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

José A. R. P. Sardinha, Alessandro Garcia, Carlos J. P. Lucena, Ruy L. Milidiú

TecComm Group (LES), Computer Science Department, PUC-Rio
Rua Marques de São Vicente 225, Gávea, Rio de Janeiro, Brazil
{sardinha,afgarcia,lucena,milidiu}@inf.puc-rio.br

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 this process, we present a design pattern that guides the structure of the learning property to object oriented design. The methodology and the pattern are used in the construction of an open MAS for the Trading Agent Competition environment in order to illustrate the suitability of the approach.

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. 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 in the environment. Machine learning is time consuming and it is unfeasible to allow every agent in the society to use system 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. This methodology also helps to define some important issues that appear when learning techniques are introduced in large scale multi-agent systems, such as: (i) How to evaluate the goal of the multi-agent system? (ii) How to define the individual goal of each agent in the system and evaluate it? (iii) How the knowledge of each individual agent is going to be modeled? (iv) How this

agent will acquire the knowledge? (v) To which agent(s) the knowledge will be associated with?

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 design and implementation 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 simple 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 novice engineers of intelligent systems. Software engineers need large experience on the application of learning machine techniques to use a toolkit; they need 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 the methodology have been used in four 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) a multi agent system [13] for the Trading Agent Competition (TAC) [15], (d) a multi agent system [14] for managing the paper submission and selection process in workshops and conferences. Section 2 presents a case study that involves a multi agent system for the TAC Competition. Sections 3 and 4 present the methodology for introducing machine learning techniques in large scale multi-agent systems and the Learning Design Pattern. Section 5 presents our conclusions.

2 The Trading Agent Competition MAS – A Case Study

The Trading Agent Competition (TAC) [15] 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. In the analysis phase, we identified the trading problems, such as: calculate the 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. [16], which called the previous version of this agency as SIMPLE. This MAS was implemented with reactive agents [1], and a service of an integer programming model to obtain the optimal allocation of available goods for the customers.

An agent called LA-Clone with similar features to LivingAgents [17], the winner of the 2001 TAC competition, was implemented to be used as a benchmark and enhance the testing environment of SIMPLE. The strategy of SIMPLE already reported a 56,2% winning rate in [16], when competing against one LA-clone and six Dummies [15]. After 32 instances played, SIMPLE obtained an average score of 1740, and LA-Clone had an average score of 1390.

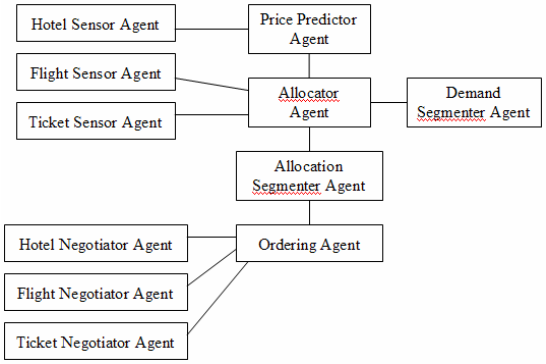

Fig. 1. The TAC Agency

The agency presented in this paper is called TAC Agency, and is depicted in figure 1. The Hotel Sensor Agent, Flight Sensor Agent and Ticket Sensor Agent are responsible for collecting data, monitoring the market, and updating a common

knowledge base of the agency. The Price Predictor Agent has the duty of predicting hotel auction prices. The first plan of this agent is to predict prices in the first minute of the instance of the game. This strategy predicts closing ask prices (ask price when hotel auctions have closed - calculated as the 16th highest price among all bid units). The second plan of this agent is executed after the third minute and in every successive minute until the hotel auctions are closed. This strategy of predicting the ask prices is extremely important for the Allocator Agent, Allocation Segmenter Agent, and Ordering Agent. Together they cooperate to buy flight tickets in the first minute and hotel rooms after the third minute.

The Demand Segmenter Agent classifies the customer preferences in order to minimize the risk of buying flight tickets and not being able to purchase hotel rooms afterwards. The segmentation of the demand transforms customer preferences that are considered to by very risky to preferences of goods that are easier to buy. These preferences are also saved in the knowledge base of the agency.

The Allocator Agent uses a service of an integer programming model to obtain the optimal allocation of available goods for the customers. The service is set in motion not only to search for the optimal allocation, but as many best allocations as the solver can optimize in 25 seconds. Such model is executed during the game instances to indicate which goods the agents must acquire. The model receives price quotes, predicted prices, goods already bought and preferences of the clients. All this information is obtained through the knowledge base of the agency.

The Allocation Segmenter Agent receives all of the optimal allocations and classifies them. Based on these segmentation results, the Ordering Agent calculates the amounts of required goods, maximum and minimum prices of the bids and importance of goods. Moreover, the Ordering Agent sends purchase orders to the negotiating agents. These orders are received and converted to bids for the auctions.

3 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 [2]. Machine learning is time consuming and it is unfeasible to allow every agent in the organization [2] 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.

3.1 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 [2] implemented as a reactive agent [1]. This methodology has six phases: Agent Selection, Problem Domain Analysis, Machine Learning Design, Implementation, Training, and Testing & Evaluation. In figure 2, we depict all the phases of the methodology.

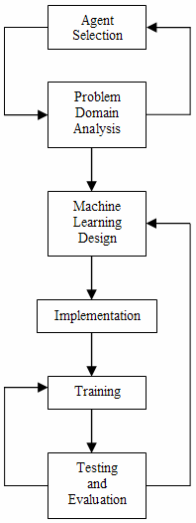

Fig. 2. The methodology for introducing Learning Techniques

3.2 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. The first step of our methodology is to define the *Organization's Learning Problem*. This learning problem is decomposed in two important definitions: (i) the *Organization's Goal*, *OG*, and (ii) the *Organization's Performance Measure*, *OP*, which will measure how well the society is achieving the predefined goal. 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. In our TAC Agency, we define this *Organization's Learning Problem* as: (i) *OG* : Obtain first place in the competition; (ii) *OP* : Average Score.

In the Agent Selection phase, this process chooses the agents that have complex plans to perform and need a learning technique to perform them. In the following phases, these complex plans are associated to a knowledge and a learning technique. Afterwards, for every selected agent, a Problem Domain Analysis is performed to identify important learning issues. The goal of this phase is to establish a well-defined *Agent Learning Problem*. This definition has been extended from [18], and defines three features for each agent: (i) an Agent Goal, *G*; (ii) a Performance Measure, *P*, which will measure how well the agent is performing to achieve its predefined goal; and (iii) a Training Experience, *E*, which defines how the agent will acquire its knowledge.

We notice that two agents have complex plans to perform and need a machine learning technique in order to improve the performance of our TAC Agency. The first agent selected is the Hotel Negotiator Agent and the second one is the Price Predictor Agent. The Hotel Negotiator Agent requires a complex plan, because hotels rooms are probably the most difficult goods to buy in the TAC environment. The hotel auctions have two characteristics that introduce extreme complexity in executing a plan to buy hotel rooms with minimum expenditure: (i) in most hotel auctions, there are not enough hotel rooms for all participants in the game; (ii) hotel auctions start closing in a random order after the forth minute of the instance. Consequently, we use a learning technique in this agent to help the execution of this plan. The *Agent Learning Problem* for the Hotel Negotiator Agent is defined by: (i) *G* : buying hotel rooms with the minimum expenditure; *P* : number of goods purchased over all the requested orders from the Ordering Agent; *E* : information obtained from final states of the game instances. We also define an *Agent Learning Problem* for the Price Predictor Agent: *G* : predict future ask prices for hotel rooms; *P* : error between the predicted and real price; *E* : history of ask prices.

3.3 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 this phase, we are faced with four decisions: (i) The ideal knowledge representation; (ii) An approximate knowledge that will be used by learning process; (iii) The learning algorithm; and (iv) the training set used by the learning algorithm.

The first decision determines exactly what type of knowledge will be learned (also known as ideal knowledge). This knowledge can be modeled as a function *F* that receives a state *S* and determines an action *A*, or $F: S \rightarrow A$. In the TAC Agency, the knowledge representation for the Hotel Negotiation Agent is defined as a function called *NextBid*, and it receives a state *S* of the TAC environment and produces a value *B* of the next bid ($NextBid: S \rightarrow B$). The knowledge for the Price Predictor Agent is also modeled as a function called *NextAskPrice* that accepts the ask price *A* of the last game instances and produces the next ask price *N* ($NextAskPrice: A \rightarrow N$).

It may be very difficult in general to perfectly learn this function for the ideal knowledge, and normally we reduce the complexity and transform the problem to only learn some approximation of this selected function. In the second decision of this phase, we must choose the most reasonable representation of the knowledge that the agent will use to describe the ideal knowledge defined above. This best representation is also called an approximate knowledge, and 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 knowledge. However, an expressive function requires more training data in the training phase.

In the TAC Agency, we use a min-max procedure [19] and reinforcement learning technique [19] to model the approximate knowledge of the Hotel Negotiator Agent. 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 [19]. In Reinforcement learning [19], 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*: the current ask price; (ii) *LastAskPrice*: the ask price of the last minute; (iii) *deltaAskPrice = AskPrice – LastAskPrice*; (iv) *Gama*: a constant; (v) *Bid*: the current bid; and (vi) *LastBid*: the bid sent in the last minute. To calculate the curent bid (MAX player decision) for the Hotel Negotiator Agent, a decision is taken using the min-max procedure on the following alternatives: (i) *Bid = LastBid + 0.5*Gama*deltaAskPrice*; (ii) *Bid = LastBid + 2*Gama*deltaAskPrice*; (iii) *Bid = LastBid + 5*Gama*deltaAskPrice*. The MIN player is modeled as the market response to the bid sent by our agent. The market response has three possible results: (i) *AskPrice = AskPrice + 0.5*Gama*deltaAskPrice*; (ii) *AskPrice = AskPrice + 2*Gama*deltaAskPrice*; (iii) *AskPrice = AskPrice + 5*Gama*deltaAskPrice*. For the evaluation function in the min-max procedure, 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 Price Predictor Agent uses a simple representation to describe the approximate knowledge. 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.

In the third decision of this phase, we have to select a learning algorithm in order to adapt the approximate knowledge. The Hotel Negotiator Agent uses a perceptron rule [19] to adapt the evaluation function of the min-max procedure: $w_i = w_i + \eta.(y(t) - d(t)).x_i$, where w_i is the i-th weight of the perceptron; x_i 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. At the end of each instance, this perceptron rule is computed for all of the traversed intermediate and final states. The Price Predictor Agent uses a Least Mean Squares (LMS) learning algorithm to adapt the value of α in the approximate knowledge. The formula used to adapt α is: $\alpha(n) = \alpha(n-1) + \beta * (AskPrice(n-1) - PredictedAskPrice(n-1))$, where β is a learning rate.

In order to learn the approximate knowledge we will need a training set, the last decision of this phase. These training examples in the set are obtained through a direct or indirect experience. In the direct experience, the designer can carefully select the best training examples that lead to a good approximate knowledge. However, the

indirect experience requires a design that suggests actions taken from the approximate knowledge that will lead to already known states that improve the performance of the system, and unknown states that guide to new experiences. The exploration is important for indirect learning agents that are willing to discover much better actions for the long run. Some learning algorithms, such as Rule Based Systems [18], do not require a training set.

In our agency, the Price Predictor Agent uses a direct experience. We selected the 500 last ask prices of the 2003 competition to be used by the learning algorithm and adapt the approximate knowledge. 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. In the perceptron rule, the value of *d(t)* is not a known value for all traversed intermediate states, and to compute it a TD-Learning [19] strategy is used: $d(t-1)=d(t)+\beta.(reward(t)+ (d(t)-d(t-1)))$, where *d(t)* is the expected output of the decision point at time *t*; β is the Reinforcement learning rate; *reward(t)* is the reward obtained in the decision point at time *t*. Moreover, the value of *d(t)* that represents final states is computed with a *reward(t)*=1 when it accomplishes the goal of purchasing all the goods in the purchase order, or a *reward(t)=0* in all other states. For final states *d(t)* is calculated using the following formula: *d(t) = reward(t)*. The goal of this technique is to adapt the weights of the evaluation function (perceptron) that lead to good final states of the tree. In these states, the agent send bids to the Game Instance that are able to purchase all ordered goods with the minimum expenditure.

3.4 Implementation, Training, Testing and Evaluation

The implementation phase transforms the models above 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.

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. The TAC Agency obtained an average score of 1920 after 32 game instances, and the LA-Clone obtained an average score of 1355.

Another test with the agent with similar features of the LivingAgents was performed against 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. The TAC Agency obtained an average score of 2592 after 32 game instances, and the LA-Clone obtained an average score of 1323.

4 The Learning Design Pattern

4.1 Intent

The Learning pattern supports the process of including the learning property in Software Agents. It defines 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 3.3, and guides the machine learning design to object-oriented components. Consequently, we can achieve an easy mapping to code.

4.2 Context

Cognitive agents [2] 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 separation of these key learning aspects creates components that are easier to design, maintain, and reuse.

4.3 Motivation Example

In the TAC Agency, presented in more detail in section 2 and 3, the Hotel Negotiator Agent needs an efficient strategy to send bids to auctions. The environment is constantly changing because every new game instance can have different participants. The knowledge of the agent has to adapt to unknown situations and lead to a good performance. In order to design this agent, we used a combination of a Temporal Difference Learning [19] and Neural Network [19] strategy.

4.4 Problem

The introduction of learning 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 in a systematic way.

4.5 Solution

The design pattern separate key aspects that are involved in the learning 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 the approximate knowledge 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 knowledge modeled in the Knowledge Representation. The learning algorithm alters this approximate knowledge through a direct or indirect experience. This is modeled as a separate entity called Training Experience.

4.6 Structure

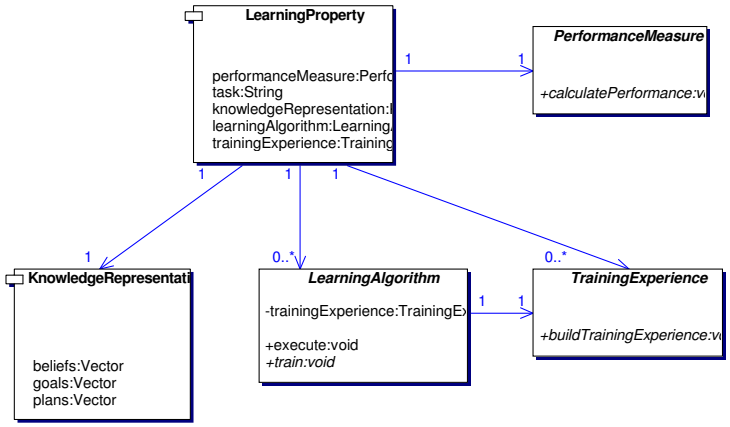

Fig. 3. Learning Design Pattern

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 [20].

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 [20].

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.

In figure 4 we present the class diagram used for the Hotel Negotiator Agent. We used the MAS Framework [21] to implement the TAC Agency, and the classes *HotelNegotiatorAgent* and *HotelNegotiatorAgentIP* are specialized classes that code a software agent. Details on how to instantiate a software agent with the MAS Framework can be found in [21].

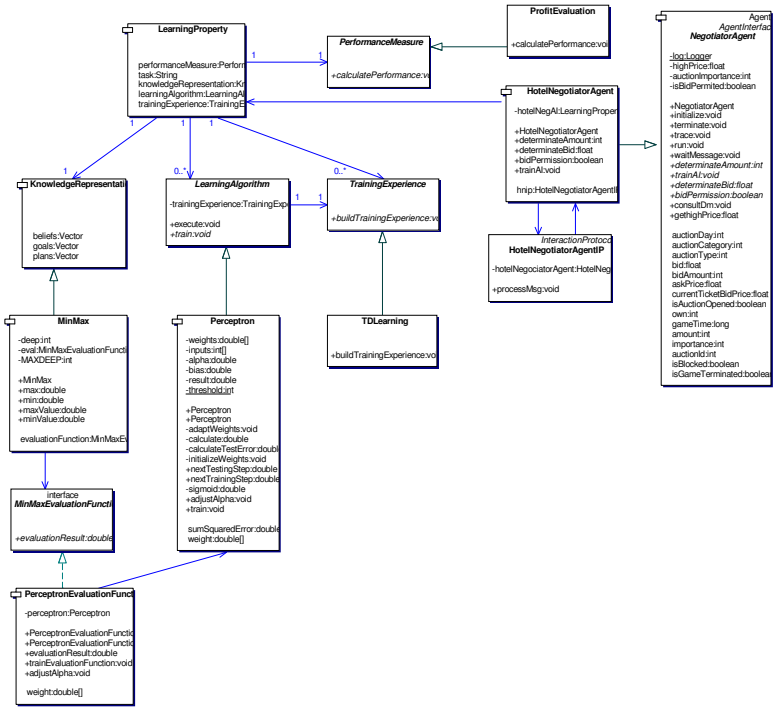

Fig. 4. The Class Diagram of the Hotel Negotiator Agent

The MinMax and PerceptronEvaluationFunction are the classes that implement the Knowledge Representation as described in section 3.3. The Learning Algorithm as explained in section 3.3 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 3.2.

4.7 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.

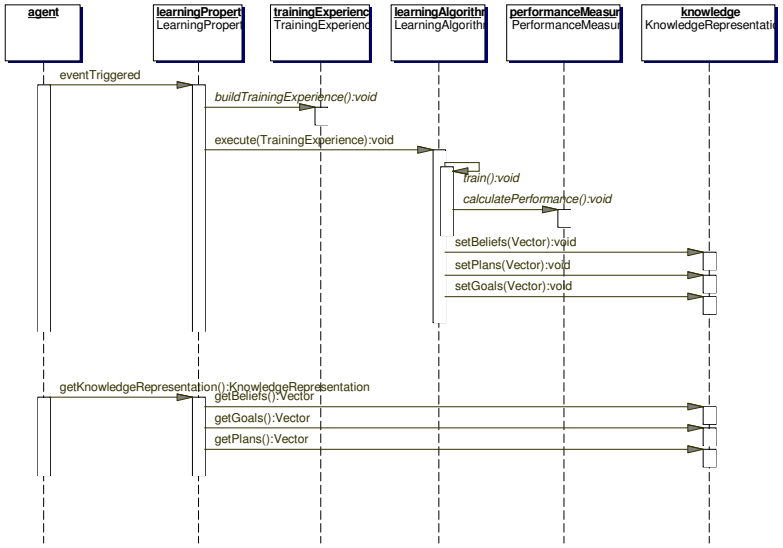

Fig. 5. The Sequence Diagram of the Learning Design Pattern

4.8 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 other agency concerns such as 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.

4.9 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.

5 Final Comments

This paper presents a methodology for building intelligent agents in a multi-agent system. We use machine learning techniques in software agents to perform complex plans, adapt the beliefs and achieve the predefined goals. 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 multi-agent systems since they provide well-known strategies to support the construction of adaptable agents.

Our methodology is unique because it guides the introduction of learning properties in software agents through design and implementation stages. With this methodology we are able to evaluate the goal of the multi-agent system and calculate the performance gain. It also defines the learning goal of each agent in the system and creates an evaluation measure. Moreover, the methodology guides the engineer in the definition of key aspects of the learning property, such as: (i) the knowledge of each individual agent; (ii) the selection of the process to acquire the knowledge. (iii) the association between agent(s) and knowledge.

The design pattern separate key aspects that are involved in the learning design of each individual agent, and it presents an easy mapping of the Problem Domain Analysis and Machine Learning Design phase in the methodology to an object-oriented design. The pattern modularizes a generic learning property that can be reused and refined in different contexts.

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. 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., Cowan, D. Agents in Object-Oriented Software Engineering. Software: Practice & Experience, Elsevier, Volume 34, Issue 5, May 2004, pp. 489 - 521.
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 the International Journal of Software Engineering & Knowledge Engineering.
14. Garcia, A. From Objects to Agents: An Aspect-Oriented Approach. Doctoral Thesis, PUC-Rio, Computer Science Department, Rio de Janeiro, Brazil, April 2004.
15. TAC web site.: http://www.sics.se/tac.
16. 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.

17. 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.
18. Mitchell, T. M.: Machine Learning. McGraw-Hill, 1997. ISBN 0070428077.
19. Russell, S. et al.: Artificial Intelligence. Prentice Hall, 1995. ISBN 0-13-103805-2.
20. Gamma, E. et al.: Design Patterns – Elements of Reusable Object-Oriented Software. Addison Wesley, 1995. ISBN 0201633612.
21. Sardinha, J.A.R.P.; Ribeiro, P.C.; Lucena, C.J.P.; Milidiú, R.L.: An Object-Oriented Framework for Building Software Agents, in Journal of Object Technology, vol. 2, no. 1, January-February 2003, pp. 85-97.