The Agent Learning Pattern

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 in agents 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 achieve their goals. In this paper, we present the Agent Learning Design Pattern that guides the object-oriented design of machine learning algorithms in Software Agents.*

Intent

The intent of the Agent Learning Pattern is to add machine learning algorithms to an object-oriented agent design. The design separates key aspects in machine learning: knowledge representation, algorithm, performance evaluation and training example generator.

Context

Multi Agent Systems [Garcia et al. 2004a] is a new technology that has been recently used in many simulators and intelligent systems to help humans perform several time-consuming tasks. 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 Multi Agent Systems, since they provide well-known strategies to support the construction of adaptable agents.

In the design of machine learning [Mitchell 1997] in software agents, we are normally faced with some key aspects, such as: (i) the knowledge representation of the agent; (ii) the learning algorithm; and (iii) a training example generator used by the learning algorithm; (iv) the performance monitor.

Problem

The design of machine learning in agent object-oriented architectures is not straightforward when reuse and maintenance is required. How do we design machine learning with several algorithms? How do we design different example generators for the same algorithm? How do we design a monitor to evaluate the performance of the agent when machine learning is used?

Forces

- The design should be general enough to encompass all machine learning strategies.
- It should be easy to map the learning machine components to abstractions of object-oriented programming languages.

- The design should support improved reuse and maintenance of the learning algorithm and training example generator.
- Specific learning techniques might need more classes to implement the learning algorithm and training example generator.
- Software agents must be able to learn automatically in complex and unpredictable environments.

Solution

Software agents are generally implemented in object oriented frameworks [Jade 2003] [Sardinha et al. 2003a] using inheritance. In general, a concrete class *Agent* extends a superclass of the OO framework as depicted in figure 1. This concrete class implements basic services of the software agent, such as: sensory of the environment, event handling, message handling, etc. The class called *KnowledgeRepresentation* implements the data structure of the agent's knowledge (examples: linear weighted function, a collection of rules, a neural network, or a quadratic polynomial function).

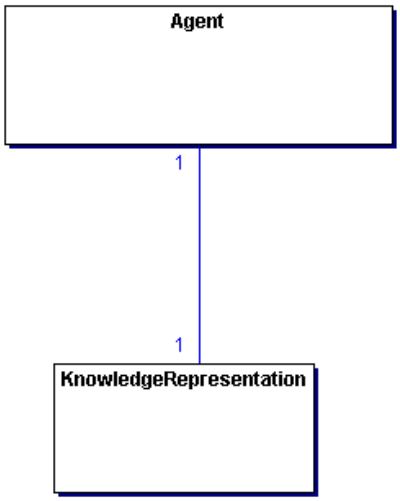

Figure 1. A typical object oriented agent design

Figure 2 is used to include a learning algorithm, a performance monitor, and a training example generator. The monitor of the agent's performance is coded in a separate class called *PerformanceMeasure*, and is used in the learning property to guarantee the agent is achieving the predefined goal. It also has code that implements standard rules of performance for the learning algorithm. The learning algorithm is a separate class called *LearningAlgorithm*, and this class is responsible for modifying the already designed *KnowledgeRepresentation* class. The example generator is modeled as a separate class called *Training Experience*.

Structure

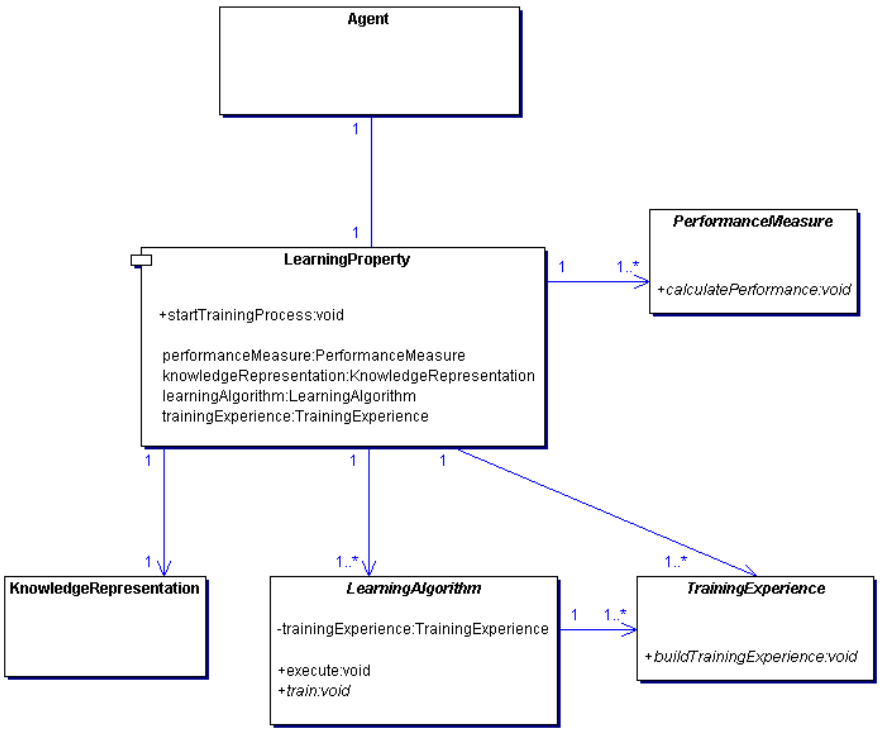

Figure 2. The Learning Design Pattern

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

Main Participants:

Learning Property

Defines the class of the agent's learning property, and it is responsible for accessing all other components

Performance Measure

An algorithm that implements rules for performance measurement.

Learning Algorithm

The machine learning algorithm.

Training Experience

An algorithm that generates training examples for the learning algorithm.

Client Participants:

Agent

Defines the class that implements the basic services of a software agent, for example: thread control; communication and interaction protocols; internal actions.

Knowledge Representation

The representation of the knowledge.

Example

The Trading Agent Competition (TAC) is designed to promote and encourage high quality research into the trading agent problem. There are two scenarios in the annual competition: TAC Classic [Wellman et al. 2001] - a "travel agent" scenario based on

complex procurement on multiple simultaneous auctions; and, TAC SCM [Arunachalam et al. 2004] - a PC manufacturer scenario based on sourcing of components, manufacturing of PC's and sales to customers.

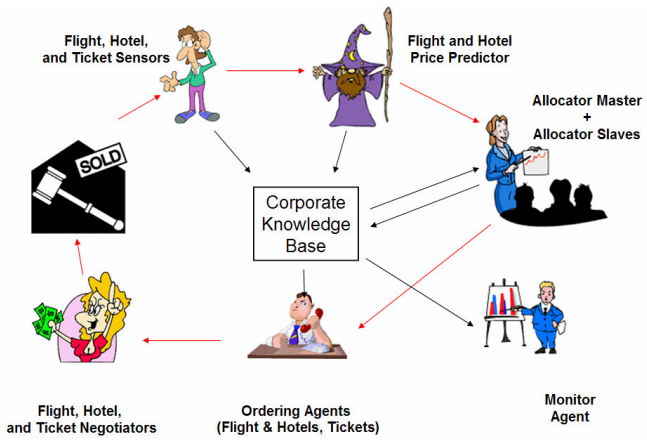

Figure 3. The *LearnAgents* architecture

We present in figure 3 the architecture of the *LearnAgents* [Sardinha et al. 2004], a multi agent system with modular entities that are asynchronous, distributed, reusable, and easy to interoperate. We define agent roles that tackle sub problems of trading, such as price prediction, bid planning, goods allocation, bidding, among others. The system's goal is to acquire travel packages for clients with as much profit as possible. This profit is defined as the sum of the utilities of the eight clients in the TAC game, minus the costs of acquiring the travel goods in the auctions.

We introduced machine learning techniques in the Flight and Hotel Price Predictor Agent, the Allocator Master and Slave Agents, and the Hotel Negotiator Agent. The addition of machine learning in software agents permit the creation of a highly adaptable system that is extremely easy to evolve when performance improvement is needed.

Dynamics

Figure 4 presents the basic pattern dynamics of machine learning in agents. Several events can trigger the agent learning [Mitchell 1997], including the execution of internal agent actions, throwing of exceptions, messages exchanged between agents, and events sensed in the external environment. The concrete class *Agent* must access the *LearningProperty*, which is the main interface to the learning pattern.

The main goal of the Flight and Hotel Price Predctor Agent is to predict auction prices based on past price sequences. The method *buildTrainingExperience* implements code that generates training examples (price sequences) for the machine learning algorithm. The machine learning algorithm is coded in the method *train*, but is called through the method *execute*. This method also calls the method *calculatePerformance* that will calculate the prediction error. If a small error is achieved then the method *adaptKnowledge* is responsible for changing knowledge attributes with the new prediction of auction prices.

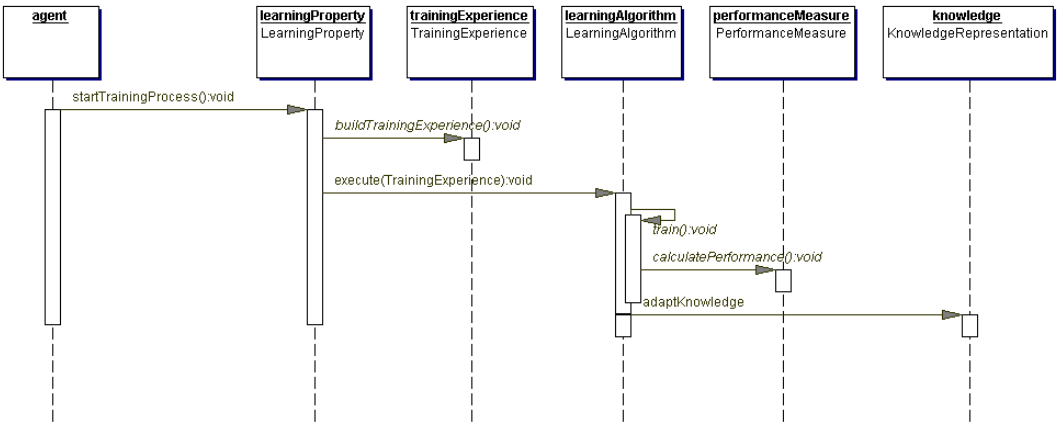

Figure 4. The Sequence Diagram of the Learning Design Pattern

Implementation

Figure 5 presents the class diagram used for the Flight and Hotel Price Predctor Agent. We used the MAS Framework [Sardinha et al. 2003a] and the classes *PricePredictorAgent* and *PricePredictorAgentIP* are specialized classes that code a software agent's basic services.

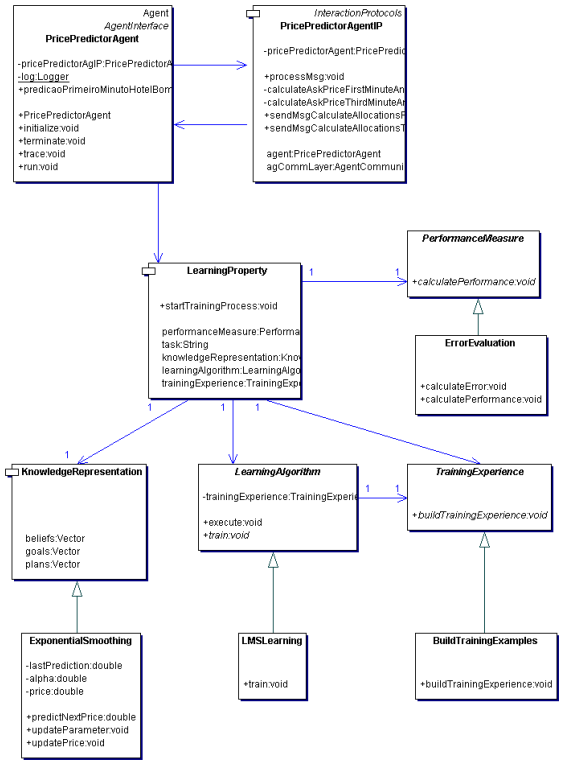

Figure 5. The Class Diagram of the Price Predictor Agent

ExponentialSmoothing is the class that implements the *KnowledgeRepresentation*. The learning algorithm is coded in the class *LMSLearning*. To generate the training examples for the LMS algorithm [Mitchell 1997], we implemented the class *BuildTrainingExamples* that executes a query in a database with auctions prices fo games already played. The *ErrorEvaluation* class has the performance measure.

We describe below pieces of code that are relevant to understand how to implement the learning pattern using the Java language [Sun 2004]. The *LearningProperty* class is the main interface to all components. The method *startTrainingProcess* (line 9) implements the process of collecting data (line 10) for the machine learning algorithm (line 11). This class also keeps a reference to all components.

```
1. public class LearningProperty {
2.      private String task;
3.      private TrainingExperience trainingExperience;
4.      private LearningAlgorithm learningAlgorithm;
5.      private KnowledgeRepresentation knowledgeRepresentation;
6.      private PerformanceMeasure performanceMeasure;
7.      // Getters and Setters
8.      …
9.      public void startTrainingProcess(){
10.             trainingExperience.buildTrainingExperience();
11.             learningAlgorithm.execute(trainingExperience);
12.     }
13. }
```

The knowledge is modeled in the class *ExponentialSmoothing* and is a specialization of the class *KnowledgeRepresentation*. The method *predictNextPrice* (line 5) implements the algorithm called Exponential Smoothing [Bowerman et al. 2004] to predict the next price:

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

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

```
1. public class ExponentialSmoothing extends KnowledgeRepresentation {
2.      private double lastPrediction;
3.      private double alpha;
4.      private double price;
5.      public double predictNextPrice(){
6.        double nextPrice = price*alpha + (1-alpha)*lastPrediction;
7.        lastPrediction = nextPrice;
8.        return(nextPrice);
9.      }
10.     public void updateParameter(double alpha){
11.       this.alpha=alpha;
12.     }
13.     public void updatePrice(double price){
14.       this.price=price;
16.     }
17. }
```

The class *BuildTrainingExamples* is a specialization of *TrainingExperience*, and implements the process of building the training examples used by the learning algorithm. The method *buildTrainingExperience* (line 3) queries a database with the 2003 competition auction prices, and store these price sequences in to the *askPriceHstory* (line 2) vector.

```
1. public class BuildTrainingExamples extends TrainingExperience {
2.      public Vector askPriceHistory;
3.      public void buildTrainingExperience() {
4.        // JDBC code
5.        …
```

```
6.      }
7. }
```

The Price Predictor Agent uses a Least Mean Squares (LMS) learning algorithm to adapt the value of alpha in the knowledge. The class *LMSLearningAlgorithm* class is a specialization of the class *LearningAlgorithm*, and has a method called train (line 11) that implements the LMS algorithm.

```
1. public class LMSLearning extends LearningAlgorithm {
2.      private BuildTrainingExamples bte;
3.      private ExponentialSmoothing es;
4.      private double alpha;
5.      private double lastAlpha;
6.      private double beta;
7.      private double predictedPrice;
8.      private double lastAskPrice;
9.      es.updateParameter(20);
10.     alpha=20;
11.     public void train() {
12.       for(int i=0;i<bte.askPriceHistory.size();i++){
13.         lastAskPrice=
14.         ((Double)bte. askPriceHistory.elementAt(i)).doubleValue();
15.         es.updatePrice(lastAskPrice);
16.         predictedPrice = es.predictNextPrice();
17.         lastAlpha=alpha;
18.         alpha=lastAplha+(beta*(lastAskPrice-predictedPrice));
19.     }
20. }
```

This class *ErrorEvaluation* is a specialization of the class *PerformanceMeasure*, and is used by the class *LeaningProperty* to calculate the error between the predicted price and actual ask price. The method *calculateError* (line 5) implements this calculation.

```
1. public class ErrorEvaluation extends PerformanceMeasure {
2.      private double predictedPrice;
3.      private double askPrice;
4.      private double error;
5.      public void calculateError() {
6.              error = Math.abs(predictedPrice-askPrice)/askPrice;
7.      }
8.      …
9. }
```

Consequences

Uniformity and Generality. The Learning pattern provides a uniform solution that is general enough to support all the machine learning techniques.

Reusability. The pattern modularizes a generic design for the learning property, which can be reused and refined to different contexts and applications.

Improved Separation of Concerns. The learning property is entirely separated from other agency concerns such as interaction and autonomy.

Straightforward implementation. The pattern presents an easy mapping of the *Agent's Goal Learning Problem* and the *Agent Learning Model* to an object-oriented design and implementation.

Known Uses

A design for introducing machine learning algorithms is presented in chapter 1 (section 1.2.5) of the book Machine Learning [Mitchell 1997]. Four modules are presented in the design: (i) the Performance System – module to solve the given performance task. It takes an instance of a new problem as input an produces a trace of its solution; (ii) the Critic – takes as input the history or trace of the solution and produces as output a set of training examples; (iii) the Generalizer – takes as input the training examples and

produces an output hypothesis; (iv) the Experiment Generator – takes as input the current hypothesis and outputs a new problem for the Performance System to explore. In our design pattern, the critic module can be coded in the class Training Experience, and the Generalizer is coded in the class LearningAlgorithm. The module Experiment Generator has to be included in the class Agent, and is the module that starts the training process. The only module that does not have a direct mapping is the Performance System. Actually, the code of this module is coded in the class PerformanceMeasure and another part is coded in the class TrainingExperience.

The Agent Learning Design Pattern has been used in four implementations: (a) a multi agent system [Milidiú et al. 2001] [Sardinha 2001] that uses evolutionary techniques to build offerings in a retail market (b) an agent system [Sardinha et al. 2003b] that learns to play Tic-Tac-Toe with no prior knowledge; (c) a multi agent system for the Trading Agent Competition (TAC) [Sardinha et al. 2004], and (d) a multi agent system for managing the paper submission and selection process in workshops and conferences [Garcia 2004].

Related Patterns

Learning Aspect. [Garcia et al 2004b] The Learning Property entity can be implemented as an aspect [Kiczales 1997] and improve the separation of concerns [Garcia et al. 2004a] [Garcia 2004] [Garcia et al. 2004c]. A LearningProperty aspect can be used to replace the LearningProperty class, and 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 the aspect-oriented variant of the Learning pattern: (i) *transparency* – the use of aspects turns out to be an elegant and powerful approach to introduce the learning behavior into agent classes in a transparent way [Garcia et al 2004b] [Milidiú et al 2001]; the description of which agent classes need to be affected is present in the aspect and the monitored classes are not modified**;** (ii) *ease of evolution* **-** as the MAS evolves, new agent classes may have to be monitored and trigger the learning process; MAS developers only need to add new pointcuts in the Learning Property aspect in order to implement the new required functionality.

References

Garcia, A., Lucena, C.J.P., and Cowan, D. (2004a) "Agents in Object-Oriented Software Engineering", In: Software: Practice & Experience, Elsevier, Volume 34, Issue 5, May 2004, pp. 489 - 521.

Mitchell, T. M. (1997), Machine Learning, McGraw-Hill. ISBN 0070428077.

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

Sardinha, J.A.R.P., Ribeiro, P.C., Lucena, C.J.P., and Milidiú, R.L. (2003a), "An Object-Oriented Framework for Building Software Agents", In: Journal of Object Technology, vol. 2, no. 1, January-February 2003, pp. 85-97.

Wellman, M.P., Wurman, P.R., O'Malley, K., Bangera, R., Lin, S., Reeves, D., Walsh, W.E. (2001), "Designing the Market Game for a Trading Agent Competition", In: IEEE Internet Computing, pp. 43-51, March/April 2001 (Vol. 5, No. 2).

Arunachalam, R., Sadeh, N. (2004), "The 2003 Supply Chain Management Trading Agent Competition", In: Trading Agent Design and Analysis workshop proceedings

at The Third International Joint Conference on Autonomous Agents & Multi Agent Systems. July 2004, New York, USA.

Sardinha, J.A.R.P., Milidiú, R.L, Lucena, C.J.P., Paranhos, P.M., Cunha, P.M. (2004), "An Agent Based Architecture for Highly Competitive Electronic Markets", Submitted to FLAIRS-2005.

Sun Microsystems (2004), Java Programming Language, http://java.sun.com.

Bowerman, B.L., O'Connell, R., Koehler, A. (2004), Forecasting, Time Series, and Regression. Duxbury Press, 4th edition, ISBN: 0534409776.

Milidiú, R.L., Lucena, C.J., Sardinha, J.A.R.P. (2001), "An object-oriented framework for creating offerings", In: Proceeding of the 2001 International Conference on Internet Computing (IC'2001), June 2001.

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

Sardinha, J. A.R.P., Milidiú, R. L., Lucena, C. J. P., Paranhos, P. M. (2003b), "An OO Framework for building Intelligence and Learning properties in Software Agents", In: Proceedings of the 2nd International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS 2003) at ICSE 2003, Portland, USA, May 2003.

Garcia, A. (2004), From Objects to Agents: An Aspect-Oriented Approach, Doctoral Thesis, PUC-Rio, Computer Science Department, Rio de Janeiro, Brazil, April 2004.

Garcia, A., Kulesza, U., Sardinha, J.A.R.P., Milidiu, R.L., Lucena, C.J.P. (2004b), "The Learning Aspect Pattern", In: Proceedings of the 11th Conference on Pattern Languages of Programs (PLoP2004), September 2004, Allterton Park, Monticello, Illinios.

Kiczales, G. et al (1997), "Aspect-Oriented Programming", In: Proceedings of the European Conference on Object-Oriented Programming - ECOOP'97, LNCS (1241), Springer-Verlag, Finland., June 1997.

Garcia, A., Sant'anna, C., Chavez, C., Silva, V., Lucena, C.J.P., Staa, A. (2004c), "Separation of Concerns in Multi-Agent Systems: An Empirical Study", In: C. Lucena et al (Eds)., Software Engineering for Multi-Agent Systems II, Springer, LNCS 2940, March 2004, pp. 49-72.