On the Incorporation of Learning in Open Multi-Agent Systems: A Systematic Approach

José Alberto Rodrigues Pereira Sardinha
sardinha@inf.puc-rio.br

Alessandro Fabricio Garcia
afgarcia@inf.puc-rio.br

Carlos José Pereira de Lucena
lucena@inf.puc-rio.br

Ruy Luiz Milidiú
milidiu@inf.puc-rio.br

PUC-RioInf. MCC07/04, April 2004

Abstract: The development of large scale multi-agent systems (MASs) requires the introduction and structuring of the learning property throughout the design and implementation stages. In open systems and complex environments, agents have to reason and adapt through machine learning techniques in order to perform their goals. In this paper, we present a methodology to introduce learning techniques into software agents. To assist the design phase, we present a design pattern that guides the design of the learning property. The methodology and pattern are used in the construction of an open MAS for the Trading Agent Competition environment in order to illustrate the suitability of our approach.

Keywords: Design Patterns, Software Agents, Machine Learning, Software Engineering Methodologies**.**

Resumo: O desenvolvimento de grandes sistemas multi agentes necessita da inclusão e estruturação da propriedade de aprendizagem nas fases de *design* e implementação. Em sistemas abertos, agentes precisam raciocinar e se adaptar através de técnicas de *machine learning* para executar planos e atingir os seus objetivos. Nesse artigo, nós apresentamos uma metodologia para incluir técnicas de aprendizado em agentes de *software*. Para auxiliar a fase de *design*, nós apresentamos também um padrão de projeto para auxiliar o *design* da propriedade de aprendizado. A metodologia e o padrão de projeto são utilizados em um estudo de caso para o TAC (Trading Agent Competition).

Palavras-Chave: Padrões de Projeto, Agentes de Software, *Machine Learning*, Metodologias de Engenharia de Software.

1. Introduction

Multi-Agent Systems [1] [2] is a new technology that has been recently used in many simulators and intelligent systems to help humans perform several time-consuming tasks. To achieve the system's goal, agents have to react to events, define strategies, interact, and participate in organizations. Software agents have gained greater importance for both academic and commercial applications with the advent of the Internet. Applications for the Internet are easily modeled with agents, mainly because of their distributed nature. We believe that many other applications based on this technology are still to be built to help leverage the use of the Internet.

We define software agents [3][4][5] as autonomous entities driven by beliefs, goals, capabilities, and plans, and agency properties, such as adaptation, interaction and learning. In complex and open environments with many cooperating agents, it is important to have a system that is able to adapt to unknown situations. Learning techniques are crucial to the development of open MASs since they provide well-known strategies to support the construction of adaptable agents.

An issue that arises in large scale Multi-Agent Systems is the availability of resources. Machine learning is time consuming and it is unfeasible to allow every agent in the society to use the resources at the same time. Therefore, there is a need for a software engineering methodology for the disciplined introduction of learning properties in software agents through different development stages. This systematic approach helps the development team of an open MAS to include machine learning techniques in adaptive environments, and consequently, leverage the performance of the system.

The learning design pattern complements the methodology in the design phase, and is used after completing the machine learning design. The pattern's design also allows an easy mapping of the definitions in the analysis and design phase of our methodology to object-oriented design and code.

Kendall et all [6] proposes an agent framework that is an architectural pattern organized in layers. In this architecture, an agent is composed of seven layers, such as the layer of sensors that are responsible for detecting changes in the environment. These patterns enable the modeling of both simple and complex agents. The reasoning layer is similar to our Learning Design Pattern, but is not general enough to encompass all machine learning strategies. Moreover, it is not clear to a system designer and programmer how to design and implement complex reasoning agents in a systematic way.

Our methodology is unique because it guides the introduction of learning properties in software agents through all development stages. In frameworks such as JADE [10] and Kendall et all [6], this easy guide is not presented in a structured format. Our design pattern is more specific and general enough to include all machine learning techniques. Consequently, the system is easier to implement and reuse.

Machine Learning toolkits [8][9] have good and efficient implementations of algorithms, but are not suitable for a novice user. To use a toolkit, the user needs to have skills to model the problem and take decisions on design issues. The most difficult phase of a machine learning implementation is the design phase, where

crucial decisions can lead to a very successful or catastrophic expert system. In our methodology, we introduce some guidelines for a good design of machine learning techniques in multi agent systems.

The design pattern and methodology is used in three implementations: (a) A multi agent system [10][11] that uses evolutionary techniques to build offerings in a retail market (b) An agent system [12] that learns to play Tic-Tac-Toe with no prior knowledge; and (c) in a multi agent system [13] for the Trading Agent Competition (TAC) [14]. In section 2 and 3, we present the methodology for introducing machine learning techniques in large scale multi-agent systems and the Learning Design Pattern. An example is presented in section 4 to explain the use of the methodology and design pattern in a multi agent system for the TAC Competition.

2. Introducing Learning Techniques in Multi-Agent Systems

An important issue that arises when we introduce learning properties in large scale Multi-Agent Systems is how to introduce intelligence and knowledge acquisition in a society of agents. Machine learning is time consuming and it is unfeasible to allow every agent in the organization to use the computational resources at the same time. We present a methodology to introduce learning techniques in multi-agent systems built over a distributed environment and with limited resources.

2.1 A Methodology for Introducing Machine Learning Techniques

We assume that an intelligent agent can improve the performance of the society when compared to a same agent role implemented as a reactive agent. This methodology has six phases: Agent Selection, Problem Domain Analysis, Machine Learning Design, Implementation, Training, and Testing & Evaluation. In figure 1, we depict all the phases of the methodology.

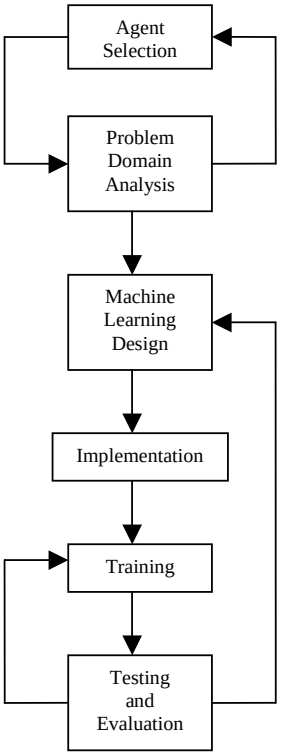

Fig. 1. The methodology for introducing Learning Techniques

2.2 Agent Selection and Problem Domain Analysis

After the definition of agent types or roles and the description of the agent behavior with a modeling language, we shall identify the best agents that will receive a learning and intelligence behavior. An important definition in this phase is the organization's learning problem. This learning problem is defined as the Organization's Goal, *OG*, and the Organization's Performance Measure, *OP*. Normally in a design of a MAS, each agent is concerned with a sub problem, which can be solved by applying a specific machine learning technique. The combination of these solutions must achieve the organization's goal and leverage the organization's performance measure.

The Agent Selection is conducted in an ad-hoc manner. The designer of the system notices that a specific agent has a complex plan to perform and needs a machine learning technique in order to improve the performance of the system. Moreover, for every selected agent, a problem domain analysis is performed to identify important learning issues. This phase is the Problem Domain Analysis and has the goal to establish a well-defined learning problem. In [15], this learning problem is defined with three features: a Task *T*, a Performance measure *P*, and a Training experience *E*. For example, a retail negotiation agent in a market place has the following training features: *T* : negotiating goods; *P* : total net income; *E* : a negotiation log of other agents in the system. In some cases, the selected agent can modify the behavior of other agents or alter interaction protocols.

2.3 Machine Learning Design

In the Machine Learning Design Phase, we make the following questions: *How am I going to get my agent to learn this?* and *What kind of evaluation function should I use to measure the performance of the agent after the learning process?* In [16], a general model of learning agent is presented that can be used in this phase for both logical systems and belief networks. A performance element is designed to inform the learning component how well the agent is doing. A problem generator is also designed to suggest actions that will lead to new experiences. Exploration is an important element for learning agents that are willing to discover much better actions for the long run.

In [15], another interesting Machine Learning Design phase is proposed in four phases: 1- Determine a Type of Training Experience; 2 – Determine a Target Function; 3 - Determine a Representation of Learned Function; and 4 - Determine a Learning Algorithm. These four phases illustrate some basic issues and approaches to machine learning systems.

In the first phase, we deal with the design choice of which type of training experience the system will have to learn. A key attribute of the training experience is whether it is direct or indirect from the training examples. For example, our Negotiation Agent can learn directly from a detailed behavior log of other agents in the system. Alternatively, this agent might not have the access to such information and has to learn with its own behavior and the interaction with other agents. This learning experience is indirect because the agent will build its knowledge through the final results of the negotiations and its own behavior that led to such results.

The second phase determines exactly what type of knowledge will be learned and how it will be used by the performance program. In our example, the Negotiation Agent needs to learn how to choose the best bid in order to buy a required good. Our target function for the Negotiation Agent can be called *NextBid*, and could be modeled as a function that accepts a state S of the environment and produces a value B of the next bid. This gives us the following target function *NextBid*: $S \rightarrow B$. It may be very difficult in general to perfectly learn this target function, and normally we reduce the complexity and transform the problem to learn only some approximation of the target function. In the negotiation agent, it might only use only a limited subset of its environment, such as the current prices of the market and with this information produce the next bid.

In the third phase, we must choose a representation that the agent will use to describe the approximate target function. This representation can be described as a linear weighted function, a collection of rules, a neural network, or a quadratic polynomial function. In general, this design choice involves an important tradeoff because we would like to pick a representation that is as close as possible to the ideal target function. However, an expressive target function requires more training data in the training phase.

In order to learn the approximate target function we will need a training set. These training examples are obtained through a direct or indirect experience. In the direct

experience, the designer can carefully select the best training that leads to the approximate target function. However, the indirect experience requires a design that suggests actions that will lead to already known states that improve the performance of the system, and unknown states that guide to new experiences. As mentioned above, exploration is important for indirect learning agents that are willing to discover much better actions for the long run.

2.4 Implementation, Training, Testing and Evaluation

The implementation phase transforms the models into code, and leads to the next phase that is training. The training experience selected in the Problem Domain Analysis and Machine Learning Design is presented to the agent through training sessions. This training session consists of one or more Agents that learn based on a selected training set, or through interaction with other agents in a controlled training environment. Some adjustments in learning parameters are made at this time.

Testing phase starts with unit testing of the implementation and learning property. After the unit testing, integration with the Multi-Agent System is done and the testing of the performance is evaluated. A performance improvement indicates that the learning property is a successful strategy. Normally, some modifications in the machine learning design and learning parameters are made to improve the performance of the system.

In some systems, our selected agents might need to learn from the interaction with other agents or the environment through all the life cycle of the system. Consequently, the agents have to undergo training sessions from time to time. It is important to remember that machine learning techniques are time consuming and that computational resources are always limited. A careful policy is required to organize which agents are submitted to these training sessions and at what time. Moreover, a performance measure has to be monitored at all time in order to detect flaws of the system.

3. The Learning Design Pattern

3.1 Intent

The Learning pattern supports the process of including cognition in Software Agents. It separates the key aspects involved in the design of machine learning techniques. This pattern shall be used in the design phase of the methodology in section 2, and guides the machine learning design to object-oriented design. Consequently, we can achieve an easy mapping to code.

3.2 Context

Cognitive agents need to modify their knowledge based on their experience on the course of its interaction with the environment and other agents. When a relevant internal or external event is triggered, the learning process in a software agent starts through a direct or indirect experience. This learning process adapts the agent's

knowledge and enables the agent to new conclusions and decisions. A performance measure is also needed to detect the efficiency of the machine learning design. The separations of these key learning aspects create components that are easier to design, maintain, and reuse.

Although this pattern is designed for each agent individually, a designer must not forget to make these intelligent agents cooperate and achieve the organization's goal. Every agent needs an individual performance measure that calculates if it is learning well or not, but another important entity, which is not in this design, is a performance measure of the organization. This entity is able to evaluate if the organization is achieving the predefined goals.

3.3 Motivation Example

In the TAC Agency, presented in more detail in section 4, the Hotel Negotiator Agent needs an efficient strategy to send bids to auctions. The open environment is constantly changing because every new game instance can have different participants. A fixed plan would not lead the agent to a good performance. In order to design this agent, we used a combination of a Temporal Difference Learning [15] and Neural Network [16] strategy.

3.4 Problem

The introduction of cognition in agents through machine learning techniques is not straightforward. There are several algorithms and many different ways to design the knowledge and training experience that will be used by the learning strategy. Many commercial and academic toolkits with learning algorithms are available in the market. However, none of these toolkits guides the MAS designer how to introduce the learning property into the system in a systematic way.

3.5 Solution

The design pattern separates the key aspects that are involved in the cognition design of each individual agent, and it presents an easy mapping of the Problem Domain Analysis and Machine Learning Design phase to an object-oriented design.

The performance measure modeled in the Problem Domain Analysis is coded in a separate entity called Performance Measure. It used by the learning property to guarantee it is achieving the predefined goals. The representation of learned function or approximate target function in the Machine Learning Design phase is implemented in an entity called Knowledge Representation.

The learning algorithm is a separate entity called Learning Algorithm, which is responsible for modifying the approximate target function modeled in the Knowledge Representation. The learning algorithm alters this approximate target function through a direct or indirect experience. This is modeled as a separate entity called Training Experience.

3.6 Structure

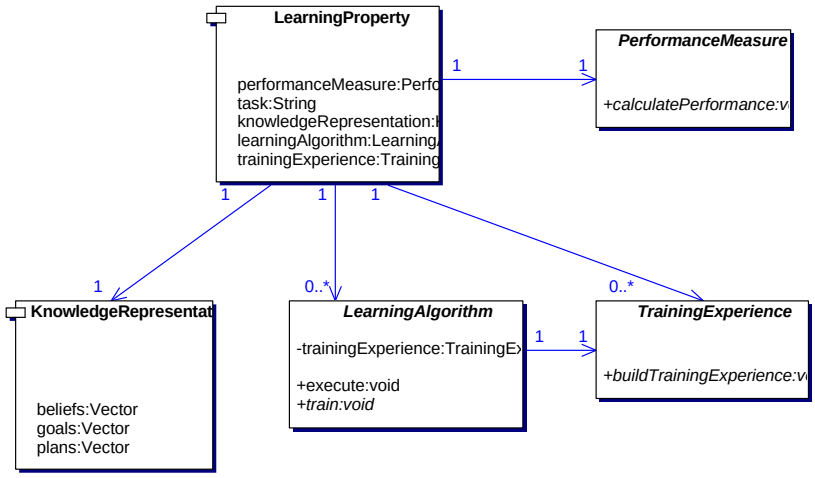

Fig. 2. Learning Design Pattern

The Learning pattern has three main participants and two client participants:

Main Participants:

Learning Property

Defines the main interface of Learning Property and implements the Façade design pattern [17].

Performance Measure

Defines the implementation of an algorithm that is going to evaluate the Learning property performance gain.

Learning Algorithm

This algorithm implements one or more machine learning algorithms that will modify the Knowledge Representation. It implements the Strategy pattern [17].

Client Participants:

Knowledge Representation

The learning knowledge entity. It is also used by the Performance Measure.

Training Experience

Learning data, or an algorithm that gathers information for the learning process.

The Knowledge Representation is an important entity. It determines exactly what type of knowledge will be learned and how it will be used by the Performance Measure. This entity can be modeled as a linear weighted function, a collection of rules, a neural network, or a quadratic polynomial function. The design choice involves an important tradeoff because we would like to pick a representation that is as close as possible to an ideal representation. However, an expressive representation requires more training data in the training phase.

The training examples are obtained through a direct or indirect experience, and are modeled as the Training Experience entity. In the direct experience, the designer can carefully select the best training that leads to the best representation of the Knowledge Representation. However, an indirect experience requires a design that suggests actions that will lead to new experiences in unknown states, and also guide to already known states that improve the performance of the system. Exploration is important for indirect learning agents that are willing to discover much better actions for the long run. The Training Algorithm implements the code that will use the Training Experience to build the Knowledge Representation.

3.7 Dynamics

Every time an important event is triggered, the agent starts the training process. The main interface to the learning pattern is the LearningProperty. This LearningProperty calls the method *buildTrainingExperience* that is responsible for implementing code that will obtain examples through the direct or indirect experience. These training examples are used by the LearningAlgorithm, which is called through the method *execute*. The actual training algorithm is coded in train and is evaluated by PerformanceMeasure to calculate the achievement of the training algorithm. If the training algorithm is evaluated well, then the KnowledgeRepresentation is updated.

Now the agent can use the updated knowledge to perform different plans, use different goals and beliefs. In environments that are changing constantly, it is important to have agents that are able to learn with these changes and start performing new plans to achieve the goal of the organization.

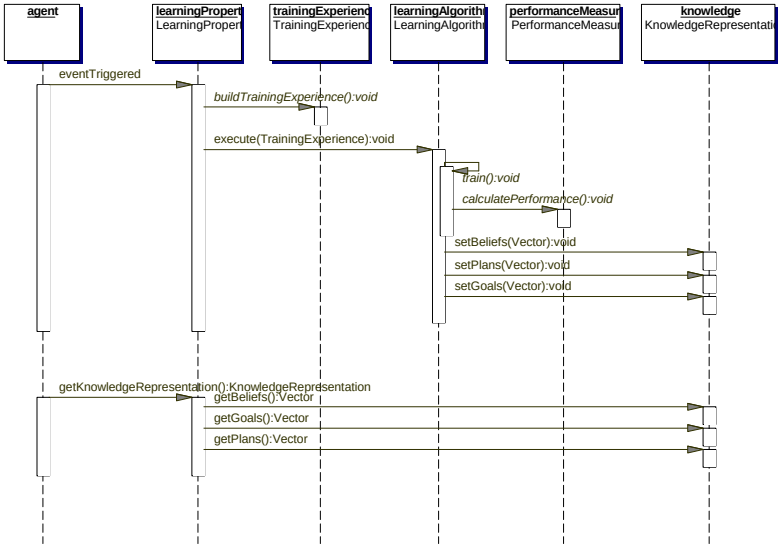

Fig. 3. The Sequence Diagram of the Learning Design Pattern

3.8 Consequences

Reusability. The pattern modularizes a generic learning property that can be reused and refined in different contexts.
Improved Separation of Concerns. The learning property is entirely separated from interaction and adaptation.
Easier implementation. The pattern presents an easy mapping of the Problem Domain Analysis and Machine Learning Design phase to an object-oriented design.

3.9 Variants

Aspect-Oriented Solution. The Learning Property entity can be implemented as an aspect and improve the separation of concerns. It can connect the executions points (events) on different agents classes, and identify when the learning process should be triggered. These are some of the additional advantages of using an aspect oriented solution:

> ***Transparency***. Aspects turns out to be an elegant and powerful approach that can be used to introduce the learning behavior into agent classes in a transparent way. The description of which agent classes need to be affected is present in the aspect and these monitored classes are not modified.
> ***Ease of Evolution.*** As the MAS evolves, new agent classes may have to be monitored and trigger the learning process. Developers only need to add new pointcuts in the Learning Property aspect in order to implement the new required functionality.

4. The Trading Agent Competition MAS

The Trading Agent Competition (TAC) [14] is an international forum designed to encourage high quality research on competitive trading agents. The multi-agent system in TAC operates in a shopping scenario of goods for traveling purposes. The artificial agents are travel agents that buy and sell airplane tickets, hotel rooms, and entertainment tickets for clients. TAC scores are based on the client's preferences for trips, and net expenditures in the travel auctions.

In TAC, each agent has a goal of assembling travel packages. Every package is from TACtown to Tampa, for a 5-day period. Each agent is acting on behalf of eight clients, who express their preferences for various aspects of the trip. The objective of the travel agent is to maximize the total satisfaction of its clients, with the minimum net expenditure in the travel auctions. The satisfaction is defined as the sum of all client utilities.

A run of the game is called an instance. Several instances of the game are played during each round of the competition in order to evaluate each agent's average performance and to smooth the random variations in clients' preferences. Each game instance takes twelve minutes.

Our travel agent is modeled as a multi-agent system that trades in the related auctions of an instance. Trading problems were identified in the goal view of A-note, such as: calculate best allocations, predict auction prices and calculate demand segmentation. Each agent in our TAC Agency is concerned with one or more of these trading sub-problems. This allows us to apply different computational techniques to solve the sub-problems separately and then combine the solutions. This is an evolution of the previous work conducted by Milidiu et al. [18].

4.1 Introducing Machine Learning Techniques in the TAC Agency

A similar version of the agency modeled above was implemented by Milidiu et al. [18]. The agency is called SIMPLE and is composed of reactive agents with a service of an integer programming model to obtain the optimal allocation of available goods for the customers.

An agent with similar features (LA-clone) of the LivingAgents [19], the winner of the 2001 competition, was implemented to enhance the testing environment of SIMPLE. The best strategy of SIMPLE provided a 56,2% winning rate, when competing against one LA-clone and six Dummies. Table 2 presents the results of SIMPLE and LA-Clone.

Agent	**Average Score**	**Games**
SIMPLE	1740	32
LA-Clone	1390	32

Table 1. Experimental results of SIMPLES against LA-Clone and 6 dummies

Our goal is to improve the performance of the agency by using the methodology and design pattern presented in section 2 and 3. Therefore, our first step in the methodology is to define the organization's learning problem:

Organizations's Goal, *OG* : Obtain first place in the competition
Organization's Performance Measure, *OP* : Average Score

4.1.1 Agent Selection and Problem Domain Analysis

We notice that two agents have complex plans to perform and need a machine learning technique in order to improve the performance of the system. The first agent is the Hotel Negotiator Agent and the second is the Price Predictor Agent. A problem domain analysis is performed to identify the type of problem, performance measure and training experience.

The learning problem is defined with three features: a Task *T*, a Performance measure *P*, and a Training experience *E*. The Hotel Negotiator Agent has the following training features:

T : negotiating hotel rooms;
P : number of goods purchased from the Ordering Agent;
E : interaction in the game instance.

The Price Predictor Agent has the following training features:

T : predict future ask prices for hotel rooms;

P : difference between the predicted and real price;
E : history of ask prices.

4.1.2 Machine Learning Design

In the Machine Learning Design Phase, we use the four phases presented in [15]: 1- Determine a Type of Training Experience; 2 – Determine a Target Function; 3 - Determine a Representation of Learned Function; and 4 - Determine a Learning Algorithm.

4.1.2.1 Training Experience Type

In the first phase, we deal with the design choice of which type of training experience the system will have to learn. The training experience of the Hotel Negotiator Agent is indirect from the training examples, because the agent will build its knowledge through the final results of the negotiations. To be exact, it compares the number of acquired goods with the number of goods to be purchased from the Ordering Agent. The Price Predictor Agent uses a direct knowledge building, since it uses the ask prices in the past to predict the next one.

4.1.2.2 Target Function

The second phase determines exactly what type of knowledge will be learned and how it will be used by the performance program. The target function for the Hotel Negotiation Agent is called *NextBid*, and it is modeled as a function that accepts a state S of the environment modeled in the Ontology Diagram and produces a value B of the next bid. Therefore, the target function is *NextBid: S → B*. The target function for the Price Predictor Agent is called *NextAskPrice*, and accepts the ask price A of the last game instances and produces the next ask price N. Consequently, the target function for the Price Predictor Agents is *NextAskPrice:A→N*.

4.1.2.3 Learned Function Representation

In the third phase, we must choose a representation that the agent will use to describe the approximate target function. In the Hotel Negotiator Agent, we use a min-max procedure [16] and reinforcement learning technique as the evaluation function for the min-max algorithm.

The min-max procedure is a search technique for a two player game that decides the next move. In this game there are two players: MAX and MIN. A depth-first search tree is generated, where the current game position is the root. The final game position is evaluated from MAX's point of view, and the inner node values of the tree are filled bottom-up with the evaluated values. The nodes that belong to the MAX player receive the maximum value of the children. The nodes for the MIN player will select the minimum value of the children. The min-max procedure is also combined with a pruning technique called Alpha-Beta [16].

Reinforcement learning [15] is different from supervised learning [15], since the agent is not presented with a learning set. Instead, the agent must discover which actions yield the most reward by trying them. Consequently, the agent must be able to learn

from the experience obtained from the interaction with the environment and other agents. A challenge that arises in reinforcement learning is the tradeoff between exploration and exploitation. Most rewards are obtained from actions that have been experienced in the past. But to discover such actions and to earn better selections, the agent must explore new paths and eventually fall in to pitfalls.

Every state of the min-max search tree is modeled with the following information obtained from the environment: (i) AskPrice – current Ask Price; (ii) LastAskPrice – Ask Price of the last minute; (iii) deltaAskPrice = AskPrice – LastAskPrice; (iv) Gama – A Constant; (v) Bid – Current Bid; and (vi) LastBid – Bid sent in the last minute.

The calculation of the next bid sent to the game instance is based on the decision taken with the min-max procedure. Three decisions can be taken by the decision tree:

1 – Bid = LastBid + 0.5*Gama*deltaAskPrice
2 – Bid = LastBid + 2*Gama*deltaAskPrice
3 – Bid = LastBid + 5*Gama*deltaAskPrice

The MIN player is modeled as the market response to the bid sent by our player (representing the MAX player). The market response is modeled as three possible results:

1 – AskPrice = AskPrice + 0.5*Gama*deltaAskPrice
2 – AskPrice = AskPrice + 2*Gama*deltaAskPrice
3 – AskPrice = AskPrice + 5*Gama*deltaAskPrice

For the evaluation function, we use a single-layer perceptron with only one artificial neuron. This perceptron receives a state of the min-max search tree as an input and scores the state between 0 and 1. The main goal of this learning technique is to adapt the weights of the perceptron to encounter an evaluation function that leads to good final states. These good final states represent environment states where bids are sent to the Game Instance and are able to purchase all ordered goods with the least net expenditure.

The Price Predictor Agent uses a simple representation to describe the approximate target function. The following formula is used to predict the next price:

$$PredictedAskPrice(n) = \alpha * AskPrice(n-1) + (\alpha - 1) * PredictedAskPrice(n-1)$$

where α is a number between 0 and 1; and n is the n-th game instance.

4.1.2.4 Determine a Learning Algorithm

In order to learn the approximate target function we will need a training set. These training examples are obtained through a direct or indirect experience. The Hotel Negotiator Agent uses an indirect experience because it uses data from the environment to adapt the weights of the perceptron rule:

$$wi = wi + \eta . (y(t) - d(t)) . xi,$$

where wi is the i-th weight of the perceptron; xi is the i-th data input of the state in the min-max search tree; y(t) is the actual result of the evaluation function; d(t) is the expected result of the evaluation function; and η is the learning rate.

Whenever the game instance ends up, a Perceptron rule is computed for all of the traversed intermediate and final states. Moreover, the perceptron adapts its weight using the Perceptron rule above and a TD-Learning strategy:

d(t-1)=d(t) + ß (reward(t) + (d(t)-d(t-1))),

where d(t) is the expected output of the decision point at time t; ß is the Reinforcement learning rate; reward(t) is the reward obtained in the decision point at time t.

The value of d(t) that represents a final states is computed with a reward value of 1 when it accomplishes the goal of purchasing all the goods in the purchase order, or a reward value of 0 in all other states. For final states d(t) is calculated using the following formula:

d(t) = reward(t)

The Price Predictor Agent uses a Least Mean Squares (LMS) strategy to compute the value of α:

α(n) = α(n-1) + ß * (AskPrice(n-1)-PredictedAskPrice(n-1))

where ß is a learning rate.

In order to learn the approximate target function we will need a training set. The Price Predictor Agent uses a direct experience. We selected the 50 last ask prices and predicted ask price to train the approximate target function.

4.1.3 Testing and Evaluation

To evaluate the performance of each learning strategy, we first tested the TAC Agency only with the Hotel Negotiator Agent. Consequently, we excluded the Price Predictor Agent and the Ordering Agent only sent purchase orders for flight at the end of the Game Instance. This strategy is used because the Allocation Agent is using the current ask prices of the environment, and it is safer to buy the flight tickets after the hotel rooms are acquired.

The agent with similar features (LA-clone) of the LivingAgents was tested with the TAC Agency with only the Hotel Negotiator Agent. The best strategy of the TAC Agency provided a 62,3% winning rate, when competing against one LA-clone and six Dummies. Table 2 presents the results of TAC Agency and LA-Clone.

Agent	Average Score	Games
TAC Agency	1920	32
LA-Clone	1355	32

Table 2. Experimental results of TAC Agency against LA-Clone and 6 dummies

The agent with similar features of the LivingAgents was again tested with the TAC Agency with both the Hotel Negotiator Agent and the Price Predictor Agent. The best strategy of the TAC Agency provided a 73,4% winning rate, when competing against one LA-clone and six Dummies. Table 3 presents the results of TAC Agency and LA-Clone.

Agent	Average Score	Games
TAC Agency	2592	32
LA-Clone	1323	32

Table 3. Experimental results of TAC Agency against LA-Clone and 6 dummies

4.2 The Learning Property of the Hotel Negotiator Agent in the TAC Agency

In this section we present the class diagram used for the Hotel Negotiator Agent. We used the MAS Framework [20] to implement the TAC Agency. The HotelNegotiatorAgent and HotelNegotiatorAgentIP are specialized classes that code the software agent. Details on how to instantiate a software agent with the MAS Framework can be found in [20].

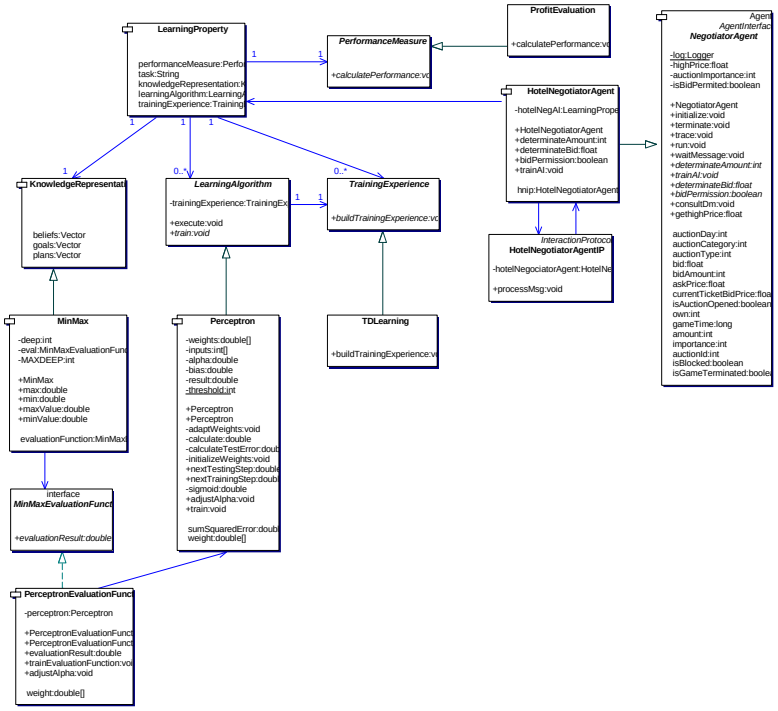

Fig. 4. The Class Diagram of the Hotel Negotiator Agent

The MinMax and PerceptronEvaluationFunction are the classes that implement the Knowledge Representation as described in section 4.1.2.3. The Learning Algorithm as explained in section 4.1.2.4 is coded in the class Perceptron. Although the TD Learning is a Learning Algorithm, we use this Reinforcement Learning strategy as method to build an indirect experience for the Perceptron. Therefore, we implement it as a specialization of the Training Experience class. The ProfitEvaluation class has the Performance measure described in section 4.1.1.

5. Final Comments

The methodology and pattern emerged from the long-term application of our method to different multi-agent systems. We believe there is a need for a software engineering methodology for the disciplined introduction of learning properties in software agents through different development stages. This systematic approach helps the development team of an open MAS to include machine learning techniques in adaptive environments, and consequently, leverage the performance of the system.

The learning property is entirely separated from other agent concerns and environment classes. Consequently, the code can modularize a generic learning property that can be reused and refined in different contexts. The pattern also presents an easy mapping of the Problem Domain Analysis and Machine Learning Design phase to an object-oriented design. Our case study explains in detail the use of our methodology and pattern, and presents results that encourage the use of learning in a multi agent system. The TAC Agency has a 48.97% performance gain over a similar agency with only reactive agents.

References

[1] Weiss, G.: Multiagent systems: a modern approach to distributed artificial intelligence. The MIT Press, Second printing, 2000.

[2] Ferber, J.: Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence. Addison-Wesley Pub Co, 1999.

[3] Garcia, A.; Silva, V.; Lucena, C.; Milidiú, R.: An Aspect-Based Approach for Developing Multi-Agent Object-Oriented Systems. Simpósio Brasileiro de Engenharia de Software, Rio de Janeiro, Brasil, Outubro 2001.

[4] Garcia, A.; Lucena, C. J.; Cowan, D.D.: Engineering Multi-Agent Object-Oriented Software with Aspect-Oriented Programming. Practice & Experience, Elsevier, May 2001.

[5] Garcia, A.; Lucena, C. J.: An Aspect-Based Object-Oriented Model for Multi-Agent Systems. 2nd Advanced Separation of Concerns Workshop at ICSE'2001, May 2001.

[6] Kendall, E.; Krishna, P.; Pathak, C.; Suresh, C.: A Framework for Agent Systems. In: Implementing Application Frameworks – Object-Oriented Frameworks at Work, M. Fayad et al. (editors), John Wiley & Sons, 1999.

[7] Telecom Italia Lab: JADE Programmer's Guide, http://sharon.cselt.it/ projects/ jade/ doc/ programmersguide.pdf, Feb. 2003.

[8] Computer Associates (CA) CleverPath web site: http://www.ca.com/

[9] DB2 Business Intelligence web site: http://www-306.ibm.com/ software/ data/ db2bi/

[10] Milidiu, R.L.; Lucena, C.J.; Sardinha, J.A.R.P.: An object-oriented framework for creating offerings. 2001 International Conference on Internet Computing (IC'2001) June 2001.

[11] Sardinha, J. A. R. P.: VGroups – Um framework para grupos virtuais de consumo. Master's dissertation – Departamento de Informática – PUC-Rio. March 2001.

[12] Sardinha, J. A.; Milidiú, R. L.; Lucena, C. J. P.; Paranhos, P. M.: An OO Framework for building Intelligence and Learning properties in Software Agents. Proceedings of the 2nd International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS 2003) at ICSE 2003, Portland, USA, May 2003.
[13] Sardinha, J.A.R.P.; Choren, R.; Milidiú, R.L.; Lucena, C.J.P.: Engineering Machine Learning Techniques into Multi Agent Systems. Submitted to INTERNATIONAL JOURNAL OF SOFTWARE ENGINEERING & KNOWLEDGE ENGINEERING.
[14] TAC web site.: http://www.sics.se/tac.
[15] Mitchell, T. M.: Machine Learning. McGraw-Hill, 1997. ISBN 0070428077.
[16] Russell, S. et al.: Artificial Intelligence. Prentice Hall, 1995. ISBN 0-13-103805-2.
[17] Gamma, E. et al.: Design Patterns – Elements of Reusable Object-Oriented Software. Addison Wesley, 1995. ISBN 0201633612.
[18] Milidiú, R. L.; Melcop, T.; Liporace, F.; Lucena, C.: SIMPLE – A Multi-Agent System for Simultaneous and Related Auctions. IV Encontro Nacional de Inteligência Artificial 2003. SBC 2003.
[19] Fritschi, C.; Dorer, K.: Agent-oriented software engineering for successful TAC participation. Proceedings of the first international joint conference on Autonomous agents and multiagent systems. 2002.
[20] Sardinha, J.A.R.P.; Ribeiro, P.C.; Lucena, C.J.P.; Milidiú, R.L.: An Object-Oriented Framework for Building Software Agents. Journal of Object Technology. January - February 2003, Vol. 2, No. 1.