The Interaction Aspect Pattern

Alessandro Garcia[1] Uirá Kulesza[2] Christina Chavez[3] Carlos Lucena[2]

[1]Computing Department
Lancaster University
InfoLab 21
Lancaster
United Kingdom
afgarcia@comp.lancs.ac.uk

[2]Computer Science Department
Pontifical Catholic University of
Rio de Janeiro (PUC-Rio)
Rio de Janeiro
Brazil
{uira, lucena}@inf.puc-rio.br

[3]Computer Science Department
Institute of Mathematics
Federal University of Bahia
Salvador
Brazil
flach@im.ufba.br

Interaction Aspect

A multi-agent system (MAS) is composed of autonomous agents that have the ability to interact with its surrounding environment. They need to be interactive in order to communicate with other agents and observe external events in its environment. However, as the agents' complexity increases, object-oriented abstractions cannot modularize their interactive behavior that usually tends to spread across several classes and methods of each agent design. The Interaction Aspect pattern supports the separate definition of interaction-related concerns through the use of aspects. It decouples the agent's interactive behavior from the implementation of its basic functionality and other agent-specific concerns, which in turn improves the system reusability and maintainability.

Keywords Software Agents, Interactive Agents, Multi-Agent Systems, Aspect-Oriented Software Development.

Example Consider a multi-agent system that supports the management of paper submissions for conferences as well as the reviewing process. This system 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* [5, 20] 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*. Roles are associated with *plans*, which 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 negotiates with reviewers for performing reviews. Figure 1 shows classes representing the agents' basic functionalities and some examples of roles and plans.

In the EC system, interaction is required in several circumstances. Messages need to be sent from different agent plans and actions. In addition, user

agents need to send messages to each other, for example, when playing the reviewer and chair roles. The chair sends messages with revision proposals to the reviewers and they send the responses back to the chair. Each reviewer uses a different communication language. Moreover, an agent, as a reviewer, needs to observe different external software components: (i) the user agenda which is implemented by a scheduling software system, (ii) the user interface, and (iii) a component that manages the user curriculum. Figure 1 illustrates a representative example of the interactive behavior of EC agents. The agents' interaction capabilities are supported by a set of sensors and effectors.

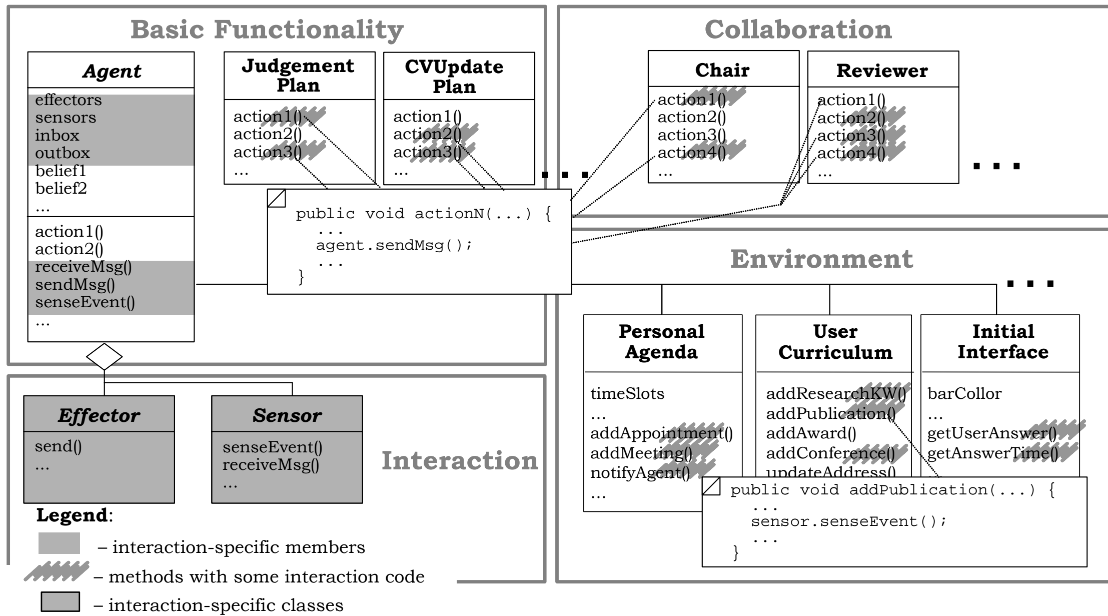

Figure 1. The Object-Oriented Design of the Interaction Concerns.

Object-oriented designs and implementations of the interaction property are intrusive [12, 16]. Figure 1 shows a partial OO representation of the EC system, where each set of classes, surrounded by a gray rectangle, has the main purpose of modularizing a specific agent-related concern, namely: the agents' basic functionality, the collaboration concern, and the environment concern. Although part of the interaction concerns are localized in some specific modules (`Sensor` and `Effector` classes), the interaction issues crosscuts the basic classes, the roles, and the environment objects. Figure 1 illustrates how the interaction behavior is tangled and scattered through the basic agent classes, the collaboration classes, and the environment classes.

The methods, attributes and code affected by the interaction property are shadowed in the figure. The agent classes, which represent the agent's basic functionality, are usually intermingled with interaction methods, such as `sendMsg()`, `receiveMsg()`, and `senseEvent()`, and interaction attributes, such as `inbox` and `outbox`. Moreover, the interaction implementation crosscuts several agent plans and actions, which send messages to collaborative agents. The interaction behavior also crosscuts the role methods, which are intermingled with calls to the `sendMsg()` method. It also crosscuts multiple environment objects which are monitored by the agent (sensory behavior). The objects need to notify the agents about relevant events by invoking the methods of `Sensor` classes.

Kendall [16] proposes the realization of the Adapter pattern [6] to improve the separation of the sensory behavior. This pattern is used to create a reusable class that cooperates with environment classes. Figure 2 illustrates the pattern instantiation for the EC system. The key abstraction of the Kendall's solution is a set of specific sensors that need to cooperate with domain-specific classes. This solution supplies a `Sensor` class with the abstract method `sense()`, while the application developer provides the domain-specific sensor adapters.

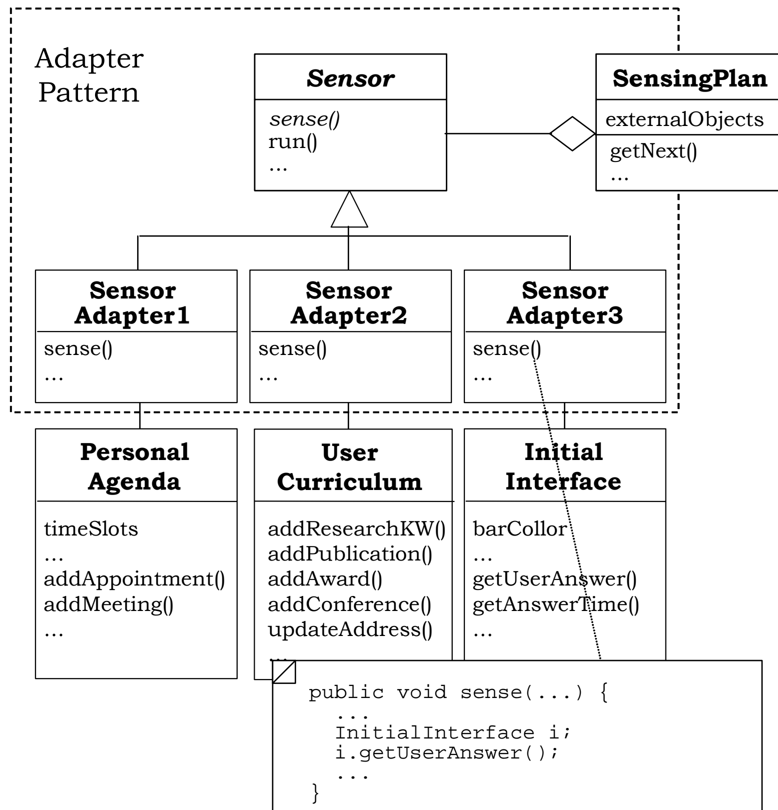

Figure 2. Sensing Behavior in the EC Agents: The Adapter-Based Solution

There is a list of external objects that need to be sensed; these are stored in the `SensingPlan` class. The `SensingPlan` may be just a collection of the sensors. Figure 3 shows a `Sensor` object utilizes a `SensingPlan` and a `SensorAdapter` to sense external events regarding changing information in an environment object, i.e. an `InitialInterface` object. The corresponding adapter obtains information about new user commitments and preferences according to the data entered in the user interface. For additional details on this pattern solution, refer to [16].

The solution improves the separation of concerns, since it isolates the sensory behavior in the adapter subclasses and in the `SensingPlan` class. This structure leads to less tangled code since non-interaction code does not interlace with interaction code, but does not completely avoid it. The code for sending messages within agent classes cannot be untangled. It is because this solution only tries to solve the problem related to sensory behavior. It does not modularize the interactive behavior that is tangled to basic agent classes. Moreover, in the cases where it can be untangled (sensory behavior), the

designer has to pay a high price for that: adapters have to be written just to take care of the sensing functions.

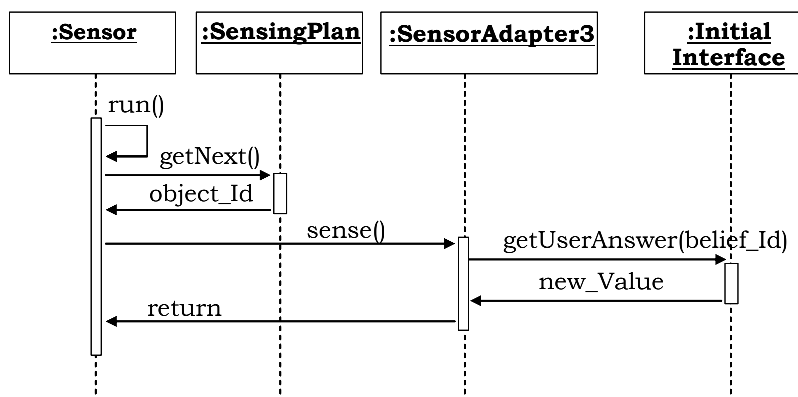

Figure 3. The Dynamic View of the Adapter Pattern.

In addition, the adapter-based proposal itself introduces other problems. Each sensor is a thread which works as a pooling mechanism that waits until some information is gathered from the external environment. This monitoring is expensive and resource-consuming because environment changes may not occur in a frequent way. The sensing behavior should only be activated when some change happens in the environment. Finally, this solution assumes that there are public methods on the domain-specific classes that provide the services and information needed. This assumption may not be always true in certain applications.

Context When using object-oriented abstractions to design the agent's interactive behavior.

Problem Object-oriented abstractions do not support the modularization of interaction concerns. The basic implementation of agent plans and associated actions are intermingled with invocations to the interaction-specific methods responding for sending messages to other agents. In addition, the basic functionalities of environment classes are changed in order to incorporate calls to the sensing methods in an intrusive way. As a consequence, the agent kernel and environment components are highly coupled to the interaction behavior. How to decouple the interaction behavior from the agents' basic structure? The following forces are associated with this problem:

- *Separation of Concerns*. The design solution should separate the interaction behavior from the agents' basic structure.
- *Transparency*. The basic agent structure should be unaware of the interaction behavior. For example, external classes should not be changed in the implementation of the sensory agent capability.
- *Reusability and Maintainability.* MAS engineers should define agent interaction in a reusable and maintainable manner.
- *Ease of Evolution*. It should be easy to evolve the multi-agent system, for example, to allow new plans to send messages or new external objects to be sensed by the agents.
- *Reduced Number of Components and Minimized Coupling.* Explosion in the number of agent components should be avoided as well as in the number of relationships between them.

- *Uniformity.* Received messages and sensed events should be handled in a uniform way.
- *Support for Observing Third-Party Components.* Sometimes the observed classes may be part of third party components, of which we do not have access to the source code. The solution should deal with this situation.

Solution

Use aspects[1] [17] to improve the separation of the interactive agent behavior (Figure 4). Interaction aspects are used to cleanly capture the interaction behavior which affects many parts of a software agent. An Interaction aspect separates the interaction behavior from agent's basic elements, such as actions, plans, and from roles and environment components. In other words, aspects are used not only to modularize the core of the interaction behavior, but also to isolate the whole behavior related to the interaction concerns, which includes message sending, message reception, and sensory behavior.

By using Interaction aspects, we define how the agent actions and plans interact with the external environment. These aspects are able to crosscut some agent execution points – e.g. method invocations on Plan classes - and change their normal execution in order to send and receive messages, and capture external events. The Interaction aspects monitor these execution points in order to identify when a message and events must be handled by using auxiliary classes. Auxiliary classes are used to implement sensors and effectors, and implement both the translation and the (un)marshalling processes.

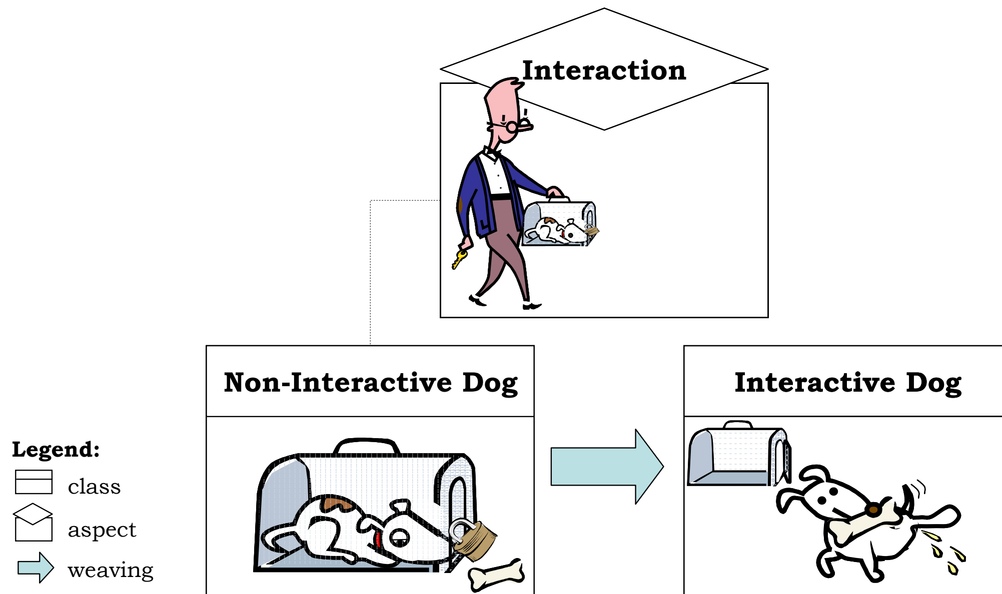

Figure 4. Diagram for Interaction Aspect using the "Dog Interaction" Example

Structure

Figure 5 illustrates the structure of the Interaction Aspect pattern. The design notation is based on an aspect-oriented 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

[1] Appendix A presents a brief overview of terminology related to aspect-oriented design.

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 5). A crosscutting interface is composed of inter-type declarations, pointcuts 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.

The Interaction Aspect pattern has four participants:

- **Interaction Aspect**
  - defines the basic logic of the interaction-specific concerns.
- **Interaction Subaspect**
  - implements the interaction behavior specific to an agent type or role.
- **Sensor**
  - collaborates with communication mechanisms and environment classes in order to receive messages and external events relevant to the agent.
- **Effector**
  - collaborates with communication mechanisms and environment classes in order to send messages and invoke methods on environment classes (i.e. generate events in the external environment).

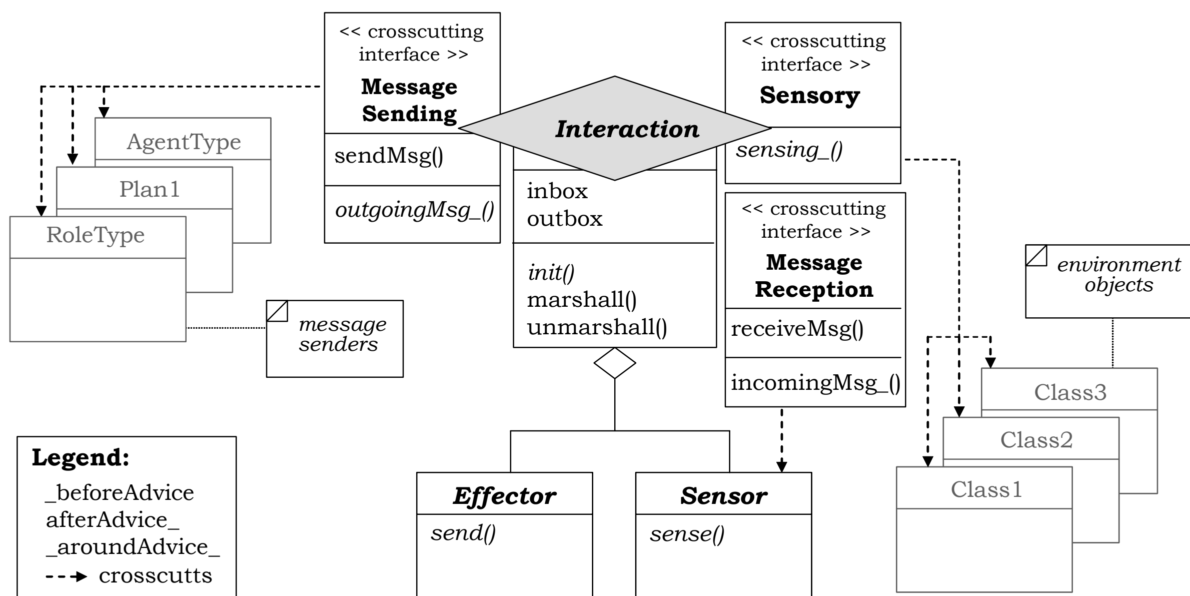

Figure 5. The Static View of the Interaction Aspect Pattern.

Figure 5 shows the parts that are common to all potential instantiations of the pattern, and other parts that are specific to each instantiation. The common parts are:

1. The existence of sensors and effectors (i.e. the fact that some classes

act as sensors and some as effectors).

2. The general interaction logic:
   a. messages are sensed, unmarshaled, and stored.
   b. messages are stored, marshaled, and propagated to the environment.
3. The interface to send and receive messages.

The parts specific to each instantiation of the pattern are:

1. Which classes are sensors and which are effectors in the context of an agent type or specific roles.
2. A set of pointcuts where messages should be sent to the external environment.
3. A set of pointcuts depicting events associated with the environment objects that need to be sensed.

The purpose of the Interaction aspect is to make `Agent` instances interactive. In other words, the Interaction aspect extends the `Agent` class's behavior to both send and receive messages, and sense external events. This aspect send and receive messages and senses environment changes by means of sensors and effectors. Inter-type declarations (static crosscutting) are used to add the new interaction-specific methods: the methods `receiveMsg()` and `sendMsg()`. The `Sensor` and `Effector` classes represent sensors and effectors respectively, and cooperate with environment classes. Sensors and effectors are classes and their isolation from the aspect is intended to improve reuse.

The Interaction aspect has four parts: the aspect itself and three crosscutting interfaces. The aspect holds an inbox, an outbox, initialization methods, methods to marshal and unmarshal the messages, and defines the interaction logic. Since the Interaction aspect implements the interaction logic, it crosscuts the `Agent` class, sensors, effectors, `Agent` or `Plan` classes that need to send a message to other agents, and environment classes which need to be observed by the agent.

The crosscutting interfaces define how the Interaction aspect crosscut different classes of the multi-agent system. The `MessageSending` interface defines an `outgoingMsg` pointcut that specifies the message senders; it specifies join points in the agent classes from which messages need to be sent to the external world. Examples of join points are methods of Plan classes and Agent classes. Note that the `outgoingMsg` pointcut is abstract because the join points depend on the specific agent types and roles. The pointcut is concretized in the Interaction subaspects. The interface contains an advice which runs after executions of actions on agent classes, actions on plan classes, another aspects associated with the agent (for example, Role aspects [8] or Mobility aspects [10]). The purpose of the advice is capturing the information needed to send the message to the agents and updating the `outbox`.

The `MessageReception` interface defines an `incomingMsg` pointcut for intercepting executions of the method `sense()` on the `Sensor` classes; the goal is to detect the arrival of messages. This pointcut is associated with an after advice responsible for processing the incoming messages and updating

the inbox. The `Sensory` interface implements the `sensing` pointcut that declares which methods of the environment classes must be monitored. The `sensing_` advice processes the external events and updates the inbox. The `sensing` pointcut is also declared as abstract since the join points depend on the specific agent types and roles.

Dynamics The following scenarios depict the dynamic behavior of the Interaction pattern.

Scenario I – Receiving a Message from another Agent, which is illustrated in Figure 6, presents the pattern behavior when the interaction aspects receive messages from other agents at specific points in the execution flow:

- The sensor receives a message from the communication platform.
- The sensor delegates the message disassembling and translating processes to auxiliary classes.
- The interaction aspect detects that a message has been received by intercepting the method `sense()` (join point) of the `Sensor` class.
- This aspect captures the received message and updates the `inbox`.

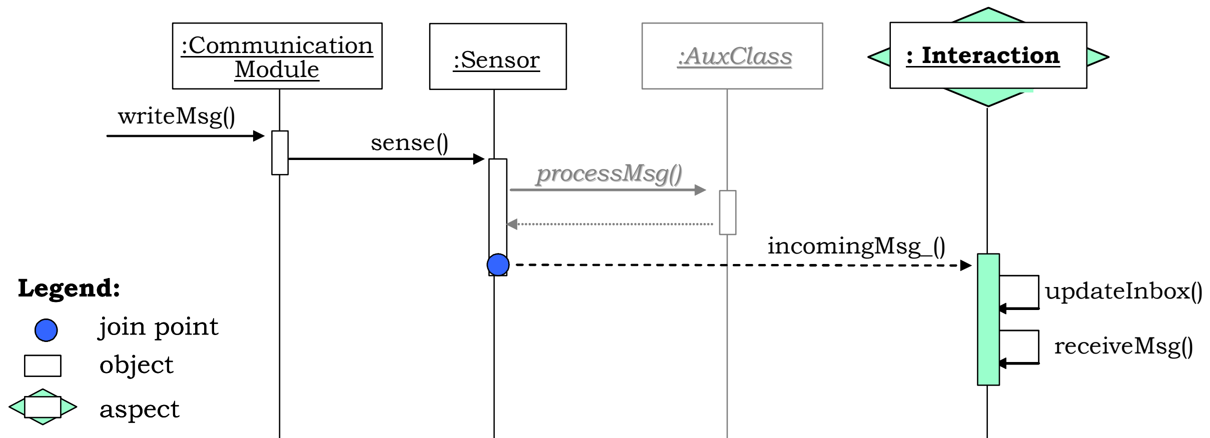

Figure 6. Receiving a Message

Scenario II – Sending a Message, which is illustrated in Figure 7, presents the pattern behavior when the interaction aspects detect the need for sending messages at specific points in the execution flow:

- The agent starts executing one of its actions or plans.
- The Interaction aspect detects points in the execution flow (join points) of the agent actions or plans where a message must be sent.
- The aspect assembles the message and updates the agent outbox.
- The aspect sends the message by selecting the appropriate effector.
- The effector delegates the message translation to auxiliary classes.
- The effector sends the message to the environment using the associated communication platform.

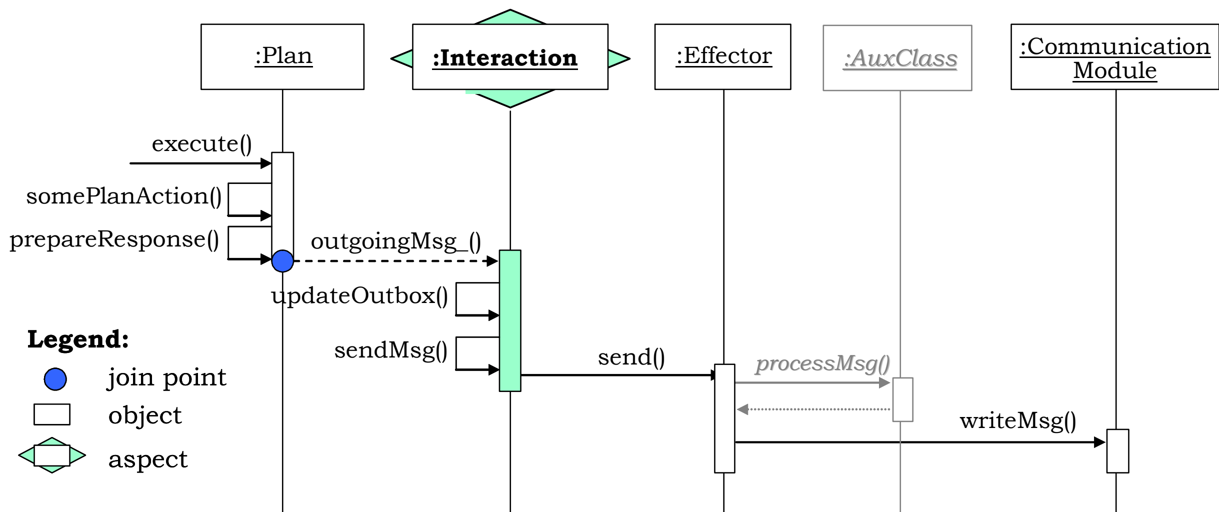

Figure 7. Sending a Message

Scenario III – Sensing an External Event in an Environment Object, which is illustrated in Figure 8, presents the pattern behavior when the interaction aspects senses stimuli from classes in the environment:

- The interaction aspect observes an event from the environment by intercepting some method execution or attribute update on environment classes.
- The interaction aspect delegates the message translation to auxiliary classes.
- This aspect captures the received message and updates the inbox.

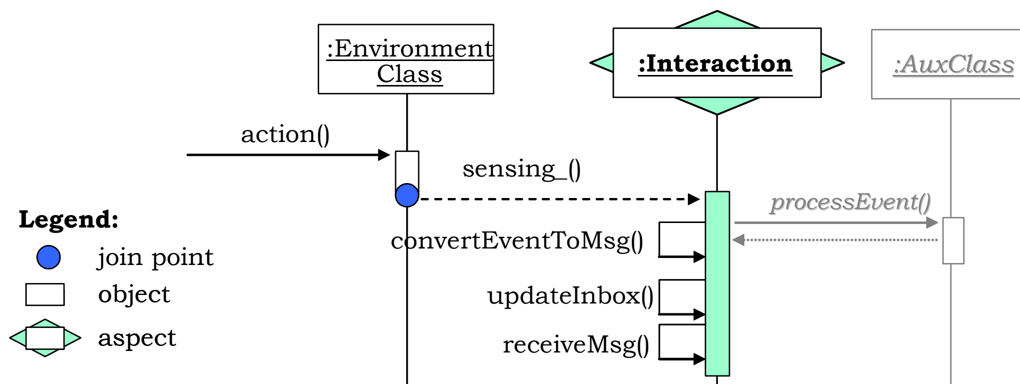

Figure 8. Sensing an External Event

Note that the sensory behavior and messages reception are handled in a uniform way. The only difference is that aspects need to directly inspect environment classes in order to transparently detect external events, while messages are received by sensor classes which are associated with communication platforms.

Solved Example

Figure 9 illustrates the pattern instantiation for the user agent in the Expert Committee system. Note that differently from Figure 1 interaction concerns are modularized here in aspects. This system has a generic Interaction aspect and several subaspects, such as the `ReviewerInteraction` aspect and the `ChairInteraction` aspect. However, for simplification reasons, figure 9 only presents some representative aspects and classes; it has also been omitted the auxiliary classes, which support the message translations.

Each `Interaction` aspect crosscuts different classes, about 15 components in the EC system. Figure 9 shows the `ReviewerInteraction` aspect and some of the classes that it crosscuts. The way the `ChairInteraction` aspect affects classes essentially follows the same pattern. The `ReviewerInteraction` aspect affects, for example, the `judgeProposal()` method of the `JudgementPlan` class to send revision proposal evaluations to the chair agent. `ReviewerInteraction` also intercepts the constructor of the `UserAgent` class in order to initialize the sensors and effectors of a reviewer agent when a new instance of a `UserAgent` is created. Finally, this aspect affects different environment classes to sense stimuli from classes in the environment. As an example, it intercepts changes in the `Agenda` class by intercepting the `updateAgenda()` method in order to update the reviewer agent beliefs related to its schedule planning.

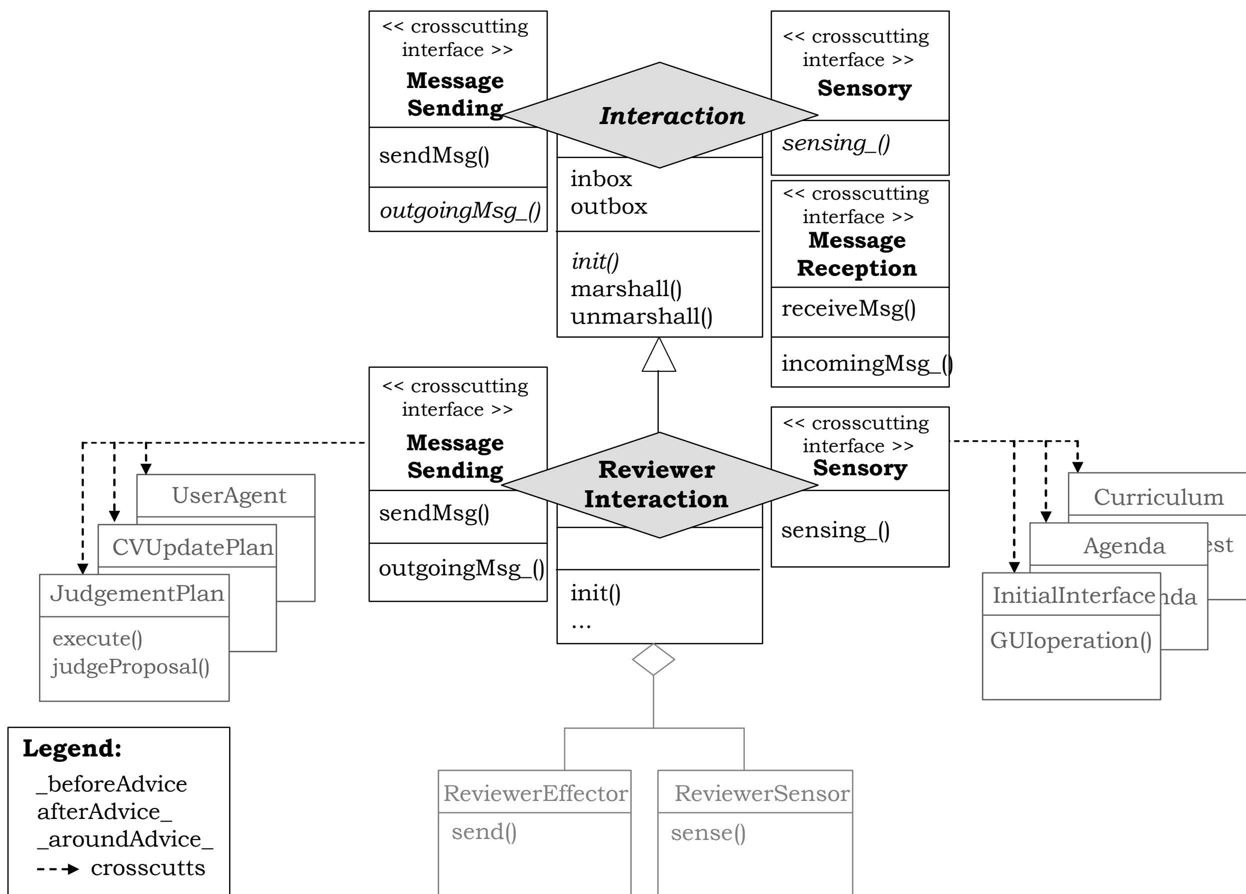

Figure 9. The Interaction Pattern for the EC's User Agent.

Figure 10 presents the pattern behavior when the `ReviewerInteraction` aspect receives messages from the chair agent:

- An effector of the chair agent sends a message to a specific agent reviewer using an agent platform, such as JADE [ref];
- The agent platform notifies the reviewer sensor the incoming of a new message.
- The sensor delegates the message disassembling and translating processes to auxiliary classes.
- The `ReviewerInteraction` aspect detects that a message has been received by intercepting the method `sense()` (join point) of the `Sensor` class.
- The `ReviewerInteraction` aspect captures the received message and updates the agent inbox.

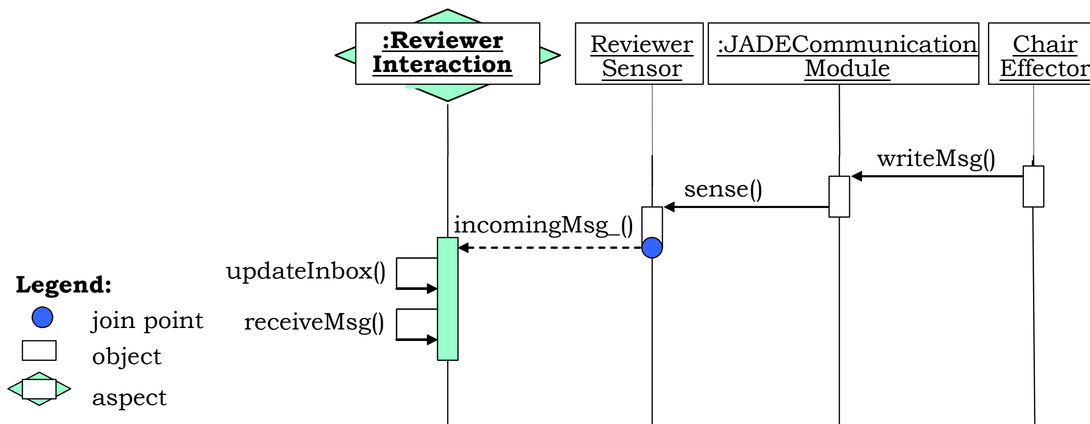

Figure 10. RevieverInteraction receiving a Message from the chair agent

Figure 11 presents the pattern behavior when the `ReviewerInteraction` aspect detects the need for sending messages after the execution of a judgement plan:

- The reviewer agent starts executing its judgement plan.
- The `ReviewerInteraction` aspect detects after the execution of the judgement plan the need to send the results to the chair agent. Thus, to make it possible, it intercepts the `prepareResponse()` method of the `JudgementPlan` class.
- The `ReviewerInteraction` aspect assembles the message containing the revision proposal evaluation and updates the agent outbox.
- The `ReviewerInteraction` sends the message by selecting the appropriate reviewer effector.
- The reviewer effector delegates the message translation to auxiliary classes.
- Finally, the effector sends the message to the environment using the associated platform.

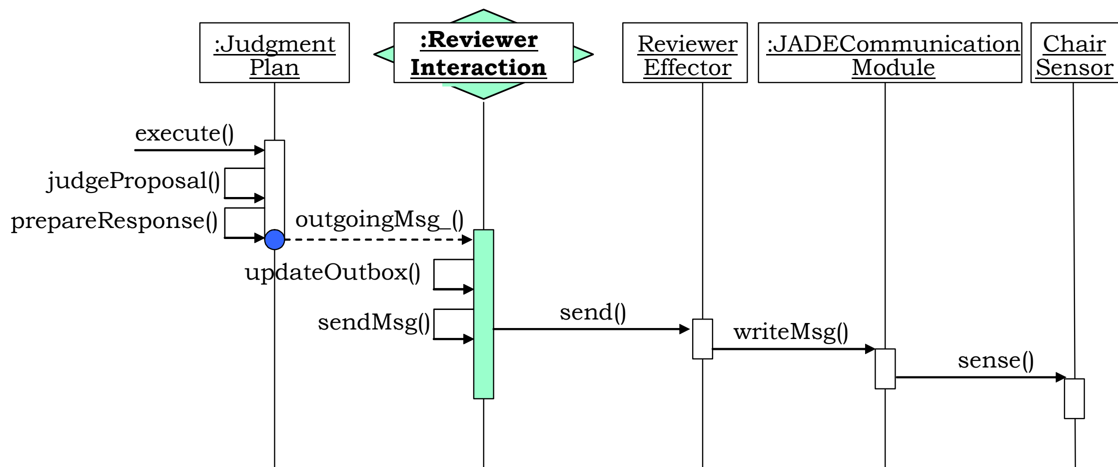

Figure 11. ReviewerInteraction sending a message to the chair agent

Figure 12 presents the pattern behavior when the `ReviewerInteraction` aspects senses some updates in the user agenda object and decides to gather those changes in order to update the reviewer beliefs:

- The `ReviewerInteraction` aspect observes changes in the user agenda object by intercepting the `updateAgenda()` method of the `Agenda` class.
- Using the `convertEventToMsg()` method, the `ReviewerInteraction` aspect converts the information related to the agenda update in a specific message format (using the `AuxClass class`), which the reviewer agent is able to process.
- The `ReviewerInteraction` aspect updates the agent inbox with a new message.

Consequences The use of the Interaction pattern provides the following advantages:

- *Improved Separation of Concerns.* The interaction protocol is entirely separated from the other agent concerns and environment classes. The aspect-oriented constructs support the separate definition of message sending and sensory behavior that affect several units of the system. This separation of concerns allows for better modularity, avoiding tangled code and code spread over several units.

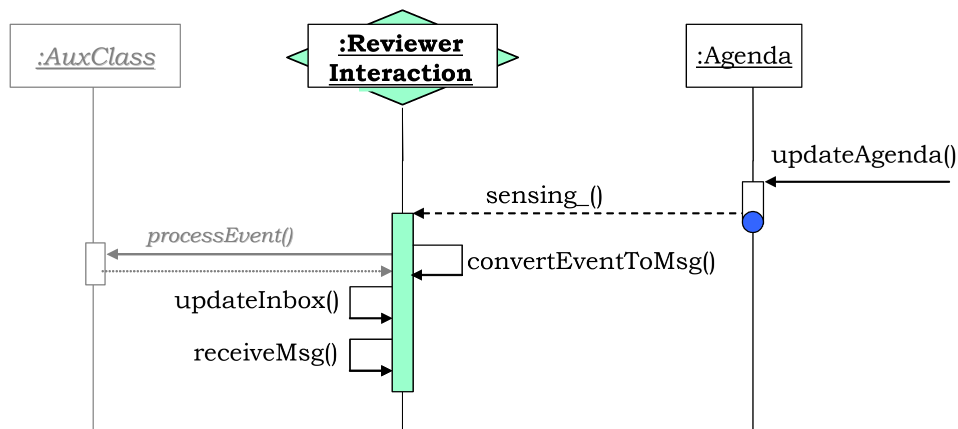

Figure 12. Sensing an External Event

- *Transparency.* The use of aspects is an effective solution to introduce the sensing behavior into application classes not designed to be sensed. Environment classes are observed in a transparent way.
- *Reusability and Maintainability.* The improved separation of concerns provided by the Interaction pattern also contributes to increase system maintainability and reusability.
- *Ease of Evolution.* MAS developers need only to add new pointcuts in the Interaction aspect in order to implement new required functionalities, such as the need for new plans to send messages or new external objects to be sensed by the agents.
- *Reduced Number of Components and Minimized Coupling.* The interaction pattern reduces the number of additional classes and interrelationships existing in the adapter-based solution [16]. No sensing plan or additional adapters need to be created to implement the sensing behavior.
- *Uniformity.* Received messages and sensed events are handled in a uniform way.
- *Support for Observing Third-Party Components.* The Interaction pattern is still applicable with third-party components since aspect-oriented languages, such as AspectJ, support bytecode weaving. In this case, only the .class files are required to perform the weaving.

However, this pattern solution has the following drawbacks:

- *Inseparable Concern.* The Interaction Aspect pattern does not modularize the message assembling from different plans or roles; the message needs to be prepared within a method on plan classes or on role aspects because its assembling is very coupled to the role or plan context. The solution could separate the message assembling with aspects but it would result in higher complexity.
- *Repetitive and Time-Consuming Definitions.* All the message senders of the system must be specified in the pointcut inside the Interaction aspect. This might indeed be repetitive and tedious, suggesting that AspectJ should have more powerful metaprogramming constructs. However, this is not an unsolvable problem because code-generation tools can assist MAS engineers in this development step. In addition, we can establish a naming convention and use wildcards supported by most aspect-oriented languages [1]. The implementation of Portalware [9] and Expert Committee [7] used naming conventions.

Known Uses Portalware [9] and Expert Committee [7] have implemented the Interaction pattern. The implementation of a traffic simulator architecture [4] has also been used the Interaction Aspect solution.

Counter-Indications The Interaction Aspect pattern should not be used when your agents interact directly with each other. When there is a direct communication, the agents do not send messages to each other and directly invoke the provided agent services through a public agent interface. Although there is still a crosscutting problem here, it seems to be an inherent part of the adopted interaction protocol.

See Also The Interaction Aspect pattern is related to the Mobility Aspect pattern [10] because a message needs to be sent to the environment agents when an agent decides to move to a remote environment. This message notifies the other agents that the services of the mobile agent are no longer available. The Interaction Aspect needs to intercept the moment when the mobile agent decides to migrate to the new environment. Finally, the Interaction Aspect pattern is related to the Role Aspect pattern [8] since collaborative actions into the roles need to send messages to other agents.

Implementation We describe below some guidelines for implementing the Interaction Aspect pattern. We give AspectJ [1, 18] code fragments to illustrate a possible implementation of the pattern, describing details of the EC example. Although we illustrate an implementation of the Interaction Aspect pattern in AspectJ, the pattern can be specified using a different aspect-oriented programming language following the guidelines presented.

Step 1: How to define an abstract Interaction Aspect?

The `Interaction` aspect is declared as abstract (line 1) since it needs to be redefined in different contexts, i.e. different agent types and roles. The aspect defines different attributes, such as `inbox` (line 2), `outbox` (line 3), `sensors` (line 4), and `effectors` (line 5). Update methods are also defined, for instance, for updating inbox and outbox (lines 7-14).

```
1. public abstract aspect Interaction {
2.    private Vector Agent.inBox = new Vector();
3.    private Vector Agent.outBox = new Vector();
4.    private Vector Agent.sensors = new Vector();
5.    private Hashtable Agent.effectors = new Hashtable();
6.
7.    private void Agent.updateInBox(Message msg) {
8.          inBox.addElement(msg);
9.    }
10.
11.   private void Agent.updateOutBox(Message msg) {
12.         outBox.addElement(msg);
13.   }
14.   ...
```

Each agent instance must have its own inbox, outbox, sensors and effectors. As a consequence, the `Interaction` aspect must be instantiated per `Agent` instance. The current version of AspectJ supports the specification of per-object aspects. In our case, we could describe the instantiation of the `Interaction` aspect using `perthis`:

```
public abstract aspect Interaction 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 interaction protocol crosscuts several classes, not only the `Agent` class, the `perthis` clause cannot be used in this context. As a result, we had to declare our aspect as a singleton and introduce the methods and attributes (lines 2-14) to the `Agent` class.

Note that although the structure of the Interaction pattern does not describe these aspect members as part of a crosscutting interface, they had to be introduced in our AspectJ implementation due to AspectJ restrictions. They are declared as private, which means that "they are private to the aspect": only code in the aspect can see these fields and methods. If the `Agent` class has other private members named in the same way (declared in the `Agent` or in another aspect) there will not be a name collision, since no reference to these members will be ambiguous.

The use of inter-type declarations complicated the design of the `Interaction` aspect since it requires the agent instance to be exposed as a parameter in each advice of the `Interaction` aspect. Our design principle consists of introducing methods only when they are to be intercepted or modified by another aspect. From our point of view, classes should not be aware of the aspects. Even the behavior introduced by the aspects should not be accessed directly by the classes because it increases the system coupling. This design decision can be avoided by using pointcuts and advices which glue aspects and classes [13, 15].

```
15.   private void Agent.receiveMsg(Message msgUnmarshalled) {
16.         updateInBox(msgUnmarshalled);
17.   }
18.   public void sendMsg(Message msg, Agent agent){
19.         agent.updateOutBox(msg);
20.         agent.sendMsgEffector(msg, agent);
21.   }
22.
23.   public abstract void sendMsgEffector(Message msg, Agent agent);
24.   public void init(Agent agent) {
25.         agent.inBox = new Vector();
26.         agent.outBox = new Vector();
27.         agent.sensors = new Vector();
28.         agent.effectors = new Hashtable();
29.
30.         // Init sensors and effectors
31.         initSensorsAndEffectors(agent);
32.
33.         // Set the agent reference in every sensor
34.         Enumeration sensorsEnum = (agent.sensors).elements();
35.         Sensor sensor;
36.         while (sensorsEnum.hasMoreElements()) {
37.               sensor = (Sensor) sensorsEnum.nextElement();
38.               sensor.setAgent(agent);
39.         }
40.   }
41.
42.   public abstract void initSensorsAndEffectors(Agent agent);
```

The `Interaction` aspect defines two fundamental methods:

- `receiveMsg()` – this method (line 15-17) is invoked by the aspect advices responsible for receiving external messages from the environment (agent sensors or domain-specific classes);
- `sendMsg()` – this method (line 18-21) is called by the aspect advices responsible for sending messages to the environment. The source-code shows that this method calls the abstract method `sendMsgEffector()` (line 23). This abstract method implements the

logic to decide which agent effector must be used to send a specific message. Interaction subaspects must implement this abstract method.

Besides, the `Interaction` aspect defines pointcuts and advices in order to: (i) initialize the agent's interaction capabilities; (ii) define the crosscutting interface `IncomingMessage`; and (iii) define the crosscutting interface `OutgoingMessage`. We describe below (steps 2, 3 and 4) the implementation of these elements. ❑

Step 2: How to initialize the agent's interaction capabilities?

The initialization of the interaction capabilities requires the definition of the abstract pointcut `agentInstantiation()` (line 61). This pointcut define the join point in the agent or role execution where the `Interaction` aspect is initialized. In general, it acts on the constructors of specific agent classes. `Interaction` subaspects (step 5) must implement this abstract pointcut. The `Interaction` initialization also needs the definition of an advice attached to the `agentInstantiation()` pointcut. It must be defined as an after advice (lines 62-64) to call the `init()` method of the `Interaction` aspect.

The `init()` method initializes interaction attributes, such as, data structures to store messages (`inbox`, `outbox`), sensors and effectors (lines 43-59). Additionally, it also calls the abstract method `initSensorsAndEffectors()` (line 42). This method initializes sensors and effectors used by a specific type of agent or role. It needs to be implemented by `Interaction` subaspects (step 5).

```
40.    }
41.
42.    public abstract void initSensorsAndEffectors(Agent agent);

43.    public void init(Agent agent) {
44.          agent.inBox = new Vector();
45.          agent.outBox = new Vector();
46.          agent.sensors = new Vector();
47.          agent.effectors = new Hashtable();
48.
49.          // Init sensors and effectors
50.          initSensorsAndEffectors(agent);
51.
52.          // Set the agent reference in every sensor
53.          Enumeration sensorsEnum = (agent.sensors).elements();
54.          Sensor sensor;
55.          while (sensorsEnum.hasMoreElements()) {
56.                sensor = (Sensor) sensorsEnum.nextElement();
57.                sensor.setAgent(agent);
58.          }
59.    }
60.
61.    protected abstract pointcut agentInstantiation(Agent agent);

62.    after(Agent agent) : agentInstantiation(agent) {
63.          init(agent);
64.    }  ❑
```

Step 3: How to define the crosscutting interface IncomingMessage?

The crosscutting interface `IncomingMessage` defines how to intercept sensors and domain-specific classes to receive messages or sense external events. It defines two poincuts in the Interaction aspect: (i) the poincut `incomingMsgSensor()` (line 65-67), which intercepts every subclass of `Sensor`; and (ii) the abstract pointcut `incomingMsgDomainSpecific()` (line 77-78), which intercepts domain-specific classes. The latter must be implemented by `Interaction` subaspects (step 5).

Both pointcuts are associated with after advices. These advices call the `receiveMsg()` method in a specific agent instance (lines 69-75).

```
65.   pointcut incomingMsgSensor(Sensor sensor, Message msg):
66.         (this(sensor) && args(msg) &&
67.          execution(void Sensor.receiveUnmarshalledMsg(Message)));
68.
69.   after(Sensor sensor, Message msg):
70.               incomingMsgSensor(sensor, msg){
71.         Agent agent = sensor.getAgent();
72.         if (msg != null) {
73.               agent.receiveMsg(msg);
74.         }
75.   }
76.
77.   protected abstract pointcut incomingMsgDomainSpecific
78.                               (Agent agent, Object interceptedObject);  ❑
```

Step 4: How to define the crosscutting interface OutgoingMessage?

The crosscutting interface `OutgoingMessage` defines how to intercept agent knowledge classes interested to send internal messages to the environment. The `Interaction` aspect defines two abstract pointcuts `outgoingMsgFromRoles()` and `outgoingMsgFromPlans()` to accomplish this required behavior (lines 79 and 86, respectively). These pointcuts specify the join points in agent and plan classes, respectively, where a message should be sent by an agent effector. They must also be implemented by `Interaction` subaspects (step 5). Each one of the pointcuts has an associated after advice that calls the method `sendMessage()` of the `Interaction` aspect (lines 81-84 and 88-91, respectively).

```
79.   protected abstract pointcut outgoingMsgFromRoles(Agent agent);
80.
81.   after (Agent agent) returning (Message msg):
82.                          outgoingMsgFromRoles(agent) {
83.         sendMsg(msg, agent);
84.   }
85.
86.   protected abstract pointcut outgoingMsgFromPlans(Plan plan);
87.
88.   after (Plan plan) returning (Message msg):
89.                          outgoingMsgFromPlans(plan) {
90.         Agent agent = plan.getAgent();
91.         sendMsg(msg, agent);
92.   } ❑
```

Step 5: How to define a concrete Interaction Aspect?

`Interaction` subaspects define concrete implementations of the `Interaction` aspect. We can specify a different `Interaction` subaspect for each of the agent types or agent roles defined in our MAS. The subaspects must implement the abstract pointcuts and methods of the Interaction aspect, as follows:

- `agentInstantiation()` pointcut – define a join point to initialize interaction capabilities of the agent or role;
- `outgoingMessageFromRoles()` and `outgoingMessageFromPlans()` pointcuts – define, respectively, join points to send messages from agent roles and from agent plans;
- `initSensorsAndEffectors()` method – creates and initializes the sensors and effectors used by a specific agent type or role;
- `sendMsgEffector()` method – defines the logic to decide which agent effector will be used to send a specific message.
- In the EC system, we defined the subaspects `ChairInteraction` and `ReviewerInteraction` to specify different interaction capabilities for the agent roles (chair and reviewer). We present the `ReviewerInteraction` source code below.

```
1. public aspect ReviewerInteraction extends Interaction  {
2.    public pointcut agentInstantiation(Agent agent):
3.                this(agent) &&
4.                initialization(ResearcherUserAgent+.new(..));
5.
6.    protected pointcut outgoingMsgFromRoles(Agent agent):
7.                      target(agent) &&
8.                      execution(Message Agent+.prepareMessage(..));
9.
10.   protected pointcut outgoingMsgFromPlans(Plan plan) :
11.       this(plan) &&
12.       execution(Message ProposalReceptionPlan.prepareMessage(..));
13.
14.   public void sendMsgEffector(Message msg, Agent agent) {
15.         Hashtable effectors = agent.getEffectors();
16.         Effector agentEffector =
17.                     (Effector) effectors.get("Reviewer");
18.               agentEffector.sendMsg(msg);
19.   }
20.
21.   public void initSensorsAndEffectors(Agent agent) {
22.         Effector agentEffector = new BlackboardEffector();
23.
24.         Hashtable effectors = agent.getEffectors();
25.         effectors.put("Reviewer", agentEffector);
26.   }
27. } ❑
```

Acknowledgments We would also like to thank Daniela Brauner for helping us to produce Figure 4. This work has been partially supported by European Commission grant IST-2-004349: European Network of Excellence on Aspect-Oriented Software Development (AOSD-Europe), 2004-2008. This work has also been partially supported by CNPq under grant No. 141457/2000-7 for Alessandro Garcia, grant No. 140252/2003-7 for Uirá Kulesza, 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] AspectJ Team. The AspectJ Programming Guide. Mar 2003, eclipse.org/aspectj

[2] AspectWerkz Website. Simple, Dynamic, Lightweight and Powerful AOP for Java. http://aspectwerkz.codehaus.org/

[3] Chavez, C. A Model-Driven Approach to Aspect-Oriented Design. PhD Thesis, Computer Science Department, PUC-Rio, April 2004, Rio de Janeiro, Brazil.

[4] Costa, A. An Aspect-Oriented Software Architecture for Traffic Simulators. Master's Dissertation, University of Sao Paulo, October 2003. (In Portuguese)

[5] Fowler, M. Dealing with Roles. Proceedings of the 4th Annual Conference on the Pattern Languages of Programs, Monticello, Illinois, USA, September 2-5, 1997.

[6] Gamma, E. et al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA, 1995.

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

[8] Garcia, A., Chavez, C., Kulesza, U., Lucena, C. The Role Aspect Pattern. 10th European Conference on Pattern Languages of Programs (EuroPLoP2005), July 2005, Isree, Germany. (submitted)
[9] 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.
[10] Garcia, A., Kulesza, U., Sant'Anna, C., Lucena, C. The Mobility Aspect Pattern. Proceedings of the 4th Latin American Conference on Pattern Languages of Programming (SugarLoafPLoP'04), August 2004, Fortaleza, Brazil.
[11] Garcia, A., Lucena, C., Cowan, D. Agents in Object-Oriented Software Engineering. Software: Practice & Experience, Elsevier, Vol. 34, Issue 5, April 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] Garcia, A., Sant'Anna, C., Figueiredo, E., Kulesza, U., Lucena, C., von Staa, A. Modularizing Design Patterns with Aspects: A Quantitative Study. Proc. of the 4th Intl. Conference on Aspect-Oriented Software Development (AOSD'05), Chicago, USA, March 2005.
[14] Hanenberg, S., Unland, R., Schmidmeier, A. AspectJ Idioms for Aspect-Oriented Software Construction. Proc. of the EuroPlop'03, Irsee, Germany, June 2003.
[15] Hannemann, J., Kiczales, G. Design Pattern Implementation in Java and AspectJ. Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02), November 2002, pp. 161-173.
[16] 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.
[17] Kiczales, G. et al. Aspect-Oriented Programming. Proc. of the European Conference on OO Programming - ECOOP'97, LNCS 1241, Springer, Finland, June 1997.
[18] Kiczales, G. et al. An Overview of AspectJ. Proceedings of the European Conference on Object-Oriented Programming (ECOOP'01), Budapest, Hungary, 2001.
[19] Masuhara, H., Kiczales, G. Modeling Crosscutting in Aspect-Oriented Mechanisms. In Proc. of ECOOP2003, LNCS 2743, pp.2-28, Darmstadt, Germany, 2003.
[20] 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.

Appendix A – Aspect Terminology

This appendix contains a brief overview of the terminology associated with aspect-oriented software development. We have used the terminology described by Kiczales et al [1, 18] and adopted by many aspect-oriented programming languages, such as AspectJ [1] and AspectWerkz [2]. We present below the main terms that are usually considered as a conceptual framework for aspect-orientated design and programming [3, 14, 17, 18, 29].

Aspects. Aspects are modular units that aim to support improved separation of crosscutting concerns. An aspect can affect, or crosscut, one or more classes and/or objects in different ways. An aspect can change the static structure (static crosscutting) or the dynamics (dynamic crosscutting) of classes and objects. An aspect is composed of internal attributes and methods, pointcuts, advices, and inter-type declarations.

Join Points and Pointcuts. Join points are the elements that specify how classes and aspects are related. Join points are well-defined points in the dynamic execution of a

system. Examples of join points are method calls, method executions, exception throwing and field sets and reads. Pointcuts have name and are collections of join points.

Advices. Advice is a special method-like construct attached to pointcuts. Advices are dynamic crosscutting features since they affect the dynamic behavior of classes or objects. There are different kinds of advices: (i) before advices - run whenever a join point is reached and before the actual computation proceeds; (ii) after advices - run after the computation “under the join point” finishes; (iii) around advices run whenever a join point is reached, and has explicit control whether the computation under the join point is allowed to run at all.

Inter-Type Declarations. Inter-type declarations either specify new members (attributes or methods) to the classes to which the aspect is attached, or change the inheritance relationship between classes. Inter-type declarations are static crosscutting features since they affect the static structure of components.

Weaving. Aspects are composed with classes by a process called weaving. Weaver is the mechanism responsible for composing the classes and aspects. Weaving can be performed either as a pre-processing step at compile-time or as a dynamic step at runtime.