# 5
# Suporte de Software

Avanços na tecnologia de agentes dependem do desenvolvimento de modelos, mecanismos e ferramentas para a construção de sistemas multi-agentes de alta qualidade. O projeto e implementação desses sistemas é caro e bastante sujeito a falhas. Frameworks lidam com esta complexidade através do fatoramento de partes do software que já foram testadas e usadas em várias implementações de maneira que reduzem o custo de se produzir novos software e também aumentar a qualidade. Desta forma, um framework pode ser definido como sendo uma aplicação semi-completa e reutilizável que pode ser especializada para produzir aplicações específicas [64].

Nesta seção, apresenta-se um framework que fornece suporte para o desenvolvimento de sistemas multi-agentes considerando o modelo conceitual de leis apresentado no Capítulo 4.

Antes de apresentar o framework em detalhes, é importante definir quais são os tipos de usuários do framework e qual o suporte fornecido pelo framework para cada um deles. Sendo assim, o framework pode ser utilizado por três tipos diferentes de usuários:

– "Desenvolvedor das leis" representa o desenvolvedor responsável por especificar as leis do sistema multi-agente. Estes desenvolvedores precisam conhecer muito bem a aplicação que está sendo desenvolvida, pois as leis deverão ser efetivas na aplicação. Além disso, também é preciso conhecer o modelo conceitual de leis e entender o relacionamento entre seus elementos, assim como a linguagem XMLaw que permite a especificação das leis.
– "Desenvolvedor dos agentes" representa o desenvolvedor responsável por desenvolver os agentes que compõem o sistema. Estes desenvolvedores são especialistas na aplicação em si, embora também precisem ter conhecimento das leis que foram especificadas, portanto o papel deles é construir os agentes em conformidade com as leis previamente estabelecidas.
– "Desenvolvedor da infra-estrutura" lida com o software que dá suporte a interpretação, monitoramento e cumprimento das leis. Este desenvolvedor

deve utilizar um framework de leis, como o proposto neste trabalho, ou mesmo desenvolver outro mecanismo caso os existentes não sejam adequados.

O framework provê suporte para cada um destes desenvolvedores de diferentes maneiras. Primeiro, é fornecida uma linguagem declarativa chamada XMLaw para uso dos desenvolvedores de leis. Essa linguagem permite a especificação e manutenção das interações de um sistema multi-agente através dos conceitos de leis. As especificações realizadas utilizando o XMLaw são interpretadas e seu cumprimento é garantido através de uma infra-estrutura de software que media a interação entre os agentes. Esta infra-estrutura é extensível através de vários *hotspots* (pontos de flexibilização de um framework) que podem ser implementados por desenvolvedores de infra-estrutura. E finalmente aos desenvolvedores de agentes, o framework fornece um conjunto de classes e interfaces em Java que permite a construção de agentes integrados com a infra-estrutura de leis.

As funcionalidades do framework são providas através da implementação de vários módulos que estão ilustrados na Figura 5.1. Esta figura também mostra quais módulos estão acessíveis a quais desenvolvedores, além de destacar quais deles podem ser estendidos (*hotspots*). É importante ressaltar que os módulos estão distribuídos em duas partes: o conjunto de módulos disponível para o desenvolvedor de agentes e o conjunto de módulos disponível para os outros dois papéis. Na próxima seção, é apresentada a arquitetura utilizada para que a interação dos agentes seja mediada pelo framework. E nas próximas seções, são apresentados os detalhes de cada módulo, desde o seu funcionamento a instruções de como implementar os *hotspots*.



## 5.1 Modelo de Interação

Uma das informações acessíveis sobre os agentes é o seu comportamento observável através da troca de mensagens. O mecanismo de leis deve então interceptar estas mensagens e garantir o cumprimento das leis. Desta forma, o mecanismo age como um mediador entre os agentes, onde todas as mensagens passam primeiro pelo mecanismo antes de chegar aos agentes destinatários da mensagem. A idéia de usar mediadores na comunicação de agentes foi publicada anteriormente com algumas diferenças em um grande número de publicações tais como controllers [65], governors [9] e inter agents [38].

O mecanismo de leis possui um ciclo de vida constituído por três macro-atividades (Figura 5.2): (i) interceptação; (ii) aplicação das leis; (iii) redirecio-

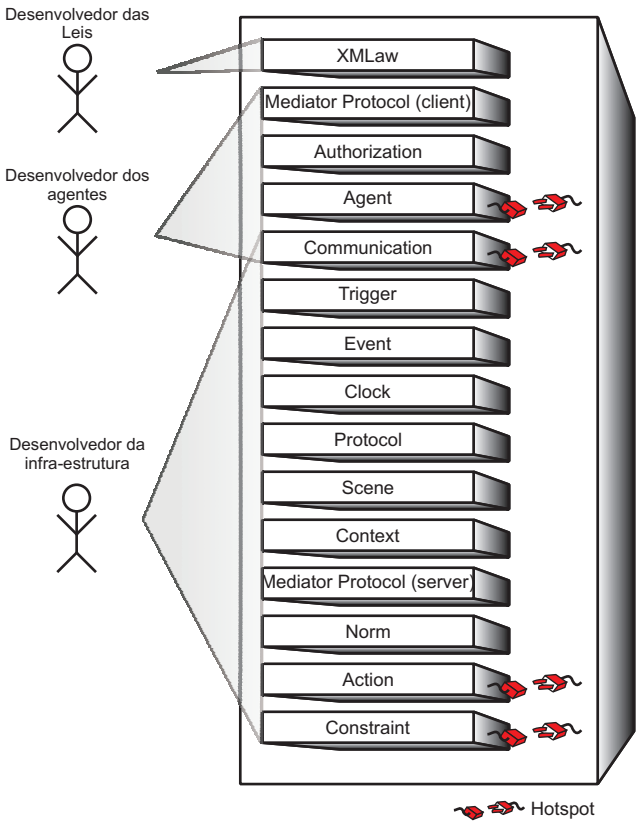


Figura 5.1: Framework: Módulos e Desenvolvedores

namento. Na primeira atividade, o mecanismo intercepta as mensagens trocadas entre os agentes. Então, na segunda fase, o mecanismo, de acordo com as especificações das leis, verifica se a mensagem é permitida assim como quais as conseqüências da mensagem no sistema de leis, por exemplo, a mensagem pode disparar uma transição e mudar o estado de um protocolo; ela pode ativar uma norma, ação, dentre outros. Por último, se as leis permitirem, a mensagem é redirecionada aos destinatários reais. Desta forma, através dessas três macro-atividades o comportamento observável dos agentes pode ser monitorado e controlado.

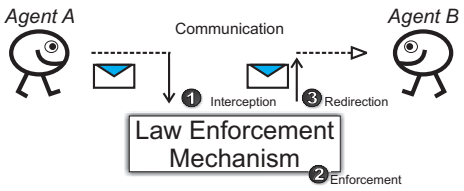


Figura 5.2: Modelo de Interação para a Aplicação das Leis

O framework apresentado neste trabalho segue exatamente a arquitetura apresentada nesta seção. Desta forma, uma vez que a sua arquitetura foi apresentada e, portanto, uma visão geral do seu funcionamento já é conhecida, nas próximas seções detalham-se os principais módulos que compõem o framework.

## 5.2 Módulo de Comunicação

O módulo de comunicação é composto por dois sub-módulos: a camada de comunicação e o módulo de mensagens. O primeiro é relacionado com questões de heterogeneidade da comunicação entre agentes e o segundo descreve qual o tipo de mensagem que agentes devem utilizar para se comunicar.

### 5.2.1 Camada de Comunicação

Agentes podem utilizar diferentes formas de comunicação. Eles podem utilizar uma infra-estrutura baseada nos padrões SOAP [66], podem implementar infra-estruturas proprietárias utilizando *sockets* [67], ou mesmo padrões de comunicação entre agentes tais como os definidos pela FIPA [68]. Cada aplicação possui diferentes requisitos de performance, flexibilidade, interoperabilidade, entre outros. Uma camada de comunicação deve se adequar a esses requisitos. Desta forma, a camada de comunicação proposta nesta seção fornece uma interface que permite que as aplicações mudem a implementação de determinadas partes para que os requisitos específicos de cada aplicação possam ser atendidos.

A interface *ICommunication* (Figura 5.3) define métodos para o envio e recebimento de mensagens. Por se tratar apenas de uma interface [1], as implementações deste método não são fornecidas por *ICommunication* e, portanto, precisam ser providas por implementações da interface para que as funcionalidades sejam fornecidas. O método de envio de mensagem é o método *send(Message msg)*. O comportamento esperado deste método é o envio de uma mensagem ao destinatário especificado na mensagem. Para o recebimento de mensagens, três métodos foram definidos, sendo dois métodos "bloqueantes" e um método "não bloqueante". O método *waitForMessage():Message* bloqueia a execução do programa que invocou este método até o recebimento de alguma mensagem. Quando uma mensagem for recebida, a chamada é desbloqueada e a mensagem é retornada. Uma versão um pouco modificada deste método é o método *waitForMessage(long milliseconds)*. Este método funciona de forma análoga ao *waitForMessage()* exceto que ele bloqueia o programa que o chamou até que uma mensagem seja recebida ou que o tempo especificado no parâmetro tenha se esgotado. Desta forma, este método retorna a mensagem recebida caso ela tenha chegado ou *null* caso nenhuma mensagem tenha chegado e o tempo te-

[1] Interfaces definem tipos de dados abstratos. Desta forma, tipos de dados específicos precisam implementar as operações definidas na interface.

nha expirado. Os métodos *wait* são bloqueantes uma vez que o programa que os invoca têm a sua execução bloqueada. Porém, esta interface define mais um método denominado *nextMessage():Message*. Este método, ao contrário dos outros dois, não bloqueia a execução, mas retorna a próxima mensagem que ainda não foi lida, ou *null* caso nenhuma mensagem tenha chegado até o momento da invocação do método. Este comportamento do método *nextMessage* sugere que implementadores da interface *ICommunication* implementem uma estrutura de fila para que todas as mensagens recebidas sejam armazenadas e quando o método *nextMessage* for invocado, a primeira mensagem da fila seja retornada.

Na Figura 5.3 é mostrado o diagrama de classes da camada de comunicação. Devido à definição de seus métodos, esta interface possui um relacionamento de dependência com a classe *Message* (detalhada na Seção 5.2.2). Além disso, nesta figura também mostra-se como o framework JADE [44] foi utilizado para prover as funcionalidades definidas pela interface da camada de comunicação. O JADE implementa um mecanismo de comunicação compatível com os padrões definidos pela FIPA. Este mecanismo é acessível de forma relativamente transparente através da classe *jade.core.Agent*, provida pela implementação JADE. Então, a classe *JadeCommunication* reutiliza a implementação desse mecanismo de comunicação JADE através da delegação dos métodos da interface *ICommunication* para uma instância da classe *jade.core.Agent*. Na verdade, implementando a interface da camada de comunicação e delegando boa parte da funcionalidade para o JADE, a classe *JadeCommunication* implementa o padrão de projeto *wrapper* [69]. Na Tabela 5.1, apresenta-se um fragmento de código da classe *JadeCommunication* e como esse padrão é implementado. A classe *JadeCommunication* possui o atributo *jadeAgent* do tipo *jade.core.Agent*. Esse atributo é inicializado no momento da instanciação da classe *JadeCommunication*. No método *send*, primeiro o objeto da classe *Message* é transformado para o formato de mensagem que JADE suporta, *jade.lang.acl.ACLMessage*, e depois a chamada é delegada para o atributo *jadeAgent*.

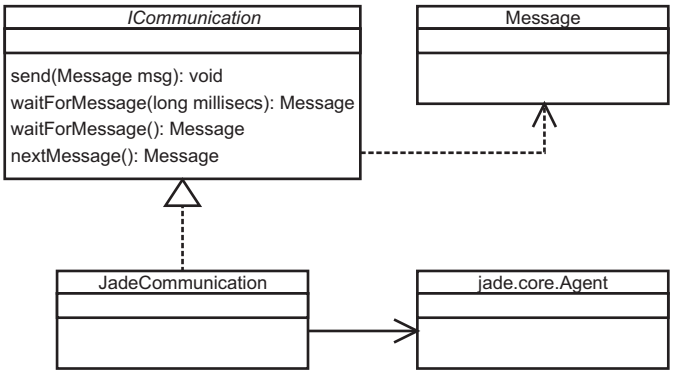


Figura 5.3: Camada de Comunicação

```
public class JadeCommunication implements ICommunication{
    ...
    protected Agent jadeAgent;

    ...

    public JadeCommunication(String name) {
        ...
        jadeAgent = new Agent();
        ...
    }

    ...

    public void send(Message msg) {
        ACLMessage message = this.messageWrapper.transform(msg);
        jadeAgent.send(message);
    }
    ...
}
```

Tabela 5.1: Fragmento de Código da Camada de Comunicação Utilizando JADE



### 5.2.2 Módulo de Mensagens

Mensagens são uma parte crucial da abordagem de leis. A abordagem de leis é concebida para o controle do comportamento dos agentes de um ponto de vista de interação. Uma vez que agentes interagem através da troca de mensagens[2], o controle do comportamento é realizado através da análise das mensagens trocadas no contexto da conversação. A classe *Message* representa uma mensagem trocada entre agentes.

[2]Apesar de existirem outras formas de comunicação entre agentes que não envolvem a troca direta de mensagens, este trabalho considera apenas a comunicação entre agentes através do envio e recebimento de mensagens.

A classe *Message* contém campos pré-definidos que são utilizados para uma comunicação efetiva entre os agentes. Estes campos são especificados em um formato chave-valor e, embora o número de campos usados em uma mensagem possa variar, existem cinco campos que precisam estar presentes em todas as mensagens: *performative*, *sender identification*, *sender role*, *receiver identification* e *receiver role*. Estes campos refletem a definição de mensagem especificada no modelo conceitual da Seção 4.2. Outros campos pré-especificados incluem *conversation identification*, *content* e *scene authorization*. O primeiro identifica contextos de conversação. Agentes podem manter o controle de todas as mensagens trocadas em uma conversação utilizando o campo *conversation identification*. O campo *content* contém a informação que um agente está enviando na mensagem. O campo *scene authorization* representa a autorização que os agentes precisam possuir para interagir com os outros agentes de uma cena. Esta autorização é emitida pelo agente mediador e pode ser requisitada utilizando-se o protocolo *Mediator Protocol* detalhado na Seção 5.4.2.

XMLaw define um elemento denominado *message-pattern* que permite a especificação de quais padrões de mensagens agentes podem enviar. O framework faz o reconhecimento deste padrão utilizando tanto o método *format()* da classe *Message* quanto o método *match()* da classe *MessagePattern*. O método *format()* retorna a mensagem de acordo com o formato de texto mostrado na Tabela 5.2.

```
message(performative,
        sender(sender identification, sender role),
        receiver(receiver identification, receiver role),
        content(content) ).
```

Tabela 5.2: Valor de Retorno do Método *format()*

A classe *MessagePattern* gerencia a criação e destruição de implementações da interface *IMessagePatternMatcher*. Esta interface define apenas um método, ilustrado na Tabela 5.3.

```
match(String formattedMessage, String template): Hashtable
```

Tabela 5.3: Método da Interface *IMessagePatternMatcher*

Este método contém dois parâmetros, o *formattedMessage* que é resultante da invocação do método *format()*, e o template pelo qual a implementação de *IMessagePatternMatcher* deverá verificar se a *formattedMessage* está em conformidade. Se a mensagem está de acordo com o padrão e o padrão permite o uso de variáveis, então este método retorna uma *Hashtable* contendo pares variável-valor para a mensagem em questão.

Uma alternativa para implementar esta abordagem é utilizar um padrão de mensagem baseado na sintaxe de *Prolog* [31]. Por exemplo, em *Prolog* variáveis

são representadas por uma cadeia de caracteres iniciadas com a primeira letra em maiúsculo. Então, um possível padrão poderia ser *price(Value)*, onde *Value* é a variável do padrão. Desta forma, caso o método *match* fosse invocado como em: *match("price($34,45)", "price(Value)")*, o retorno esperado seria uma Hashtable contendo o par *["Value","$34,45"*].

Em suma, os padrões das mensagens permitidas são especificados usando-se o XMLaw. Estes padrões precisam ser "entendidos" por uma implementação concreta da interface *IMessagePatternMatcher*, e o framework é responsável por gerenciar as invocações dos métodos no momento certo. As classes que compõem o módulo de mensagens e os seus relacionamentos estão representadas na Figura 5.4.

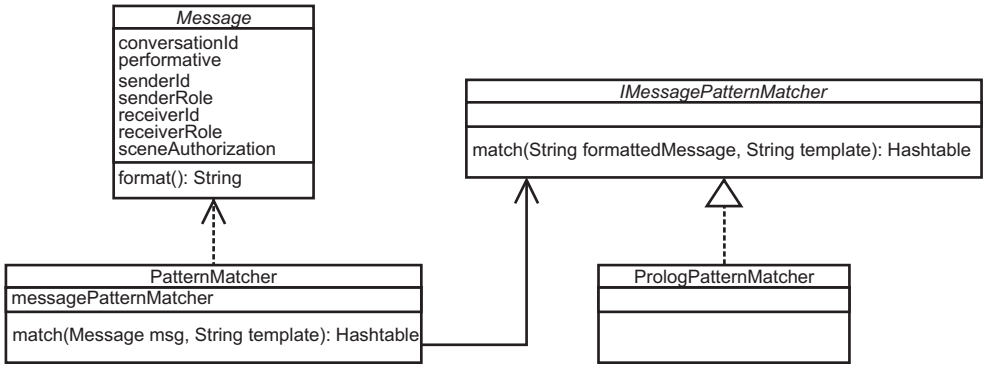


Figura 5.4: Módulo de Mensagens

## 5.3 Agente Mediador

A maior parte do framework é implementada no agente mediador que é responsável pelo monitoramento e aplicação das leis na interação dos agentes dos sistemas. Desta forma, esse agente contém a maioria dos módulos do framework.

O agente mediador realiza um certo número de atividades ilustradas na Figura 5.5. Primeiro o mediador espera pelo recebimento de mensagens. Uma vez que uma mensagem é recebida, ele checa se a mensagem pertence ao protocolo do mediador (explicado na Seção 5.4.2). Se ela é uma mensagem do protocolo do mediador, a resposta é enviada ao agente que enviou a mensagem de acordo com as definições do protocolo. Se não for uma mensagem do protocolo do mediador e caso seja uma mensagem que pertença a uma conversação entre agentes, o mediador inicia o processo de interpretação e aplicação das leis. Caso as leis permitam que a mensagem seja enviada, então a mensagem é redirecionada para o agente destinatário. Esta seqüência de atividades é repetida durante toda a "vida" do agente mediador.

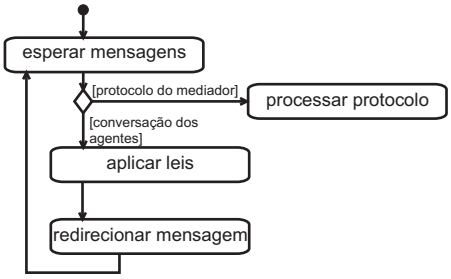


Figura 5.5: Atividades do mediador

Para executar estas atividades o mediador tem que ser capaz de realizar tarefas como interceptar mensagens e interpretar as leis, porém uma vez que a abordagem de leis continua em evolução e para acomodar as necessidades específicas de cada aplicação, este agente precisa ser flexível e expansível para acomodar mudanças futuras. Desta forma, os módulos do framework que foram implementados no mediador foram projetados visando alcançar estes objetivos. Os principais módulos estão descritos das próximas seções.



### 5.3.1 Eventos

A comunicação entre os módulos do agente mediador é principalmente baseada em notificações de eventos. Esta abordagem leva a um baixo nível de acoplamento entre os módulos e também a sistemas mais flexíveis [70]. Em sistemas baseados em eventos, existe um módulo chamado Gerenciador de Eventos (*Event Manager*) que fornece uma interface contendo todas as operações necessárias para habilitar a comunicação baseada em eventos. Esta interface contém as operações abstratas *fireEvent*, *subscribe* e *unsubscribe*. A operação *fireEvent* é invocada por clientes denominados produtores e seu comportamento é avisar a todos os interessados a ocorrência de um determinado evento no sistema. Produtores são os clientes que geram os eventos no sistema e consumidores são os clientes que registram seus interesses em certos tipos de eventos através da operação *subscribe*. Consumidores também podem eliminar o interesse em certos tipos de eventos através de chamadas a operação *unsubscribe*.

O framework implementa um sistema baseado em eventos através da colaboração de várias classes e interfaces. A interface *IEvent* define o comportamento básico de todos os eventos (Figura 5.7). Esta interface estende a interface *IIdentifiable*, o que significa que todos os eventos possuem identificações. Além disso, *IEvent* define três métodos:

– *getType():int* - retorna o tipo do evento. Embora todos os eventos possam ser identificados pelas suas classes, o uso de números inteiros para identifi-

car os eventos permite a implementação eficiente da notificação de eventos através de operações de aritmética binária, além de permitir módulos mais desacoplados uma vez que os módulos precisam saber apenas o tipo do evento ao invés de conhecer suas classes. A class *Mask* possui constantes que identificam todos os tipos de evento contidos no framework. Na Figura 5.6 ilustra-se os eventos existentes.

| Mensagem | Descrição |
|---|---|
| message_arrival | Mensagem chega ao mediador |
| transition_activation | Transição é disparada |
| clock_tick | Tempo especificado expirou |
| deactivation_event | Evento genérico para desativações |
| norm_activation | Norma foi ativada |
| clock_activation | Relógio foi ativado |
| final_state_reached | Um estado final de um protocolo foi alcançado |
| time_to_live_elapsed | Tempo de vida da cena expirou |
| successful_scene_completion | Cena terminou com sucesso |
| failure_scene_completion | Cena terminou com falha |
| action_activation | Ação foi executada |

Figura 5.6: Tipos de Eventos



– *getInfo():InfoCarrier* - Eventos transportam informações que podem ser importantes para os consumidores dos eventos. Esta informação é encapsulada em objetos da classe *InfoCarrier*. Este objeto é uma tabela onde a informação pode ser armazenada e obtida. O método *getInfo* da interface *IEvent* retorna o objeto *InfoCarrier* contido em um evento.

– *getEventGenerator(): IIdentifiable* - Conceitualmente, eventos são gerados por produtores de eventos. No framework estes produtores podem ser qualquer objeto que implemente a interface *IIdentifiable*. Desta forma, este método retorna o produtor deste de um evento.

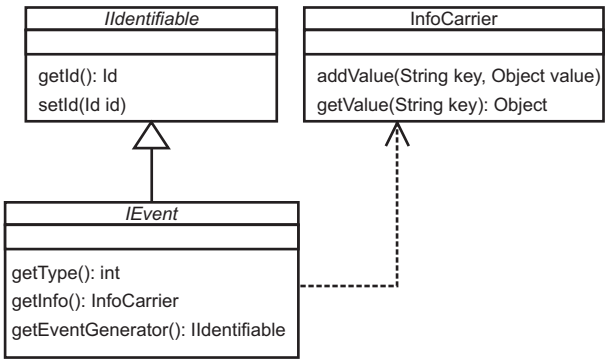


Figura 5.7: IEvent Interface

As interfaces *ISubject* e *IObserver*, ilustradas na Figura 5.9, definem respectivamente o comportamento de gerenciadores de eventos e de consumidores

de eventos. Consumidores registram seus interesses em certos tipos de eventos utilizando o método *attachObserver*. Este método possui dois parâmetros, o primeiro é o consumidor do evento e o segundo é uma máscara relacionada aos tipos de eventos que o consumidor está interessado em ser avisado. A máscara pode ser formada usando tantos eventos quantos forem necessários. Esta máscara é construída utilizando o operador binário '|'(OR) em conjunto com constantes definidas na classe *Mask*. Então, por exemplo, se um consumidor está interessado em eventos de ativação de clocks e normas, a chamada ao método *attachObserver* deveria ser:

*attachObserver(this, Masks.CLOCK_ACTIVATION | Masks.NORM_ACTIVATION );*

No framework, existem várias implementações para a interface *IObserver* tais como as classes *Transition* e *Clock*, que são detalhadas em seções posteriores. Entretanto, o framework fornece apenas uma implementação para a interface *ISubject*. Esta implementação é fornecida pela classe *Subject*. Esta classe tem uma implementação peculiar para o método *fireEvent*, uma vez que as implementações tradicionais para este tipo de método executam um loop notificando todos os consumidores sobre a ocorrência de um evento, tal como na Tabela 5.4.

```
... for (int i = 0; i < observers.size(); i++) {
    IObserver anObserver = (IObserver) observer.get(i);
    anObserver.update(event);
} ...
```

Tabela 5.4: Implementação Usual do Método fireEvent

Entretanto, a implementação apresentada na Tabela 5.4 possui alguns problemas. Do ponto de vista do programa que invoca o método *fireEvent*, ele pode ter que esperar muito pela execução deste método se existirem tarefas que requeiram muito processamento na implementação dos métodos *update* em algum consumidor. Além disso, comportamentos inesperados também podem ocorrer: implementações do método update nos consumidores podem também gerar outros eventos. Isto significa que eventos gerados depois, podem chegar a observadores primeiro que eventos gerados anteriormente.

Por exemplo, na Figura 5.8 mostra-se um diagrama de seqüência no qual um produtor gera o evento *A*. Existem dois consumidores registrados no objeto *subject* que faz o papel de gerenciador de eventos. O primeiro consumidor é o *observer1* e ele está interessado em eventos *A*s. O segundo consumidor é o *observer2* que está interessado tanto em eventos *A*s e *B*s. Logo, dada a geração

do evento *A*, o subject primeiro notifica o *observer1* da ocorrência de *A*. A implementação do método *update* do *observer1* produz um evento *B*. O objeto subject recebe a geração deste novo evento e verifica que somente o *observer2* está interessado neste tipo de evento e, portanto, notifica o *observer2* sobre a ocorrência de *B*. No entanto, a implementação do método *update* do *observer2* requer que o evento *A* tenha ocorrido antes do evento *B* para que a operação *operation()* seja executada. Mas, uma vez que o *observer2* foi notificado primeiro da ocorrência de *B*, a operação *operation()* não é chamada, mesmo que de fato o evento *A* tenha ocorrido antes de *B*.

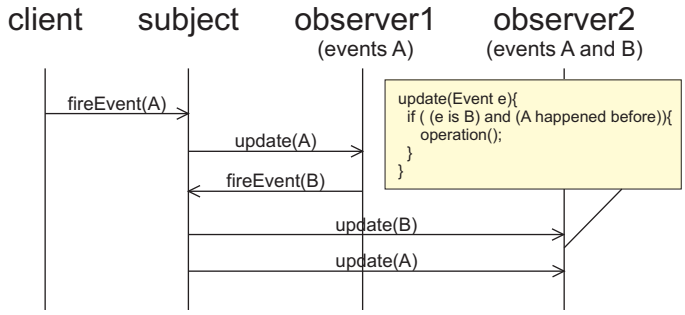


Figura 5.8: Problema com a Implementação Usual da Notificação de Eventos

Para evitar este problema, o *subject* (gerenciador de eventos) fornecido pelo framework utiliza uma estratégia diferente. Quando produtores geram eventos, o *subject* adiciona o evento a ser divulgado para os consumidores em uma fila. Este *subject* é implementado pela classe *Subject* conforme ilustrado na Figura 5.9. *Subject* estende a classe Java *Thread* e sobrescreve o método *run()*. Isto significa que a classe *Subject* executa em uma linha de execução diferente. O pseudo-código para a implementação do método *run()* é mostrado na Tabela 5.5. Este método contém um loop que verifica constantemente se existem eventos na fila. Os eventos são colocados na fila através do método *fireEvent*. Esta técnica garante que os consumidores irão ser notificados sobre a ocorrência dos eventos na ordem correta e os produtores de eventos não precisam esperar que os consumidores sejam notificados quando o método *fireEvent* é chamado.

```
while (true){
    while (events.size()>0){
        IEvent event = (IEvent)events.pop();
        Vector observers = get all observers for this event;
        for (int i = 0; i < observers.size(); i++) {
            IObserver anObserver = (IObserver) observers.get(i);
            anObserver.update(event);
        }
    }
    wait();
}
```

Tabela 5.5: Pseudo-código para o método run() do Subject

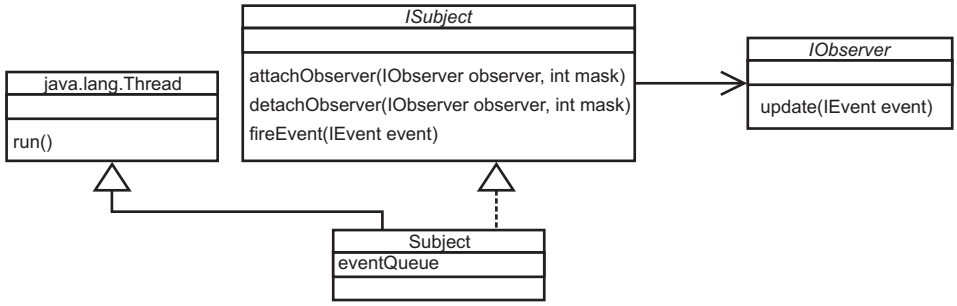


Figura 5.9: Observer e Subject

### 5.3.2 Trigger

*Triggers* são componentes do framework que podem ser ativados por eventos quando determinadas condições são satisfeitas e uma vez ativados eles executam alguma ação. Esta idéia é similar aos sistemas do tipo Evento-Condição-Ação (ECA) e é largamente utilizada em sistemas de banco de dados [71]. O framework contém um módulo para o uso de triggers que especifica um conjunto de classes e interfaces que são estendidas por outros módulos do framework para a criação de triggers.

Na verdade, como a comunicação dos módulos do framework é principalmente baseada em notificações de eventos, o módulo de trigger age como um consumidor dos eventos gerados e então ativa e desativa triggers. Muitos elementos do modelo conceitual de leis, tais como relógios e normas, são implementados como triggers. Desta forma, uma vez que o módulo de triggers está sendo utilizado e testado por muitos outros módulos, benefícios como reúso e aumento na confiabilidade do sistema são alcançados.

Um trigger é representado pela interface *ITrigger*. Esta interface contém métodos para adicionar condições que ativam ou desativam um trigger. Condições de ativação e desativação de triggers são, na verdade, a ocorrência de certos tipos de eventos gerados por algum produtor de eventos. A razão para incluir tanto o evento quanto quem produz o evento em uma condição é que diferentes produtores podem gerar o mesmo tipo de evento, assim, especificar tanto o produtor quanto o evento torna possível especificar que somente um produtor específico ativa o trigger.

Triggers também são capazes de gerar eventos indicando a sua ativação. O método *getActivationEvent* retorna o evento que é gerado quando o trigger é ativado. De acordo com o modelo ECA discutido anteriormente, quando um trigger é ativado uma ação deve ser executada. No framework, as ações de triggers são representadas pela classe abstrata *TriggerFiring*, e os triggers possuem o método *getTriggerFiring* que retorna uma instância da classe *TriggerFiring*. Então, por

exemplo, sabendo que um relógio é um trigger, uma vez que ele é ativado ele cria uma instância de uma subclasse de *TriggerFiring* que começa a contar o tempo (ação do trigger = contar o tempo).

Porém, as instâncias da classe *TriggerFiring* possuem um ciclo de vida próprio, podendo estar basicamente em um desses três estados: *criado*, *executando* e *parado*. O estado *criado* ocorre logo após a instanciação. As instâncias de *TriggerFiring* entram no estado *executando* quando o método *start()* é chamado, e somente deixam este estado e entram no estado *parado* quando o método *stop()* é chamado.

Existe um componente responsável por controlar quando chamar os métodos das instâncias de *ITrigger* tal como *getTriggerFiring*, além de controlar o ciclo de vida das instâncias de *TriggerFiring*. A classe do framework responsável por essas tarefas é o *TriggerManager*. Na Figura 5.10 mostra-se o diagrama de classes do módulo de triggers, no qual a classe *TriggerManager* também pode ser vista.

O *TriggerManager* mantém uma lista de todos os triggers. Esta classe implementa a interface *IObserver* e "escuta" por todos os eventos que acontecem no framework. Na implementação do seu método *update*, o *TriggerManager* controla a ativação e desativação de *triggers*. Na Tabela 5.6 ilustra-se como acontece a ativação de *triggers*. Basicamente, quando um evento ocorre, para cada trigger é verificado se o trigger é ativado pelo evento que ocorreu, ou seja, se o tipo de evento e o produtor do evento fazem parte das condições de ativação do trigger. Se o trigger for ativado, o *TriggerManager* requisita ao trigger para que retorne a ação que é consequência de sua ativação, uma instância da classe *TriggerFiring*. A ação retornada é, então, inicializada e um evento relacionado à ativação do trigger é agendado para ser disparado. A desativação de *triggers* ocorre de maneira bastante similar e é ilustrada na Tabela 5.7.

```
for (int i = 0; i < triggers.size(); i++) {
    ITrigger trigger = (ITrigger) triggers.elementAt(i);
    if (trigger.isActivatedBy(event)){
        TriggerFiring activeTrigger = trigger.getTriggerFiring(info);
        enabledElements.add(activeTrigger);
        activeTrigger.start();
        IEvent eventToThrow = trigger.getActivationEvent(info); // fires it
            latter
        eventsToThrowList.add(eventToThrow);
    }
}
```

Tabela 5.6: Ativação de Triggers

```
for (int i = 0; i < enabledElements.size(); i++) {
    TriggerFiring enabledEvent = (TriggerFiring) enabledElements.elementAt(i);
    if (enabledEvent.isDeactivatedBy(event)){
        this.enabledElements.remove(enabledEvent);
```

```
            enabledEvent.stop();
            IEvent eventToThrow = enabledEvent.getDeactivationEvent(info);
            eventsToThrowList.add(eventToThrow);
        }
    }
```

Tabela 5.7: Desativação de Triggers

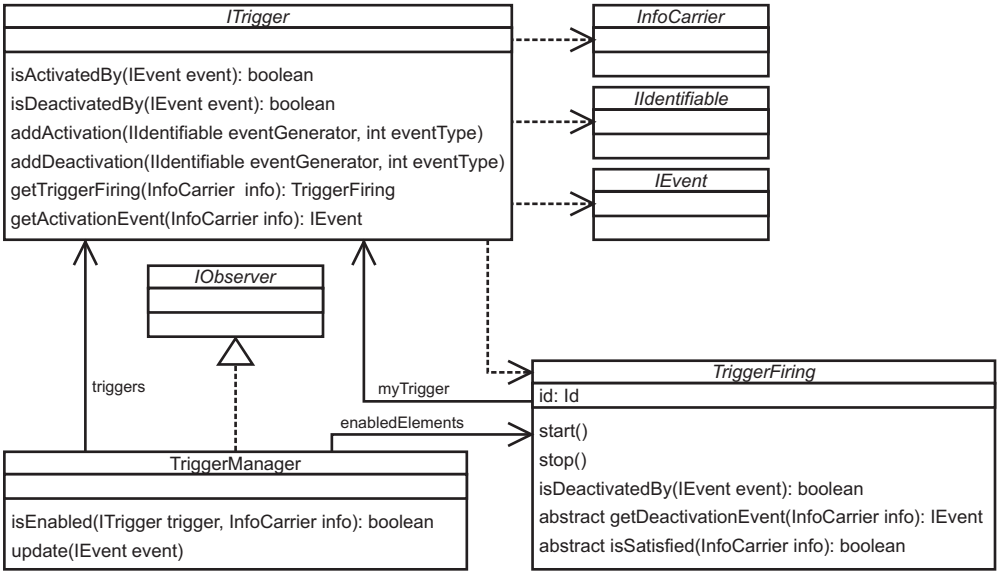


Figura 5.10: Módulo de Triggers

### 5.3.3 Contextos

Contextos são geralmente hierárquicos e limitam a visibilidade de informações e operações em um determinado sistema. A idéia deste tipo de contexto pode ser explicada fazendo-se uma analogia à estrutura dos sistemas de arquivos. Em um sistema de arquivo, cada diretório pode conter arquivos ou outros diretórios. Desta forma, cada diretório fornece um contexto para os arquivos e diretórios contidos nele.

A especificação de leis pode ser composta pela definição de leis para várias organizações, e cada lei de uma organização é composta por cenas. Essas composições definem contextos, onde os elementos da lei definidos no escopo de uma organização são visíveis para todas as cenas, mas os elementos definidos no contexto de uma cena são somente visíveis na cena em si.

No framework, contextos limitam a visibilidade de eventos e *triggers*. Cada instância de uma cena possui um contexto associado, que é filho do contexto da organização. Tudo que acontece em um cena é visível para a própria cena e para a organização da qual a cena pertence. Devido a esse esquema de visibilidade,

é possível que eventos que ocorrem dentro de uma cena afetem elementos definidos no contexto da organização. Por exemplo, o disparo de transição de um protocolo de uma cena pode ativar uma norma definida no contexto da organização.

A classe *Context* representa esses contextos e a sua estrutura é exibida na Figura 5.11. Como contextos limitam a visibilidade de eventos e triggers, a classe *Context* reutiliza a implementação das classes *Subject* e *TriggerManager*. Um contexto encapsula estas classes de forma que cenas, organizações e outros elementos precisem conhecer apenas a classe *Context* para gerar e receber eventos. A classe *Context* delega a implementação dos métodos *attachObserver*, *detachObserver* e *fireEvent* para a classe *Subject*, que age como gerenciador de eventos; e também delega a implementação dos métodos *isEnabled* e *addTrigger* para a classe *TriggerManager*.

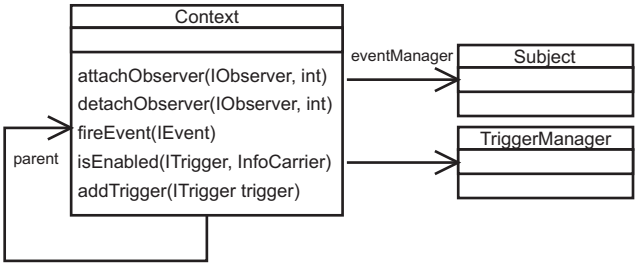


Figura 5.11: Classe *Context*

A classe *Context* possui um atributo denominado *parent* que referencia o contexto "pai" caso ele exista, ou o valor *null* caso o contexto seja um contexto de uma organização e, portanto, é um contexto raiz. Quando um evento é gerado em um objeto *Context*, o evento é também propagado para o contexto pai. Além disso, a verificação se um certo trigger está ativo ou não em um determinado contexto, também é feita no contexto "pai". Estas propagações são implementadas nos métodos *fireEvent* e *isEnabled*, respectivamente. A pseudo-implementação destes dois métodos está ilustrada na Tabela 5.8.

```
public boolean isEnabled(ITrigger trigger, InfoCarrier info) {
    if (!triggerManager.isEnabled(trigger,info)){
        if (parentContext!=null){
            return parentContext.isEnabled(trigger,info);
        }else{
            return false;
        }
    }
    return true;
}

public void fireEvent(IEvent event) {
    eventManager.fireEvent(event);
    if (parentContext!=null){
        parentContext.fireEvent(event);
```

```
        }
}
```

Tabela 5.8: Propagação de Eventos e Verificação de Triggers

### 5.3.4 Relógios

Relógios (Clocks) são implementados estendendo os módulos de triggers e de eventos. A classe *Clock* representa o elemento *Clock* do modelo conceitual. Relógios geram instâncias da classe *ActivatedClock* em resposta a chamadas ao método *getTriggerFiring*, que é herdado da interface *ITrigger*.

*ActivatedClock* implementa a interface Java *Runnable* e, portanto, executa em uma linha de execução diferente. Uma vez iniciado pelo *TriggerManager* (Figura 5.10), um *ActivatedClock* começa a contar o tempo e quando o tempo contado for igual ao valor do atributo *timeout*, gera-se uma instância da classe *ClockTick*, que é um evento. O diagrama de classes do módulo de relógios é ilustrado na Figura 5.12.

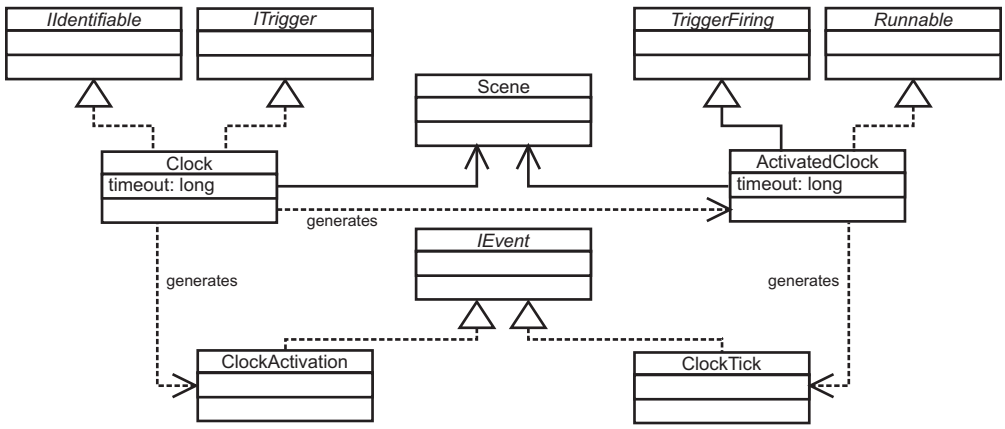


Figura 5.12: Módulo de Relógio

### 5.3.5 Normas

Normas (Norms) são representadas pela classe *Norm*. Como no módulo de relógio, normas são implementadas estendendo os módulos de triggers e de eventos. Normas são triggers "escutando" por eventos que as ativam e desativam. Uma vez ativadas, normas geram instâncias da classe *NormActivation*, e então uma instância da classe *ActivatedNorm* é criada e colocada em algum contexto (organização ou cena). De acordo com o modelo conceitual as normas podem ser

de três tipos: *Obrigação* (*Obligation*), *Permissão* (*Permission*) e *Proibição* (*Forbidden*). As classes que representam estes três tipos de normas estão ilustradas na Figura 5.13.

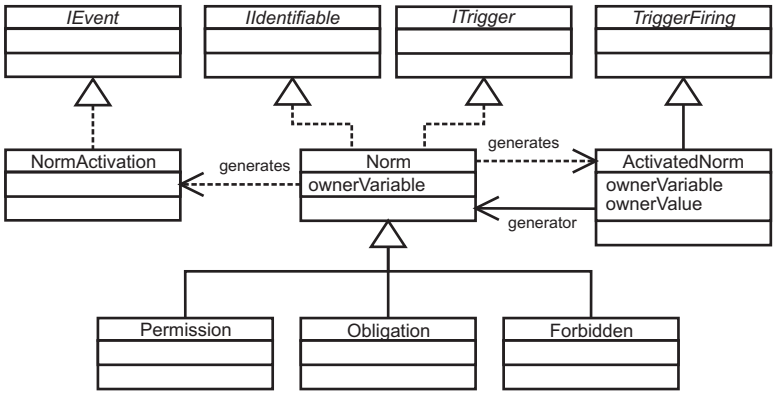


Figura 5.13: Normas

Um aspecto muito importante no funcionamento das normas é relacionado com os atributos *ownerVariable* e *ownerValue* das classes *Norm* e *ActivatedNorm* respectivamente. Normas, tais como permissões, são concedidas a um agente específico ou a um determinado papel de agente. Isto significa que normas permitem especificar que um agente *X* possui a permissão *P*, ou mesmo que todo agente desempenhando o papel *R* possui a obrigação *O*. A maneira de obter esta funcionalidade é implementada através dos atributos *ownerVariable* e *ownerValue*. O atributo *ownerVariable* especifica quem é o dono de uma certa norma e o valor deste atributo deve conter o nome de uma variável. Esta variável deve estar presente na informação carregada através do objeto *InfoCarrier* (Seção 5.3.1). Usualmente, o primeiro evento no sistema é a chegada de uma mensagem (*MessageArrival*). Este evento coloca no *InfoCarrier* informações sobre a mensagem que chegou. O protocolo de uma cena, recebe este evento e verifica se alguma transição será ativada. Cada transição, por sua vez, solicita a execução do casamento de padrão da mensagem recebida com o padrão de mensagem especificado para a transição. Então, é feita uma associação de todas as variáveis definidas no padrão da mensagem com seus respectivos valores provenientes da mensagem. Essas associações são colocadas em um objeto *InfoCarrier*. A partir deste momento, consumidores de eventos de ativação de transição têm acesso a essas associações, que serão propagadas para eventuais futuros eventos. Quando a norma é ativada, ele procura no objeto *InfoCarrier* o valor da associação atribuída a variável especificada em *ownerVariable*. O par *ownerVariable* e seu valor *ownerValue* é armazenado na instância da classe *ActivatedNorm*. Isto significa que a norma foi ativada para um agente específico ou para um papel de agente.

### 5.3.6
### Protocolo

No módulo de protocolo, implementa-se um autômato finito não determinístico [36] onde o protocolo de interação entre os agentes pode ser especificado. A classe *Protocol* (Figura 5.14) representa este autômato. A classe *Protocol* implementa a interface *IObserver* [3] e escuta apenas eventos de chegada de mensagens. Esta classe mantém uma referência para o estado inicial e para uma lista com todos os estados atuais do protocolo[4].

Estados são representados pela classe *State*. Cada estado possui uma lista de todas as transições que se originam naquele estado em direção a outros. Logo, quando o protocolo recebe a notificação da chegada de uma mensagem, ele executa o algoritmo mostrado na Tabela 5.9. O protocolo controla a lista dos estados atuais, mas a lógica de mudança de estados é delegada para os próprios estados. Os estados, por sua vez, executam o algoritmo mostrado na Tabela 5.10. Neste algoritmo, os estados gerenciam uma lista de próximos estados delegando a responsabilidade para as transições. De fato, é a classe *Transition* que "sabe" se o protocolo deve mudar ou não de estado. Transições executam as atividades mostradas na Figura 5.15. Primeiro, verifica-se se a mensagem que chegou está em conformidade com o padrão da mensagem especificada no XMLaw para aquela transição. Então verifica-se se todas as normas que deveriam estar ativas estão realmente ativas e o mesmo se faz para as normas que deveriam estar inativas. Por último, executam-se as restrições (constraints) e verifica-se se todas elas permitem que a transição seja disparada. Se todas as condições anteriores forem satisfeitas, então a transição é disparada e um evento de ativação de transição é gerado.

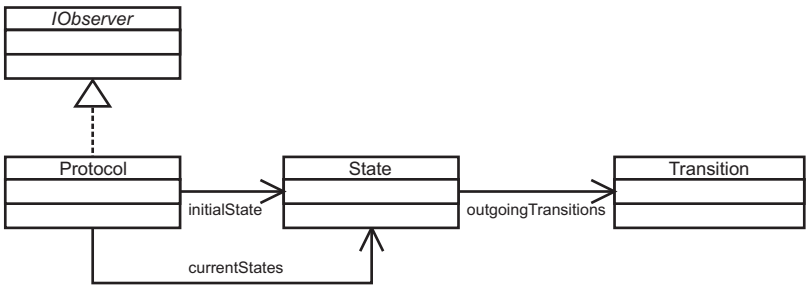


Figura 5.14: Módulo de Protocolo

[3] IObserver é descrito na Seção 5.3.1

[4] Como se trata de um autômato não determinístico, a lista de estados atuais representa justamente as opções de caminhos que o não determinismo impõe

```
List futureStates = new List(); for each current state{
    futureStates.add( state.step() );
}

if (futureStates.size == 0){
    sendMessage ( Message not allowed );
}else{
    currentStates <-- futureStates;
    redirectMessage;
}
```

Tabela 5.9: Pseudo-Código do Protocolo

```
List nextStates = new List(); for each outgoing transition{
    if transition fires{
        nextStates.add( transition.getDestinationState() )
    }
} return nextStates;
```

Tabela 5.10: Pseudo-Código do Estado

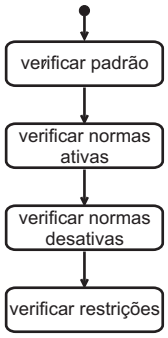


Figura 5.15: Atividades de uma Transição



### 5.3.7 Cena

Cenas são representadas pela classe *Scene*. Esta classe define o contexto para eventos, normas, relógios e outros elementos das leis. A classe *Scene* age como uma consumidora de eventos do seu próprio contexto e age também como um *ISubject*, delegando as requisições para o seu objeto *Context*. Desta forma, outras classes como *Norm* e *Clock* precisam conhecer apenas a classe *Scene* para gerar e receber eventos.

Na Figura 5.16 mostra-se o diagrama de classe para as classes relacionadas às cenas. Cenas possuem uma referência para o seu protocolo de interação (classe *Procotol*), para o contexto definido para a cena, para uma lista de todas as normas necessárias para a execução da cena e para a organização da qual a cena pertence.

O ciclo de vida de uma cena é implementado pelos métodos *initialization()*, *start()* e *finalization()*. Eles são chamados pelo framework quando a cena precisa ser criada, iniciada ou destruída.

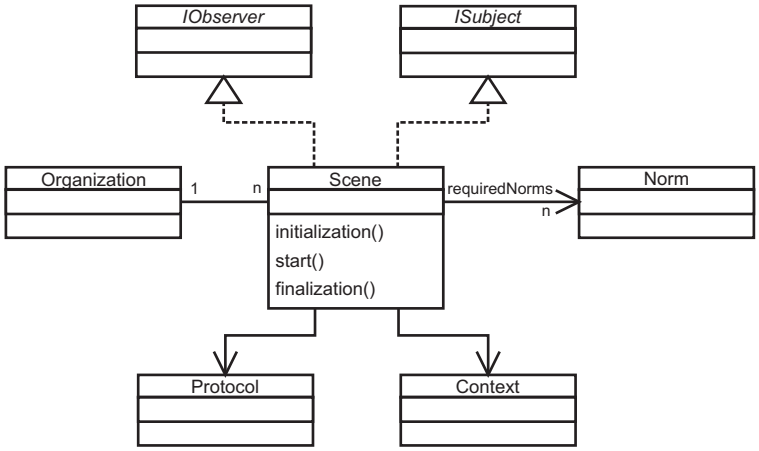


Figura 5.16: Módulo de Cenas



### 5.3.8 Restrições

O módulo que implementa as restrições (*Constraints*) é ilustrado no diagrama de classes da Figura 5.17. A interface *IConstraint* define somente uma operação: *constrain*. Esta operação é chamada por instâncias da classe *Transition* e deve retornar o valor *true* se a transição deve ser impedida de ser disparada pela restrição ou o valor *false* caso contrário. Um exemplo de implementação de uma restrição é ilustrado na Tabela 5.11. Neste exemplo, as mensagens de uma determinada transição devem possuir o padrão: *content(television,brand(AnyBrand),price(Amount))*. A implementação da restrição certifica que o valor da variável *Amount* esteja entre 50 e 100.

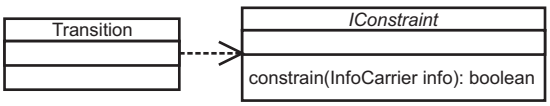


Figura 5.17: Módulo de Restrições

```
public class RangeValue implements IConstraint{

  public boolean constrain(InfoCarrier info) {
    String value = (String)info.getValue("Amount");
    if (value==null) return true;
    int intValue = Integer.parseInt(value);
    if (intValue>50 && intValue<100){
      return false;
    }else{
      return true;
    }
  }
}
```

Tabela 5.11: Exemplo da Implementação de uma Restrição

### 5.3.9 Ações

Mais uma vez o módulo de triggers é reutilizado, desta vez para a implementação do módulo de ações. O módulo de ações reutiliza o módulo de triggers mas ao mesmo tempo esconde esta reutilização dos desenvolvedores de ações. Do ponto de vista de um desenvolvedor que vai implementar uma ação, apenas uma interface está disponível: *IActionFiring*. Esta interface fornece o método *execute* que deve ser implementado para prover as funcionalidades de recuperação automática de software (*self-healing*).

Esta transparência e facilidade de uso para a implementação de ações é alcançada devido às classes mostradas na Figura 5.18. A classe *Action* implementa a interface *ITrigger* e, portanto, é capaz de escutar eventos e executar algum comportamento uma vez que seus eventos de ativação ocorram. O comportamento a ser executado é representado pela classe *TriggerFiring*, que é estendida pela classe *ActionFiringAdapter*. A classe *ActionFiringAdapter* age como um *Adapter* [69] e delega seu comportamento para a interface *IActionFiring*. Então, o comportamento da ação é definido implementando-se a interface *IActionFiring*.

## 5.4 Módulo de Suporte ao Desenvolvedor de Agentes

Devido à natureza distribuída e concorrente de sistemas multi-agentes, o desenvolvimento de agentes envolve os desenvolvedores em problemas que não são relacionados com as funcionalidades presentes completamente na aplicação a ser desenvolvida. Tratamento de concorrência e gerenciamento da distribuição são alguns dos problemas que os desenvolvedores enfrentam.

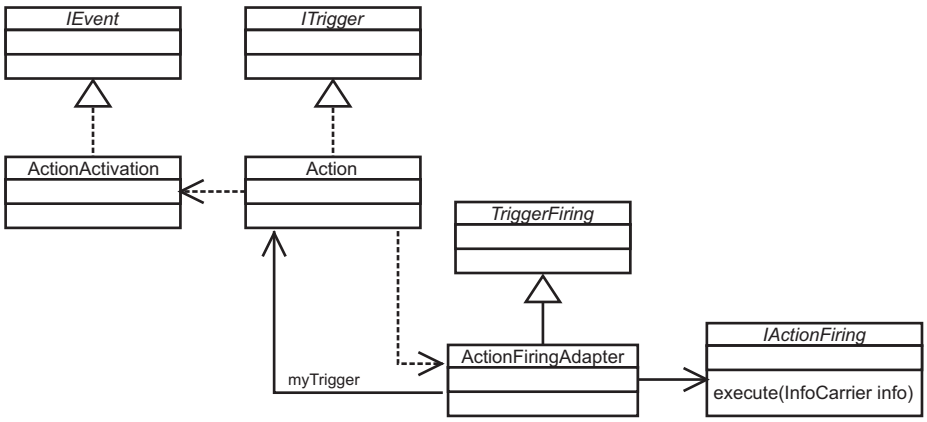


Figura 5.18: Módulo de Ações

Quando os agentes estão interagindo em um sistema que faz a aplicação de leis de interação, um componente de complexidade é adicionado e os desenvolvedores devem projetar os seus agentes sabendo que existem leis que podem impedir determinadas ações dos agentes.

O módulo de suporte ao desenvolvedor de agentes auxilia desenvolvedores na programação dos agentes, de forma que detalhes da comunicação entre os agentes possam ser abstraídos e que a integração com o sistema que aplica as leis (neste caso, o agente mediador) seja suavizada buscando a transparência.

O suporte oferecido por este módulo é disponibilizado para o desenvolvedor por um conjunto de classes, que são apresentadas nas seções 5.4.1 e 5.4.2.



### 5.4.1 Agente

Existem duas maneiras de criar um novo agente: construindo-o desde o início ou usando a classe *Agent* provida por este módulo de agentes do framework. Esta classe fornece métodos que auxiliam no envio e no recebimento de mensagens, é completamente integrada com a abordagem de leis, além de fornecer algumas outras pequenas facilidades. Caso o agente a ser desenvolvido já herde de alguma outra classe, ainda assim é possível utilizar a classe *Agent* provida pelo módulo através da técnica de delegação de chamada de métodos e, assim, alcançar os benefícios fornecidos pela classe *Agent*.

O envio e o recebimento de mensagens utilizando a classe *Agent* é realizado com apenas uma chamada de método. A classe *Agent* possui uma instância da classe *ICommunication* como um de seus atributos. Este atributo representa o canal por onde o agente envia e recebe mensagens. O envio de uma mensagem utilizando este canal pode ser feito da seguinte forma:

```
this.communication.send(message);
```

Entretanto, o canal de comunicação está encapsulado na classe *Agent*, sendo disponibilizado pelos métodos mostrados na Tabela 5.12.

```
public void send(Message msg){
    communication.send(msg);
}
public Message nextMessage() {
    return communication.nextMessage();
}
public Message waitForMessage() {
    return communication.waitForMessage();
}
public Message waitForMessage(long milliseconds) {
    return communication.waitForMessage(milliseconds);
}
```

Tabela 5.12: Implementações dos Métodos de Envio e Recebimento de Mensagens na Classe *Agent*

### 5.4.2 Protocolo do Mediador

Na maior parte do tempo os agentes não estão cientes da existência de um mediador. Isto acontece porque a interceptação e aplicação das leis nas mensagens é feita transparentemente pela classe *Agent* através do módulo de comunicação (Seção 5.2). Entretanto, agentes também podem se comunicar com o agente mediador para pedir informações sobre as especificações das leis, solicitar autorizações para criar e participar de cenas, ou mesmo para saber o estado de execução da aplicação das leis. Além disso, o mediador pode autonomamente enviar mensagens para os agentes contendo informações sobre as leis. Estas mensagens são compostas por três partes: a performativa que especifica a intenção do agente; o conteúdo ou operação que indica a operação sendo requisitada ou as informações sendo transmitidas; e por último um conjunto de parâmetros, que pode inclusive ser vazio.

Na Figura 5.19, ilustram-se as mensagens que os agentes podem enviar para o mediador. Entre estas mensagens estão mensagens que solicitam uma autorização para entrar em uma organização; requisitam a lista de todas as organizações que o mediador está mediando; solicitam a criação de uma instância de uma cena; solicitam a participação de um agente em uma cena; requisitam a lista de todas as cenas especificadas em uma determinada organização; e requisitam a lista de todas as instâncias de cenas.

O agente mediador, por sua vez, responde a estas requisições dos agentes, através de um conjunto de mensagens mostradas na Figura 5.20. Além de responder às requisições, os mediadores também podem enviar mensagens in-

formando que alguma coisa ocorreu durante a aplicação das leis. Por exemplo, a mensagem *lawException* é enviada a um agente quando a mensagem que o agente tentou enviar para algum outro membro da organização de agentes não é válida de acordo com as leis.

| Performative | Message | Description |
|---|---|---|
| request | enterOrganization | Request authorization to enter in an organization |
| request | listOrganizations | Request a list of all organizations |
| request | startScene | Request to start a scene execution |
| request | enterInScene | Request authorization to enter in a scene |
| request | listScenes | Request a list of all scenes of an organization |
| request | listRunningScenes | Request a list of all running scenes of a specific scene |

Figura 5.19: Protocolo do Mediador - Mensagens que os Agentes Podem Enviar

| Performative | Message | Description |
|---|---|---|
| inform | sceneAuthorization | Inform a scene authorization |
| failure | invalidSceneAuthorization | Inform the scene authorization provided is not valid |
| inform | organizationAuthorization | Inform a organization authorization |
| failure | invalidOrganizationAuthorization | Inform the organization authorization provided is not valid |
| failure | organizationDoesNotExist | Organization that agent said does not exist |
| failure | sceneDoesNotExist | Scene agent said does not exist |
| inform | organizationList | List of all organizations loaded in the mediator |
| inform | sceneList | List of all scenes of a certain organization |
| inform | runningSceneList | List of all running scenes of a specific scene |
| failure | lawException | Law definitions do not allow the message sent |
| failure | undefined | Unexpected behavior of the mediator agent |

Figura 5.20: Protocolo do Mediador - Mensagens que os Mediadores Podem Enviar

A classe *Agent* fornece também um conjunto de métodos que auxiliam no envio das requisições que um agente pode enviar ao mediador. Cada um destes métodos envia a mensagem adequada, aguarda a resposta do agente mediador e encapsula a resposta em objetos Java para facilitar o manuseio das informações pelos desenvolvedores dos agentes. A classe *Agent* é ilustrada na Figura 5.21.

### 5.4.3
### Guia para o Desenvolvimento de Agentes: *Ping Pong*

O objetivo desta seção é ilustrar os passos necessários para a criação de agentes, para isto uma aplicação onde se realiza a comunicação entre dois agentes é apresentada.

Como o foco é ilustrar os passos para a criação dos agentes de forma a guiar desenvolvedores na criação de novos sistemas multi-agentes, optou-se por apresentar uma aplicação simples que funciona da seguinte maneira: dois agentes, denominados *ping-agent* e *pong-agent*, fazem parte de uma organização de

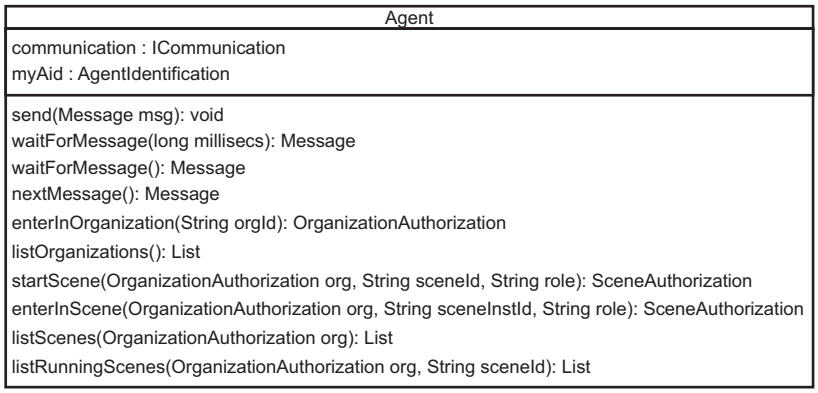


Figura 5.21: Classe *Agent*

agentes chamada *International Ping-Pong Organization*. Estes agentes desempenham os papéis *ping* e *pong* respectivamente. A única interação existente nesta organização é o envio de uma mensagem contendo o valor *ping* que o *ping-agent* envia ao *agent-pong* e a resposta do *pong-agent* através de uma mensagem contendo o valor *ping-pong*.

A interação entre os agentes desta aplicação pode ser especificado como ilustrado na Tabela 5.13. Nesta tabela, especificada utilizando-se XMLaw, define-se uma cena, denominada *game*, onde apenas o agente desempenhando o papel de *ping* pode iniciá-la. Após iniciada, a cena conta com dois tipos de participantes: *ping* e *pong*, que entram nos estados *s0* e *s1*, respectivamente. O restante da lei que regula a interação nesta cena especifica que um agente desempenhando o papel de *ping* envia uma certa mensagem e um outro agente desempenhando o papel de *pong* responde a esta mensagem.

```
<Laws>
    <LawOrganization id="org-ping-pong" name="International␣Ping-Pong␣
        Organization"/>
        <Scenes>
            <!-- ++++++++++++++++++++++++++++++++++++++++++++++++++++++ -->
            <Scene id="game" time-to-live="infinity">
                <Creators>
                    <Creator agent="any" role="ping"/>
                </Creators>
                <Entrance>
                    <Participant agent="any" role="ping">
                        <States>
                            <State ref="s0"/>
                        </States>
                    </Participant>
                    <Participant agent="any" role="pong">
                        <States>
                            <State ref="s1"/>
                        </States>
                    </Participant>
                </Entrance>
                <Messages>
                    <Message id="m1" template="message(request,sender(_,ping),
                        receiver(_,pong),content(ping))."/>
                    <Message id="m2" template="message(inform,sender(_,pong),
                        receiver(_,ping),content(ping-pong))."/>
                </Messages>
                <Protocol>
                    <States>
                        <State id="s0" type="initial" label="Initial␣state"/>
                        <State id="s1" type="execution" label="Ping␣sent"/>
                        <State id="s2" type="success" label="Pong␣sent"/>
                    </States>
                    <Transitions>
                        <Transition id="t1" from="s0" to="s1" message-ref="m1"/>
                        <Transition id="t2" from="s1" to="s2" message-ref="m2"/>
                    </Transitions>
                </Protocol>
            </Scene>
    </LawOrganization>
</Laws>
```

Tabela 5.13: Definição de Lei do Ping-Pong

O agente que envia o *ping* é o agente que inicia a conversação. Existem algumas diferenças entre criar um agente que inicia a conversação e entre um agente entrar na conversação em resposta a uma mensagem recebida por um agente. Desta forma, a criação destes dois agentes é mostrada em separado. A criação do *ping-agent* é mostrada nas etapas que seguem.

1. Estender a classe *Agent* - neste primeiro passo, reaproveita-se a implementação da classe *Agent* provida pelo framework. O parâmetro *name* do construtor da classe *PingAgent* refere-se ao nome pelo qual o agente será

identificado, o que neste caso será *ping-agent*. Porém, optou-se por passar esta informação no momento da instanciação.

```
public class PingAgent extends Agent {

    public PingAgent(String name) {
        super(name);
    }

}
```

Tabela 5.14: Implementando o Agente Ping - Passo 1

2. Solicitar a entrada na organização *org-ping-pong* - como o agente mediador pode estar gerenciando muitas organizações, é preciso informar-lhe em qual organização que está se querendo interagir. Isto é feito simplesmente invocando o método *enterInOrganization* informando-se o identificador da organização definido na lei.

```
public class PingAgent extends Agent {

    public PingAgent(String name) {
        super(name);
    }

    public void run(){
        OrganizationAuthorization orgAuth = enterInOrganization("org-
            ping-pong");
    }

}
```

Tabela 5.15: Implementando o Agente Ping - Passo 2

3. Criar uma instância da cena *game* - desta vez, como o agente *ping-agent* é o iniciador da conversação, é preciso que ele solicite a criação de uma nova cena. Isto é feito invocando o método *startScene*, onde informa-se nos parâmetros a organização à qual a cena pertence e que o agente já foi autorizado a entrar, o identificador da cena que se deseja criar, e qual o papel que o agente *ping-agent* está exercendo.

```
public class PingAgent extends Agent {

    public PingAgent(String name) {
        super(name);
    }

    public void run(){
        OrganizationAuthorization orgAuth   = enterInOrganization("org-
            ping-pong");
        SceneAuthorization sceneAuth        = startScene(orgAuth,"game","
            ping");
    }

}
```

Tabela 5.16: Implementando o Agente Ping - Passo 3

4. Solicitar a entrada na cena *game* - dado que a cena já está em execução, agora é necessário a solicitação de participação na cena. É importante perceber que, embora um agente possa ter permissão para criar uma cena, pode não ser permitida sua participação na mesma.

```
public class PingAgent extends Agent {

    public PingAgent(String name) {
        super(name);
    }

    public void run(){
        OrganizationAuthorization orgAuth   = enterInOrganization("org-
            ping-pong");
        SceneAuthorization sceneAuth        = startScene(orgAuth,"game","
            ping");
        sceneAuth                           = enterInScene(sceneAuth,"
            ping");
    }

}
```

Tabela 5.17: Implementando o Agente Ping - Passo 4

5. Enviar uma mensagem para o agente *pong-agent* contendo *ping* - nesta etapa, as classes que auxiliam na comunicação entre os agentes são utilizadas. Uma instância da classe *Message* é criada passando a performativa como parâmetro e depois os seus parâmetros são preenchidos. No fim, invoca-se o método *send* da classe *Agent* e a mensagem é enviada para o destino.

```
public class PingAgent extends Agent {

    public PingAgent(String name) {
        super(name);
    }

    public void run(){
        OrganizationAuthorization orgAuth   = enterInOrganization("org-
            ping-pong");
        SceneAuthorization sceneAuth        = startScene(orgAuth,"game","
            ping");
        sceneAuth                           = enterInScene(sceneAuth,"
            ping");

        Message pingMessage = new Message(Message.REQUEST);
        pingMessage.setSceneAuthorization(sceneAuth);
        AgentIdentification pongAgentAid = new AgentIdentification("pong-
            agent");
        pingMessage.setReceiver(pongAgentAid,"pong");
        pingMessage.setContent("ping");
        send(pingMessage);
    }

}
```

Tabela 5.18: Implementando o Agente Ping - Passo 5

6. Esperar a resposta do agente *pong-agent* e imprimi-la - como o agente *ping-agent* enviou uma requisição ao *pong-agent*, é natural que ele espere pela resposta a sua requisição. Para isto, usa-se o método *waitForMessage*, também da classe *Agent*.

```
public class PingAgent extends Agent {

    public PingAgent(String name) {
        super(name);
    }

    public void run(){
        OrganizationAuthorization orgAuth   = enterInOrganization("org-
            ping-pong");
        SceneAuthorization sceneAuth        = startScene(orgAuth,"game","
            ping");
        sceneAuth                           = enterInScene(sceneAuth,"
            ping");

        Message pingMessage = new Message(Message.REQUEST);
        pingMessage.setSceneAuthorization(sceneAuth);
        AgentIdentification pongAgentAid = new AgentIdentification("pong-
            agent");
        pingMessage.setReceiver(pongAgentAid,"pong");
        pingMessage.setContent("ping");
        send(pingMessage);

        Message pongMessage = waitForMessage();
        System.out.println(pongMessage.getContent());
    }

}
```

Tabela 5.19: Implementando o Agente Ping - Passo 6

A criação do agente *pong-agent* é parecida com a do agente *ping-agent*. Porém, como o *pong-agent* não inicia a conversação e, portanto, não cria uma cena. Ele precisa de outra forma de obter uma referência a instância correta da cena. Isto pode ser feito de duas maneiras. A primeira é utilizando o método *getAllRunningScenes* da classe *Agent*, e a segunda é através do recebimento de uma mensagem de um agente que já está participando de uma cena. No caso do *pong-agent*, a segunda opção é adotada. Os passos a seguir ilustram a criação do agente Pong.

1. Estender a classe *Agent*.

```
public class PongAgent extends Agent {

    public PongAgent(String name) {
        super(name);
    }

}
```

Tabela 5.20: Implementando o Agente Pong - Passo 1

2. Solicitar a entrada na organização *org-ping-pong*.

```
public class PongAgent extends Agent{

    public PongAgent(String name) {
        super(name);
    }

    public void run(){
        OrganizationAuthorization orgAuth   = enterInOrganization("org-
            ping-pong");
    }

}
```

Tabela 5.21: Implementando o Agente Pong - Passo 2

3. Esperar a mensagem do agent *ping-agent* e solicitar a entrada na cena *game* - é importante observar que na chamada do método *enterInScene*, a referência à instância da cena que o *ping-agent* criou está contida na autorização que veio na mensagem. Desta forma, ao passar essa referência como parâmetro, entra-se na mesma instância de cena do *ping-agent*.

```
public class PongAgent extends Agent{

    public PongAgent(String name) {
        super(name);
    }

    public void run(){
        OrganizationAuthorization orgAuth   = enterInOrganization("org-
            ping-pong");

        Message pingMessage = waitForMessage();
        SceneAuthorization sceneAuth = enterInScene(pingMessage.
            getSceneAuthorization(),"pong");
    }

}
```

Tabela 5.22: Implementando o Agente Pong - Passo 3

4. Enviar uma mensagem contendo *ping-pong* como resposta.

```
public class PongAgent extends Agent{

    public PongAgent(String name) {
        super(name);
    }

    public void run(){
        OrganizationAuthorization orgAuth    = enterInOrganization("org-
            ping-pong");

        Message pingMessage = waitForMessage();
        SceneAuthorization sceneAuth = enterInScene(pingMessage.
            getSceneAuthorization(),"pong");

        Message pongMessage = pingMessage.createReply();
        pongMessage.setPerformative(Message.INFORM);
        pongMessage.setContent(pingMessage.getContent()+"-pong");
        pongMessage.setSceneAuthorization(sceneAuth);

        send(pongMessage);
    }

}
```

Tabela 5.23: Implementando o Agente Pong - Passo 4

Para executar os dois agentes e o agente mediador, cria-se uma classe que contenha o método Java *main*, seguindo-se os passos a seguir.

1. Iniciar o agente mediador e informar onde está a definição da lei da organização do ping-pong - a criação do agente mediador é realizada invocando o método estático *main* da classe *RunGod* passando o endereço onde a lei está localizada. A lei pode estar localizada inclusive em algum servidor web, como por exemplo: *http://someplace/ping-pong-law.xml*.

```
public class RunPingPong {

    public static void main(String[] args) {
        RunGod.main("C:/dev/ping-pong/law.xml");
    }

}
```

Tabela 5.24: Executando a Aplicação - Passo 1

2. Iniciar o agente *pong-agent*.

```
public class RunPingPong {

    public static void main(String[] args) {
        RunGod.main(null);

        PingAgent pongAgent = new PingAgent("pong-agent");
        pongAgent.start();
    }

}
```

Tabela 5.25: Executando a Aplicação - Passo 2

3. Iniciar o agente *ping-agent*.

```
public class RunPingPong {

    public static void main(String[] args) {
        RunGod.main(null);

        PingAgent pongAgent = new PingAgent("pong-agent");
        pongAgent.start();

        PingAgent pingAgent = new PingAgent("ping-agent");
        pingAgent.start();
    }

}
```

Tabela 5.26: Executando a Aplicação - Passo 3

A execução destes passos finaliza o guia de desenvolvimento exemplificado. O desenvolvimento de sistemas multi-agentes mais complexoes segue um processo análogo ao apresentado, salvo uma quantidade maior de implementação.