Neural Network Framework Components

Fábio Beckenkamp, Wolfgang Pree, Sérgio Viademonte

Software Engineering Group
University of Constance
D-78457 Constance, Germany
Voice: +49.7531.88.40 79 Fax: +49.7531.88.35 77
E-mail: *LastName*@acm.org

Abstract. The goal of this paper is to describe the design and implementation aspects of a framework architecture for decision support systems that rely on artificial neural network technology. Besides keeping the design open for supporting various neural network models, a smooth integration of neural network technology into a decision support system forms another important design goal. Many conventional implementations of such decision support systems suffer from a lack of flexibility, that is, they are built for a particular application domain and rely on one specific algorithm for the intelligent engine. In general, for different application domains, large portions of the decision support system have to be reimplemented from scratch. The principal contributions of this paper are: the description of flexible and reusable components for core aspects of neural networks implementations, the integration of different neural network models in a decision support system, and the presentation of a decision support system architecture that can be easily adapted to handle different domain problems. The chapter first outlines the flexibility problems of a typical conventional implementation, and then goes on to discuss in detail the overall architecture of the object-oriented redesign together with some relevant implementation aspects.

Key words: frameworks, object-oriented design, object-oriented architectures, decision support systems, artificial neural networks, hybrid intelligent systems, software reusability.

1 An overview of the domain area

One characteristic of computer-based decision support systems is that they deal with complex, unstructured real-world tasks (Bonzek, 1981). Let's take a look at a sample application of a decision support system in the realm of a mail order reseller. Customers order goods typically via telephone or by sending an order form. The information associated with an order, such as the value of ordered goods, the customer's home address and the customer's age are useful for customer's classification. For example, the marketing department needs such a classification to optimize the selection of customers who receive information brochures regularly. In order to implement a computer-based decision support system for the classification task, the mail order reseller provides a large amount of customer data from recent years that describes customer behavior.

The construction of decision support systems requires the integration of several methods from knowledge engineering and artificial intelligence (AI) research areas. An adequate decision support system should:

- Have sufficient knowledge about the problem domain
- Be able to learn
- Have logical, deductive, and inductive reasoning capabilities
- 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 proposed (Leão and Rocha, 1990). 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 excel in the ability to do deductive learning (Lawrence, 1991). Case-based reasoning remembers previous problems and applies this knowledge to solve or evaluate new problems.

Why build a framework for decision support systems?

One difficulty in implementing hybrid systems is the smooth integration of the various single AI technologies. A hybrid system should also be flexible enough to solve problems in several application domains. Medsker and Bailey (1992) discuss the former aspect, whereas this chapter focuses on the latter aspect. We assume that the reader is familiar with the most basic concepts of artificial neural networks and expert systems.

In artificial neural networks (ANNs) software development it is common to redevelop models from scratch each time a different application must be accomplished. There are some tools that tries to avoid this and help on the main ANN development aspects offering some pre-defined building blocks. Unfortunately, in general, these tools are commercialized software and their structure is not open for analysis. Various important aspects of ANN development must be cited here. ANN software developers usually:

- Think about only one neural model to solve a specific application problem.

- Come up with too specific implementations for a particular problem.
- Are worried about the ANN performance and not about the construction of different ANN models and its reusability in different problem domains.

Thus object-oriented (OO) design and implementation have hardly been applied so far in this domain area. Our intention is to build a flexible OO architecture in order to solve the following problems related to the implementation of decision support systems:

- The architecture should be extensible to deal with additional neural models.
- Flexible ANN architectures that can change their structure and behavior at run time allow experiments for gaining better results.
- It should be easy to have different models of neural network models, distributed over a network. In this way, a suitable solution of a decision problem can be found more quickly.

In the long term, the OO architecture could form the basis of building up hierarchies of ANNs working together, cooperating, and acting as intelligent agents in a distributed environment.

We chose Java as implementation language. Neural networks form a good test bad to test the performance of Java, in particular when just-in-time (JIT) compilation is applied. The first comparisons with the conventional C implementations revealed that there are no significant performance losses, if the virtual machine and JIT compiler are reasonably well implemented.

2 Characteristics and problems of a conventional architecture

Hycones (short for Hybrid Connectionist Expert System; Leão 1993) is a sample hybrid system that is especially designed for classification decision problems. The core technology is artificial neural networks based on the Combinatorial Neural Model (Machado and Rocha 1989, 1990). The experiments with this model proved that it is powerful for classification problems, having good results from medical diagnosis to credit analysis (Leão 1993a, Reategui 1994, da Rosa 1995). This section starts with a discussion 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.

2.1 Hycones as generator of decision support systems

The fact that Hycones is a generator system already indicates that the design of a generic architecture constitutes an important goal right from the beginning. Unfortunately, the conventional design and implementation did not provide the required flexibility as will be outlined in section 2.2.

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

hypotheses, correspond to the desired decision support. In the case of the customer classification system, the customer categories become the output nodes.

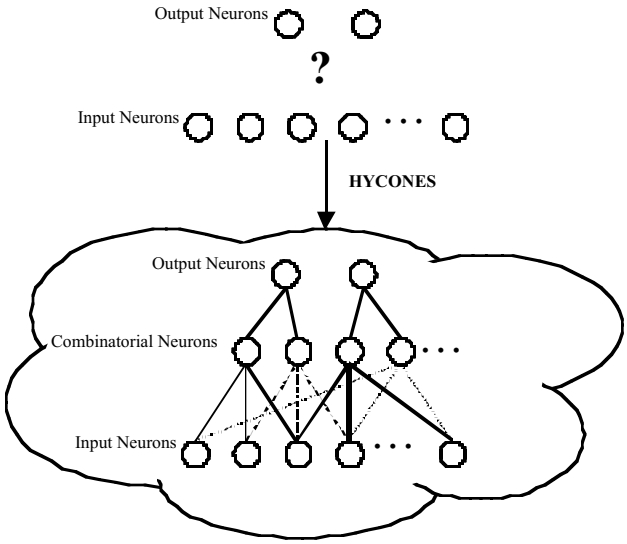

Figure 1 Hycones as ANN generator.

Based on the information described above, Hycones generates the Combinatorial Neural Model (CNM) topology depending on some additional parameters (various thresholds, etc.). Figure 1 schematically illustrates this feature of Hycones. Each of the combinatorial nets contributes to the overall decision.

Learning

We discern inductive and deductive learning mechanisms. Inductive learning is performed through the training of the generated ANN based on available data 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 ANN, creating or restoring connections between neurons. Deductive learning is not implemented in the current version of Hycones.

Inference

Once the generated ANN is trained, Hycones pursues the following strategy to come up with a decision for one specific case (e.g. a customer): The ANN evaluates the case and calculates a confidence value for each hypothesis. The inference mechanism finds the winning hypothesis and returns the corresponding result.

Expert rules

Additional expert knowledge can be modeled in expert rules (Leão and Rocha, 1990). For example, rules describing typical attributes of customers belonging to a particular

category could be specified for the mail order decision support system. Such rules imply modifications of the weights in the ANN. Figure 2 exemplify this Hycones property. The expert rule I3 & I4 & In => O2 corresponds to the strengthened connections among the input nodes I3, I4 and In, one of the combinatorial nodes, and the output node O2 of an ANN.

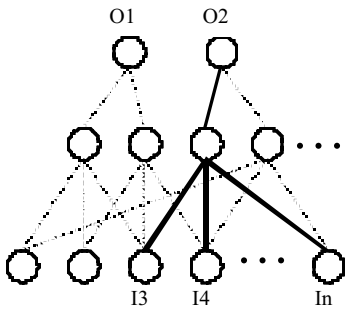

Figure 2 Incorporating expert rules into the ANN topology.

2.2 Adaptation problems

Despite of 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 is implemented in CLOS. CLOS simplifies the implementation of core parts of Hycones, but the execution time turns out to be insufficient for the domain problems at hand.

In subsequent versions of Hycones, parts of the system are even implemented on different platforms to overcome performance problems and memory limits. For example, the ANN training algorithm is implemented in C on a Unix workstation. C was chosen to gain execution speed. Other parts of Hycones, such as an interactive tool for domain modeling by means of specifying expert rules, are implemented on PCs and use Borland Delphi for building the GUI.

Complex conceptual modeling

This issue is also related to performance problems: Hycones manages the complex ANN topology by storing the information of all the connections and their corresponding weights in main memory. (Hycones provides no parallelization of ANN training and testing.) Due to memory limits, only parts of the ANN structure can be kept there and rest is stored in a database. Roughly speaking, database records represent the connections and weights of one ANN topology. Overall this forms a quite complex conceptual model, involving overhead when swapping ANN weights between memory and database. The way the information about the generated ANN is stored in database tables has had to be

changed several times to be optimal for the database system in use. These changes are not only tedious but error-prone.

The fact that Hycones became a hybrid system also regarding its implementation implies complex data shifting between different computing platforms. The parameters comprising the specification for the ANN generation are entered on PCs, but the ANN training and testing is done on Unix workstations. Finally, if the user prefers to work with the decision support system on PCs, the generated and trained ANN have to be transferred back from the Unix platform to the PC environment.

Neural network models

Hycones supports only one ANN model, the Combinatorial Neural Model, but it should be possible to choose from a set of ANN models the one that is best suited for the decision support problem at hand.

Conversion of data

Companies that want to apply Hycones have to provide data for ANN training and testing. Of course, various different ways of dealing with this data have to be considered. For example, some companies provide data in ASCII-format, others as relational database tables, others as object databases. The data read from these sources must be converted to valid data for the ANN input. This conversion is done based on the domain knowledge, which also changes from application to application. Though this seems to be only a minor issue and a small part of the overall Hycones system, experience has proven that a significant part of the adaptation work deals with the conversion of training and test data.

In order to overcome the problems mentioned above, Hycones is completely redesigned based on framework construction principles. The problems of the conventional Hycones implementation form a good starting point for identifying the required hot spots (see another chapter in the book: Hot-Spot-Driven framework Development). The next section presents the hot-spot-driven redesign and implementation of Hycones in an object-oriented way.

3 Design of a neural network framework architecture

We use the term Java-*A*NN-*B*usiness-*C*omponents, Java-ABC for short, because the resulting set of frameworks is implemented in Java. The hot spots of Java-ABC can be summarized as follows:

- ANN training: Java-ABC should support several ANN models. As mentioned above, Hycones is restricted to one specific model, the Combinatorial Neural Model.
- Data conversion: Java-ABC should provide flexible mechanisms for converting data from various sources.
- The ANN internal structure and behavior changes from model to model, but some significant aspects can be kept flexible in order to facilitate any ANN model implementation.

Modeling the core entities of the ANN, neurons and synapses, as objects solves the complex conceptual modeling of Hycones. Instead of storing the generated ANN in

database tables, the topologies are saved as objects via Java's serialization mechanism. The object-oriented model also forms the basis for the parallelization and distribution of ANN learning and testing, however those aspects are not related to frameworks, we won't discuss them in this paper. Finally, being Java a portable language and system solves the problem of the splitting of the original system into subsystems implemented in various programming paradigms on different platforms. Java-ABC runs on all major computing platforms.

3.1 Object-oriented modeling of the core entities of neural networks

Neurons and synapses of ANNs mimic their biological counterparts and form the basic building blocks of ANNs. Java-ABC provides two classes, Neuron and Synapse, whose objects correspond to these entities. Both classes are abstract and offer properties that are common to different neural network models. The idea is that these classes provide basic behavior independent of the specific neural network model. Subclasses add the specific properties according to the particular model.

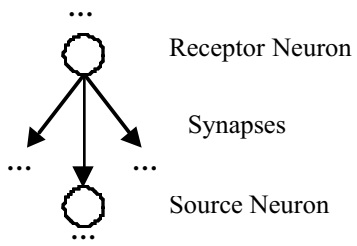

Figure 3 The relationship between Neuron and Synapses objects.

An object of class Neuron has the activation as its internal state, provides methods to calculate its activation and to manage a collection of Synapses objects that process outgoing signals of a source neuron. A Synapse object represents a directed connection between two neurons (see Figure 3). The incoming signal from one neuron is processed and forwarded to the neuron on the outgoing side of the synapse. Thus, a Synapse object has exactly one Neuron object connected to it, that is the source of the computational flow. The receptor Neuron manages a list of synapses and computes its activation from all incoming synapses. Example 1 shows some aspects of the Neuron and Synapse classes.

Example 1 Neuron and Synapse classes.

```
public class Neuron extends Object implements Serializable {
        float currentActivation; // stores the resulting activation computation
        Vector incomingSynapses; // vector of input Synapse objects
        ComputationStrategy  compStrategy;
                // strategy for processing input values (see explanation in the text)
        Neuron () {incomingSynapses = new Vector();}
        Neuron (Vector sourceNeurons) {
                incomingSynapses = new Vector();
                generateSynapses(sourceNeurons);
        }
```

```
        void compute() {
                Synapse s;
                for (int i=0; i<incomingSynapses.size; i++) {
                        s = incomingSynapses.elementAt(i);
                        s.compute();// calculates next pathway current flow
                }
                // take Synapses currentFlow and apply its own calculation strategy
                currentActivation = compStrategy.compute(incomingSynapses);
        }
        void generateSynapses(Vector sourceNeurons) {
                Synapse s;
                for (int i=0; i<sourceNeurons.size; i++) {
                        s= new Synapse(sourceNeurons.elementAt(i));
                        incomingSynapses.addElement(s);
                }
        }
        float getCurrentActivation() {return (currentActivation);}
        …// other methods are implemented
}

public class Synapse extends Object implements Serializable {
        Neuron sourceNeuron; // the neuron that the synapse receives computation
        ComputationStrategy  compStrategy;
                // strategy for processing input values (see explanation in the text)
        float weight;                   // synaptic weight
        float currentFlow;              // stores the result of synapsis computation
        Synapse(Neuron newSourceNeuron) {
                weight = (float)1.0;
                sourceNeuron = newSourceNeuron;
        }
        void compute(){
                sourceNeuron.compute();
                currentFlow = compStrategy(getCurrentActivation(),weight);
                        // calculate currentFlow using the incoming activation from the
                        // sourceNeuron and the synaptical weight
        }
        void setWeight(float newWeight) {weight = newWeight;}
        float getWeight() {return (weight);}
        float getCurrentFlow() {return (currentFlow);}
}
```

As the synapse knows its source neurons, different neuron architectures can be build such as multilayer feedforward or recurrent networks. The process of creating the neural network architecture is controlled by a method called generateNet() and belongs to the class NetImplementation that is explained in the section 3.2. Each neural network model is responsible by its architecture construction. Different neural models use the Neuron and Synapse classes as the basic building blocks for the neural network structure and behavior construction.

Using Neuron and Synapse classes to create a feedforward architecture

In case of a multilayer feedforward neural network, initially the neurons for all necessary neuron-layers are created. Later, the necessary synapses to connect the neurons at

different layers are created and correctly connected to the neuron layers. A list of synapses (called incommingSynapses) controls each instance of Synapse that connects an output neuron to a hidden neuron. The class Neuron (see Example 1) implements this list. When creating instances of the class Synapse (see Example 1 – class Synapse), it is informed in its constructor to which hidden neuron it must be connected. The reference to the hidden neuron is stored in the instance variable sourceNeuron. This process is repeated for all network layers. The method generateSynapsis(Neuron sourceNeurons) in class Neuron, is responsible for the generation of Synapse instances and its appropriate connection to source neurons.

The software architecture explained above was successfully used to implement different neural network models involving different neural network architectures. Besides the CNM model, the Backpropagation (Rumelhart and McClelland, 1986) network was implemented as another feedforward network. The Self-Organizing Feature-Mapping (Kohonen, 1982) was implemented representing lattice structures with two-dimensional array of neurons. Finally the Hopfield Network (Hopfield, 1982) was implemented as recurrent network architecture example.

The implementation of recurrent computation in the proposed architecture implies synchronization of the computational flow by choosing which neuron is going to process in a learning step. Randomly choosing the next neuron to compute is the typical solution (Haykin, 1994).

A class called NetImplementation is tightly associated with class Neuron (this class is explained in the section 3.2). Roughly speaking, a NetImplementation object harnesses the ANN in order to make decisions. The NetImplementation object represents the inference engine (= neural network model) of the running Java-ABC system. Its abstract interface reflects the needs of the decision making process. For example, a method getWinner() computes which output neuron has the maximum activation. Due to the abstract design of NetManager and Neuron, Java-ABC supports different inference engines. How to switch between different ANN models is discussed in the next section.

3.2 Support of different neural network models through the Separation pattern

Java-ABC should be flexible regarding its underlying ANN model. It depends on the particular decision problem at hand, which ANN model proves to be the most appropriate one. Thus the design should allow the trial of different ANN models.

Java-ABC should be able to manage various ANN models trying the solution for a specific problem at the same time. Several neural models can run at the same time in a distributed way. To handle this idea the class NetManager was created. The NetManager class is responsible for controlling, at run time, a set of instances of different neural models, or even a set of instances of the same neural model but with different configurations.

As the learning of a neural model can take days, it can be interesting to change the underlying ANN model at run-time. It is necessary to allow the user to add new ANN models at run-time in order to start different learning trials during the learning or testing process of other neural models, without stopping the processes already started. It is also

desirable to allow changes in the ANN that is already learning. This means the possibility of adding or deleting neurons and synapses on the ANN structure and changing its behavior by changing learning strategies and tuning learning parameters. To have these kinds of simulation characteristics, it is necessary to have a quite flexible architecture design. This design was obtained through the hot-spot-driven design methodology.

To permit the addition of new neural models at run-time, it is necessary to implement the NetManager with the Separation pattern. The NetManager have a list of instances of the NetImplementation class that are responsible for different ANN implementations. A specific ANN model is defined by subclassing NetImplementation and overriding the corresponding hook methods such as getWinner() and compute(). Figure 4 exemplifies how three commonly used ANN models can be incorporated in Java-ABC: Backpropagation, SOM and CNM.

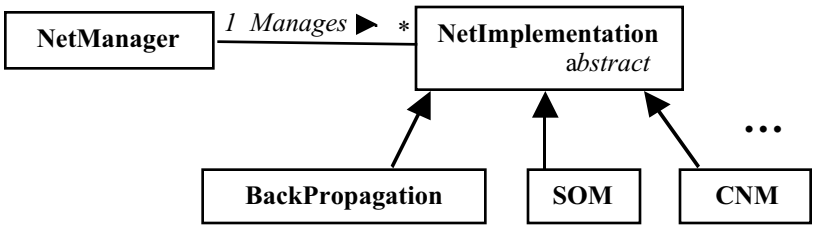

Figure 4 ANN models as subclasses of NetImplementation.

The neural network behavior

Specific ANN models also imply the need for specific behavior of the classes Neuron and Synapse. A simple solution would be to create subclasses of Neuron and Synapse, but this solution would generate a nested hierarchy because for each neural model, similar subclasses of Neuron and Synapses would be created. For example, in the case of CNM the subclasses would have the names CNMNeuron and CNMSynapse. CNMNeuron would factor out commonalties of the CNM-specific classes CNMInputNeuron, CNMCombinatorialNeuron and CNMHypothesisNeuron. For Backpropagation the same would be required.

To avoid this, the Bridge pattern was used (Gamma et Al. 1995). This pattern is equivalent to the separation Metapattern (Pree, 1996), and thus has the ability to change neural network behavior at run-time. Figure 5 shows the application of this pattern to the class Neuron. Its application is similar to the class Synapse. Neuron is an abstract class having the three most common neuron types as its subclasses: InputNeuron, HiddenNeuron, and OutputNeuron. These three classes are the most common in the ANN implementations. The names refer to the layers they belong to. The hidden layer can be of any size, and always reuse the HiddenNeuron class to build it. Any neural model is based on these Neuron classes.

The necessary behavior of each neural model is added to these classes by composition through the associated abstract class ComputationStrategy(see Figure 5). The different behaviors are implemented in subclasses of ComputationStrategy and can be used by different model implementations through the classes Neuron and Synapse. The class Synapse also has a relationship with the class ComputationStrategy.

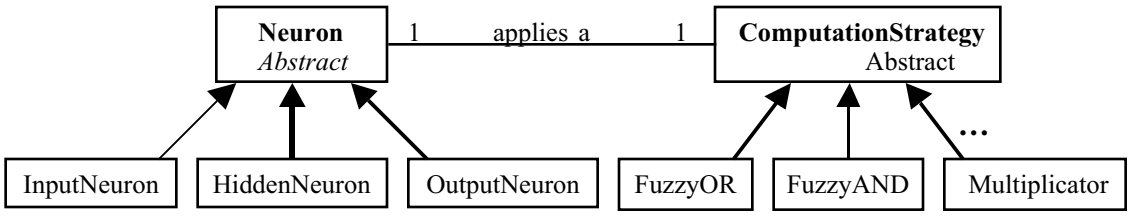

Figure 5 Design of flexible behavior based on Bridge Pattern.

The ComputationStrategy class also implements the pattern Flyweight (Gamma et Al. 1995). The framework has only one instance of each of its subclasses. Each instance is shared by a large number of Neuron and Synapse instances. This keeps the memory footprint significantly smaller and improves the behavior reusability.

An important design issue is that a developer who uses Java-ABC does not have to worry about which specific subclasses of Synapse and Neuron are associated with a particular ANN model. In order to resolve this, the Factory pattern (Gamma et al. 1995) was applied. A concrete subclass of NetImplementation such as CNM already takes care of the correct instantiation of Neuron and Synapse subclasses (see below).

CNM-Adaptation of Java-ABC

A sample adaptation of Java-ABC exemplifies the necessary steps to adjust the framework components to a specific neural network model. For this reason, it is necessary to discuss the inner working (see Figure 6):

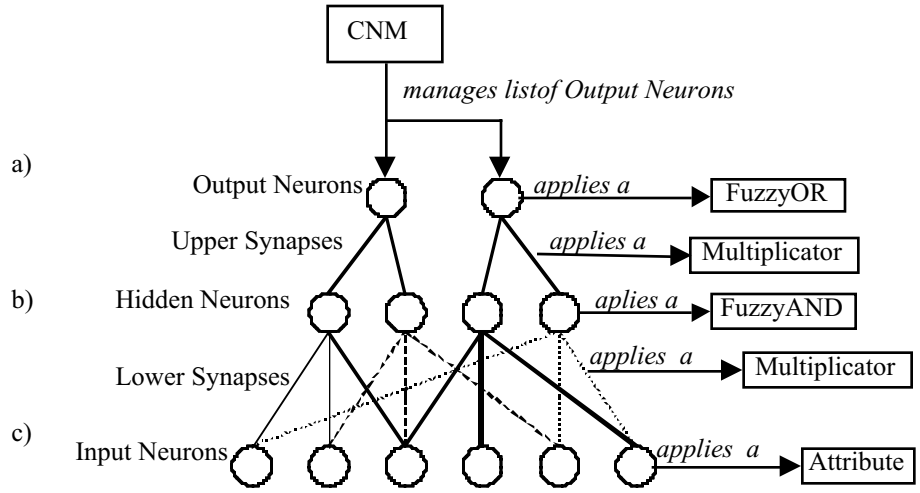

Figure 6 Building CNM architecture.

a) The CNM object is responsible for the creation of OutputNeuron objects (called Hypothesis neurons in the CNM definition, Machado 1990), with the FuzzyOR behavior.

b) An OutputNeuron instance then creates Synapse objects that automatically create HiddenNeuron instances (called combinatorial neurons in the CNM definition, Machado 1990), with the FuzzyAND behavior.

c) The connections between the HiddenNeuron instances and the InputNeuron instances are established in an analogous way. The Synapse instances of a CNM model have similar behavior to the Backpropagation model using the same Multiplicator behavior, that simple do the multiplication of the input activation with the synaptic weight and return the result.

For the neural network generation process, the NetImplementation subclasses (in this case the CNM object) rely on the problem-specific domain knowledge, whose representation is discussed in Section 3.3. The basic idea behind the CNM inference machine is that numerous combinations (the small networks that form the CNM hidden layer) all test a certain case for which a decision is necessary. Adding the activation from all combinations amounts to the OutputNeuron activation. The OutputNeuron with the maximum accumulated activation is the winner (FuzzyOR). The CNM object also provides an explanation by calculating those input neurons that most strongly influenced the decision. Machado and Rocha (1989, 1990) discuss the CNM-specific algorithm.

Adding behavior to a multilayer feedforward architecture

The computation of a case, either learning or testing, is done in the following way: The object NetImplementation knows the instances of output neurons that are implemented in the structure already created. The result value of a case computation is implemented by the output neurons' compute() method and can be retrieved by the getCurrentFlow() method (see Example 1 – class Neuron). The NetImplementation object requests computation

from the output neurons by calling the compute() methods of all existent output neurons. When the output neuron is requested to do its computation, it first requires its list of incoming synapses to do the same. The synapses also have a source neuron that they request to do its own computation (see Example 1 – class Synapse).

The source neuron is a hidden neuron in the architecture and its compute() method implementation also requests the computation of the connected synapses. In this way, the request of computation flows from the output neurons to the input neurons. The input neuron instance's behavior is simply to take the activation from outside to be able to start the ANN data processing. This activation comes from the class Attribute, explained in Section 3.3. In short, the Attribute classes prepare the activation values from the data read in the data sources. These prepared data (activation) is transferred to the input neuron instances on demand.

When the computation flows goes back from the input neurons to the output neurons, each synapse and neuron object then is able to do the necessary calculation it is supposed to do and return it to the object that requested (other neurons and synapses). The instances of class ComputationStrategy do this calculus. Finally, the compute method of each output neuron gets the computational results of all connected synapses and does its appropriate computation. The resulting values then can be consulted through the output neurons getCurrentFlow() method. The NetImplementation object is able to evaluate these values and make a decision.

The computational flow explained above is a parallel process internal to the neural network architecture. Instances of Synapse and Neuron in the same layer can be completely independent processes. Depending on the neural model, synchronization must be implemented in order get the correct results and to have optimal performance. The parallel implementation strategy of the computational flow is specific to each model and is not explained here.

To complete the appropriate behavior of the implemented neural networks, it is necessary to have them related to the knowledge representation of the problem domain. The next section explains how the domain model influences and interacts with the neural network architecture.

3.3 Domain representation and data conversion

As the principal application domain of Java-ABC is classification problems, the chosen object-oriented design of this system aspect reflects common properties of classification problems. On one hand, so-called evidences form the input data. Experts use evidences to analyze the problem in order to come up with decisions. Evidences in the case of the customer classification problem would be the age of a customer, his home address, etc. One or more Attribute objects describe the value of each Evidence object. For example, in the case of the home address, several strings form the address evidence. The age of a customer might be defined as a fuzzy set (Kosko 1992; da Rosa 1997) of values: child, youth, adult, and senior.

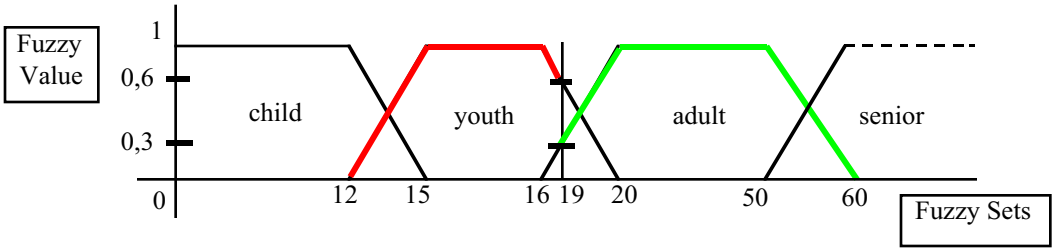

Figure 7 Fuzzy set example.

On the other hand, the hypotheses (classification categories) constitute a further core entity of classification problems. In Java-ABC an instance of class Domain represents the problem by managing the corresponding Evidence and Hypothesis objects. Class Domain again applies the Singleton pattern. Even based on classification problems and focused on neural networks learning algorithms, the design presented here can also be extended to support general domain representation for symbolic learning strategies. Edward Blurock (1998) goes deep in the domain representation for machine learning algorithms. Even being completely independent works, both lead to quite similar designs. Figure 8 shows the relationship among the classes involved in representing a particular domain.

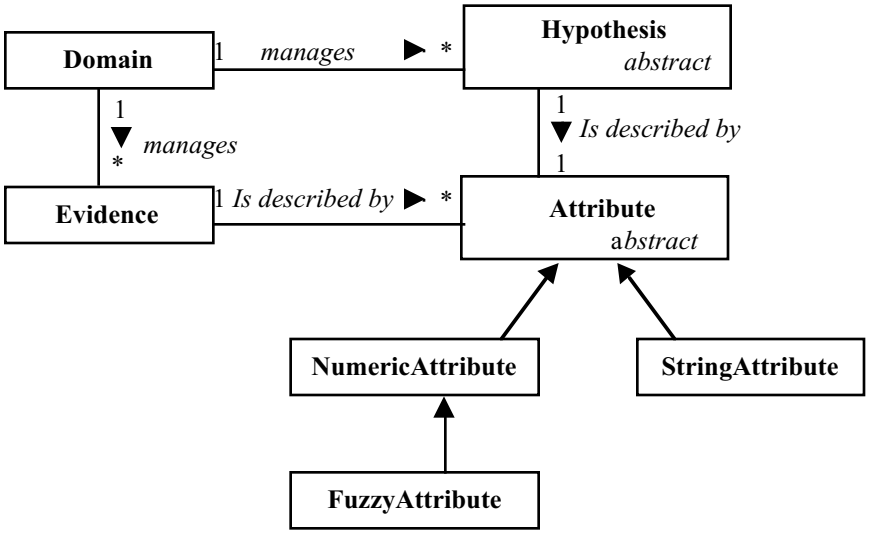

Figure 8 Domain representation.

The training and testing of an ANN are the principal features of Java-ABC. For both tasks, data must be provided. For example for training an ANN to classify customers, data might come from an ASCII file. One line of that file represents one customer, i.e. the customer's evidences and the correct classification. After training the ANN, customer data should be tested. To do that, Java-ABC gets the evidences of a customer as input data and must classify the customer. The data source might, in this case, be a relational database management system. It should be clear from this scenario that Java-ABC has to

provide a flexible data conversion subsystem. Data conversion must be flexible at run time, as the user may wish to change the data source anytime during learning or testing. Thus the Separation pattern discussed in Section 3 is the appropriate construction principle underlying this framework.

Two abstract classes constitute the framework for processing problem-specific data, class Fetcher and class EvidenceFetcher. Class Fetcher is abstractly coupled with the class Domain (see Figure 9). A Fetcher object is responsible for the preparation/searching operations associated with a data source. If the data source is a plain ASCII file, the specific fetcher opens and closes the file. This includes some basic error handling.

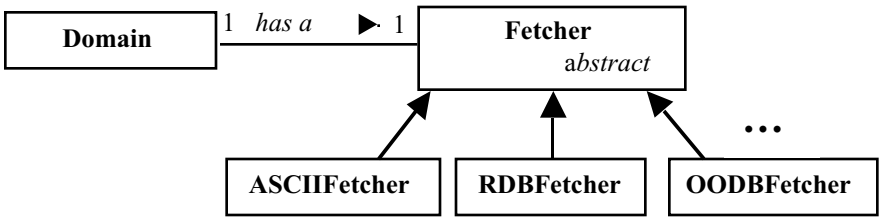

Figure 9 Dealing with different data sources.

Class Evidence and class Hypothesis are abstractly coupled with Class EvidenceFetcher (see Figure 10). Specific subclasses of EvidenceFetcher know how to access the data for the particular evidence. For example, an EvidenceFetcher for reading data from an ASCII file stores the position (from column, to column) of the evidence data within one line of the ASCII file. An EvidenceFetcher for reading data from a relational database would know how to access the data by means of SQL statements. Figure 10 shows the design of these classes, picking out only class Evidence. The class Hypothesis has an analogous relationship with the class EvidenceFetcher.

Note that the Attribute objects prepare the data from external sources so that they can be directly feed to the input neurons of the ANN (see Figure 6). This works in the following way: each Evidence instance fetches its value from the data source, and this value is applied automatically to all attributes the evidence has. Each attribute applies the conversion function that is inherent to the specific Attribute class. For example, the StringAttribute conversion function receives the string from the database and compares it to a given string modeled by the expert, returning 1 or 0 based on whether the strings match. This numeric value is stored by the attribute object and will be applied in the ANN input by request. The ANN input nodes have a direct relationship with the attributes of the evidence (see Figure 6). When the learning or testing is performed, each input node requests from its relative attribute the values previously fetched and converted. The attribute simply returns the converted value.

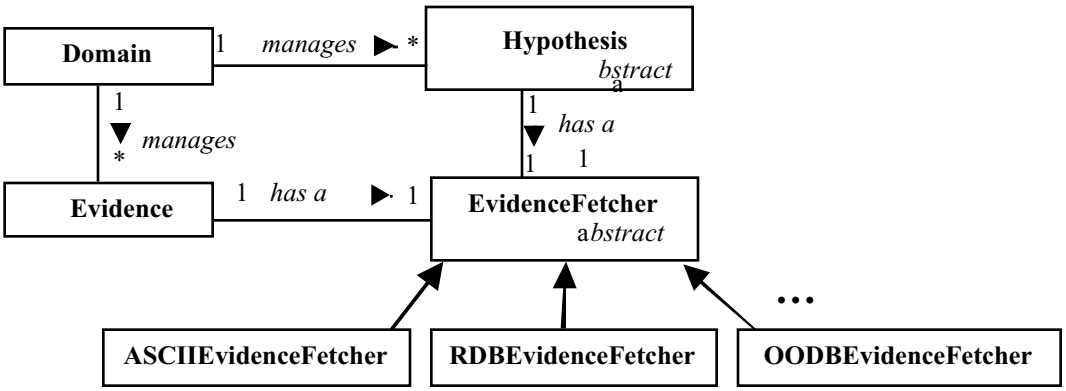

Figure 10 Data conversion at the evidence level.

Visual/interactive tools support the definition of the specific instances of EvidenceFetcher and Fetcher subclasses. For example, in the case of fetching from ASCII files, the end-user of Java-ABC who does the domain modeling, simply specifies the file name for the ASCIIFetcher object and, for the ASCIIEvidenceFetcher objects, specify the column positions in a dialog box.

4 Summary and conclusion

The chapter illustrates how the uncompromising application of framework technology leads to systems with appropriate flexibility. The chosen case study corroborates that a sound object-oriented design of neural network components delivers the expected benefits a conventional solution could not provide. The current design and implementation of Java-ABC is a generic system decision-making system based on neural networks. An ambitious goal is to enhance further the framework so that other decision support problems such as forecasting can be supported. Also ambitious is to allow the implementation of other learning mechanisms that do not rely only on neural networks such as machine learning algorithms.

This framework developed in the Java-ABC project is the base for the new Hycones implementation that is planned to be a new product for decision support. The framework design is giving flexibility and reliability to the system. Its goals are being expanded from a classificatory system with only one learning algorithm to the possibility of implementing many different learning algorithms such as clustering. Furthermore, some experiments of the technology as a data mining tool have been done. The design also allows the framework to easily add other implementation facilities such as parallelization and distribution.

Currently, most excellent frameworks are products of a more or less chaotic development process, often carried out in the realm of research-like settings. In the realm of designing and implementing Java-ABC we found that hot-spot-analysis is particularly helpful for coming up with a suitable framework architecture quicker. An explicit capturing of flexibility requirements can indeed contribute to a more systematic framework development process.

References

Blurock, Edward S., 1998. ANALYSIS++: Object-Oriented Framework for Multi-Strategy Machine Learning Methods. Paper submitted to OOPSLA'98. http://www.risc.uni-linz.ac.at/people.

da Rosa, S.V., Beckenkamp, F.G., and Hoppen, N. 1997. The Application of Fuzzy Logic to Model Semantic Variables in a Hybrid Model for Classification Expert Systems. Proceedings of the *Second International ICSC Symposium on Fuzzy Logic and Applications* (ISFL'97). Zurich, Switzerland

da Rosa, S.V., Leao, B.F., and Hoppen, N. 1995. Hybrid Model for Classification Expert System**.** Proceedings of the *XXI Latin American Conference on Computer Science*. Canela, Brasil.

Gamma, E., Helm R., Johnson R. and Vlissides J., 1995. *Design Patterns—Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison-Wesley

Haykin, S., 1994. *Neural Networks A Comprehensive Foundation*. Upper Saddle River, NJ, Prentice-Hall

Hopfield, J.J., 1982. Neural networks and phisical systems with emergent collective computational abilities. Proceedings of the *National Academy of Sciences of the U.S.A.* 79, 2554-2558.

Kohonen, T., 1982. *Self-organized formation of topologically correct feature maps*. Biological Cybernetics 43, 59-69.

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

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

Leao, B. F. and Reategui, E. 1993. Hycones: a hybrid connectionist expert system. Proceedings of the *Seventeenth Annual Symposium on Computer Applications in Medical Care - SCAMC*, IEEE Computer Society, Maryland.

Leão, B. F. and Reátegui, E. 1993a. A hybrid connectionist expert system to solve classificational problems. Proceedings of *Computers in Cardiology* , IEEE Computer, IEEE Computer Society, London.

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

Machado, R. J. and 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. and Rocha, A. F., 1990. The combinatorial neural network: a connectionist model for knowledge based systems. In B. Bouchon-Meunier, R. R. Yager, and L. A. Zadeh, editors, *Uncertainty in knowledge bases*. Springer Verlag.

Medsker, L. R. and 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.

Pree, W., 1995. *Design Patterns for Object-Oriented Software Development*. Reading, MA: Addison-Wesley/ACM Press

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

Pree, W., 1997. *Komponentenbasierte Softwareentwicklung mit Frameworks*. Heidelberg: dpunkt

Reategui, E. and Campbell, J. 1994. A classification system for credit card transactions. In Haton, J-P., Keane, M., Manago, M. (eds). *Advances in Case-Based Reasoning. Second European Workshop EWCBR-94.* Chantilly, France, November 94. Springer Verlag

Rumelhart, D.E., and McClelland, J.L., 1986. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Vol. 1. Cambridge, Ma: MIT Press.

Wirfs-Brock, R. and Johnson, R., 1990. Surveying Current Research in Object-Oriented Design. *Communications of the ACM*, 33(9)