

The Learning Aspect Pattern

Alessandro Garcia Uirá Kulesza José Sardinha Carlos Lucena Ruy Milidiú

*Software Engineering Laboratory – SoC+Agents Group
Pontifical Catholic University of Rio de Janeiro - PUC-Rio - Brazil
{afgarcia, uira, sardinha, lucena, milidiu}@inf.puc-rio.br*

Abstract

An intelligent agent has the ability to learn and adapt itself as a result of several events, including its own actions, its mistakes, its successive interactions with the external world and collaborations with other agents. As the agents' complexity increases, object-oriented abstractions cannot modularize the learning concern, which tends to spread across several system classes. This paper presents the Learning Aspect pattern, which documents an aspect-oriented solution for the separation of the learning concern, which in turn improves the system reusability and maintainability.

Intent

The Learning Aspect pattern modularizes the learning concern, totally decoupling the basic agent structure from the learning protocol.

Context

Engineers of intelligent agents [2, 3, 4] must deal with the agents' basic functionality, the agent services that are made available to the clients, and a number of additional concerns, such as learning, which greatly increase the system complexity. Agents need to learn based on internal and external events, including their own actions, their mistakes, the successive interactions with the external world and the collaborations with other agents [1, 2, 3]. The agent design is typically based on learning machine techniques [2, 3].

In this context, many facets of the learning concern [2, 3, 4] need to be considered, including the definition of events that trigger the agent learning, the information gathering to enable the learning process, the specification of the learning knowledge, the implementation of the learning algorithms to process the gathered information and refine the agent knowledge, and the adaptation of the current agent knowledge. In this context, the separation of the learning concern is crucial to make the agent components easier to maintain and reuse.

Example

This section introduces an open multi-agent system that supports the management of paper submissions for conferences as well as the reviewing process. It is from herein referred to as Expert Committee (EC). The EC system encompasses *user agents* that are software assistants to represent system users in reviewing processes. The basic functionality of the user agents is to infer and keep information about the corresponding users related to their research interests and their participation in scientific events.

In addition to their basic functionality, user agents can collaborate with each other; the collaboration concern comprises the *roles* [26, 27] played by the agents. Each role represents collaborative activities in specific contexts. Each EC agent plays different roles, but the main ones are chair and reviewer. User agents play these roles in order to cooperate with each other. Classes are used to represent the basic functionalities of the agent types and the different roles.

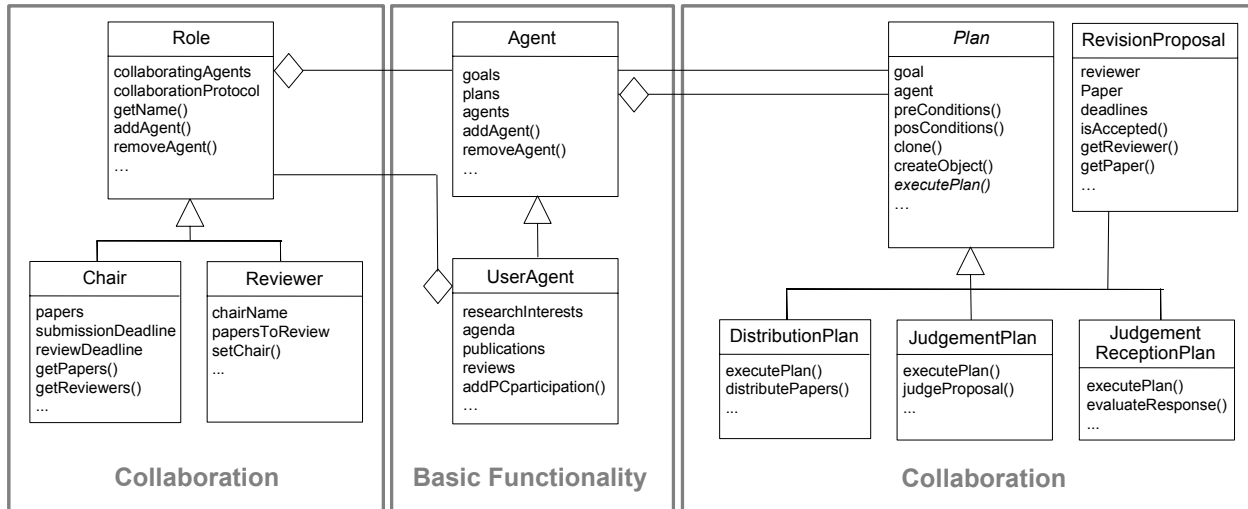


Figure 1. Object-Oriented Design for the Agent Types and Roles (Without Learning)

Agent types and roles are associated with *plans*, which are also represented by separate classes. Each role has a set of plans which are used to implement more sophisticated collaborative activities. The chair role has plans for distributing review proposals; the reviewer role has plans for judging the chair proposals. The chair and reviewers negotiate with each other for performing reviews. There are other plans to address user workloads and invitations to new reviewers. Figure 1 shows classes representing the basic functionalities of the agent types and some examples of roles and plans in the EC system; it does not address the learning concern.

EC agents also incorporate the learning property, using two widely-applied learning techniques to learn the user preferences: Temporal Difference Learning (TD-Learning) [2] and Least Mean Squares (LMS) [2]. The reviewer role uses TD-Learning in order to learn the user preferences in the subjects he/she likes to review. The chair role uses LMS to learn the reviewer preferences. In order to gather information to the learning purpose, user agents supervise the executions of their own actions (methods), the feedback from the system users, the interactions with multiple environment components and the inter-agent collaborations. Figure 2 presents the learning-related components in addition to the design introduced in Figure 1.

The combination of the Observer pattern [5] with the Strategy pattern [5] is a flexible approach to the object-oriented design of the learning concern [6, 7, 8]. The Observer pattern implements the mechanism for event monitoring and information gathering, while the Strategy pattern makes it flexible with respect to the learning strategies. Consider a concrete example of this approach in the context of the EC system, as shown in Figure 2. In such a system, the goal of the Observer pattern is to notify the learning components of relevant events that trigger the learning process. Operations on Plan, Agent, Role classes are monitored to provide the learning component with contextual information and start the learning process. The Observable component is an interface and not an abstract class because the observable classes already extend an abstract class (JADEAgent). Several object-oriented programming languages do not support multiple inheritance. Java, the most used language for implementing software agents [4, 15, 18], does not support this feature. The Agent and Role classes do not directly implement the Observable interface because some agents and roles have not the learning property. The LearningComponent class implements the Strategy pattern and represents a family of different algorithms that implement the learning techniques. The TD-Learning and LMS subclasses implement the specific learning algorithms.

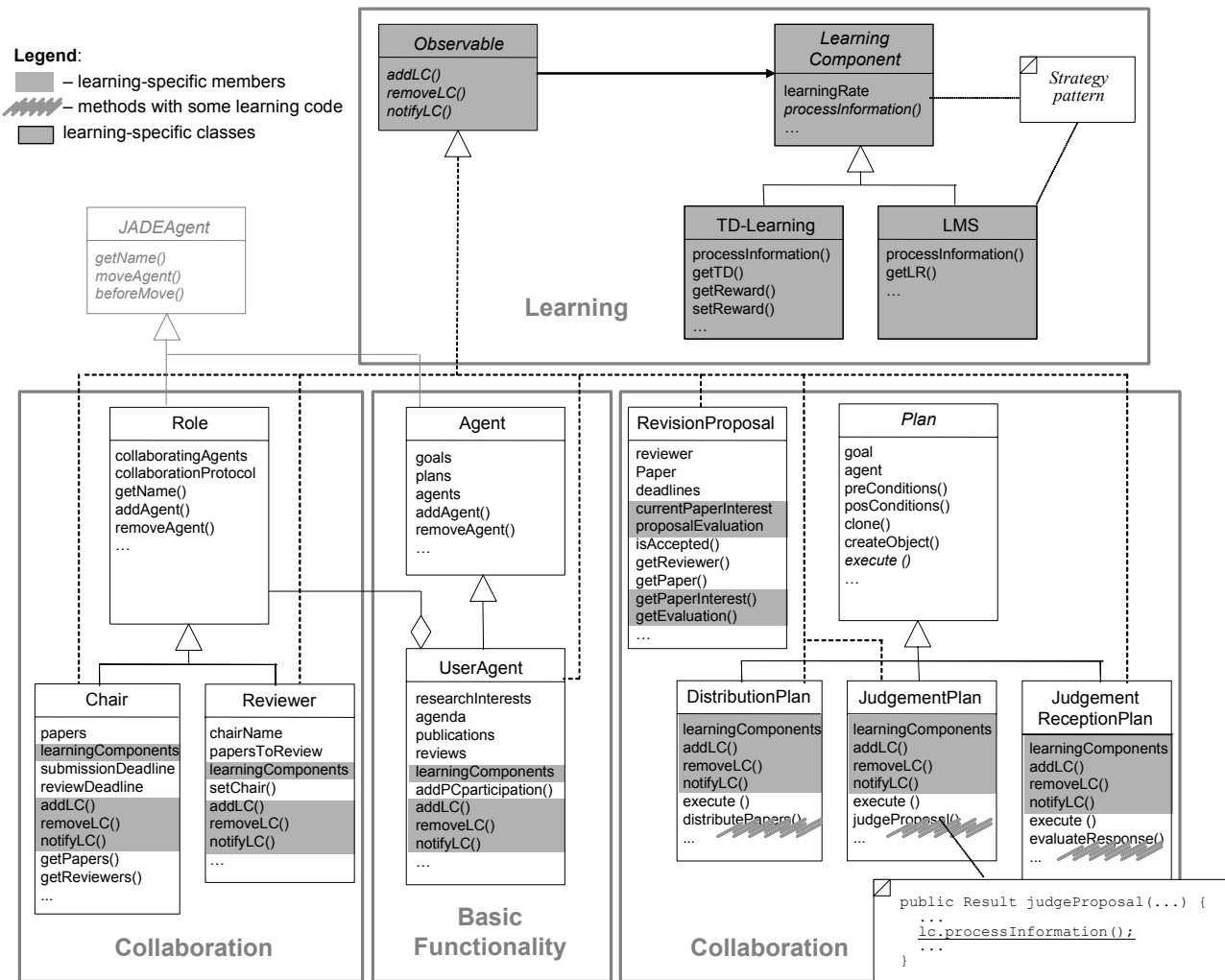


Figure 2. Learning: the Observer Pattern with the Strategy Pattern.

However, the object-oriented design of the learning concern has a huge impact on the agent structure. As shown in the figure, although part of the learning concern is localized in the classes of the Strategy pattern, learning-specific code replicates and spreads across several class hierarchies of a software agent. Several participants (e.g. Chair, Reviewer, UserAgent, and Plan subclasses) have to implement the observation mechanism and consequently have learning code in them. Some classes (e.g. the RevisionProposal class) have learning-specific knowledge. Adding or removing the learning code from classes requires invasive changes in those classes.

Note that even if we try to refactor the object-oriented solution presented in Figure 2, we cannot find a more modular solution. One alternative solution is to try to move the learning-specific methods and attributes from the agent classes to a new class. However, the following problems still remain: (i) the agent classes need to keep an attribute with a reference to this new learning-related class, and (ii) the code relative to information gathering remains scattered over the methods on other agent classes (for example, the method `judgeProposal()` in Figure 2). As a result, the learning concern still crosscuts multiple class hierarchies representing other agent concerns, such as collaboration and the agent's basic functionality. This problem happens because learning is a crosscutting concern independently of the object-oriented decomposition used.

Problem

Object-oriented abstractions do not support the separation of the learning concern and other agent concerns. The design and implementation of the learning concern tend to affect or crosscut many agent classes and methods. This makes it hard to distinguish between the learning protocol and other agent concerns involved [9, 11, 20, 21]. Adding, removing or modifying the learning concern to/from a system is often an invasive, difficult to reverse change. How do we separate the learning concern from the other agency concerns? The following forces emerge from this problem:

- *Reusability*. The basic learning protocol should be easy to reuse to different agent types and agent roles.
- *Readability and Maintainability*. Agent classes, which modularize the agent's basic functionality, should not be polluted with learning-specific knowledge. Moreover agent classes should not be mixed with invocations of learning-specific methods in order to improve the system readability and maintainability.
- *Ease of Evolution*. The design of the learning concern should be easy to evolve as new learning-related requirements need to be satisfied. Changes on the definition of observed events and on the learning strategies should not affect the basic agent functionality.
- *Code Replication*. The design solution should minimize code replication across different classes and methods of the multi-agent system.
- *Flexibility*. The design should be flexible enough to support the association of different learning strategies with distinct agent types and role classes.
- *Transparency*. The design solution should support the learning behavior into existing systems in a way that is transparent to the rest of the system.
- *Generality*. The solution should be general enough to support the modularization of the learning concern independent of the used machine learning techniques.

Solution

Use aspects to improve the separation of the learning concern. Learning aspects are used to modularize the entire learning concern, including the learning-specific knowledge and the information gathering. The Learning aspect separates the learning protocol from agent classes, such as agent types, plans, and roles. By using Learning aspects, we define when and how the agent learns. They specify how to extract information from diverse agent components which are necessary to enable the agent learning. The Learning aspects connect the execution points (events) on different agent classes with the corresponding learning components, making it transparent to the agent's basic functionality the particularities of the learning algorithms in use. These aspects are able to crosscut some agent execution points in order to change their normal execution and invoke the learning components. The execution points include the change of a knowledge element, execution of actions on plans, roles, and agent types, or still some throw exception. Auxiliary classes are used to implement different learning techniques.

Structure

Figure 3 illustrates the structure of the Learning Aspect pattern. The design notation is based on the ASideML modeling language [16, 17], which is used throughout this paper. This language extends UML with notations for representing aspects. The notations provide a detailed description of the aspect elements. In this modeling language, an aspect is represented by a diamond; it is composed of internal structure and crosscutting interfaces. The internal structure declares the internal attributes and methods. A crosscutting interface specifies when and how the aspect affects one or more classes [16, 17]. Each crosscutting interface is presented using the rectangle symbol with compartments

(Figure 3). A crosscutting interface is composed of inter-type declarations, pointcuts (set of join points) and advices. The first compartment of a crosscutting interface represents inter-type declarations, and the second compartment represents pointcuts and their attached advices. The notation uses a dashed arrow to represent the crosscutting relationship, which relates one aspect to classes and/or aspects.

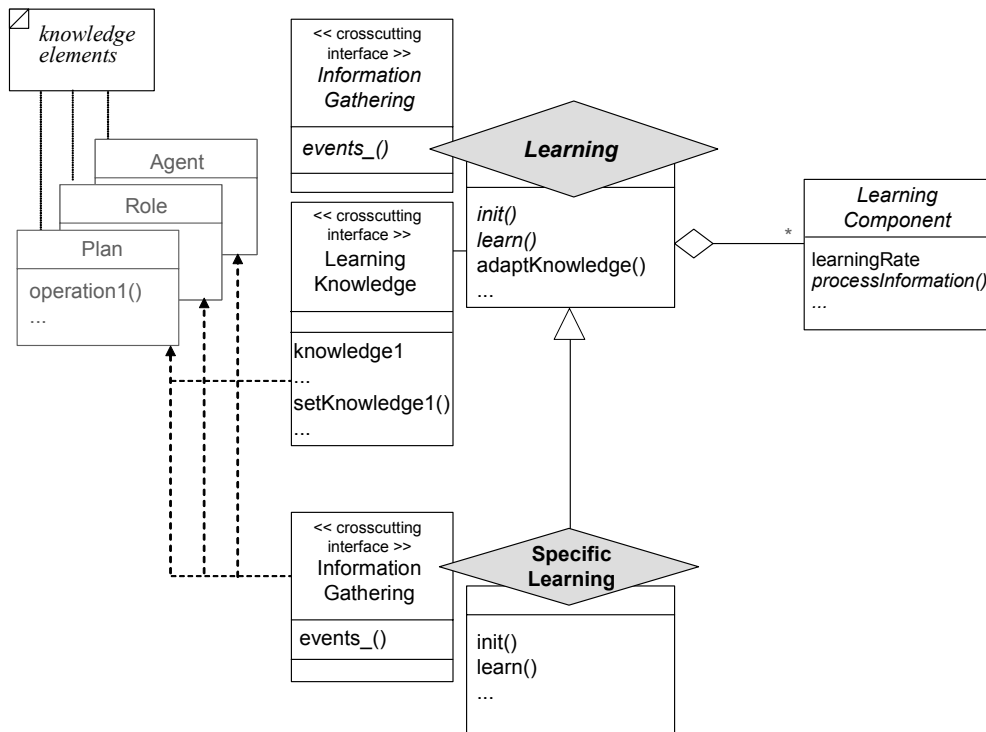


Figure 3. The Static View of the Learning Pattern.

The Learning Aspect pattern has four participants:

- **Learning Aspect**
 - defines the general learning protocol.
- **Specific Learning Subaspect**
 - implements the part of the learning concern that is specific to an agent type or role.
- **Learning Component**
 - implements a specific learning technique.
- **Knowledge Element**
 - provides relevant events and contextual information for learning purposes – this element can be a plan class, an agent class, a role class, or other classes that are part of the agent. They do not have any learning-specific code.

In the structure of the Learning pattern (Figure 3), some parts are common to all instantiations of the pattern, and other parts are specific to each instantiation. The common parts are:

1. The general learning protocol (Learning Aspect):
 - a. learning components are initialized,
 - b. events are sensed,

- c. contextual information is gathered,
 - d. learning components are called, and
 - e. the agent knowledge is adapted.
2. The list of Learning Components in the Learning Aspect, i.e. the references to components that implement more sophisticated learning strategies.
 3. The learning-specific knowledge.
 4. The general structure of the Learning Components.

The specific parts are:

5. The definition of the specific events associated with an agent type or role.
6. The specific information gathering.
7. The initialization of specific learning components used.
8. The adaptation of the agent knowledge.
9. The implementation of the specific Learning Components.

The purpose of the `Learning` aspect is to make the agents able to learn. The `Learning` aspect extends the agent classes to introduce the learning protocol to them. The `Learning` aspect has three main parts: the aspect itself and two crosscutting interfaces. The aspect holds the list of specialized learning components, and the methods to update the agent knowledge since new conclusions are obtained from the learning components. The crosscutting interfaces define how the `Learning` aspect crosscut different classes of the software agents.

The `InformationGathering` interface defines the join points that describe the relevant events and the information which must be gathered from the agent/role classes in order to enable the learning process. This interface contains the advices which invoke either methods responsible for implementing a learning behavior or a specific learning component. The advices usually run after executions of methods on agent classes, role classes and plan classes, and other classes eventually associated with the agent. The `LearningKnowledge` interface introduces different learning-specific attributes and methods into different agent/role classes based on inter-type declarations.

Note that all the learning code is removed from the agent classes and is separately implemented in associated learning aspects, as explained above. The learning code consists of learning aspects and auxiliary classes devoted to implement specific learning strategies. When the learning aspects are woven with the system code, they essentially affect several agent classes; the weaving process is required to compose the learning concern with the other agent concerns, such as the agent's basic functionality and roles.

Dynamics

Figure 4 presents the basic pattern dynamics: (i) the Learning Aspect detects that a relevant operation (join point) on an agent/role class was performed, (ii) the Learning Aspect intercepts this operation, (iii) the Learning Aspect gathers event-related information through the advice parameters, (iv) the Learning Aspect optionally updates some learning-specific knowledge, (v) the Learning Aspect selects and calls the corresponding Learning Components, providing them with the event-related information, (vi) the Learning Components process the new information, (vii) if they get a conclusion, the Learning Aspect updates the attributes of the agent/role classes.

Several events can trigger the agent learning [1, 2, 3, 4], including the execution of internal agent actions, throwing of exceptions, messages exchanged between agents, and events sensed in the external environment. The pattern dynamics is illustrated in the next section in terms of the example.

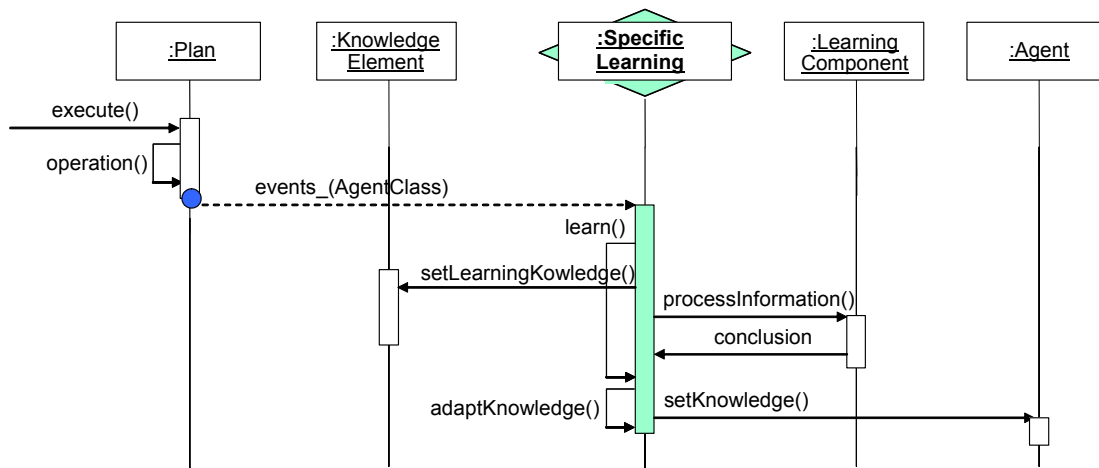


Figure 4. Dynamic View of the Learning Aspect Pattern

Solved Example

Figure 5 illustrates the pattern instantiation for the EC system. The Learning aspect and its subspects crosscut about 12 different classes in this system. However, the figure only presents a partial set of the classes affected by the learning aspects; it shows the Reviewer class, the RevisionProposal class, the UserAgent class, and the JudgementPlan class. The Learning aspect has two subspects: ChairLearning and ReviewerLearning; Figure 5 illustrates only the ReviewerLearning subspect.

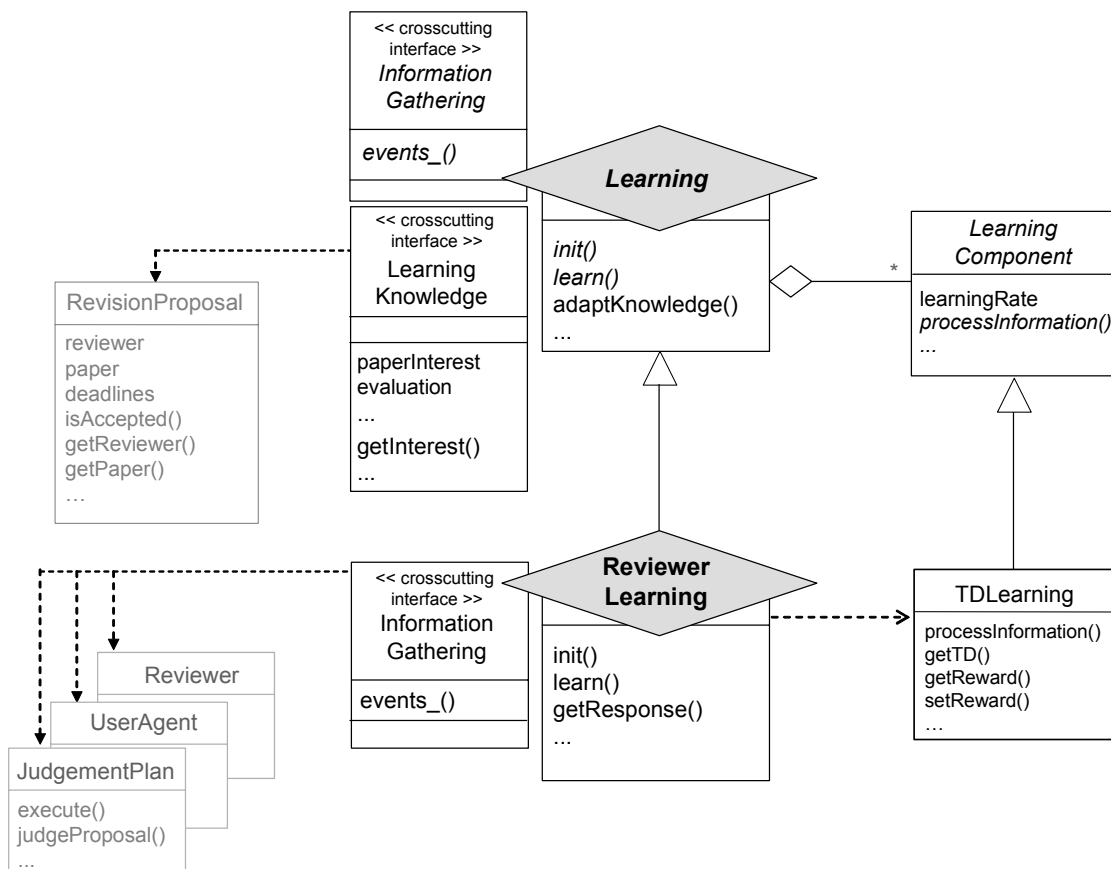


Figure 5. The Learning Pattern for the Reviewer Role.

The `ReviewerLearning` aspect affects the action of judging a proposal in order to learn the user preferences. The execution of the `judgeProposal()` method on the `JudgementPlan` class is an important event for the learning purpose; once the judgment is concluded, the judgement-related information is used by the learning aspect in order to learn about the user preferences. The `ReviewerLearning` aspect catches the information associated with the proposal judgement and the associated learning component is invoked (the `TDLearning` class in this case). The `ReviewerLearning` aspect also intercepts methods on the `Reviewer` class, and on the `UserAgent` class. Figure 5 also illustrates how the `LearningKnowledge` interface of the Learning aspect modifies the structure of the `RevisionProposal` class. This interface introduces the attributes `paperInterest` and `evaluation` and the associated “setters” and “getters” so that the chair role can learn based on the reviewer evaluation.

Figure 6, presents the pattern behavior when the `ReviewerLearning` aspect detects that an important action on an agent plan was performed and learning is required:

- The judgement plan is executed.
- Judgement actions are performed by calling the method `judgeProposal()`.
- The `ReviewerLearning` aspect detects the judgement result by intercepting the end of the method execution.
- This aspect gathers the information needed from the plan context, i.e. the `RevisionProposal` object.
- The aspect updates the `RevisionProposal` object so that the chair can learn based on the reviewer judgement – it updates this object state by invoking the methods `setPaperInterest()` and `setEvaluation()`, both of them introduced by the Learning aspect.
- The `ReviewerLearning` aspect selects and calls the corresponding learning components, the `TDLearning` class in this case, and provides them with the contextual information.
- The aspect executes its specific algorithms and alternatively gets a conclusion which leads to the adaptation of the agent knowledge, in this example the update of the user’s research interests in the `UserAgent` class.

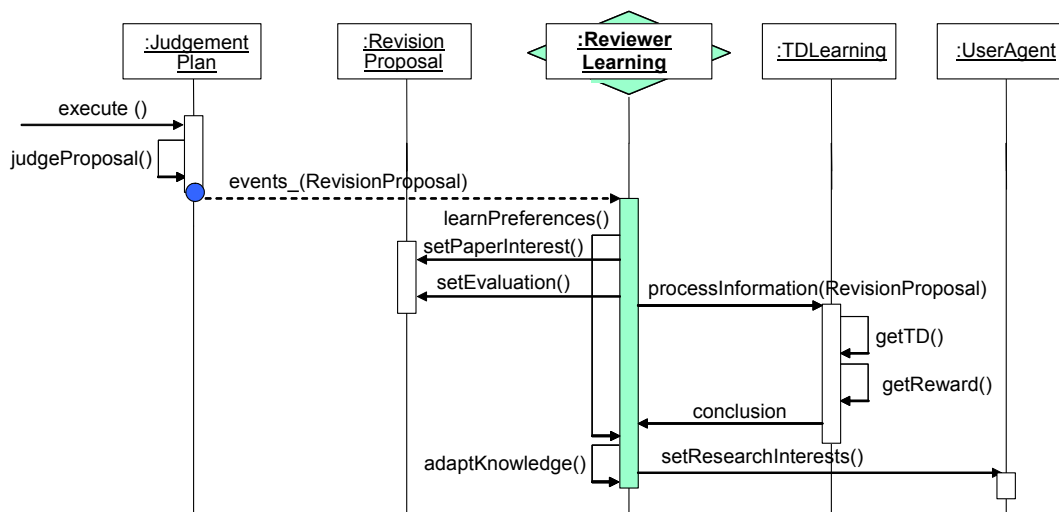


Figure 6. Learning the Reviewer Preferences.

Consequences

The Learning pattern has the following consequences:

- *Improved Separation of Concerns.* The learning protocol is entirely separated from the other agent concerns, such as the agent's basic concerns and interaction. The classes and aspects associated with other agent concerns have no learning code.
- *Reusability.* The basic learning protocol is modularized in a generic learning aspect, which can be reused and refined to different contexts.
- *Readability and Maintainability.* The agent kernel is not intermingled with invocations of methods responsible for the learning implementation. As a consequence, the pattern solution improves readability, which in turn improves maintainability.
- *Ease of Evolution.* As the multi-agent system evolves, new agent classes may have to be monitored and trigger the learning process. Agent developers need only to add new pointcuts in the learning aspects in order to implement the new required functionality.
- *Reduced Code Replication.* The pattern supports the isolation of the learning protocol in learning aspects, minimizing the code replication.
- *Flexibility.* The pattern solution is flexible enough to support the association of different learning strategies with distinct agent types and role classes.
- *Transparency.* Aspects are 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 agent classes are not intrusively modified.
- *Generality.* The solution of the Learning Aspect pattern is general enough to support the modularization of the learning concern independent of machine learning techniques in use. The pattern solution presents the central components required in the learning techniques.

Although the learning concern is completely defined apart from other agent concerns, the use of the pattern imposes some problems to the agent designer:

- *Required Refactoring.* In some circumstances, the realization of the Learning Aspect pattern requires restructuring of the base code associated with other agent components in order to expose suitable join points. In this way, capturing the learning concern as aspects sometimes requires restructuring of the classes and methods to expose suitable join points. For instance, we have extracted code from existing methods of a plan class into a new method to expose a method-level join point so that the learning aspects can intercept it. Tools to help in the refactoring would make it easier to introduce aspects into an existing system.
- *Description of Learning Aspects Depends on Specific Core Classes.* The names of agent classes, role classes and plan classes appear in the definition of pointcuts in the learning aspects. The description of a Learning Aspect cannot be directly applied to other agents.
- *Introduction of More Design Elements.* The Learning Aspect pattern introduces new design elements (aspects) to promote the separation of the learning concern. This solution introduces another level of indirection.

Variants

Reflective Learning. This variant is similar to the aspect-oriented solution presented here. However, this variant rests on the use of the Reflection architectural pattern [29]. This reflective solution uses learning meta-objects as an alternative to learning aspects. Each learning aspect is a meta-class and learning subaspects are defined subclassing this meta-class. The `LearningKnowledge` crosscutting

interface is defined as attributes internal to the learning meta-classes. The `InformationGathering` crosscutting interface is defined using the meta-object protocol that intercepts the methods calls (events) to objects and redirects the control flow to meta-objects. The disadvantage of this reflective variant is that it requires a meta-object protocol which usually introduces changes to the virtual machine. In addition, reflective solutions do not directly support the composition of the learning meta-classes with other meta-classes modularizing other crosscutting concerns. As the agents' complexity increases, good composition mechanisms are essential to the system reusability and maintainability.

Direct Learning. The Learning Aspect pattern's basic solution considers indirect training for knowledge acquisition [2, 3]; it does not explicitly address direct training [2, 3]. The *Direct Learning* variant includes two additional classes `TrainingExperience` and `PerformanceMeasure` to support the direct learning [19]. The basic solution of the pattern is reused; however, these additional classes are associated with the learning component. The `PerformanceMeasure` class implements an algorithm that manages rules for standard performance [19]. The `TrainingExperience` class implements the algorithm that generates training examples for the learning algorithm [19].

Known Uses

Developers have been using a design solution similar to the Learning Aspect pattern to implement the Brainstorm framework for multi-agent systems [20]. This framework implements the reflective learning variant. The `LearningAspect` elements are implemented as meta-objects. We have also implemented the Learning Aspect pattern both in the EC system [21] and in the Portalware system [10, 11]. The Portalware system has learning aspects associated with information agents in order to optimize user queries. The queries are intercepted by the aspects, which is the information used by learning components to build the user profiles. The user profiles are used to optimized the next user queries.

The EC system is used through this paper to illustrate the pattern application. The "Solved Example" section presents the pattern instantiation for the `ReviewerLearning` aspect. There are other learning aspects in the EC system, such as the `ChairLearning` aspect - the learning aspect associated with the chair role. This aspect is connected to a LMS class that implements the LMS learning technique. This class extends the `LearningComponent` abstract class. The agent playing the chair role learns new research interests of a specific reviewer based on the reply of the paper review proposal. Hence, the implementation of the `ChairLearning` aspect accesses the attributes `paperInterest` and `evaluation`, which were updated by the `ReviewerLearning` aspect, so that the chair role can learn new research interests of the reviewers. The `ChairLearning` aspect detects that important actions on chair-specific classes were performed, gets information for the learning purpose, and triggers the learning process. While the `ReviewerLearning` aspect "updates" a specific object (`:RevisionProposal`) with learning-related information, while the `ChairLearning` aspect "accesses" the same object to learn based on the information appended by the reviewer side.

We know other software projects that implement learning in an object-oriented manner and could use this pattern. Some of these systems are the following:

- A real system [7, 13] developed for the participation in the Trading Agent Competition (TAC) [30]. TAC 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. There are two types of intelligent agents in this system which incorporates learning machine techniques: the Hotel Negotiator Agent and the

Price Predictor Agent. The former uses: (i) a minimax decision tree [3] and an evaluation function based on perceptrons [2] (neural networks) to model the *agent knowledge*, (ii) a Learning aspect to modularize the auction history and the final results of the auctions (*learning-specific knowledge*), and the specification of methods called to finalize the auctions (*information gathering*) - the events that trigger the agent learning, and (iii) a Learning component that implements the TD-Learning algorithm. The second agent uses: (i) an exponential smoothing technique [34] to model the *agent knowledge*, (ii) a Learning aspect to separate the ask prices and last predicted ask price (*learning-specific knowledge*), and the specification of auction-related methods that are called in each minute of the game (*information gathering*), and (iii) a Learning component which implements the Back Propagation [2, 4] and LMS algorithms.

- A system [14] that implements the Tic-Tac-Toe game. The agents here use a minimax decision tree [3] and neural networks to implement the *agent knowledge*, and a Learning aspect to encapsulate the player trajectories and the final result of the game (*learning-specific knowledge*), and the specification of methods called to make new plays and to finalize the game (*information gathering*). A Learning component was used to implement an algorithm for adaptive dynamic propagation [3].

See Also

The Learning Aspect pattern is a variant of the Learning pattern [19]. The Learning Aspect pattern is alternatively related to the Role Object pattern [20] when this pattern solution is used to structure the agent roles; the learning aspects learn based on the execution of role methods. The Learning Aspect pattern contains the aspect-oriented implementation of the Observer pattern [22]. The Strategy pattern [5] can be used to implement different learning strategies. Finally, the implementation of the Learning Aspect pattern (see below) uses some idioms [23] for the AspectJ language [24], like *Template Advice*, *Composite Pointcut*, and *Advice Method*.

Implementation

We describe below some guidelines for implementing the Learning Aspect pattern. We give AspectJ [24] code fragments to illustrate a possible implementation of the pattern, describing details of the EC example.

Step 1: How to define a Learning Aspect?

A Learning Aspect must define the general learning protocol. This aspect must define the attributes and methods common to all the learning aspects in the system. For example, it holds a reference to the associated learning components, an abstract method to initialize these components, and an abstract method to invoke the learning components.

➔ The EC system contains the implementation of a general `Learning` aspect to both chair and reviewer agent roles. This aspect is declared as abstract. Note that the initialization method is called by an after advice, which is in turn associated with an abstract pointcut. In AspectJ, pointcuts are used to define which join points on the object execution the aspect is interested to observe. These pointcuts must expose as parameters the information (object instances) necessary to be used in the aspect context. Advices associated with these pointcuts invoke methods on aspects and classes, and if it is necessary they pass the information gathered in the pointcuts as arguments. The `learningInstantiation` pointcut describes when a specific learning aspect should be initialized; it is abstract because it depends on the agent type or role class associated with the specific learning aspect. This aspect also specifies the methods: (i) `learnPreferences()` - which is responsible for

invoking the learning components; and (ii) `updatePreferences()` – which updates the user research interests, after the execution of the learning algorithm.

```
public abstract aspect Learning {
    ...
    protected Hashtable Role.learningComponents = new Hashtable();

    protected void abstract init(Role role);

    protected abstract pointcut learningInstantiation(Role role);

    after(Role role): learninInstantiation(role) {
        System.out.println("<* Learning *> initialization:" + ((Role)role).getName());
        init(role);
    }

    public Hashtable abstract learnPreferences(Hashtable currentInterests,
        Vector my_keywords, boolean newDecision, int currentPaperInterestDegree);

    public void updatePreferences(Hashtable currentInterests, Hashtable newPreferences)
    { ... }
    ...
} □
```

Step 2: Why the learning aspect must be singleton?

In general, each agent instance must have its own learning aspect. As a consequence, learning aspects must be instantiated per Agent instance. The current version of AspectJ supports the specification of per-object aspects. We could describe the instantiation of the `Learning` aspect using `perthis`:

```
public abstract aspect Learning perthis(Agent) {...}
```

However, the use of `perthis` restricts the scope of the aspect. When one AspectJ aspect is declared to be singleton or static, its scope is the whole system and the aspect can crosscut all system classes. Per-object aspects can only crosscut the object with which it is associated. Since the learning protocol crosscuts several classes, not only the Agent class or the Role class, the `perthis` clause cannot be used in this context. As a result, you have to declare learning aspects as singletons and introduce the methods and attributes to the Agent and Role classes. This was the strategy followed in the definition of the `learningComponents` attribute described in Step 1.

Note that although the structure of the Learning Aspect pattern does not describe these aspect members as part of a crosscutting interface, they have to be introduced due to AspectJ restrictions. They are declared as protected, which means that “they are protected to the aspects”: only code in the aspect and subaspects can see these fields and methods. If the Agent or Role class has other protected members named in the same way (declared in the Agent or in another class) there will not be a name collision, since no reference to these members will be ambiguous. The use of inter-type declarations complicates the design of the Learning aspect since it requires the agent or role instance to be exposed as a parameter in each advice of the Learning aspect.

➔ The `Learning` aspect in the EC system is implemented in AspectJ as a singleton aspect since it crosscuts many system classes. In this sense, the `Role` instance is passed as a parameter in the advices of the `Learning` aspects so that the advice code can determine which system’s role is in charge of being adapted. □

Step 3: How to define the general crosscutting interface for information gathering?

The `Learning` aspect must define the abstract pointcut `events()` responsible to declare join points in the object execution where the learning algorithms must be invoked. This pointcut must be refined in the concrete learning subspects. You should use: (i) the *Composite Pointcut* idiom [23] when there are several events to be monitored, and (ii) the *Advice Method* idiom [23] for deciding whether an event is relevant or not for the learning process.

```
public abstract aspect Learning {
    ...

    protected abstract pointcut events(RevisionProposal proposal, Plan plan);

    ...
} □
```

Step 4: How to define the crosscutting interface for specifying the learning knowledge?

You must define each element of the learning knowledge as an inter-type declaration in `AspectJ`. Sometimes, the learning knowledge affects several agent classes and role classes. In these cases, you should use the *Introduction Container* idiom [23].

↪ In the EC system, the abstract `Learning` aspect introduces the attributes `evaluation` and `paperInterest` in the `RevisionProposal` class, in order to become possible that the chair and reviewer roles can learn based on the paper evaluation.

```
public abstract aspect Learning {
    ...
    private Hashtable RevisionProposal.evaluation = new Hashtable();

    private int RevisionProposal.paperInterest = 0;

    public Hashtable RevisionProposal.getEvaluation() {
        return evaluation;
    }

    public void RevisionProposal.setEvaluation(Hashtable evaluation){
        this.evaluation = evaluation;
    }

    public int RevisionProposal.getPaperInterest(){
        return paperInterest;
    }

    public void RevisionProposal.setPaperInterest(int interest) {
        this.paperInterest = interest;
    }
    ...
} □
```

Step 5: How to define a specific Learning aspect?

You must create learning subspects to define the learning behavior specific to an agent type or role context by extending the abstract `Learning` aspect.

↪ In the EC system, we implemented the specific learning aspects to both chair and reviewer agent roles. An agent playing the chair role learns new research interests of a specific reviewer based on the reply of the paper review proposal. An agent playing the reviewer role learns new research

interests of a user based on its feedback. Since the chair and reviewer learning have common aspects and are inter-related, we implemented their behavior in an aspect hierarchy, composed by the Learning aspect and ChairLearning and ReviewerLearning subaspects.

For example, the ChairLearning aspect implements the abstract pointcut events(), defined in the abstract Learning aspect, by intercepting the method verifyReviewerResponse() of the ProposalJudgementReceptionPlan class. This pointcut is associated with an after advice, which is responsible for evaluating the paper revision proposal returned by the reviewers (invoking the method learnPreferences()). This advice is also responsible for updating their research interests (invoking the method updatePreferences()), using the information (evaluation and paperInterest attributes) introduced in the RevisionProposal class.

```
public aspect ChairLearning extends Learning {
    ...
    // (reviewer name, table of research interests)
    public Hashtable Chair.reviewers = new Hashtable();

    protected pointcut events(RevisionProposal proposal, Plan plan): (
        this(plan) && args(proposal) &&
        execution(void JudgementReceptionPlan.evaluateResponse(RevisionProposal)));

    after (RevisionProposal proposal, Plan plan): events(proposal, plan) {
        boolean acceptedProposal = proposal.isAccepted();
        Paper paper = proposal.getPaper();
        ResearchArea area = paper.getResearchArea();
        Vector paperKeywords = area.getResearchKeywords();
        Hashtable reviewerEvaluation = proposal.getEvaluation();
        int reviewerInterest = proposal.getPaperInterest();
        Reviewer reviewer = proposal.getReviewer();
        String reviewerName = reviewer.getName();

        //getting the reviewer' current interests
        Hashtable reviewerPreferences = (Hashtable)reviewers.get(reviewerName);

        //learning the new user interests
        Hashtable newPreferences = learnPreferences(reviewerPreferences,
            paperKeywords, acceptedProposal, reviewerInterest);

        //update my preferences
        updatePreferences(reviewerPreferences, newPreferences);
    }

    public Hashtable learnPreferences(Hashtable currentInterests, Vector my_keywords,
        boolean newDecision, int currentPaperInterestDegree) {...}
}
```

The ReviewerLearning subaspect defines the pointcut events() by intercepting the method judgeProposal() of the JudgementPlan class. This pointcut is associated with an after advice, which invokes the method learnPreferences() passing the information about the paper revision proposal. This invocation results in the update of the reviewers' research interests based on their evaluation. □

Step 6: How to define learning knowledge specific to an agent type or role?

In general, the learning knowledge is defined only at the Learning aspect (Step 4). However, there is sometimes a need for defining learning knowledge specific to an agent type or role. In this case, you only need to define this specific learning knowledge as inter-type declarations in the subaspects.

➔ The ChairLearning aspect specifies the reviewers attribute which maintains the learning knowledge about the research interests of the reviewers. This attributed is introduced to the Chair

class. The attribute `reviewer` is initialized through the pointcut `learningInitialization()` and its respective advice; this advice runs before the execution of the method `sendPapersToReviewer()` of the `PaperDistributionPlan` class.

```
public aspect ChairLearning extends Learning {
    ...
    // (reviewer name, table of research interests)
    public Hashtable Chair.reviewers = new Hashtable();
    ...
} □
```

Step 7: How to initialize a specific learning aspect?

A specific Learning Aspect needs to be initialized when a given event happens. The initialization involves attributes of the specific Learning Aspect, and the associated Learning Components. Use a pointcut to define when the aspect should be initialized. Use an advice to implement the initializations, and associate this advice with the initialization pointcut.

➔ The `ChairLearning` aspect specifies an initialization pointcut. The triggering event is the beginning of the paper distribution, i.e. the execution of the `sendPapersToReview()` method.

```
public aspect ChairLearning extends Learning {
    ...
    protected pointcut learningInitialization(Agent agent, Reviewer reviewer, List papers):
        args (agent, reviewer, papers) &&
        call(public void PaperDistributionPlan.sendPapersToReviewer(Agent,Reviewer,List));

    before (Agent agent, Reviewer reviewer, List papers):
        learningInitialization(agent, reviewer, papers) {
            String reviewerName = reviewer.getName();
            Hashtable reviewer_interests =
                (Hashtable) reviewers.get(reviewerName);
            if (reviewer_interests == null) {
                reviewer_interests = new Hashtable();
                reviewers.put(reviewerName, reviewer_interests);
            }
            // Initialize the research interest of the reviewer
            ...
        }
} □
```

Acknowledgments

We would like to give special thanks to James Noble, our shepherd, for his important comments, helping us to improve our pattern. This work has been partially supported by CNPq under grant No. 141457/2000-7 for Alessandro Garcia, grant No. 140252/2003-7 for Uirá Kulesza, grant No. 140601/2001-5 for José Sardinha, and by FAPERJ under grant No. E-26/150.699/2002 for Alessandro. The authors are also supported by the PRONEX Project under grant 7697102900, and by ESSMA under grant 552068/2002-0 and by the art. 1st of Decree number 3.800, of 04.20.2001.

References

- [1] Camacho, D. et al. MAPWEB: Cooperation between Planning Agents and Web Agents. *Information & Security: An International Journal*, Vol 8, N 2, pp. 209-238, 2002.
- [2] Mitchell, T. *Machine Learning*. McGraw Hill, New York, 1997.
- [3] Russell, S., Norvig, P. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2nd ed., 2002.
- [4] Bigus, J., Bigus, J. *Constructing Intelligent Agents Using Java: Professional Developer's Guide Series*. 2nd Edition, John Wiley & Sons, 2001.

- [5] Gamma, E. et al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA, 1995.
- [6] Kendall, E. et al. A Framework for Agent Systems. Implementing Application Frameworks – Object-Oriented Frameworks at Work, M. Fayad et al. (eds.). John Wiley & Sons: 1999.
- [7] Sardinha, J., Milidiú, R., Lucena, C. Engineering Machine Learning Techniques into Multi-Agent Systems. Submitted to Intl. Journal of Software Eng. and Knowledge Eng., 2004.
- [8] Sardinha, J., Ribeiro, P., Lucena, C., Milidiú, R. An Object-Oriented Framework for Building Software Agents. Journal of Object Technology, Jan - Feb 2003, Vol. 2, No. 1.
- [9] Pace, A., Campo, M., Soria, A. Architecting the Design of Multi-Agent Organizations with Proto-Frameworks. In: Software Engineering for Multi-Agent Systems II, LNCS 2940, Berlin, Feb 2004, pp. 75-92.
- [10] Garcia, A., Cortés, M., Lucena, C. An Environment for the Development and Maintenance of E-Commerce Portals based on a Groupware Approach. Proc. of the IRMA'01 Conference, Toronto, May 2001, pp. 722-724.
- [11] 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.
- [12] Garcia, A., Sant'Anna, C., Chavez, C., Lucena, C., Staa, A. Separation of Concerns in Multi-Agent Systems: An Empirical Study. In: Software Engineering for Multi-Agent Systems II, LNCS 2940, Jan 2004.
- [13] Sardinha, A., Garcia, A., Lucena, C., Milidiú, R. On the Incorporation of Learning in Open Multi-Agent Systems: A Systematic Approach. Proc. of the 6th AOIS Workshop at CAiSE'04, Riga, Latvia, June 2004.
- [14] Sardinha, J., Milidiú, R., Lucena, C., Paranhos, P. An OO Framework for Building Intelligence and Learning Properties in Software Agents. Proc. of the SELMAS'03 Workshop, Portland, USA, May 2003.
- [15] Bellifemine, F., Poggi, A., Rimassi, G. JADE: A FIPA-Compliant Agent Framework. Proc. of the Practical Applications of Intelligent Agents and Multi-Agents, April 1999; pp. 97-108.
- [16] Chavez, C. A Model-Driven Approach to Aspect-Oriented Design. PhD Thesis, Computer Science Department, PUC-Rio, April 2004, Rio de Janeiro, Brazil.
- [17] Chavez, C., Lucena, C. Design-level Support for Aspect-oriented Software Development. Proc. of the Workshop on Advanced Separation of Concerns at OOPSLA'2001, Tampa Bay, USA, October 14, 2001.
- [18] Nwana, H. et al. ZEUS: An Advanced Toolkit for Engineering Distributed Multi-Agent Systems. Applied Artificial Intelligence Journal, 1999, 13(1):129-186.
- [19] Sardinha, J., Garcia, A., Milidiú, R., Lucena, C. The Learning Pattern. 4th Latin American Conference on Pattern Languages of Programming, SugarLoafPLoP'04. August, 2004, Fortaleza, Brazil.
- [20] Amandi, A., Price, A. Building Object-Agents from a Software Meta-Architecture. In: Advances in Artificial Intelligence, LNAI, vol. 1515, Springer, 1998.
- [21] Garcia, A. From Objects to Agents: An Aspect-Oriented Approach. Doctoral Thesis, Computer Science Department, PUC-Rio, Rio de Janeiro, Brazil, April 2004.
- [22] Hannemann, J., Kiczales, G. Design Pattern Implementation in Java and AspectJ. Proc. of OOPSLA'02, November 2002, pp. 161-173.
- [23] Hanenberg, S., Unland, R., Schmidmeier, A. AspectJ Idioms for Aspect-Oriented Software Construction. Proc. of the EuroPlop'03, Irsee, Germany, June 2003.
- [24] AspectJ Team. The AspectJ Programming Guide. March, 2003. <http://eclipse.org/aspectj/>
- [25] Kiczales, G. et al. Aspect-Oriented Programming. Proceedings of the European Conference on Object-Oriented Programming - ECOOP'97, LNCS (1241), Springer-Verlag, Finland., June 1997.
- [26] Fowler, M. Dealing with Roles. Proceedings of the 4th Annual Conference on the Pattern Languages of Programs, Monticello, Illinois, USA, September 2-5, 1997.
- [27] Odell, J., Parunak, H., Fleischer, M. The Role of Roles in Designing Effective Agent Organizations. Software Engineering for Large-Scale Multi-Agent Systems, LNCS 2603, Springer, April 2003, pp. 27-38.
- [28] Bowerman, B., O'Connell, R. Forecasting and Time Series: An Applied Approach. Thomson Learning; 3rd edition, Massachusetts: Duxbury Press, March 1993.
- [29] F. Buschmann et al. Pattern-Oriented Software Architecture: A System of Patterns. John Wiley Sons, 1996.
- [30] TAC web site.: <http://www.sics.se/tac>.