaptations can be achieved by exchanging components.

3.3 Design of OO-Hycones

The conventional design of Hycones has undergone a major object-oriented redesign based on the concepts presented above. Parts of the resulting OO-Hycones system became even pure black-box frameworks. In the following we outline the relevant aspects of the redesign and show how the problems discussed in Section 2.2 were solved.

OO modeling of the ANN representation

The core entity of an ANN, a neuron, is modeled as class. This forms a straightforward design, as neurons in ANNs are self-contained: A neuron knows its connections to other nodes and can (re-)calculate the weight of the connections based on various algorithms (eg, backpropagation). Thus an ANN is not represented by entries in relational database tables, but as collection of instances of class Neuron.

The Strategy design pattern [Gamma et al., 1995] was applied to keep the training strategy flexible (see Figure 5): the class TrainingStrategy is abstractly coupled with class Neuron. This framework construction is related to black-box frameworks and allows a flexible change of the training strategy by plugging in a specific strategy component (eg, an instance of class Backpropagation) into the neuron objects of an ANN.

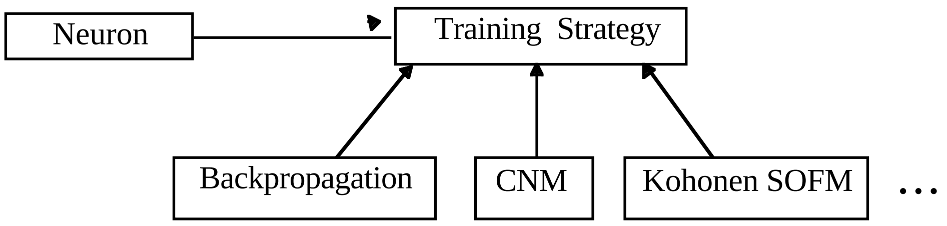

Figure 5 A black-box mini-framework for keeping the training strategy flexible.

Besides the advantage of having a natural model of ANNs, the object-oriented design of this aspect of Hycones leads to another significant improvement: OO-Hycones generates physically separated ANNs that have the training and testing capability contained in their neurons. Thus the parallelization of ANN training and testing becomes feasible. Though this feature is not implemented yet, the object-oriented design forms the precondition for this enhancement. Thus hardware and software limitations can easily be surpassed through parallel training and testing on networked computers.

Converter framework

In order to cope with different data formats, a black-box framework for format conversions was designed according to the Chain-of-Responsibility design pattern [Gamma et al., 1995]. Class Converter applies this design: The class contains a reference to itself which is used to build a chain of converters (see Figure 6). Each converter in the chain tries to do the conversion. The conversion request is forwarded in the chain until a converter is found that actually converts the data.

Thus even data coming from different sources (e.g, data bases, ASCII-files) can be processed intertwined. The Hycones part that requires the data just asks the converter component to provide the data. The converter component consists of a chain of converters. The one that can convert the data converts them and passes the result on to the test and training subsystems. Note that converters can be rearranged or added on demand.

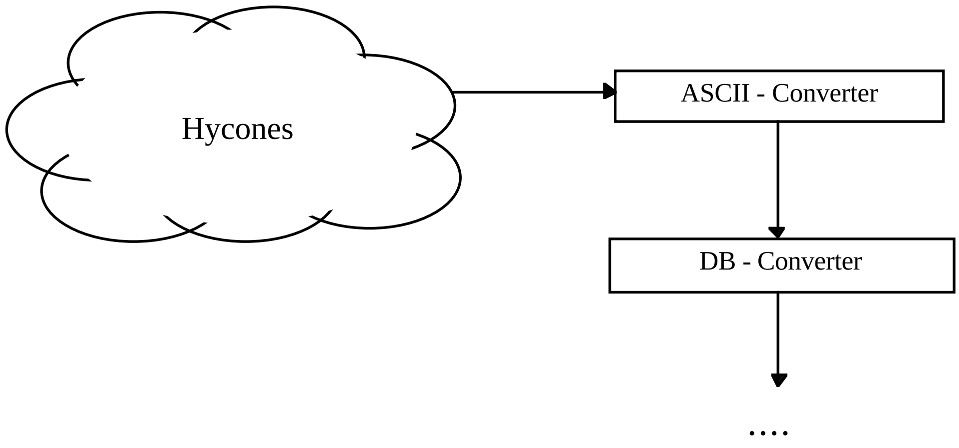

Figure 6 Hycones conversion framework.

Another important design aspect is that the class Converter defines an abstract interface, no matter how the data are actually converted. Thus the Hycones system itself has not to be changed to cope with unforseen data formats.

This design solves the problem that conversion routines have to be implemented always from scratch. The number of black-box converter components will increase over time so that an application of Hycones to another domain is likely to be able to reuse the already existing converter components.

3.4 Implementation issues

The object-oriented implementation of Hycones was done in Java with Microsoft’s Visual J++ development environment. Using Java (without native code add-ons) allowed to create a truly portable system, solving the problem of having parts of Hycones implemented for different platforms. Overall, the run-time efficiency of the Just-In-Time compiled Java code came very close to the reference implementation of Hycones in C.

Knowing the troubles one encounters with memory management in C++, Java’s garbage collection mechanism was really appreciated, in particular, in the realm of developing frameworks. The authors cannot imagine to go back to a system without this feature.

At the time the implementation started, Java Beans (Sun, 1997) were not available. Thus OO-Hycones uses a persistence mechanism which was implemented by the authors according to the Java Beans specification. Thus, using the serialization mechanism of Java Beans instead of the preliminary one turned out to be easy.

4 Summary and outlook

Overall, it was only possible to come up with a well-designed OO-Hycones system, because the system aspects that lacked flexibility were known from previous attempts to adapt Hycones to specific situations. Besides the gained flexibility, portability of OO-Hycones was another major goal of the reimplementation. This came for free due to the usage of Java as implementation language.

Currently, add-on tools, in particular interactive editors for domain modeling, are developed, and the parallelization of network training and testing is under way. In the process of adapting OO-Hycones to further domains, OO-Hycones can prove that it indeed deserves the attribute *generic*.

References

Boehm B. (1994): *Megaprogramming*. Video tape by University Video Communications (http://www.uvc.com), Stanford, California

Bonczek, R. H. (1981): *Foundations of Decision Support Systems*. New York: Academic Press.

Goldberg A., Rubin K. (1995): *Suceeding with Objects: Decision Frameworks for Project Management*. Reading, Massachusetts: Addison-Wesley

Kosko, B. (1992): *Neural Networks and Fuzzy Systems*. NJ: Prentice Hall, Englewood Cliffs

Lawrence D. (1991): *The Handbook of Genetic Algorithms*. New York: Van Nostrand Reinhold

Le„o, B. F. & Rocha, A. F. (1990): *Proposed Methodology for Knowledge Acquisition*: A Study on Congenital Heart Disease Diagnosis. Methods of Information in Medicine, V. 29, n.1, p. 30-40

Lewis T., Rosenstein L., Pree W., Weinand A., Gamma E., Calder P., Andert G., Vlissides J., Schmucker K. (1995): *Object-Oriented Application Frame-works*. Manning Publications/Prentice Hall

Machado, R. J., Rocha, A. F. (1989): *Handling Knowledge in High Order Neural Networks: The Combinatorial Neural Model*. Rio de Janeiro: IBM Rio Scientific Center. (Technical Report CCR076).

Machado, R. J. e Denis, F. A. R. M. (1991): *O modelo Conexionista Evolutivo*. Rio de Janeiro: IBM Rio Scientific Center. (Technical Report CCR-128).

Medsker L. R. & Bailey D. L. (1992): Models and Guideliness for Integratig Expert Systems and Neural Networks. In: Kandel A. & Langholz G. *Hybrid Architectures for Intelligent Systems*, CRC Press.

Nierstrasz O., Dami L. (1995): *Component-Oriented Software Technology*. In: Object-Oriented Software Composition, Nierstrasz O, Tsichtitzis D, Prentice Hall, 3-28

Parnas D.L. (1972): *On the Criteria to be Used in Decomposing Systems into Modules. Communications of the ACM*, 15(12), 1053-1058

Pree W. (1995): *Design Patterns for Object-Oriented Software Development*. Reading, Massachusetts: Addison-Wesley

Pree W. (1996): *Framework Patterns*. New York City: SIGS Books

Re·tegui, E. B. (1993): *Um modelo para sistemas especialistas conexionistas hÌbridos*. Porto Alegre: Instituto de Inform·tica da UFRGS (Master Thesis, Computer Science).

Rosa, SÈrgio I. V. (1994): *AplicaÁ„o de Sistemas Especialistas no processo decisÛrio: uma abordagem hÌbrida*. Porto Alegre: Programa de PÛs-GraduaÁ„o em AdministraÁ„o da UFRGS (Master Thesis, Business).

Sun (1997): *The Java Language*; *Java Beans*. White Papers at http://java.sun.com, Sun Microsystems

Weinand A., Gamma E. and Marty R. (1989): *Design and Implementation of ET++, a Seamless Object-Oriented Application Framework*. Structured Programming, 10(2), Springer Verlag