UNIVERSIDAD NACIONAL DEL CENTRO DE LA PROVINCIA DE BUENOS AIRES
FACULTAD DE CIENCIAS EXACTAS
MAGISTER EN INGENIERÍA DE SISTEMAS

Brainstorm/J: Un framework para agentes inteligentes

por
Alejandro Zunino

Trabajo sometido a evaluación como requisito parcial
para la obtención del grado de
Magister en Ingeniería de Sistemas

Prof. Dra. Analía Amandi
Director

Tandil, Marzo del 2000

Agradecimientos

En primer lugar quiero agredecer a Analía Amandi por sus innumerables ideas, inagotable paciencia, gran dedicación y fundamentalmente, por el apoyo y amistad brindada durante la realización de este trabajo.

Un agradecimiento muy especial a Marcelo Campo, quien participó de este trabajo con sugerencias, discusiones e innumerables buenos momentos.

Quiero expresar mi gratitud tanto a Analía como a Marcelo por haberme enseñado todo lo que sé sobre investigación y por ser grandes amigos con quienes compartí momentos muy gratos. Realmente es un placer poder trabajar con personas que ponen todo de sí para contribuir con la formación de sus alumnos, tanto profesionalmente como personalmente.

Agradezco a Claudia Marcos, quien supo evacuar mis dudas sobre re¤exión computacional, meta-objetos y UML, estando siempre dispuesta a ofrecer su ayuda. A Edgardo Belloni y Ramiro Iturregui debo agradecerles por su amistad, los buenos momentos compartidos y las numerosas conversaciones sobre este trabajo.

A mis compañeros del Laboratorio de Agentes Inteligentes y del Grupo de Objetos y Visualización del Instituto ISISTAN, con quienes compartí innumerables conversaciones sobre este trabajo y muy gratos momentos.

Un agradecimiento muy especial a mi novia Julia, quien soportó mi mal humor durante este último año, comprendiéndome y apoyándome incondicionalmente. A mis padres y hermanos debo agredecer todo el apoyo y cariño brindado.

Agradezco a la Facultad de Ciencias Exactas de la Universidad Nacional del Centro y al Instituto ISISTAN, por haberme dado un lugar de trabajo que se convirtió en mi segundo hogar. Finalmente, agradezco a la Comisión de Investigaciones Científicas de la Provincia de Buenos Aires, por haberme apoyado con una beca de iniciación a la investigación.

Gracias

Índice General

Agradecimientos iii

Índice General v

Índice de Figuras xi

Índice de Tablas xv

Índice de Algoritmos xvii

Resumen xix

Abstract xxi

Glosario xxiii

1 Introducción 1

1.1 Organización de este trabajo 8

2 Contexto 9

2.1 Agentes 9

2.1.1 Capacidad de acción 12

2.1.2 Percepción 12

2.1.3 Comunicación 13

2.1.3.1 KQML 13

2.1.3.2 COOL 15
2.1.4 Movilidad 16
2.1.5 Reacción 17
2.1.6 Deliberación 18
2.1.6.1 Planning 18
2.1.7 Aprendizaje 20
2.2 Frameworks 21
2.3 Diseño de frameworks 22
2.4 La arquitectura Brainstorm 24
2.4.1 MetaAgent 27
2.4.2 Representación y manipulación de estados mentales 27
2.4.3 Percepción 27
2.4.4 Comunicación 28
2.4.5 Situaciones 28
2.4.6 Comportamiento reactivo 28
2.4.7 Deliberación 28
2.4.8 Aprendizaje 29
2.5 Conclusiones 29

3 Herramientas para construcción de agentes **31**
3.1 Características analizadas 31
3.2 AgentBuilder 33
3.3 DECAF 35
3.4 FraMaS 37
3.5 JAF 38
3.6 JAFIMA 40
3.7 JAFMAS 41
3.8 MadKit 42
3.9 ZEUS 42
3.10 Conclusiones 43

4 Brainstorm/J 49

4.1 Materialización de los componentes arquitectónicos 50
4.2 Creación de un agente 54
4.3 Capacidad de acción 58
4.4 Percepción 59
4.5 Situaciones 61
4.6 Comportamiento reactivo 63
4.7 Movilidad 64
4.8 Deliberación 66
4.8.1 Planes abstractos 67
4.8.2 Arquitectura 69
4.8.3 Fuentes de conocimiento 72
4.8.4 Eventos 75
4.8.5 Creencias 76
4.8.6 Objetivos 78
4.8.7 Reparación de planes 79
4.8.7.1 Planes en construcción 79
4.8.7.2 Planes en ejecución 81
4.9 Comunicación 83
4.10 Conclusiones 87

5 Instanciación de Brainstorm/J 89

5.1 Forklifts 89
5.1.1 Nivel base 90
5.1.2 Creación de un agente 91
5.1.3 Capacidad de acción 92
5.1.4 Percepción 92
5.1.5 Situaciones 94
5.1.6 Comportamiento reactivo 95
5.1.7 Deliberación 96
5.1.7.1 Creencias 96
5.1.7.2 Objetivos 96
5.1.7.3 Fuentes de conocimiento 97
5.1.7.4 Planes abstractos 97

5.2 *n*-Queens . . . 98
5.2.1 Implementación . . . 99
5.2.2 Comparación . . . 103
5.3 Conclusiones . . . 104

6 Especi£cación en Object-Z 105
6.1 JMOP . . . 105
6.1.1 Identificadores . . . 105
6.1.2 Objetos . . . 106
6.1.3 Variables referencia . . . 107
6.1.4 Sentencias . . . 108
6.1.5 Métodos . . . 109
6.1.6 Invocaciones a métodos . . . 109
6.1.7 Clases . . . 110
6.1.8 Meta-objetos . . . 112
6.2 Brainstorm/J . . . 113
6.2.1 Brain . . . 114
6.2.2 MetaAgent . . . 115
6.2.3 Perceptor . . . 116
6.2.4 SituationListener . . . 116
6.2.5 SituationGenerator . . . 116
6.2.6 SituationManager . . . 117
6.2.7 Task . . . 118
6.2.8 Reaction . . . 119
6.2.9 Reactor . . . 119
6.2.10 DEventDispatcher . . . 120
6.2.11 DEvent . . . 121
6.2.12 Deliberator . . . 121
6.2.13 DelibKS . . . 122

7 Conclusiones y trabajos futuros 125
7.1 Contribuciones . . . 126
7.2 Limitaciones . . . 126
7.3 Trabajos futuros . . . 127

A Meta-Objetos en Java **129**

B JavaLog **133**

B.1 Módulos 133

B.1.1 Módulos lógicos en variables de instancia y en métodos 134

B.1.2 Objectos como cláusulas 135

B.1.3 Cláusulas con objetos 135

B.1.4 Composición de módulos 135

C Object-Z **137**

C.1 El lenguaje Z 137

C.1.1 Tipos de datos 137

C.1.2 Esquemas de estado 138

C.1.3 Esquemas de operaciones 139

C.1.4 Cálculo de esquemas 139

C.2 El lenguaje Object-Z 140

D Notación UML **143**

D.1 Correspondencia tipo-instancia 143

D.2 Diagramas de estructura estática 143

D.2.1 Diagrama de clases 144

D.2.2 Diagrama de objetos 144

D.2.3 Clases 144

D.2.4 Interfaces 145

D.2.5 Clases parametrizadas (*templates*) 145

D.2.6 Elemento ligado 146

D.2.7 Objetos 147

D.2.8 Objeto compuesto 148

D.2.9 Asociación 148

D.2.10 Asociación binaria 148

D.2.11 Extremo final de la asociación 149

D.2.12 Multiplicidad 150

D.2.13 Calificador 150

D.2.14 Clase asociación 151

D.2.15 Asociación n-aria 151

D.2.16 Composición 152
D.2.17 Links 152
D.2.18 Generalización 154
D.2.19 Dependencia 154
D.3 Diagrama de interacción 154

Bibliografía **157**

Índice de Figuras

1.1 Brainstorm y su materialización en Brainstorm/J 6
1.2 Método *template* responsable de crear e inicializar un agente 8

2.1 Clasificación de agentes [9] 11
2.2 Representación de una conversación cliente-vendedor desde el punto de vista del cliente 15
2.3 Agentes móviles para reducir el tráfico de red 17
2.4 Operadores para el dominio de cohetes [95] 19
2.5 Plan para transportar una carga desde Londres a París 20
2.6 Instanciación de un *framework* orientado a objetos 23
2.7 Diseño conducido por un modelo de dominio [16] 23
2.8 Meta-objetos 24
2.9 La arquitectura Brainstorm 25
2.10 Un objeto-agente 27

3.1 Arquitectura de DECAF [52] 36
3.2 Estructura TÆMS describiendo cómo preparar un café 39
3.3 Arquitectura de niveles de JAFIMA [60] 40

4.1 Materialización de Brainstorm 51
4.2 Creación de un agente 54
4.3 La aplicación de robots de carga 55
4.4 Diagrama de clases de la aplicación de robots de carga 56

4.5 Diagrama de interacción de la creación de un agente 57
4.6 Representación de las capacidades de acción 58
4.7 Materialización del administrador de situaciones 62
4.8 Reacciones 63
4.9 Movilidad débil 65
4.10 Estructura del deliberador 67
4.11 Plan abstracto para descargar un camión 68
4.12 Diagrama del componente deliberativo 70
4.13 Planes abstractos 71
4.14 Plan abstracto para descargar un camión (sólo caminata aleatoria) 72
4.15 Principales clases del componente de deliberación 73
4.16 Reparación de un plan (creencia falsa) 80
4.17 Reparación de un plan (acción redundante) 81
4.18 Reparación de un plan (creencia falsa) 82
4.19 Diagrama de clases del componente de comunicación y su especialización para KQML 83
4.20 Facilitador de comunicaciones (adaptado de [75]) 85
4.21 Comunicación con KQML 86
4.22 Clases de conversaciones representadas como planes abstractos 87

5.1 Sistema multi-agente FORKS 90
5.2 Clases de los objetos situados en el nivel base 90
5.3 Principales clases involucradas en la creación de los agentes *forklift* 91
5.4 Capacidad de acción de los agentes 93
5.5 Reacciones de los agentes *forklift* 95
5.6 Plan abstracto para descargar un camión (sólo caminata aleatoria) 98
5.7 Representación de *FirstQueenConv* 99
5.8 Representación de *MiddleQueenConv* 99
5.9 Representación de *LastQueenConv* 100
5.10 Objetos y meta-objetos de un agente reina 100
5.11 Materialización de *MiddleQueenConv* utilizando Brainstorm/J 101
5.12 Ventana presentada por cada reina 103

A.1 Principales de clases del *framework* JMOP 130
A.2 Modificación de las clases en tiempo de ejecución 131

B.1 Módulos lógicos en clases, métodos y objetos 134
B.2 Módulos lógicos privados de un agente 134
B.3 Combinación de módulos lógicos mediante herencia 136

D.1 Clases y objetos 144
D.2 Notación de clases abstractas y concretas 145
D.3 Notación para las interfaces en los diagramas de clases 146
D.4 Notación de clases parametrizadas 146
D.5 Objeto compuesto 148
D.6 Notación de las asociaciones 149
D.7 Asociaciones 150
D.8 Asociaciones calificadas 151
D.9 Clase asociación 151
D.10 Asociación ternaria en una clase asociación 152
D.11 Diferentes formas de mostrar la composición 153
D.12 Links 153
D.13 Generalización 154
D.14 Dependencias entre clases 154
D.15 Notación de los diagramas de interacción 155

Índice de Tablas

2.2 Performativas KQML 14

3.1 Capacidades de agentes soportadas por las herramientas 44

3.2 Tipo, objetivos de diseño, arquitectura de agentes y lenguaje de cada una de las herramientas 45

4.1 Creencias adquiridas durante la percepción 61

5.1 Métricas de clases, métodos y complejidad ciclomática de las dos implementaciones del problema de las reinas 1

Índice de Algoritmos

1 Inicialización de los esquemas de acción 60
2 Ejecución de reacciones 63
3 Revisión de creencias 78

Resumen

Los agentes inteligentes y sistemas multi-agente son una de las áreas de investigación más promisorias de los últimos años. Los agentes ofrecen nuevas maneras de analizar, diseñar e implementar sistemas de software, mejorando, potencialmente, la forma en la cual se modela y materializa el software.

Las aplicaciones de agentes inteligentes son sorprendentemente variadas: desde simples asistentes personales [68], buscadores [36] y robots [13], hasta complejos controladores autónomos de propulsión para el transbordador espacial [49], asistentes para la construcción de software [74] y sistemas industriales [41]. Por otro lado, los agentes pueden realizar esta variedad de actividades de diferentes formas [102]: interacción con personas y/o agentes, reactividad, deliberación, representación y manipulación de estados mentales, y movilidad, entre otras.

Estos factores dificultan la construcción de herramientas genéricas para desarrollar agentes, por cuanto resulta problemático capturar la funcionalidad común presente en todos esos tipos de aplicaciones y, en forma simultánea, la funcionalidad común de los diversos tipos de agentes y variedad de actividades que pueden realizar [72, 88].

Como consecuencia de estos factores, en la mayoría de los casos los agentes son implementados desarrollando componentes de software en forma *ad-hoc* para cada nueva aplicación, lo que constituye uno de los mayores obstáculos para la adopción masiva de la tecnología de agentes [106].

Esta forma de desarrollo *ad-hoc* no aprovecha el hecho de que existe funcionalidad común presente en varios tipos de agentes, por ejemplo, la capacidad de comunicarse, representar y manipular estados mentales o deliberar.

En el presente trabajo se propone una infraestructura genérica, adaptable y extensible a partir de la cual es posible construir agentes inteligentes reusando la funcionalidad común de los mismos. Dicha infraestructura ha sido materializada en un *framework* orientado a objetos denominado Brainstorm/J, basado en la arquitectura de agentes Brainstorm.

Brainstorm/J permite construir agentes con capacidades de percepción, representación y manipulación de los estados mentales, deliberación, reacción, comunicación, movilidad; combinando esas capacidades de diversas formas para obtener diferentes tipos de agentes. El *framework* especifica componentes de software adaptables y ¤exibles responsables de dichas capacidades. Esos componentes pueden ser

adaptados y extendidos para construir agentes, permitiendo que el programador defina la funcionalidad de sus aplicaciones a partir de los componentes reusables provistos por el *framework*.

Brainstorm/J permite que los agentes mantengan una representación del medio ambiente en que se encuentran, asegurando la coherencia de la misma y razonando en función de ella. Los agentes pueden deliberar y, en forma simultánea, percibir, mantener conversaciones con otros agentes, actuar, etc. Además, un agente puede razonar, en forma simultánea, sobre cómo lograr sus objetivos utilizando varios mecanismos deliberativos.

Palabras clave: Orientación a Objetos, *Frameworks*, Sistemas Multi-Agente.

Abstract

Intelligent agents and multi-agent systems are one of the more promissory areas of research of the last years. Agents offers new ways to analyze, design and implement software systems, improving, potentially, the ways in which software is modeled and materialized.

Applications of intelligent agents are surprising varied: from simple personal assistants [68], internet agents [36] and mobile robots [13], to complex autonomous propulsion systems for the space shuttle [49], assistants for software development [74] and agents for process scheduling in industrial systems [41]. On the other hand, agents can carry out this variety of activities in different ways [102]: interaction with people and/or agents, reactivity, deliberation, representation and manipulation of mental states, and mobility, among others.

These factors make difficult the construction of generic tools to develop agents, inasmuch as it is problematic to capture the common functionality of all these types of applications and, at the same time, the common functionality of the diverse types of agents and the variety of activities that they can perform [72, 88].

As a result of these factors, agents are implemented by developing software components in an ad-hoc manner for each new application. This constitutes one of the most important obstacles for the massive adoption of agent technology [106].

This form of ad-hoc development does not take advantage of the fact that there is common functionality among several types of agents, for example, the communication capability, manipulation of mental states and deliberation.

This work consists on a generic infrastructure for building agents called Brainstorm/J. By using that it is posibble to develop intelligent agents based on the common functionality provided by the infrastructure. Moreover, it can be adapted and extended by the developer to be suitable for different requeriments in a veriety of application domains. The infrastructure has been materialized by an object-oriented framework based on the Brainstorm architecture.

Brainstorm/J sopports the contruction of agents with capacities of perception, representation and manipulation of the mental states, deliberation, reaction, communication, mobility; combining these capacities to obtain different types of agents. The framework specifies adaptable and ¤exible software

components in charge of these capacities. These components can be adapted and extended to develop agents, allowing the programmer to define the specific functionality of his applications from the reusables components provided by the framework.

Agents built with Brainstorm/J are able to represent its environment, assuring the coherence of this representantion and, at the same time, reason based on it. Agents can deliberate and, simultaneously, perceive, maintain conversations with other agents, act, etc. In addition, an agent can reason, at the same time, on how to achieve its goals by using several reasoning mechanisms.

Keywords: Object Orientation, *Frameworks*, Multi-Agent Systems.

Glosario

ACL Agent Communication Language

AFDR Autómata Finito Determinístico Recursivo

BDI Belief-Desire-Intention

DECAF Distributed Environment Centered Agent Framework

KQML Knowledge Query and Manipulation Language

LAN Local Area Network

NCSS Non Commenting Source Statements

RADL Reticular Agent Definition Language

RJT Redes Jerárquicas de Tareas

SMA Sistema Multi-Agente

TÆMS Task Analysis, Environment Modeling and Simulation

TCP/IP Transport Control Protocol, Internet Protocol

TCRC Teoría de Coherencia de Revisión de Creencias

TFRC Teoría Fundacional de Revisión de Creencias

UML Unified Modeling Languaje

CAPÍTULO 1

Introducción

Los agentes inteligentes y sistemas multi-agente son una de las áreas de investigación más promisorias de los últimos años. Los agentes ofrecen nuevas maneras de analizar, diseñar e implementar sistemas de software, mejorando, potencialmente, la forma en la cual se modela y materializa el software.

Un agente inteligente es una entidad computacional capaz de percibir su medio ambiente y actuar en forma *autónoma* [30] y ¤*exible* [9]. La autonomía se refiere a la capacidad de realizar la mayoría de sus actividades sin intervención humana o de otros agentes, mientras que ¤*exible* significa que el agente debe adecuarse a los cambios de su medio ambiente de manera oportunística y guiada por objetivos [56]. Además, un agente debe ser capaz de interactuar con otros agentes para encontrar diferentes modos de realizar sus actividades o colaborar con ellos [105].

Las aplicaciones de agentes inteligentes son sorprendentemente variadas: desde simples asistentes personales [68], buscadores [36] y robots [13], hasta complejos controladores autónomos de propulsión para el transbordador espacial [49], asistentes para la construcción de software [74] y sistemas industriales [41]. Por otro lado, los agentes pueden realizar esta variedad de actividades de diferentes formas [102]: interacción con personas y/o agentes, reactividad, deliberación, representación y manipulación de estados mentales, y movilidad, entre otras.

Estos factores dificultan la construcción de herramientas genéricas para desarrollar agentes, por cuanto resulta problemático capturar la funcionalidad común presente en todos esos tipos de aplicaciones y, en forma simultánea, la funcionalidad común de los diversos tipos de agentes y variedad de actividades que pueden realizar. A continuación se resumen los principales factores que dificultan la construcción de herramientas genéricas para agentes:

- los agentes pueden ser utilizados para resolver gran variedad de problemas en diversos dominios de aplicación.
- dificultad de desarrollar componentes de software responsables de las capacidades inherentes de agentes independizándose del dominio de aplicación. Por ejemplo, existen algoritmos de

planning especialmente desarrollados para determinados tipos de problemas o aplicaciones, algo similar sucede con algunos algoritmos de aprendizaje.

- variedad de capacidades que los agentes pueden tener: acción, percepción, representación y manipulación de estados mentales, reacción, deliberación, comunicación, movilidad, etc.; sumado a las innumerables formas en que es posible combinar esas capacidades para obtener diferentes tipos de agentes [72]: colaborativos, de interfaz, móviles, reactivos, deliberativos, híbridos, de información, etc. Por otro lado, cada uno esos tipos de agentes es adecuado para determinados tipos de aplicaciones. Sin embargo, estas categorías no son disjuntas. Así, por ejemplo, un agente de información puede ser móvil y deliberativo.

Como consecuencia de estos factores, en la mayoría de los casos los agentes son implementados desarrollando componentes de software en forma *ad-hoc* para cada nueva aplicación, lo que constituye uno de los mayores obstáculos para la adopción masiva de la tecnología de agentes [106].

Con el fin de reducir el esfuerzo y los costos de construcción de tales sistemas, promover la reusabilidad y la utilización de agentes, se han desarrollado herramientas que capturan la funcionalidad común presente en varios tipos de agentes. Por ejemplo: AgentBuilder [84], DECAF [52], FraMaS [6], JAF [55], JAFIMA [60], JAFMAS [21], MadKit [53] y ZEUS [71]. De esta forma, el programador construye sus agentes reutilizando la funcionalidad común definida por una herramienta y adaptándola a las necesidades requeridas.

Las herramientas existentes permiten construir agentes con un subconjunto de las capacidades de agentes mencionadas. Algunas de ellas sólo proveen un grado limitado de adaptabilidad y extensibilidad, debido a que no permiten que el programador construya agentes con otras capacidades que no sean las predefinidas por la herramienta o redefina algunos de los componentes de software responsables de las capacidades de agentes. Así, por ejemplo, ZEUS [71] permite construir agentes que usan *planning* para decidir sus acciones, sin embargo, sólo puede utilizarse el algoritmo de *planning* provisto por la herramienta; algo similar sucede con las comunicaciones, para las cuales debe utilizarse KQML, sin permitir el uso de otros lenguajes.

Considerando las herramientas existentes, pueden clasificarse en dos categorías, dependiendo de su extensibilidad:

- *herramientas no extensibles*: típicamente, están basadas en bibliotecas de componentes que implementan las capacidades de agentes. Así, por ejemplo, una herramienta puede definir componentes para comunicación con KQML, *planning* y representación de estados mentales. Para construir agentes, el programador define la funcionalidad específica de su aplicación adaptando los componentes predefinidos en función de lo permitido por la herramienta. El programador no puede extender la funcionalidad de este tipo de herramientas, por ejemplo, utilizando un algoritmo de *planning* de su preferencia o extendiendo el lenguaje de comunicación de los agentes.

  En este categoría se encuentran AgentBuilder, DECAF y ZEUS.

- *herramientas extensibles*: típicamente, definen las capacidades de agentes mediante un conjunto de componentes de software extensibles. De esta forma, el programador puede construir agentes utilizando los componentes provistos por una herramienta. Si las necesidades de una aplicación no pueden ser obtenidas a partir de la funcionalidad provista por la herramienta, entonces, el programador puede extender dicha herramienta, definiendo, por ejemplo, otros algoritmos de *planning* u otros lenguajes de comunicación.

Obsérvese que este tipo de herramientas puede resultar de gran utilidad para desarrollar agentes, debido a la variedad de aplicaciones de los mismos, sumado al hecho de que muchas de las capacidades de agentes pueden ser dependientes del dominio de aplicación (por ejemplo, un algoritmo de *planning* especialmente diseñado para una aplicación).

En este categoría se encuentran FraMaS, JAF, JAFIMA, JAFMAS y MadKit.

Dada la gran variedad de aplicaciones en las cuales es posible utilizar agentes, resultaría impensable imaginar una herramienta genérica no extensible que no limite o restrinja los tipos de agentes soportados, el dominio de aplicaciones o las capacidades de los agentes. Por otro lado, una herramienta extensible podría definir de forma genérica y reusable las características comunes de los agentes, permitiendo que el programador las adapte y extienda según las necesidades particulares de cada aplicación. Por tales razones, las herramientas extensibles parecen ser más adecuadas para desarrollar agentes que las herramientas no extensibles.

Las herramientas extensibles existentes para desarrollar agentes poseen limitaciones con respecto a:

- *extensibilidad limitada*: sólo es posible extender un subconjunto de las capacidades de agentes soportadas. Así, por ejemplo, JAF [55] define agentes que deciden qué acciones ejecutar utilizando *schedulling*. El programador no puede construir agentes con otros mecanismos de decisión; sin embargo, los mecanismos de comunicación son extensibles, permitiendo que el programador defina los protocolos y lenguajes que considere apropiados.

- *no poseen mecanismos para mantener una representación del ambiente coherente y actualizada*: la información que un agente posee sobre al ambiente, incluyendo a los agentes que lo rodean, se denominan creencias [104]. Los agentes deliberativos razonan en función de sus estados mentales, lo que incluye a las creencias. Esto implica que un agente que posee creencias incoherentes o desactualizadas respecto de lo que occure realmente en el ambiente podría actuar de manera incorrecta. Por tal razón, la representación del ambiente no debería admitir la existencia de creencias incoherentes, tal como un objeto que se encuentra en dos lugares en forma simultánea o un vehículo que se mueve y está quieto, al mismo tiempo. Además, debería definir mecanismos para que un agente actualice sus creencias utilizando sus capacidades de percepción.

  Por ejemplo, un agente a_1 que intenta tomar una caja posee un rango de percepción limitado. Dicho agente cree que la caja se encuentra en la posición (x_1, y_1) tal que esa posición está fuera del rango de sus capacidades de percepción. Otro agente a_2 toma la caja, la transporta a otra posición (x_2, y_2) y le informa a a_1 la nueva localización de la caja. El agente a_1 debería actualizar sus creencias, re¤ejando la nueva posición de la caja.

  En general, éste aspecto sólo es tratado en forma parcial por las herramientas, representando al ambiente mediante estructuras estáticas o descripciones simbólicas manipuladas por el programador en forma *ad-hoc*. Sin embargo, ninguna de las herramientas provee soporte para construir agentes que poseen una descripción simbólica del ambiente, y a la vez, mantienen esa información actualizada respecto de lo que ocurre en el ambiente utilizando las capacidades de percepción o comunicación y analizando la coherencia de las mismas.

  Obsérvese que el problema de determinar si dos creencias son incoherentes es dependiente de la aplicación [78]. Así, por ejemplo, en determinada aplicación podría ser válido creer que dos objetos se encuentran en la misma coordenada (x, y), quizás debido a que existe una coordenada z que no es utilizada por el agente. Por tal razón, es deseable que una herramienta

para construir agentes permita que el programador especifique información sobre el dominio que permita a la herramienta mantener la coherencia de las creencias.

- *no consideran la posibilidad de que un agente realice varias actividades concurrentemente* [88]. Por ejemplo, un agente podría construir un plan de acción para alcanzar sus objetivos, percibir mediante sus sensores, reaccionar a los cambios en el ambiente, modificar sus creencias y objetivos, mantener conversaciones con otros agentes, etc., todo en forma concurrente.

  Los agentes construidos con las herramientas existentes sólo pueden realizar una actividad a la vez. Por ejemplo, mientras planean analizando cuidadosamente qué acciones ejecutar para lograr sus objetivos, no perciben, no procesan comunicaciones ni son capaces de reaccionar rápidamente ante situaciones predeterminadas, tales como un choque.

- *no permiten que los agentes deliberen utilizando diferentes mecanismos concurrentes*: existen diferentes mecanismos deliberativos para que un agente decida qué acciones realizar: selección de planes estáticos en función de los objetivos [49], *scheduling* [109], planes estáticos describiendo conversaciones multi-agente [7], mecanismos de decisión utilizando la experiencia [61] y *planning on-line* [98], entre otros. Sería deseable que una herramienta genérica permitiese la construcción de agentes que razonan utilizando varios mecanismos deliberativos en forma concurrente [88], dependiendo de la naturaleza de los objetivos, del dominio de aplicación, etc.

  Por ejemplo, un agente planea la forma de caminar hacia determinado lugar de una casa, en forma simultánea, mantiene una conversación con otro agente acerca de una compra/venta de un producto, analizando cuidadosamente las propuestas y contrapropuestas realizadas por el otro agente, e intenta recordar una experiencia sobre un negocio similar resuelto satisfactoriamente en el pasado.

En el presente trabajo se propone un *framework* Java para desarrollar agentes inteligentes, denominado Brainstorm/J. Los principales objetivos del mismo consisten en reducir el esfuerzo y los costos de construcción de agentes, promoviendo la reusabilidad y la utilización de tales sistemas, tratando los aspectos no considerados por las herramientas existentes.

Los *frameworks* constituyen una de las más exitosas técnicas de reuso de código y diseño de los últimos tiempos [38]. Básicamente, un *framework* es un diseño reusable de un sistema completo, o partes del mismo, representado mediante un conjunto de clases de un lenguaje orientado a objetos. Un *framework* describe el comportamiento común de un dominio de aplicación particular [59], descomponiéndolas en objetos e interacciones entre esos objetos.

Para diseñar un *framework* pueden adoptarse dos enfoques diferentes: *conducido por ejemplos* o *conducido por un modelo de dominio de aplicación*. El primero consiste en abstraer las características comunes de un conjunto de aplicaciones pertenecientes al mismo dominio [25]. La selección de un conjunto de aplicaciones representativas del dominio de aplicación y el posterior análisis de otras aplicaciones es de fundamental importancia para que el *framework* posea abstracciones comunes a dicho dominio. En tal sentido, los frameworks diseñados de esta forma son refinados en etapas sucesivas, conforme se adquiere mayor conocimiento sobre el dominio de aplicación.

El segundo enfoque tiene sus orígenes en el concepto de *arquitectura de software* y *estilos arquitectónicos*. La arquitectura de un sistema de software se refiere a la división del mismo en componentes y los patrones de interacción entre ellos [85]. Una *arquitectura específica del dominio* define un conjunto de componentes e interfaces que caracterizan un dominio de aplicación específico, por ejemplo, agentes.

Un patrón organizacional de componentes e interacciones que se repite en varios sistemas se denomina *estilo arquitectónico*. Por ejemplo, cliente/servidor, la organización por niveles de la arquitectura para comunicaciones de la OSI, *pipes&filters*, repositorio, etc. Así, un estilo arquitectónico caracteriza una familia de sistemas relacionados por propiedades estructurales y semánticas comunes [46]. Los estilos arquitectónicos promueven el reuso de diseño, por cuanto permiten reaplicar soluciones conocidas y bien estudiadas a nuevos problemas. Al mismo tiempo, los aspectos invariantes de un estilo arquitectónico pueden ser aprovechados para reusar implementaciones.

Así, otra manera de desarrollar un *framework* consiste en partir de un modelo de dominio abstracto y cero o más estilos arquitectónicos adecuados para el dominio de aplicación, de lo cual se obtiene una primer materialización arquitectónica. Luego, se refina y materializa la arquitectura mediante un conjunto de clases que conforman el *framework*. La aplicación de patrones de diseño [45] durante esta etapa puede mejorar la ¤exibilidad y reusabilidad del *framework*.

Los estudios comparativos entre ambos procesos [16] han mostrado que el segundo de ellos produce *frameworks* más estables en menos etapas de refinamiento. Esto se debe a que las principales clases que componen el *framework* son definidas por la arquitectura. Algo similar ocurre con las clases que implementan el ¤ujo de control abstracto del dominio de aplicación.

En particular, en el dominio de agentes inteligentes existen innumerables modelos [81, 86, 104] y arquitecturas [42, 49, 4]. De esta forma, en lugar de comenzar a diseñar un *framework* a partir de un análisis de dominio, es posible reusar el diseño de una arquitectura específica de dominio existente, reduciendo el tiempo de desarrollo, el esfuerzo y los costos.

El *framework* propuesto en este trabajo está basado en la arquitectura de agentes Brainstorm [4]. Dicha arquitectura prescribe agentes capaces de percibir, reaccionar, comunicarse, razonar y aprender. Brainstorm posee una gran ¤exibilidad, lo que permite, por ejemplo, componer las capacidades de los agentes en forma ¤exible para construir diferentes tipos de agentes.

Brainstorm posee una arquitectura re¤exiva basada en meta-objetos [66], en la que un agente es representado por un objeto al cual se le asocian un conjunto meta-objetos responsables de las capacidades de agentes, como se muestra en la figura 1.1(a). Cada meta-objeto es capaz de interceptar los mensajes recibidos por los objetos a los cuales está asociado, con el fin de modificar el comportamiento de los mismos, o simplemente observar el ¤ujo de mensajes.

Brainstorm estructura un sistema multi-agente (SMA) como un conjunto de objetos y objetos-agente. Los objetos-agente son objetos a los cuales se les ha asociado un nivel re¤exivo con meta-objetos responsables de las capacidades de agentes. El nivel re¤exivo consta de tres meta-niveles. Cada uno de los meta-niveles es capaz de actuar y alterar las computaciones del nivel inmediato inferior. Así, por ejemplo, mediante el primer meta-nivel, el objeto *object1* situado en el nivel base, adquiere capacidades para representar y manipular estados mentales (*Brain* y *LogicKnowledge*), percibir otros objetos (*Perceptor*), comunicarse (*Communicator*) y detectar posibles situaciones de interés para el agente (*SituationManager*).

El segundo meta-nivel es responsable del comportamiento reactivo (*Reactor*) o deliberativo (*Deliberator*) del agente. Finalmente, el tercer meta-nivel actúa sobre el segundo en función de la experiencia adquirida.

La primera fase en el desarrollo de Brainstorm/J consistió en materializar la arquitectura Brainstorm por medio de un conjunto de clases abstractas. Luego, se realizaron varias adaptaciones y extensiones al *framework*, como se observa en la figura 1.1(b). Así, por ejemplo, se agregó un componente para

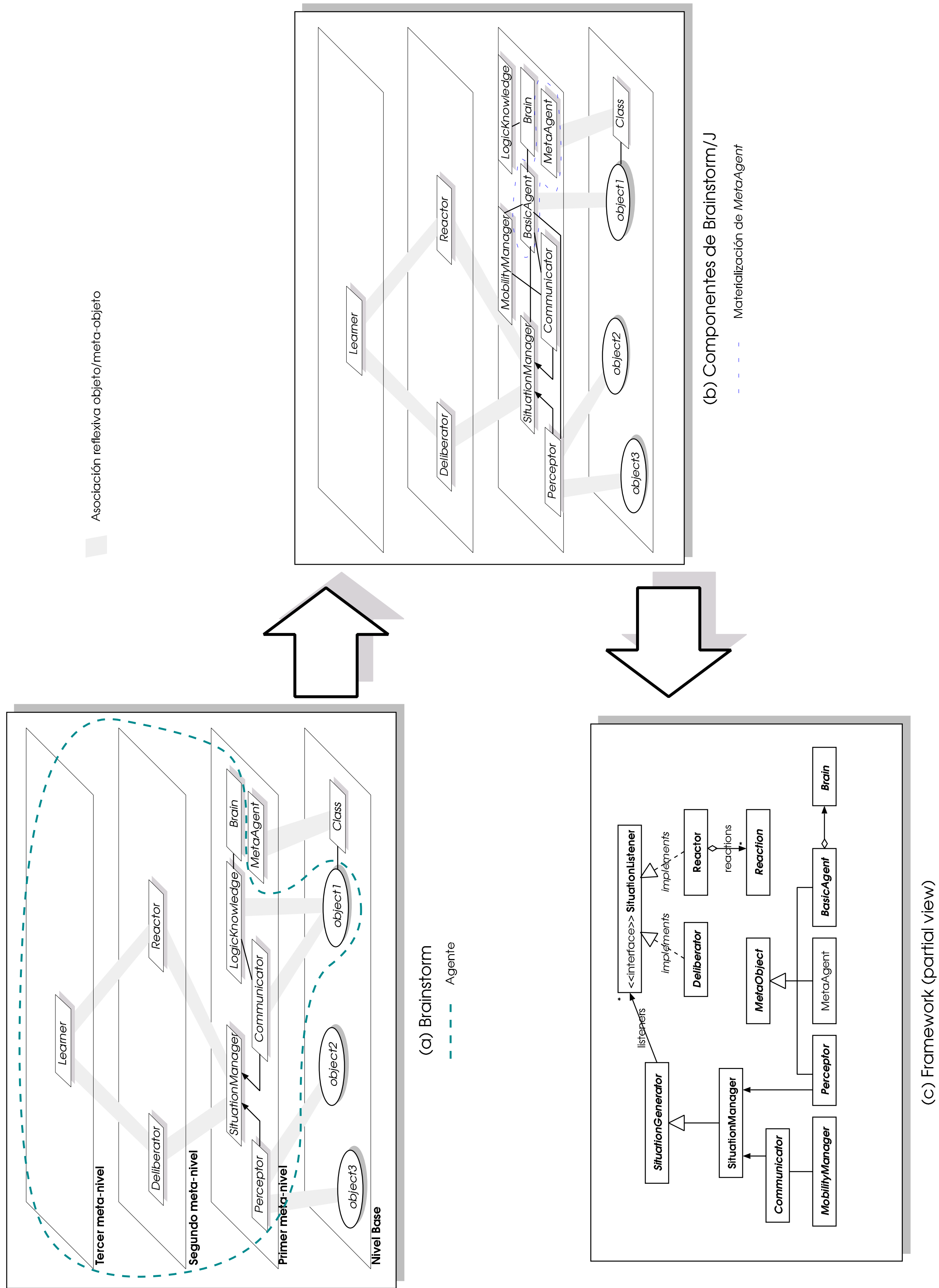


Figura 1.1: Brainstorm y su materialización en Brainstorm/J

permitir que los agentes migren entre diferentes sitios de una red (*MobilityManager*) y se extendió el componente de comunicación para soportar agentes físicamente distribuidos.

La materialización de los componentes responsables de la representación y manipulación de estados mentales se realizó mediante un lenguaje multi-paradigma basado en Java y Prolog [5, 1], llamado JavaLog. Por otro lado, se desarrolló un *framework* basado en LuthierMOPS [17] para soportar meta-objetos en Java.

En la figura 1.1(c) se muestra un diagrama UML[1] con las principales clases del *framework* y sus relaciones, las cuales constituyen la estructura abstracta de Brainstorm/J.

Para construir agentes con el *framework*, es necesario instanciar y especializar algunas de las clases definidas por el mismo. El programador debe especificar el comportamiento dependiente de la aplicación definiendo los métodos abstractos del *framework*. Básicamente, un método abstracto es un *hueco* que debe ser completado por el programador en el momento en que instancia el *framework* para construir una aplicación.

El ¤ujo de control entre los componentes de los agentes está codificado en los métodos *template*. Básicamente, un método *template* define una operación en forma independiente de la aplicación, describiendo interacciones en las que intervienen diversos objetos. Los métodos *template* se caracterizan por invocar a uno o más métodos abstractos. Los métodos abstractos son los puntos en los cuales el programador especifica el comportamiento particular de una aplicación. De esta forma, el programador se desliga de la responsabilidad de codificar el comportamiento común a todos los tipos de agentes, tales como mecanismos de comunicación o manipulación de estados mentales. Adicionalmente, por las características inherentes de la programación orientada a objetos, los *frameworks* son adaptables y extensibles, en el sentido de que es posible incorporar nueva funcionalidad o extender la existente.

En la figura 1.2 se muestra un diagrama con las principales clases involucradas en la creación de los agentes. Los agentes son creados por el método *template* `createAgent` de la clase `MetaAgent`. Obsérvese que en dicho método se invocan los métodos abstractos `initialize`, `initReactions` e `initDeliberation` de la clase `MetaAgent`. Dichos métodos deben ser definidos por el programador para inicializar los componentes de los agentes. Obsérvese que esos métodos son particulares de cada agente que se desarrolle con el *framework*, por tal razón, no forman parte del mismo.

Resumiendo, el *framework* Brainstorm/J permite construir agentes con capacidades de percepción, representación y manipulación de los estados mentales, deliberación, reacción, comunicación, movilidad; combinando esas capacidades de diversas formas para obtener diferentes tipos de agentes. El *framework* especifica componentes de software adaptables y ¤exibles responsables de dichas capacidades. Esos componentes pueden ser adaptados y extendidos para construir agentes, permitiendo que el programador defina la funcionalidad de sus aplicaciones a partir de los componentes reusables provistos por el *framework*.

Brainstorm/J permite que los agentes mantengan una representación del medio ambiente en que se encuentran, asegurando la coherencia de la misma y razonando en función de ella. Los agentes pueden deliberar y, en forma simultánea, percibir, mantener conversaciones con otros agentes, actuar, etc. Además, un agente puede razonar, en forma simultánea, sobre cómo lograr sus objetivos utilizando varios mecanismos deliberativos.

En la siguiente sección se describe la organización del trabajo.

[1] En el apéndice D se describe UML.

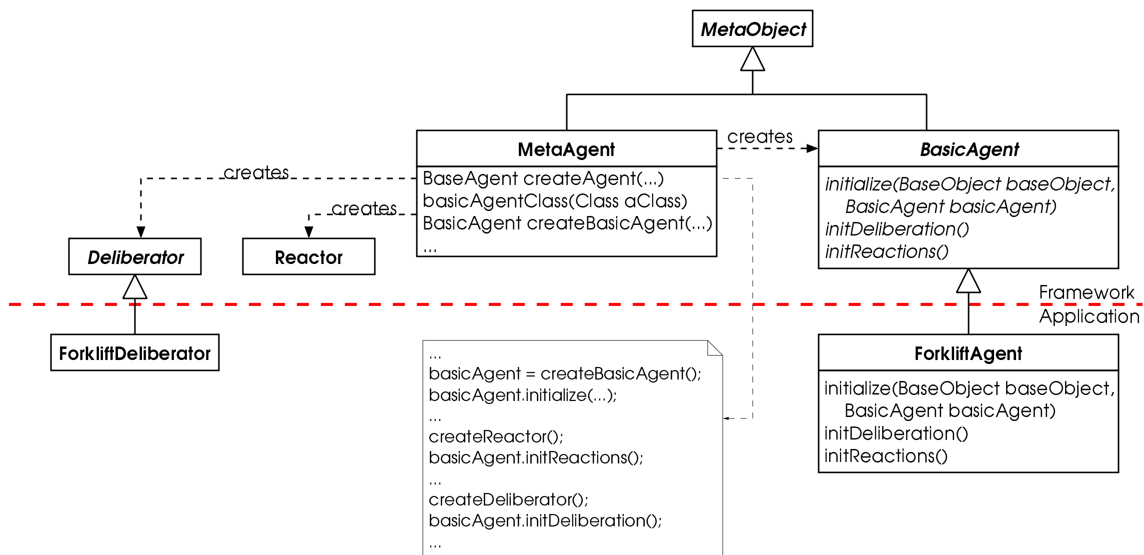


Figura 1.2: Método *template* responsable de crear e inicializar un agente

1.1 Organización de este trabajo

En el siguiente capítulo se definen los conceptos básicos en los que se basa el trabajo: agentes, *frameworks*, diseño de *frameworks* y la arquitectura Brainstorm. En el capítulo 3 se describen algunas de las herramientas existentes para desarrollar agentes, con el fin de establecer sus principales capacidades y falencias. A continuación, en el capítulo 4 se presenta el *framework* Brainstorm/J. En el capítulo 5 se describe el diseño e implementación de dos sistemas multi-agente desarrollados con Brainstorm/J y se comparan con implementaciones realizadas con otras herramientas. Luego, en el capítulo 6 se completa la descripción del *framework*, especificando formalmente sus principales clases mediante Object-Z. Finalmente, en el capítulo 7 se presentan las conclusiones y trabajos futuros.

El trabajo posee varios apéndices que describen aspectos que no son centrales al mismo. En el apéndice A se describe el *framework* para meta-objetos JMOP utilizado en la materialización de Brainstorm. A continuación, en el apéndice B se presenta el lenguaje multi-paradigma JavaLog. En el apéndice C se describe brevemente el lenguaje de especificación Object-Z. Finalmente, en el apéndice D se describe la notación UML.

CAPÍTULO 2

Contexto

El principal objetivo de este capítulo consiste en definir los conceptos en los cuales se basa el presente trabajo:

- Agentes.
- *Frameworks*.
- Diseño de *frameworks*.
- La arquitectura Brainstorm.

El capítulo está organizado de la siguiente forma: en la sección 2.1 se define qué es un agente; a continuación, en la sección 2.2 se describe una técnica de reuso de diseño y de código denominada *frameworks*; luego, en la sección 2.3 se analiza el proceso de diseño de *frameworks*; en la sección 2.4 se describe la arquitectura de agentes Brainstorm; finalmente, en la sección 2.5 se presentan las conclusiones del capítulo.

2.1 Agentes

Un agente de software es una entidad que funciona continua y autónomamente en un medio ambiente generalmente habitado por otros agentes y procesos [87]. La continuidad y autonomía están relacionadas con la posibilidad de que un agente realice sus actividades de una manera adaptativa e inteligente, percibiendo los cambios en el ambiente sin necesidad de que un humano lo guíe ni intervenga constantemente en sus decisiones [9]. Idealmente, un agente también debería ser capar de aprender de la experiencia, mejorando su performance y adaptando su comportamiento. Además, un agente que comparte el medio ambiente con otros agentes y procesos debería ser capaz de comunicarse y cooperar con ellos [105], quizás trasladándose entre diferentes lugares [99].

La mayoría de los agentes existentes poseen sólo algunas de las características mencionadas. Así, el término *"agente de software"* incluye una amplia variedad de agentes con muy diversas características, por lo que se considera que un agente de software puede poseer uno o más de los atributos que a continuación se enumeran [72, 37, 64]:

- *Capacidad de acción*: un agente es capaz de actuar, modificando su medio ambiente.
- *Percepción y reactividad*: la habilidad de percibir su medio ambiente y actuar selectivamente.
- *Comunicación a nivel de conocimiento*: capacidad de comunicarse con agentes y personas con lenguajes de alto nivel análogos a los *"speech acts"* humanos, en lugar de protocolos programa-programa. Los agentes con esta capacidad pueden compartir conocimiento, creencias, objetivos, planes, etc.
- *Comportamiento colaborativo*: capacidad de trabajar con otros agentes para alcanzar un objetivo común.
- *Capacidad de manipular sus estados mentales*: las desiciones de un agente pueden estar basadas en sus estados mentales (creencias, objetivos, intenciones, compromisos, etc). Por lo tanto es necesario expresar y manipular estos componentes, por ejemplo, en forma simbólica.
- *Movilidad*: capacidad de migrar autónomamente desde una plataforma huésped a otra.
- *Deliberación*, *racionalidad*: capacidad de un agente de actuar determinando qué acciones realizar con el fin de alcanzar sus objetivos y evitando tomar desiciones que le impidan alcanzarlos.
- *Autonomía*: capacidad de actuar sin intervención humana o de otros sistemas con el fin de alcanzar sus objetivos. Una característica clave de los agentes autónomos es su habilidad de tomar la iniciativa de sus actos, en lugar de actuar sólo en respuesta a su medio ambiente.
- *Aprendizaje (adaptabilidad)*: ser capaz de aprender, modificando su comportamiento en base a la experiencia.
- *Continuidad temporal*: persistencia de identidad y estado en largos períodos de tiempo.

Los intentos por caracterizar el espacio de tipos de agentes [43, 64] han originado innumerables taxonomías que consideran las diferentes combinaciones de atributos que un agente puede poseer. Una clasificación clásica [31, 72], considera tres tipos de agentes: *deliberativo*, *reactivo* e *híbrido*. Los agentes deliberativos poseen un modelo mental simbólico del medio ambiente y de sí mismos, del cual derivan sus acciones utilizando *planning* e interactuando con otros agentes. Por otro lado, los agentes reactivos no poseen ningún modelo mental simbólico, sino que actúan utilizando un comportamiento de tipo estímulo/respuesta, reacionando al estado actual del ambiente en el cual se encuentran. Los agentes híbridos poseen las características de los agentes deliberativos y reactivos en forma simultánea.

Otras clasificaciones, consideran combinaciones de los atributos previamente presentados. Por ejemplo, en [9] se define un espacio tridimensional compuesto por (figura 2.1):

- *agencia*: es el grado de autonomía. Puede ser medido por la naturaleza de las interacciones. Como mínimo, un agente debe operar en forma asincrónica. Esto mejora si el agente representa al usuario e interactúa con datos, aplicaciones, servicios y otros agentes.
- *inteligencia*: es el grado de razonamiento y aprendizaje que un agente posee. Como mínimo debe considerar las preferencias del usuario. Los mayores niveles de inteligencia incluyen un

modelo del usuario y razonamiento, con capacidad de aprender y adaptarse al ambiente.

- *movilidad*: se refiere a la capacidad de transportarse entre diferentes sitios de una red [44].

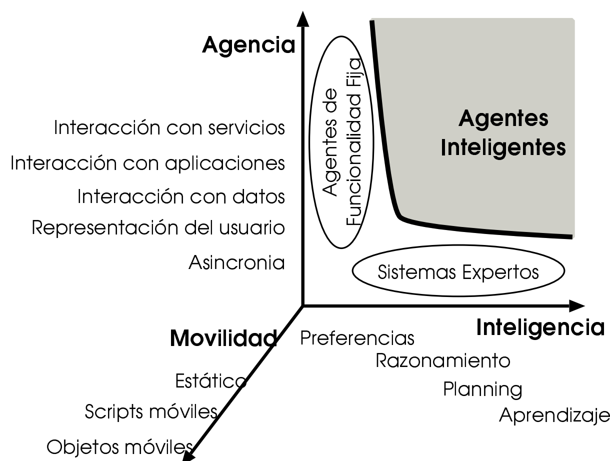


Figura 2.1: Clasi£cación de agentes [9]

De la figura 2.1 puede concluirse que es posible construir una amplia variedad agentes de características muy diferentes. El principal objetivo de este trabajo consiste en capturar las características comunes y generales de los agentes *inteligentes* en un *framework*.

Básicamente, los agentes inteligentes (también conocidos como cognitivos) poseen capacidades deliberativas. Estos agentes se caracterizan por poseer un modelo mental simbólico de su medio ambiente, de otros agentes y de sí mismos, a partir del cual deciden qué acciones realizar. Dicho modelo mental es construído a partir de las percepciones realizadas por el agente, las comunicaciones recibidas, y las propias acciones del agente.

En general, se considera que los agentes inteligentes son entidades que "parecen estar sujetas a creencias, deseos, intenciones, etc" [104]. Esto proviene de las teorías filosóficas y psicológicas por las cuales el comportamiento humano es explicado y predecido a través de la atribución de *actitudes* tales como *creer* o *desear*. Tales sistemas son denominados *sistemas intencionales*.

Las actitudes mentales de un agente pueden ser clasificadas como [104]:

- *actitudes de información*: están relacionadas con la información que el agente posee sobre el mundo en el que habita. Por ejemplo, creencias y conocimiento.
- *actitudes de orientación*: guían las acciones del agente. Por ejemplo, deseos (objetivos), intenciones, obligaciones, compromisos y elecciones.

Un agente puede ser descripto por, al menos, una actitud de información y una de orientación [104]. Nótese que las actitudes de orientación están altamente relacionadas con las de información, ya que, por ejemplo, un agente realiza elecciones y forma intenciones basado en la información que posee sobre el mundo.

La forma de actuar de un agente tiene una relación directa con la manera en que relaciona sus actitudes mentales. Por ejemplo, el modelo BDI[1] [10] considera que los agentes poseen tres actitudes

[1] Belief-Desire-Intention.

mentales: *creencias*, *deseos* e *intenciones*. Las creencias corresponden a la información que el agente tiene acerca del mundo. Los deseos corresponden a las tareas que el agente tiene que realizar (sus objetivos). Las intenciones representan a los deseos que el agente se ha comprometido llevar a cabo. Intuitivamente, un agente no será capaz de lograr todos sus deseos. Por esta razón, selecciona un subconjunto de ellos y dedica su esfuerzo para llevarlos a cabo. Esos deseos seleccionados son llamados intenciones. Típicamente, un agente intentará satisfacer una intención hasta que crea que la ha hecho, o crea que no es posible alcanzarla. Así, el agente podría ejecutar una o más acciones para satisfacer una intención.

El modelo BDI es la base de varios lenguajes de programación [79, 86] y arquitecturas [49, 11] de agentes inteligentes. Esto se debe a que captura las características esenciales del razonamiento, a la vez que resulta ser un modelo intuitivo y sencillo de comprender [103]. Por otro lado, existen numerosas formalizaciones del proceso de razonamiento que consideran otras actitudes mentales u otras relaciones entre ellas [80, 81, 100].

En las siguientes secciones se describen con mayor detalle las capacidades que caracterizan a los agentes inteligentes.

2.1.1 Capacidad de acción

Un agente tiene un repertorio de acciones disponibles [72]. Este conjunto de acciones representan la capacidad *efectora* del agente: su capacidad de modificar el medio ambiente en que se encuentra. Generalmente, algunas de las acciones de un agente pueden no estar disponibles en todo momento. Por ejemplo, una acción "tomar una carga" sólo es aplicable en situaciones en que la carga sea suficientemente liviana como para que el agente pueda tomarla, y dicha carga se encuentre al alcance del agente. Así, las acciones poseen precondiciones asociadas, las cuales definen las posibles situaciones en las que pueden ser aplicadas.

En dominios de cierta complejidad, tales como el mundo físico real o Internet, un agente no posee un control completo de su medio ambiente. En el mejor de los casos, sólo tendrá un control parcial, en el sentido de que podrá in¤uenciar lo que sucede. Desde el punto de vista del agente, esto significa que una misma acción ejecutada dos veces en circunstancias aparentemente idénticas, podría producir efectos diferentes, o incluso fallar. Así, los agentes deben estar diseñados para fallar, al menos en ambientes no triviales [64].

2.1.2 Percepción

La capacidad de percepción de un agente se refiere a la habilidad de percibir los acontecimientos que ocurren en su ambiente. Mediante dicha capacidad, un agente es capaz de conocer lo que ocurre en el ambiente, sin necesidad de que otros agentes le informen lo que sucede mediante comunicaciones. Por ejemplo, un robot que explora un terreno que desconoce debería ser capaz de percibir y distinguir los objetos que lo rodean. El robot debería poder detectar situaciones tales como colisiones, o percibir las actividades realizadas por otros robots, tales como la dirección y velocidad a la que se desplazan. Por ejemplo, un robot que tiene que tomar una caja para llevarla a un camión, podría percibir si esa caja fue movida por otro robot, de forma de no intentar alcanzarla y así, no realizar acciones redundantes.

La percepción puede ser afectada por el tipo de ambiente en el cual se encuentra el agente. Básicamente, es posible clasificar al ambiente como *accesible* o *inaccesible* [103]. Un ambiente accesible es

aquel en el que un agente puede obtener información actualizada, completa y correcta mediante sus capacidades perceptivas. Sin embargo, esto no es posible en la mayoría de los ambientes moderadamente complejos, los cuales son llamados inaccesibles (ej.: el mundo físico, o Internet).

2.1.3 Comunicación

La comunicación permite que los agentes de un sistema multi-agente intercambien información, posibilitando la coordinación sus acciones con el fin de cooperar o negociar frente a con¤ictos.

Los agentes inteligentes, generalmente se comunican a *nivel de conocimiento* [47], es decir que son capaces de intercambiar conocimiento, solicitar la colaboración de otros agentes, ofrecer servicios, entablar conversaciones, etc. Básicamente, en este tipo de comunicaciones los agentes utilizan un lenguaje de comunicación común de alto nivel que les permite interactuar en término de creencias, intenciones, planes, conocimiento, etc.

En la literatura existen varios lenguajes para comunicar agentes. Los más difundidos son ACL [93] y KQML [40]. En base a ellos, se han desarrollado lenguajes, tales como COOL [7], que permiten especificar *conversaciones* o protocolos de interacción. A continuación se describen KQML y COOL.

2.1.3.1 KQML

KQML (Knowledge Query and Manipulation Language) es un lenguaje de alto nivel diseñado para soportar interoperabilidad entre agentes inteligentes en aplicaciones distribuidas [40]. KQML especifica un formato de mensajes y un conjunto de protocolos de comunicación para permitir que diferentes agentes intercambien información y conocimiento.

Un mensaje KQML consiste de una performativa, el contenido del mensaje y un conjunto de argumentos opcionales. La performativa sugiere explícitamente una acción. Por ejemplo:

```
(tell :language prolog
      :ontology Genealogy
      :in-reply-to q1
      :sender gen1
      :receiver gen-DB
      :content "father(John,Alice)")
```

En la terminología KQML, *tell* es una performativa. El valor del campo `:content` contiene una expresión en algún lenguaje interpretado, que en este caso es Prolog. Los otros campos contienen valores que proveen un contexto para la interpretación del contenido del mensaje. En este caso el agente `gen1` le está diciendo al agente `gen-DB` que `father(John,Alice)`. Esta es una respuesta a un mensaje KQML anterior identificado por `q1`. La ontología `Genealogy` puede proveer información adicional acerca de la interpretación del contenido del mensaje. En este caso, `Genealogy` define el significado de la relación `father` y los objetos del dominio (personas).

En la tabla 2.2 se listan algunas de las performativas KQML junto con su significado.

Nombre	Signi£cado para un emisor S y un receptort R con una Base de Conocimiento Virtual (BCV)
achieve	S desea que R haga verdadero algo en su medio ambiente
advertise	S desea dar a conocer a R que S puede y procesará los mensajes como el que está en :content
ask-one	S desea una de las instanciaciones de :content verdadera según la BCV de R
ask-all	S desea todas las instanciaciones de :content verdaderas según la BCV de R
ask-if	S desea saber si :content se encuentra en la BCV de R
broadcast	S desea que R envíe un mensaje a todos los agentes que conoce
broker-all	S desea que R encuentre todas las respuestas a <performativa> (algún otro agente va a proveer la respuesta)
broker-one	S desea que R encuentre una respuesta a <performativa> (algún otro agente va a proveer la respuesta)
delete-all	S desea que R elimine de su BCV todas las sentencias coincidentes
delete-one	S desea que R elimine de su BCV una sentencia coincidente
deny	la negación de la sentencia está en la BCV de S
discard	S no quiere recibir las respuestas faltantes a un mensaje multi-respuesta previo
error	S considera que el mensaje anterior de R es incorrecto
eos	marca de £n de secuencia de respuesta multiple (stream-all)
forward	S desea que R reenvíe el mensaje al agente :to
insert	S solicita a R que agrege :content a su BCV
next	S desea la siguiente respuesta de R al mensaje previamente enviado por S
ready	S está listo para responder a un mensaje previamente recibido de R
recommend-all	S desea aprender sobre todos los agentes capaz de responder a una <performativa>
recommend-one	S desea aprender sobre un agente capaz de responder a una <performativa>
recruit-all	S desea que R obtenga todos los agentes adecuados que responden a una <performativa>
recruit-one	S desea que R obtenga un agente adecuado que responda a una <performativa>
register	S anuncia a R su presencia y nombre simbólico
rest	S desea las restantes respuestas de R
sorry	S entiende el mensaje de R, pero no puede proveer una respuesta más informativa
standby	S desea que R anuncie que está listo a proveer una respuesta al mensage :content
stream-all	versión de ask-all de respuesta múltiple
subscribe	S desea updates to Rs response to a performative
tell	la sentencia está la la BCV de S
transport-address	S asocia su nombre simbólico con una nueva dirección de transporte
unachieve	S desea que R anule algo que logró previamente
undelete	S desea que R anule un delete previo
uninsert	S desea que R anule un insert previo
unregister	S desea que R anule algo previamente registrado
untell	la sentencia no está en la BCV de S

Tabla 2.2: Performativas KQML

2.1.3.2 COOL

COOL [7] es un lenguaje basado en KQML que permite representar y utilizar conocimiento sobre interacciones multi-agente.

La idea de COOL nace en el hecho de que las interacciones entre agentes son llevadas a cabo a través de *conversaciones*. En una conversación, los agentes intercambian mensajes de acuerdo a convenciones previamente establecidas, cambian de estado y realizan acciones.

COOL provee construcciones para definir conversaciones genéricas, las cuales son representadas mediante *clases de conversaciones*. Una clase de conversación es un descripción de las interacciones y acciones que un agente realiza durante una conversación para alcanzar sus objetivos. Una clase de conversación se define mediante un autómata finito determinístico que representa los posibles estados y transiciones que podrían realizarse durante un diálogo con otros agentes.

Por ejemplo, en la figura 2.2 se muestra una representación de una clase de conversación entre un agente cliente y otro vendedor, desde el punto de vista del cliente. La conversación comienza cuando el cliente envía una orden de compra al vendedor. Luego de esto, el cliente queda en estado `propuesta` en el cual puede recibir preguntas relativas a la orden del vendedor. Si el vendedor no tiene más preguntas, envía un mensaje indicando que está trabajando en la orden, con lo que el cliente pasa a estado `trabajando`. En este instante, el vendedor puede rechazar o aceptar la orden, o realizar una contrapropuesta. Si acepta (estado `aceptada`) el cliente espera a que se ejecute la orden. En el estado `contrap`, el cliente puede rechazar la última contrapropuesta o realizar otra propuesta, retornando al estado `propuesta`. En el estado `contrap`, el cliente también puede formular preguntas al vendedor acerca de la última contrapropuesta.

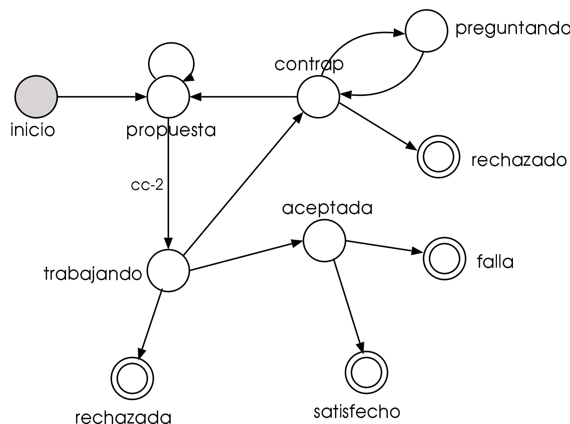


Figura 2.2: Representación de una conversación cliente-vendedor desde el punto de vista del cliente

Las transiciones entre los estados del autómata se representan mediante *reglas conversacionales*. Una regla consta de una condición de activación sobre un mensaje recibido y una acción que es ejecutada si la condición de activación es verdadera. Por ejemplo, la regla `cc-2` mostrada a continuación representa la transición entre los estados `propuesta` y `trabajando`:

```
(def-conversation-rule 'cc-2
  :name 'cc-2
  :current-state 'propuesta
  :received '(tell :sender vendedor
```

```
                    :content '(working on it))
    :next-state 'trabajando
  )
```

El campo :received especifica la condición de activación de la regla. Así, la conversación pasa del estado `propuesta` al estado `trabajando` si el cliente recibe un mensaje del vendedor indicando que está trabajando en la orden.

Cuando un agente COOL recibe un mensaje, determina a qué conversación pertenece ese mensaje. Si no existe ninguna, se determina qué *clase de conversación* podría iniciarse con ese mensaje. Luego, se crea una instancia de dicha clase, y se inicia la conversación siguiendo el protocolo especificado en la clase.

Para que dos agentes COOL dialoguen, las conversaciones de cada uno de ellos deben generar mensajes que el otro agente sea capaz de procesar. Por ejemplo, en el caso del cliente y el vendedor, el agente vendedor deberá generar mensajes que el cliente entienda y que sigan el mismo protocolo. Para lograrlo, se puede definir una clase de conversación *simétrica* a la de cliente en el vendedor, es decir, que cuando uno envía un mensaje solicitando algo, el otro responde.

2.1.4 Movilidad

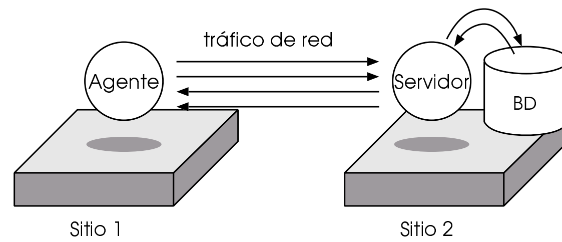
Movilidad es la capacidad de migrar una computación de un desde un sitio hacia otro durante su ejecución [99]. Diversos autores del área de sistemas distribuidos [50, 57, 44] han dado el nombre de *agentes móviles* a los sistemas con capacidad de migrar su computación durante su ejecución. Sin embargo, en el contexto de los sistemas distribuidos, el término *agente* no es utilizado con el mismo significado que en el ámbito de sistemas multi-agente. En este trabajo se utiliza el término *agente móvil* para denotar un componente de software capaz de transportarse en la red, aunque dicho agente podría no tener características típicas de un agente, sino que lo único que lo caracteriza es su capacidad de migrar autónomamente. Nótese que la movilidad puede ser considerada una característica ortogonal a las capacidades de agentes tales como reacción, deliberación, percepción, etc.

En un sistema de agentes móviles, es posible distinguir dos componentes claramente diferenciados [76]:

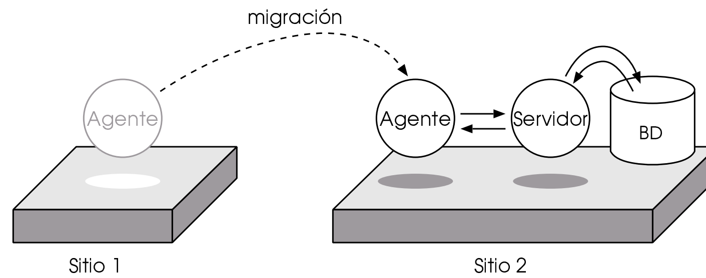
- *Agente Móvil o Unidad de Ejecución (UE)*: es un componente de software conteniendo al menos un *thread* de ejecución, que es capaz de migrar autónomamente a un sitio diferente.
- *Entorno Computacional (EC)*: es un componente que provee un entorno controlado para ejecutar agentes móviles. Típicamente, un EC ofrece un conjunto de servicios y recursos a los agentes móviles que se ejecutan en el mismo. Además, el entorno computacional está asociado con el sitio en el cual se encuentra.

  El EC provee los medios para migrar agentes, aceptar agentes móviles provenientes de otros sitios, asegurar que los agentes no interfieran entre sí, administrar los recursos y asociacionesxentre agentes-recursos, etc.

En la figura 2.3(b) se muestra un agente móvil que se desplaza desde su sitio de origen a otro sitio en el que hay un servidor de base de datos. La interacción con el servidor de base de datos se realiza localmente, por lo que no hay costo extra asociado al tráfico de red típico de las arquitecturas cliente/servidor sin agentes móviles (figura 2.3(a)).



(a) Trá£co de red en una arquitecura cliente/servidor



(b) Reducción del trá£co de red utilizando agentes móviles

Figura 2.3: Agentes móviles para reducir el trá£co de red

Los agentes móviles exhiben una serie de características que los hacen ideales para aprovechar el potencial de las redes actuales [22]. En primer lugar, permiten construir aplicaciones compuestas de múltiples componentes móviles que utilizan sus capacidades migratorias para minimizar el tráfico de red. En segundo lugar, proveen un mecanismo simple para escalar y distribuir sistemas y partes de los mismos sin mayores dificultades. El grado de ¤exibilidad y dinamicidad potencial que ofrece este nuevo paradigma es, por cierto, prometedor. Sin embargo, dichas características vienen acompañadas de nuevos problemas de diseño [77, 76]: mecanismos de movilidad, seguridad, traducción, administración de recursos, comunicación, interoperabilidad, etc., que dificultan el desarrollo de sistemas con agentes móviles.

2.1.5 Reacción

Un agente con capacidad de reacción tiene la habilidad de actuar rápidamente en función de las percepciones realizadas sobre el medio ambiente en que se encuentra para satisfacer sus objetivos de diseño [103]. Esto significa que es capaz de ejecutar una o más acciones en respuesta a los eventos ocurridos en el ambiente, si esos eventos afectan sus objetivos o los procedimientos que está ejecutando para alcanzar dichos objetivos.

Lo que caracteriza al comportamiento reactivo es el proceso de selección de acciones, el cual se realiza sin razonar y de una manera muy simple. Por ejemplo, en la arquitectura *subsumption* [13], cada agente posee un conjunto de reglas condición/acción organizadas por niveles. Ante un evento se verifica la condición de activación de la regla de nivel superior. Si se cumple, se ejecuta la acción

correspondiente; en caso contrario se verifica la regla del nivel siguiente.

Un ejemplo típico de comportamiento reactivo puede encontrarse en la naturaleza: las hormigas exhiben un comportamiento que puede ser modelado mediante un conjunto de reglas estímulo/respuesta. Así, una hormiga recolectora de alimentos que percibe un obstáculo frente a ella cambia de dirección; si se encuentra con un alimento, lo toma; si percibe un olor que le interesa, camina hacia la dirección en la que ese olor es mayor; etc.

2.1.6 Deliberación

La deliberación puede definirse como el proceso por el cual un agente razona acerca de sus estados mentales y las relaciones entre ellos para decidir qué hacer [80]. En dicho proceso, el agente podría tener que cumplir con varios, y posiblemente con¤ictivos, objetivos, para lo cual deberá tomar decisiones acerca de cómo lograrlos para obtener el efecto esperado. Esta elección, depende directamente de las creencias del agente acerca de las situaciones presentes y futuras, y de las intenciones o compromisos adquiridos en el pasado. Sin embargo, el conocimiento del agente sobre el mundo es frecuentemente incompleto, por lo que es necesario que el mismo posea la capacidad de asumir cosas sobre la posible ocurrencia de eventos o el comportamiento de otros agentes.

Básicamente, la deliberación tiene la función de decidir qué tareas realizar para cumplir con los objetivos del agente, cómo llevarlas a cabo, y cuándo ejecutarlas. Los medios para tomar esas decisiones pueden ser variados. Así, por ejemplo, podría utilizarse *planning* [98], razonamiento probabilístico [62] o *scheduling* [109], entre otros.

Entre las acciones que un agente puede realizar para cumplir sus objetivos se encuentran las capacidades comportamentales básicas, las comunicaciones y las tareas internas al agente que sólo afectan su estado mental. Esto implica, que mediante el proceso deliberativo se decide, ante una comunicación, si se responde o no, y en caso afirmativo cómo se responde. Así, por ejemplo, el conocimiento sobre las interacciones expresado mediante COOL puede ser parte del comportamiento deliberativo de un agente.

En la siguiente sección se describe la técnica de *planning*, la cual es utilizada por ciertos tipos de agentes agentes para determinar qué acciones ejecutar para cumplir con sus objetivos.

2.1.6.1 Planning

Un plan es una secuencia de acciones que transforman un estado inicial en un estado final u objetivo. El problema de *planning* consiste en encontrar las acciones tales que al ser aplicadas al estado inicial producen el estado objetivo [98]. Las acciones con las cuales se construye un plan usualmente forman parte de las capacidades comportamentales básicas del agente, las comunicaciones o las tareas internas al agente que sólo afectan su estado mental.

La capacidad de un agente de planear qué acciones realizar está estrechamente vinculada con la representación interna que el agente posee acerca del mundo y las acciones que pueden realizarse sobre él. Esto se debe a que este conocimiento es utilizado para construir el plan para alcanzar los objetivos.

Para ejemplificar la utilización de *planning*, se considerará un problema clásico. El problema consiste en transportar un conjunto de cargas situadas en varias ciudades a sus respectivos destinos por medio de cohetes. Un cohete sólo puede llevar una carga por viaje; además, puede viajar de una ciudad a

otra y cargar/descargar una carga. En la figura 2.4 se muestran las definiciones de dichas acciones en forma de *operadores* o *esquemas de acción*. Un operador en un esquema de acción que puede ser instanciado para originar una acción. Los operadores incluyen una especificación acerca de sus parámetros, precondiciones y efectos.

mover(R, $Origen$, $Destino$)
PARÁMETROS: cohete(R)$\wedge$ciudad($Origen$)$\wedge$ciudad($Destino$)$\wedge Origen \neq Destino$
PRECONDICIÓN: en(R, $Origen$)$\wedge$tieneCombustible(R)
EFECTO: en(R, $Destino$)$\wedge\neg$en(R, $Origen$)$\wedge\neg$tieneCombustible(R)
descargar(R, P, C)
PARÁMETROS: cohete(R), ciudad(P), carga(C)
PRECONDICIÓN: en(R, P)$\wedge$en(C, R)
EFECTO: $\neg$en(C, R)$\wedge$en(C, P)
cargar(R, P, C)
PARÁMETROS: cohete(R), ciudad(P), carga(C)
PRECONDICIÓN: en(R, P)$\wedge$en(C, P)
EFECTO: $\neg$en(C, P)$\wedge$en(C, R)

Figura 2.4: Operadores para el dominio de cohetes [95]

Básicamente, el problema de planing consiste en encontrar una secuencia de acciones (operadores instanciados), tales que al ejecutar esas acciones partiendo del estado inicial, se obtiene el estado objetivo. Por ejemplo, supóngase que el estado inicial del mundo para el problema de los cohetes es: en(r, londres)$\wedge$en(c, londres)$\wedge$tieneCombustible(r), indicando que en Londres hay un cohete con combustible y una carga. El objetivo es que la carga esté en París: en(c, paris). Los objetos presentes en el ambiente son: cohete(r), ciudad(londres), ciudad(parís) y carga(c).

En la figura 2.5 se muestra una representación del plan para transportar una carga desde Londres a París. En la parte superior e inferior de la misma, se muestra el estado inicial y los objetivos, respectivamente. Los arcos que unen las proposiciones representan los enlaces causales, esto es, una relación de orden entre dos acciones A y B con una proposición común p indicando que la acción A debe ser ejecutada antes que B debido a que ésta última requiere que p sea verdadera, y A tiene a p como efecto. Las acciones que forman parte del plan son: cargar(r, londres, c) $\rightarrow$ mover(r, londres, parís) $\rightarrow$ descargar(r, parís, c), es decir, que si se ejecutan esas acciones en orden, partiendo del estado inicial, se alcanzan los objetivos.

Existen numerosos algoritmos para construir planes, por ejemplo, STRIPS [39], Prodigy [19], UCPOP [98] o Graphplan [8]. En general, estos algoritmos se diferencian por el poder expresivo del lenguaje de especificación de operaciones y por la performance. Algunos de los algoritmos son independientes del dominio, mientras que otros requieren de información dependiente del dominio para obtener una buena performance. Por ejemplo, UCPOP permite definir operaciones con variables universalmente cuantificadas en las precondiciones y efectos condicionales. Sin embargo, requiere de la definición de una función *choose* dependiente del dominio, mediante la cual se selecciona una operación de un conjunto de operaciones para aplicar en cada estado del mundo y construir el plan. Por otro lado, Graphplan posee la misma capacidad expresiva, pero es independiente del dominio. Sin embargo, la performance de GraphPlan se ve afectada en forma negativa cuando hay gran cantidad de objetos en el dominio. Para resolver esto se han desarrollado varias extensiones al algoritmo, por ejemplo, IGP [89] o STAN [65], entre otros.

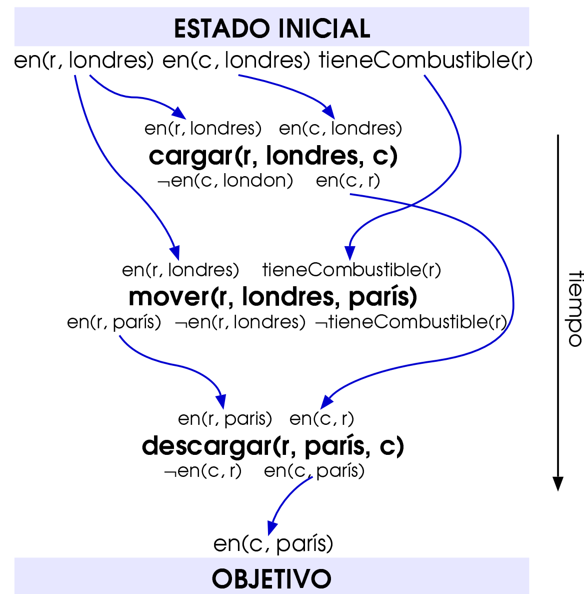


Figura 2.5: Plan para transportar una carga desde Londres a París

2.1.7 Aprendizaje

Se dice que una computadora aprende cuando cambia su estructura, programas o datos, basada en sus entradas o en respuesta a información externa, de manera tal de mejorar su eficiencia o eficacia en el futuro [70].

En un agente, es posible modificar su comportamiento o su estado mental. Por modificar su comportamiento debe entenderse, cambiar sus capacidades reactivas o deliberativas. Por ejemplo, un agente que utiliza *planning* podría almacenar los planes generados para aplicarlos a situaciones similares en el futuro [54].

Hay varias razones por las cuales el aprendizaje puede ser útil [70]:

- Algunas tareas no pueden ser bien definidas, a menos que sea a través de ejemplos.
- Es posible que existan relaciones ocultas en grandes volúmenes de información (*data mining*).
- Ciertos detalles del ambiente pueden no ser conocidos.
- La cantidad de conocimiento disponible puede ser muy grande para ser codificado explícitamente por humanos.
- Dinamicidad en el ambiente y en los requerimientos.

El aprendizaje aplicado a agentes puede ser dividido en dos categorías [97]:

- *aislado*: los agentes aprenden sin considerar ni aprovechar la presencia de otros agentes.
- *multi-agente*: un conjunto de agentes aprenden de forma comunitaria intercambiando información, compartiendo puntos de vista, siguiendo convenciones comunes, etc.

El aprendizaje en agentes aislados ha sido estudiado por décadas. Sin embargo, el aprendizaje multiagente constituye un campo de estudio muy reciente.

En aprendizaje multi-agente, aparecen dos nuevos elementos de los cuales es posible aprender: el ambiente y las interacciones. Así, por ejemplo, es posible aprender de las interacciones agente-agente y agente-ambiente.

Básicamente, el aprendizaje puede ser clasificado según la dificultad que implica:

- *aprendizaje de memoria*: implantación directa de conocimiento y habilidades sin realizar inferencias o transformaciones desde el agente.
- *aprendizaje de instrucciones y sugerencias*: transformación e integración de nueva información (instrucciones y sugerencias) no ejecutable por el agente en forma directa con conocimiento y habilidades previas.
- *aprendizaje por ejemplo y por práctica*: extracción y refinamiento de conocimiento y habilidades a partir de ejemplos positivos y negativos o de experiencia práctica.
- *aprendizaje por analogía*: transformación de conocimiento y habilidades obtenidas de problemas resueltos con anterioridad, para resolver problemas no resueltos.
- *aprendizaje por descubrimiento*: obtención de nuevo conocimiento y habilidades haciendo observaciones, experimentos, generando y verificando hipótesis o teorías en base a las observaciones y resultados experimentales.

El aprendizaje puede ser mejorado notoriamente si se provee algún tipo de *feedback*, indicando el nivel de performance alcanzado. Básicamente, se pueden distinguir tres tipos de mecanismos de *feedback*:

- *aprendizaje supervisado*: el *feedback* especifica la actividad deseada; el objetivo del aprendizaje consiste en satisfacer esto de la mejor forma posible.
- *aprendizaje por refuerzo*: el *feedback* sólo especifica la utilidad de la actividad actual; el objetivo del aprendizaje consiste en maximizar esta utilidad.
- *aprendizaje sin supervisión*: no hay *feedback*.

En todos los casos se asume que el *feedback* proviene del ambiente o de otros agente. Esto significa que el ambiente u otro agente actúa como un maestro (aprendizaje supervisado) o crítico (aprendizaje por refuerzo).

En la siguiente sección se describe una técnica de reuso de diseño y de código que se utilizó para desarrollar una infraestructura ¤exible y reusable que captura las características generales del dominio de agentes inteligentes con el fin de facilitar su desarrollo.

2.2 Frameworks

Un *framework* es una aplicación reusable *semi-completa* o *esqueleto* para una familia de aplicaciones pertenecientes a un dominio determinado [38]. A diferencia de otras técnicas de reuso orientado a objetos basados en bibliotecas de clases, los *frameworks* están orientados hacia un dominio de aplicación en particular. Los *frameworks* orientados a objetos permiten reutilizar diseño e implementaciones probadas para reducir el costo y mejorar la calidad del software [59]. Esto resulta particularmente importante si se considera la aplicación de *frameworks* en la construcción de software de cierta complejidad tal como agentes inteligentes [101, 102].

Johnson [58] define *framework* como *"un esqueleto de una aplicación que puede ser instanciado por un desarrollador para obtener una aplicación completa"*. Un *framework* está formado por un número de componentes genéricos que interactúan mediante una interfaz estable. Dichos componentes e interacciones definen la estructura abstracta de un dominio de aplicación de forma genérica y reusable. La *instanciación* del *framework* se produce cuando se especifica el comportamiento dependiente de la aplicación.

En particular, en orientación a objetos, los *frameworks* especifican los componentes e interfaces por medio de un conjunto de clases. Típicamente, un gran número de esas clases son abstractas. La instanciación del *framework* se realiza especializando un conjunto de clases, redefiniendo métodos y componiendo clases concretas.

Los mecanismos más importantes de la programación orientada a objetos utilizados en la construcción de *frameworks* son los métodos *hook*, *abstractos*, *template* y *base*. Los métodos *hook* definen comportamiento por defecto, desacoplando las interfaces estables y el comportamiento de un dominio de aplicación de las variaciones requeridas por las instanciaciones en un contexto particular. Así, el comportamiento especificado por un método *hook* podrá ser utilizado directamente o redefinido por medio de herencia. Los métodos *abstractos* sólo definen una interfaz. Puede pensarse que un método abstracto es un *hueco* que debe ser completado por el programador en el momento en que instancia el *framework*. En términos de POO esto significa que el programador deberá implementar todos los métodos abstractos de las clases que especialice para instanciar el *framework*.

El ¤ujo de control del *framework* es especificado por los métodos *template*. Un método *template* define comportamiento común al dominio de aplicación invocando a uno o más métodos abstractos. El comportamiento específico de cada aplicación se implementa redefiniendo dichos métodos abstractos. El comportamiento final de un método *template* varía en función de la implementación de los métodos abstractos que éste invoca, de acuerdo con la clase del objeto que recibe el mensaje que activa ese método *template*.

Finalmente, los métodos base proveen una implementación completa de un comportamiento común a cualquier aplicación del dominio. A diferencia de los métodos *hook*, los métodos base no deberían ser redefinidos por el programador.

En la figura 2.6 se observa un *framework* compuesto por 3 clases: `A`, `B` y `C`. Las clases `A` y `B` poseen métodos abstractos, por lo tanto ambas son abstractas. La clase `A` posee un método *template* que invoca a un método abstracto `m1` de la clase `A`, luego al método *hook* `create` de la clase `C` y finalmente al método base `update` de la clase `B`. Para especificar el comportamiento particular de cada aplicación, se deberá definir el método `m1` en las subclases de `A` y, opcionalmente, redefinir el método `create`. La instanciación del *framework* consta de tres clases: `D`, `E` y `F`. La clase `D` especializa `A`, definiendo sus dos métodos abstractos; la clase `F` redefine uno de los métodos *hook* de la clase `B`.

2.3 Diseño de frameworks

Para diseñar un *framework* pueden adoptarse dos enfoques diferentes: *conducido por ejemplos* o *conducido por un modelo de dominio de aplicación*. El primero consiste en abstraer las características comunes de un conjunto de aplicaciones pertenecientes al mismo dominio [25]. La selección de un conjunto de aplicaciones representativas del dominio de aplicación es de fundamental importancia para que el *framework* posea abstracciones comunes a dicho dominio.

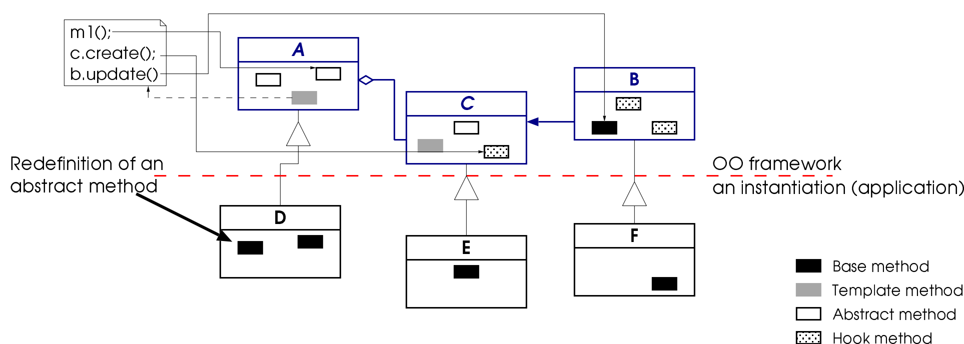


Figura 2.6: Instanciación de un *framework* orientado a objetos

El segundo enfoque tiene sus orígenes en el concepto de *arquitectura de software* y *estilos arquitectónicos*. La arquitectura de un sistema de software se refiere a la división del mismo en componentes y los patrones de interacción entre ellos [85]. Una *arquitectura específica del dominio* define un conjunto de componentes e interfaces que caracterizan un dominio de aplicación específico.

Un patrón organizacional de componentes e interacciones que se repite en varios sistemas se denomina *estilo arquitectónico*. Por ejemplo, cliente/servidor, la organización por niveles de la arquitectura para comunicaciones de la OSI, *pipes&filters*, repositorio, etc. Así, un estilo arquitectónico caracteriza una familia de sistemas relacionados por propiedades estructurales y semánticas comunes [46]. Los estilos arquitectónicos promueven el reuso de diseño, por cuanto permiten reaplicar soluciones conocidad y bien estudiadas a nuevos problemas. Al mismo tiempo, los aspectos invariantes de un estilo arquitectónico pueden ser aprovechados para reusar implementaciones.

Así, otra manera de desarrollar un *framework* consiste en partir de un modelo de dominio abstracto y uno o más estilos arquitectónicos adecuados, de lo cual se obtiene una primer materialización arquitectónica. Luego, se refina y materializa la arquitectura mediante un conjunto de clases que conforman el *framework*. La aplicación de patrones de diseño [45] durante esta etapa puede mejorar la ¤exibilidad y reusabilidad del *framework*. En la figura 2.7 se ilustra este proceso de diseño.

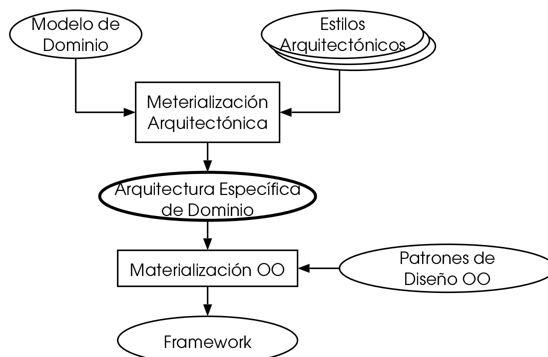


Figura 2.7: Diseño conducido por un modelo de dominio [16]

Los estudios comparativos entre ambos procesos [16] han mostrado que el segundo de ellos produce *frameworks* más estables en menos etapas de refinamiento. Esto se debe a que las principales clases que componen el *framework* son definidas por la arquitectura. Algo similar ocurre con las clases que implementan el ¤ujo de control abstracto del dominio de aplicación.

En particular, en el dominio de agentes inteligentes existen innumerables modelos [81, 86, 104] y arquitecturas [42, 49, 4]. De esta forma, en lugar de comenzar a diseñar un *framework* a partir de un análisis de dominio, es posible reusar el diseño de una arquitectura específica de dominio existente.

En la sección siguiente se describe la arquitectura de agentes Brainstorm, en la cual se basa el presente trabajo.

2.4 La arquitectura Brainstorm

En la sección anterior se describieron los dos principales enfoques para construir *frameworks*. En particular, se analizó la construcción de *frameworks* a partir de un modelo de dominio.

El *framework* propuesto en este trabajo está basado en una arquitectura específica de dominio. Esto implica que no sólo se están reusando modelos y estilos arquitectónicos, sino que se está aprovechando el resultado de un proceso de diseño cuyo resultado es una arquitectura. En particular, el *framework* propuesto está basado en la arquitectura de agentes Brainstorm.

La arquitectura Brainstorm [4] prescribe agentes con capacidades de acción, reacción, deliberación, percepción, comunicación y aprendizaje. Además, prescribe la forma en que se representan y manipulan los estados mentales de los agentes, para lo cual define una integración de programación orientada a objetos y programación lógica.

Brainstorm considera a un sistema multi-agente como un cojunto de objetos y *objetos-agente* [2, 3]. Un objeto-agente es un objeto que tiene un *meta-nivel* asociado. El meta-nivel interfiere las computaciones del objeto para alterar su comportamiento, implementando las capacidades de agentes tales como comunicación, percepción, reacción, deliberación y aprendizaje.

Para incorporar capacidades de agentes a objetos, la arquitectura utiliza un mecanismo re¤exivo basado en *meta-objetos*. Un meta-objeto es un tipo especial de objeto capaz de interceptar la invocación de métodos y alterar su ejecución [66]. Por ejemplo, en la figura 2.8 se utiliza un meta-objeto para realizar un *trace* de los mensajes recibidos por el objeto *anObject1*. Cada vez que dicho objeto recibe un mensaje, por ejemplo *update()*, su meta-objeto asociado *aMetaObject* es notificado con *handleMessage(update)*. Luego, el meta-objeto imprime por pantalla el mensaje interceptado, en este caso *update*. Finalmente, ejecuta el método original mediante *message.send()*.

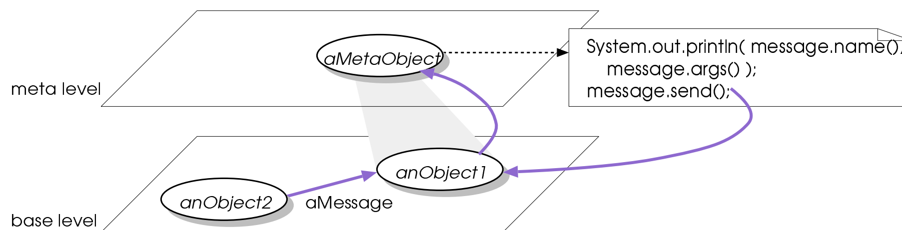


Figura 2.8: Meta-objetos

La arquitectura define varios tipos de meta-objetos responsables de las capacidades de agentes que pueden ser asociadas a los objetos. De esta manera, un objeto simple (sin inteligencia) puede ser transformado en un *objeto-agente* asociándole meta-objetos que implementan capacidades tales como:

- Comunicación utilizando un lenguaje y un protocolo uniforme tal como KQML [40]. Esto permite que los agentes interactúen a nivel de conocimiento de forma similar a lo que sucede con los *"speech acts"* humanos. Así, por ejemplo, un agente puede preguntarle a otro si conoce cierta información, o solicitarle que cumpla un conjunto de objetivos realizando las acciones que considere necesarias.
- Percepción: permite que un agente conozca hechos de su medio ambiente sin que otros se lo comuniquen. La percepción permite a un agente observar las actividades de otros agentes, sus interacciones, los cambios en su medio ambiente, etc.
- Reacción: posibilidad de actuar ante situaciones predeterminadas sin *razonar*, es decir, siguiendo patrones comportamentales fijos. Por ejemplo, si un agente percibe un obstáculo frente a él se detiene.
- Deliberación: consiste en seleccionar las acciones antes de actuar. Es decir, en lugar de reaccionar ante una situación dada, se busca un conjunto de acciones que permitan al agente lograr sus objetivos. Luego, esas acciones son ejecutadas.
- Manipulación de estados mentales tales como objetivos, conocimiento, creencias, intenciones, entre otras.
- Aprendizaje: consiste en modificar el comportamiento del agente en función de la experiencia previa.

Cada una de esas capacidades es responsabilidad de un meta-objeto. De esta forma, la combinación de los meta-objetos para construir agentes permite obtener diferentes tipos de agentes con diferentes capacidades, tales como agentes reactivos, deliberativos e híbridos. Además, es posible modificar las capacidades de un agente en forma dinámica, simplemente agregando o quitando meta-objetos.

En la figura 2.9 se muestra un diagrama de la arquitectura Brainstorm. Las zonas grises indican una asociación re¤exiva objeto/meta-objeto. El nivel inferior, denominado *nivel base*, contiene objetos simples y objetos-agente.

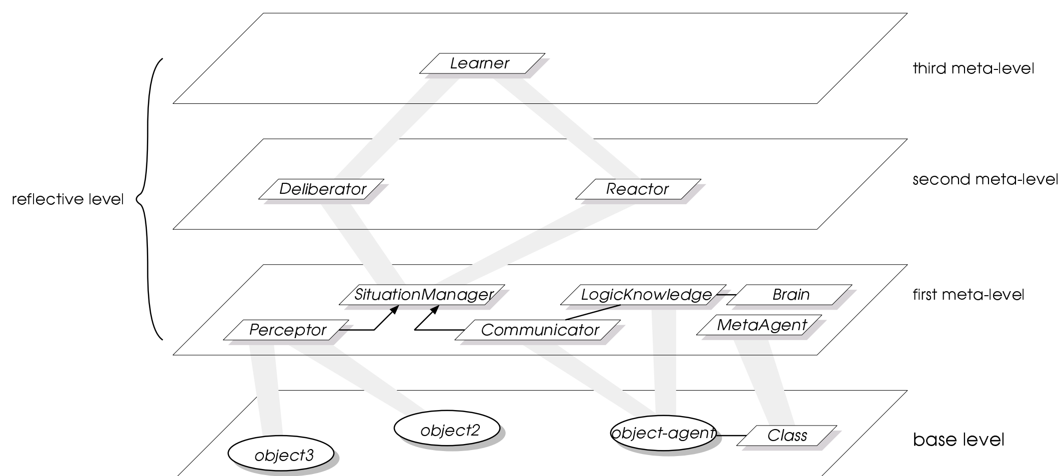


Figura 2.9: La arquitectura Brainstorm

El primer meta-nivel de la arquitectura trabaja con los mensajes recibidos por el nivel base. Consta de componentes responsables de administrar y manipular los estados mentales, percibir los eventos

ocurridos en el ambiente, administrar las comunicaciones, detectar situaciones de interés e interferir las computaciones del nivel base para incorporar capacidades de agentes a un objeto simple.

Los componentes *LogicKnowledge* y *Brain* son responsables de representar y manipular los estados mentales del agente. Dichos componentes se basan en una integración de objetos y lógica que permite manipular cláusulas lógicas en un lenguaje orientado a objetos [5].

El componente responsable de las comunicaciones, denominado *Communicator*, define servicios de recepción, envío e interpretación de mensajes, utilizando un lenguaje de alto nivel. Básicamente, dicho componente es responsable de las interacciones entre agentes basadas en conocimiento.

El componente *Perceptor* es el encargado de percibir los eventos que se producen en el ambiente del agente. Esto lo realiza observando el intercambio de mensajes entre objetos y objetos-agente del nivel base. Dicho componente puede ser utilizado para observar los mensajes recibidos por cualquier objeto en forma transparente.

El componente *SituationManager* es responsable por determinar las situaciones de interés para el agente a partir de las percepciones, comunicaciones y estado mental. Por ejemplo, para un robot que carga cajas puede ser de interés saber que hay una caja en un camión que se encuentra frente a él cuando no lleva ninguna carga y tiene como objetivo descargar todas las cajas de un camión.

El segundo meta-nivel, denominado comportamental, procesa las situaciones de interés detectadas por el *SituationManager* y las comunicaciones. Dicho meta-nivel es responsable de las acciones del agente. Está formado por un componente de reacción y otro de deliberación.

El componente de reacción, llamado *Reactor*, actúa sobre las situaciones de interés, generando una o más acciones ante una situación. El componente *Deliberator* analiza y decide qué acciones ejecutar ante una situación, una comunicación o simplemente para cumplir con los objetivos del agente. Básicamente, estos componentes son opcionales, ya que dependen del tipo de agente. Así, por ejemplo, un agente reactivo sólo utilizará un *Reactor*, ya que el componente *Deliberator* no sería necesario.

El tercer meta-nivel, formado por el componente *Learner*, es responsable del aprendizaje. Dicho componente actúa sobre los componentes de reacción y deliberación, modificando el comportamiento del agente en base a la experiencia. Este componente es opcional, ya que por ejemplo, un agente reactivo no requiere de él.

Básicamente, un objeto-agente posee un meta-objeto de conocimiento, un meta-objeto de comunicación (opcional), un conjunto de cero o más meta-objetos de percepción, un meta-objeto de reacción, uno de deliberación y un meta-objeto de aprendizaje (opcional).

Para crear un agente, se puede asociar un meta-objeto *MetaAgent* a una clase de nivel base como se muestra en la figura 2.10. Luego, cada vez que se crea una instancia de la clase de nivel base mediante el mensaje *new*, se intercepta dicho mensaje, y actúa el meta-objeto *aMetaAgent*. Este meta-objeto crea los meta-objetos necesarios para formar el agente. Por ejemplo, al objeto-agente `object1`, se le ha asociado un meta-nivel de inteligencia. Dicho agente percibe otros dos objetos mediante un meta-objeto de percepción *aPerceptor*. El meta-objeto de comunicación *aCommunicator* le da la capacidad de comunicarse utilizando KQML. El segundo nivel re¤exivo es responsable por las acciones del agente y contiene un meta-objeto de reacción *aReactor* y uno para deliberación *aDeliberator*.

En las siguientes secciones se describe con mayor detalle la funcionalidad y las iterfaces entre los componentes de la arquitectura.

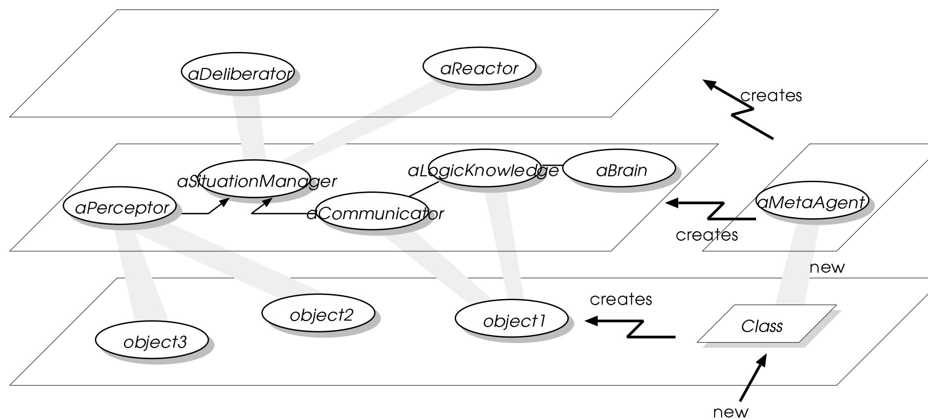


Figura 2.10: Un objeto-agente

2.4.1 MetaAgent

Este componente es responsable del proceso de instanciación de los meta-objetos. Cuando la clase de nivel base del agente recibe un mensaje de creación, su *metaAgent* asociado lo intercepta y genera el meta-nivel del nuevo objeto-agente. Un *metaAgent* crea los meta-objetos necesarios para satisfacer la especificación del agente. Así, es posible definir diversos tipos de agentes en el mismo sistema.

2.4.2 Representación y manipulación de estados mentales

LogicKnowledge y *Brain* permiten que los agentes representen y manipulen sus estados mentales. Dichos estados constan, principalmente, de creencias y objetivos.

Los estados mentales son representados mediante cláusulas lógicas y objetos, y son manipulados por el componente *Brain*. Así, estos componentes integran objetos con lógica facilitando la manipulación y representación de actitudes mentales en un lenguaje orientado a objetos [3, 5].

La representación de las actitudes mentales en términos lógicos, permite utilizar mecanismos de inferencia para obtener nueva información de la información disponible. Por otro lado, la administración de ciertos tipos de actitudes mentales requiere de mecanismos de manipulación ¤exibles que permitan, por ejemplo, detectar y eliminar inconsistencias, o utilizar el conocimiento del agente en algoritmos de *planning* o razonamiento basado en casos.

2.4.3 Percepción

En un sistema orientado a objetos todas las interacciones se realizan por medio de mensajes. Entonces, la percepción puede realizarse observando el ¤ujo de mensajes mediante meta-objetos. Así, para percibir acontecimientos, un agente debe utilizar uno o más meta-objetos de percepción. Estos meta-objetos observan otros agentes u objetos, re¤ejando los mensajes recibidos por ellos.

Los mensajes interceptados son recibidos por el meta-objeto de percepción mediante el método *handleMessage*. Los meta-objetos de percepción pueden observar a cualquier objeto del sistema en forma transparente, es decir, que el objeto observado desconoce que lo están observando. Además, es posible reconfigurar dinámicamente los meta-objetos de percepción para observar otros objetos.

2.4.4 Comunicación

El componente *Communicator* es responsable por manejar protocolos de comunicación de alto nivel. Los agentes que desean interactuar utilizan meta-objetos de comunicación de la misma clase. Cuando un objeto-agente recibe un mensaje que pertenece a su lenguaje de comunicación, este mensaje es interceptado por su meta-objeto de comunicación. Este mecanismo envía el mensaje *handleMessage* con el mensaje interceptado como argumento al meta-objeto de comunicación. Entonces, ese mensaje es analizado por el meta-objeto de comunicación para decidir qué hacer. Si el mensaje se ha definido en la clase del objeto como capacidad temporaria o en la clase del meta-objeto de comunicación como parte de su lenguaje de comunicación, se registra utilizando *register*. El meta-objeto deliberador puede interceptar el mensaje *register* para tratarlo y generar un mensaje de respuesta.

2.4.5 Situaciones

El componente *SituationManager* es responsable de buscar situaciones de interés utilizando las percepciones, mensajes recibidos y el estado mental del agente. Este componente tiene conexiones estáticas con los componentes del primer meta-nivel y puede utilizar el conocimiento y las creencias encapsuladas en el componente *Brain*.

Los meta-objetos de percepción informan al administrador de situaciones todos los eventos observados. Algo similar ocurre con las comunicaciones. Cuando se detecta una situación de interés, el *SituationManager* se envía a sí mismo el mensaje *newSituation*. Este protocolo es utilizado por los componentes de reacción y deliberación para actuar en función de las situaciones detectadas.

2.4.6 Comportamiento reactivo

El componente *Reactor* es responsable de reaccionar ante una situación dada. La reacción consiste en una secuencia de acciones que son ejecutadas cuando el *SituationManager* detecta una situación y se envía a sí mismo el mensaje *newSituation*.

El meta-objeto de reacción de un agente actúa de forma inmediata si existe una reacción definida para la situación detectada. Luego, el deliberador podría actuar.

2.4.7 Deliberación

El componente *Deliberator* es el responsable del comportamiento inteligente y autónomo del agente. Las acciones del agente son determinadas cuando: 1) percibe una situación de interés, 2) recibe un mensaje y 3) decide qué acciones realizar para alcanzar un conjunto de objetivos.

Cuando un agente detecta una situación de interés, debe decidir qué hacer, para lo cual el deliberador intercepta la situación detectada y la delega en uno de los algoritmos de decisión. Para las comunicaciones, al agente puede responder o ignorar dicha comunicación.

Además, un agente puede actuar sin necesidad de que se produzcan situaciones o comunicaciones. En este caso, el agente atiende las comunicaciones pendientes, construye planes para alcanzar sus objetivos, analiza sus objetivos, intenciones y ejecuta acciones en el ambiente

2.4.8 Aprendizaje

El componente de aprendizaje, denominado *Learner*, observa y modifica el comportamiento reactivo y deliberativo del agente en base a la experiencia. Este componente es opcional, ya que, por ejemplo, un agente puramente reactivo no es capaz de aprender.

En el caso del comportamiento reactivo, el *Learner* puede alterar o eliminar reacciones existentes, o agregar nuevas.

El comportamiento deliberativo puede ser alterado interviniendo la forma en que el agente selecciona las acciones para alcanzar sus objetivos, la forma en que actúa frente a las situaciones de interés o comunicaciones.

2.5 Conclusiones

En este capítulo se presentó el concepto de agente, describiendo sus capacidades típicas, tales como acción, representación de estados mentales, percepción, reacción, comunicación, movilidad y deliberación. Luego, se describieron los principales conceptos sobre *frameworks* orientados a objetos y el diseño de los mismos. Finalmente, se presentaron los aspectos más relevantes de la arquitectura de agentes Brainstorm, en la cual se basa el presente trabajo.

En el capítulo siguiente se describen varias herramientas para construir agentes.

CAPÍTULO 3

Herramientas para construcción de agentes

La construcción de agentes inteligentes es un proceso complejo, por cuanto se trata de software autónomo y adaptativo con variadas capacidades [106, 88]. Para facilitar la construcción de tales sistemas se han desarrollado herramientas que proveen un conjunto de componentes de software a partir de los cuales es posible construir agentes inteligentes.

Las herramientas existentes permiten construir agentes con una o más de las capacidades presentadas en el capítulo anterior: comunicación, percepción, movilidad, representación de estados mentales, reacción, deliberación y aprendizaje. Además, algunas de ellas proveen ambientes visuales de desarrollo y depuradores.

En este capítulo se describen algunas de las herramientas existentes para desarrollar agentes, con el fin de establecer sus principales capacidades y falencias. El capítulo está organizado de la siguiente manera: en la sección 3.1 se describen las características analizadas; a continuación, en las secciones 3.2 a 3.9 se describen y analizan AgentBuilder, DECAF, FraMaS, JAF, JAFIMA, JAFMAS, MadKit y ZEUS; finalmente, en la sección 3.10 se comparan las herramientas y se presentan las conclusiones del capítulo.

3.1 Características analizadas

Las herramientas analizadas en el presente capítulo permiten construir agentes con una o más de las capacidades descriptas en el capítulo 2. Con el fin de comparar las herramientas, se analizaron características generales de cada una de ellas y características referentes a cómo son soportadas cada una de las capacidades de agentes desde el punto de vista de la ¤exibilidad y extensibilidad. A continuación se describen las características generales analizadas:

- *lenguaje*: tradicionalmente, las herramientas de agentes inteligentes han estado basadas, principalmente, en lenguajes declarativos que procesan una representación simbólica del ambiente en que se encuentra el agente y una representación del comportamiento del mismo, para producir

el comportamiento *inteligente*. Como ejemplos de este tipo de lenguajes se pueden mencionar Agent0 [86], PLACA [94] y AgentSpeak(L) [79].

En general, los lenguajes para agentes comparten una característica común: el programador no puede extender ni adaptar los elementos del lenguaje con facilidad. En consecuencia, resulta difícil incorporar nueva funcionalidad a dichos lenguajes o realizar extensiones/adaptaciones a la arquitectura general de los agentes.

- *biblioteca de componentes*: consisten de componentes reusables responsables de las capacidades de agentes. Típicamente, las bibliotecas de agentes poseen componentes de uso general que pueden ser utilizados para construir cualquier tipo de agente. Por ejemplo: comunicación con KQML, repositorios de datos, *blackboards*, herramientas de visualización, lenguajes para representación de conocimiento, etc. El programador es responsable de ensamblar esos componentes y especificar el ¤ujo de control entre ellos para construir agentes [59].

- *arquitectura*: es una descripción de un conjunto de componentes de software y patrones de interacción entre ellos, que prescriben un sistema de software (agente). Por ejemplo, una herramienta basada en una arquitectura puede prescribir componentes responsables de las capacidades de comunicación y razonamiento, cómo un agente procesa las comunicaciones y la forma en que decide qué acciones realizar. Típicamente, los lenguajes para agentes especifican la arquitectura y los tipos de agentes soportados (deliberativo, reactivo, híbrido, etc.), mientras que las bibliotecas de componentes no lo hacen.

- *framework*: un *framework* consta de un conjunto de clases que definen componentes abstractos y concretos (listos para ser utilizados) y el ¤ujo de control entre esos componentes. Una característica importante de los *frameworks* consiste en que definen la arquitectura general de cierto tipo de software, en este caso particular, agentes.

  A diferencia de las bibliotecas de componentes, los *frameworks* definen los protocolos de interacción entre los componentes y las formas en que pueden ser combinados. Los *frameworks* pueden ser usados como bibliotecas de componentes, es decir, invocando código de los componentes. Sin embargo, lo que caracteriza a los *frameworks* es la inversión de control. Es decir, que el programador implementa los métodos específicos de su aplicación, los cuales son invocados por el *framework*. De esta forma, la mayor parte de la funcionalidad de las aplicaciones es heredada del *framework*, con lo que se reduce el esfuerzo de diseño e implementación.

- *toolkit*: es un ambiente gráfico de desarrollo de aplicaciones. Un *toolkit* [59] está formado por un conjunto de herramientas visuales que permiten al programador interactuar con un *framework* para configurar y construir aplicaciones, en este caso agentes.

  Las posibilidades de los *toolkits* dependen de la ¤exibilidad de las herramientas visuales y del *framework* subyacente, entre otros. Pueden clasificarse en:

  – *toolkits* que no permiten que el programador ingrese código, modifique el código generado ni extienda su funcionalidad con facilidad. Este tipo de *toolkits* intentan ocultar el *framework* subyacente con el fin de facilitar la construcción de aplicaciones. Sin embargo, la ¤exibilidad de este tipo de herramientas puede ser limitado, debido, justamente, al hecho de que el programador no tiene acceso al *framework*.

  – *toolkit* con la posibilidad de ingresar código: permiten que el programador ingrese código, construya diagramas representando autómatas, redes de petri, redes jerárquicas de tareas,

etc. En este tipo de *toolkits*, en forma similar a la categoría previa, la complejidad del *framework* es ocultada. Sin embargo, logran mayor ¤exibilidad y adaptabilidad, sin impactar negativamente la facilidad de uso.

– *toolkit* extensible: utilizan una herramienta gráfica de desarrollo para instanciar un *framework*. El programador tiene la posibilidad de adaptar y extender el *framework* a las necesidades de cada aplicación, extendiendo los componentes predefinidos, definiendo nuevos componentes e integrándolos con el *toolkit*. En este tipo de herramientas, el programador puede trabajar en forma similar que en los anteriores, sin embargo, en caso de ser necesario, tiene acceso al *framework* subyacente con una asistencia (quizás mínima) del *toolkit*.

En general, las herramientas existentes definen uno o más componentes de software responsables de cada una de las capacidades de agentes. Dichos componentes pueden ser adaptados o extendidos, en menor o mayor grado por el programador, según las necesidades requeridas. Por ejemplo, una herramienta puede permitir que se seleccione el algoritmo de *planning* que utilizan los agentes para decidir sus acciones, o permitir que el programador desarrolle otros algoritmos y los utilice con la herramienta. Así, para cada una de las herramientas es posible analizar la ¤exibilidad de la implementación de cada una de las capacidades de agentes soportada y clasificarla en las siguientes categorías:

- *Imposibilidad de adaptacióno extensión*: el programador tiene que utilizar cierta capacidad tal como está definida en la herramienta. Por ejemplo, una herramienta permite construir agentes con capacidades de comunicación con KQML utilizando TCP/IP, sin embargo, el programador no puede utilizar otros lenguajes ni protocolos, ni puede enviar mensajes utilizando otros medios (e-Mail o HTTP).
- *Posibilidad de adaptación*: el programador puede adaptar una capacidad especificando una o más propiedades predefinidas o limitadas (en número). Por ejemplo, el programador podría seleccionar el lenguaje de comunicación de los agentes (KQML o ACL) y el medio de transporte para dichos mensajes (mensajes entre objetos, TCP/IP o e-Mail); sin embargo, no podría utilizar otros lenguajes que no sean los predefinidos.
- *Posibilidad de adaptación y extensión*: el programador puede adaptar y extender una capacidad. Por ejemplo, podrían implementarse nuevos lenguajes de comunicación para ser usados por los agentes, u otros mecanismos de decisión.

En las siguientes secciones se describen y analizan algunas herramientas existentes, según las características presentadas en esta sección.

3.2 AgentBuilder

AgentBuilder [84] es un *toolkit* para desarrollar agentes cognitivos basado en una extensión del modelo BDI. El *toolkit* permite construir agentes cognitivos con capacidades de comunicación y coordinación.

AgentBuilder fue desarrollado con el objetivo de proveer un ambiente de desarrollo que facilite la construcción de aplicaciones basadas en agentes, ofreciendo herramientas gráficas que asisten al programador durante todo el ciclo de desarrollo, siguiendo una metodología específica de AgentBuilder.

AgentBuilder posee dos componentes principales: el *sistema de tiempo de ejecución* y el *toolkit*. El sistema de tiempo de ejecución es un intérprete del lenguaje RADL (Reticular Agent Definition Language), el cual está basado en Agent0 [86] y PLACA [94].

Básicamente, el *toolkit* genera un programa RADL conteniendo la definición del agente. Un programa RADL contiene una especificación completa del comportamiento de un agente y su estado mental inicial. Además, el programador puede proveer, en caso de ser necesario, clases Java accesorias definiendo comportamiento extra de un agente, interfaces gráficas de usuario o conceptos del dominio de aplicación, tales como *cheque*, *auto* o *factura*.

El *toolkit* consta de los siguientes componentes:

- Administrador de ontologías: asiste al desarrollador en el análisis del dominio del problema, definiendo los conceptos relevantes a través de ontologías. La herramienta visualiza en forma gráfica las relaciones entre los conceptos. Además, permite definir el dominio utilizando un modelo de objetos expresado en notación UML.

  Los conceptos definidos por esta herramienta pueden ser utilizados por los agentes para representar y manipular el conocimiento del dominio de aplicación.

  El administrador de ontologías produce una clase Java por cada concepto definido. Dichas clases pueden ser modificadas según las necesidades de la aplicación, sin embargo, esto no puede realizarse directamente desde la herramienta.

- Administrador de agencias: una agencia consiste de dos o más agentes que se comunican para realizar determinada tarea. Esta herramienta permite caracterizar los agentes y agencias de una aplicación.

- Administrador de protocolos: permite especificar los protocolos de interacción utilizando autómatas finitos similares a los de COOL.

- Administrador de agentes: consta de herramientas para definir el estado mental de un agente (creencias y compromisos) y su comportamiento (capacidades y reglas comportamentales).

  - Editor de compromisos: permite establecer un conjunto de acciones que deben ser realizadas por el agente en ciertos tiempos.

  - Editor de acciones: se utiliza para asociar una acción del agente con un método Java provisto por el programador.

  - Editor de reglas: permite construir reglas IF-THEN-ELSE que definen el comportamiento del agente. La condición del IF pueden incluir operadores lógicos relacionando creencias entre sí y campos del mensaje KQML recibido. Las acciones pueden modificar el estado mental del agente, generar mensajes, esperar a que algo suceda o ejecutar las capacidades del agente.

El sistema de ejecución consta de un intérprete de RADL, el programa RADL del agente y un conjunto de clases Java accesorias. Cada agente ejecuta en su propio sistema de ejecución, por lo tanto, cada agente ejecuta en su propio intérprete RADL.

Los agentes construidos con AgentBuilder poseen una base de datos de creencias y un conjunto de reglas comportamentales, que son utilizadas para enviar o responder mensajes. AgentBuilder utiliza KQML como lenguaje de comunicación de agentes.

El modelo de ejecución de los agentes es el siguiente: cuando un agente recibe un mensaje KQML, se busca una regla comportamental tal que su condición de activación sea verdadera y se ejecutan las acciones asociadas. Luego, se actualizan los estados mentales.

Una característica destacable de AgentBuilder es la posibilidad de construir agentes *casi* sin programar, ya que provee un ambiente de desarrollo para editar las capacidades y características de los agentes, y depurarlos durante su ejecución. Lo único que debe ser codificado en Java son las clases que describen el contenido de los mensajes KQML y los objetos del dominio que se desean representar. Opcionalmente, si la aplicación lo requiere, las capacidades de los agentes pueden ser codificadas en métodos Java. A tal fin, el *toolkit* provee un conjunto de servicios básicos, tales como acceder al estado mental del agente, o enviar mensajes KQML, que pueden ser utilizados en Java. Sin embargo, la arquitectura general de los agentes no puede ser redefinida ni adaptada, debido a que esto forma parte del lenguaje RADL.

La mayor limitación del *toolkit* consiste en que no posee ningún componente para *planning* o aprendizaje. Como consecuencia de esto, los agentes construidos siguen patrones comportamentales fijos.

Otra limitación consiste en que no es posible modificar ni extenter AgentBuilder, mas allá de lo permitido por la interfaz gráfica. Sin embargo, la posibilidad de definir las capacidades de los agentes utilizando métodos Java da ¤exibilidad al *toolkit*; por otro lado, el hecho de estar basado en el lenguaje RADL restringe las posibilidades y ¤exibilidad del mismo, a la vez que simplifica el desarrollo de agentes.

3.3 DECAF

DECAF (Distributed Environment Centered Agent Framework) [52] es un *toolkit* Java para construir sistemas multi-agente basado en un *framework*. DECAF define agentes inteligentes con capacidades de comunicación, reacción, deliberación y coordinación.

DECAF fue desarrollado con el fin permitir la rápida construcción de prototipos experimentales para evaluar técnicas de comunicación entre agentes, *planning*, *scheduling* reactivo y, eventualmente, aprendizaje. Para lograr dichos objetivo, DECAF provee una herramienta gráfica mediante la cual es posible especificar el comportamiento de los agentes utilizando redes jerárquicas de tareas[1] (RJT, HTN - Hierarchical Task Network). Una RJT describe un plan para alcanzar cierto objetivo, indicando cursos de acción alternativos y costos asociados. Cada una de las acciones de un plan lleva asociada una clase Java que implementa dicha acción. Esas clases deben ser provistas por el programador y definidas externamente a la herramienta gráfica.

Cada agente DECAF posee un conjunto de objetivos y una colección de acciones que pueden ser utilizadas para alcanzar los objetivos. Cada objetivo lleva asociado un plan estático RJT que indica qué acciones deben realizarse para lograr dicho objetivo. Así, para cada objetivo existe un plan que es ejecutado cuando ese objetivo se intenta hacer verdadero.

DECAF utiliza KQML como lenguaje de comunicación entre agentes. La coordinación se logra especificando clases de conversaciones mediante RJT, por lo que posee capacidades similares a las de COOL.

[1] Las redes jerárquicas de tareas están basadas en las estructuras TÆMS [29] descriptas en la sección 3.5.

En la figura 3.1 se muestra un esquema de la arquitectura de DECAF. Un agente es creado a partir de un *archivo de planes* que incluye una especificación de las capacidades del agentes, sus planes, estado inicial, etc. Dicho archivo es generado por una herramienta gráfica llamada *editor de planes*.

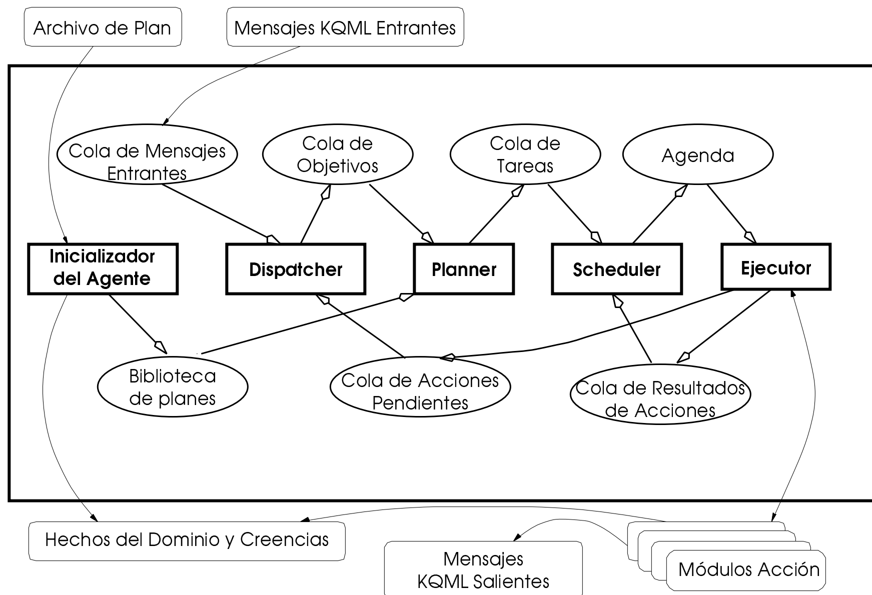


Figura 3.1: Arquitectura de DECAF [52]

A continuación se describen los principales componentes de DECAF:

- *Dispatcher*: es el encargado de procesar los mensajes KQML recibidos. Básicamente, detecta si se trata de un mensaje KQML perteneciente a una comunicación previa o una nueva conversación. En el primer caso, busca una acción en la *cola de acciones pendientes* y responde el mensaje. En el segundo caso, genera un nuevo objetivo y lo coloca en la *cola de objetivos.*
- *Planner*: monitorea la cola de objetivos. Cuando se introduce un nuevo objetivo, selecciona un plan de la *biblioteca de planes* para tratar de lograr dicho objetivo. Luego, coloca el plan en la *cola de tareas*. Dicha estructura contiene los planes en ejecución, incluyendo los originados por subobjetivos. Los cursos de acción alternativos de un plan son seleccionados por el *Planner*.
- *Scheduler*: es el responsable de determinar qué acciones pueden ser ejecutadas, cuáles deberían ser ejecutadas en cierto instante y en qué orden deberían ser ejecutadas. En la versión actual, simplemente toma las acciones de la *cola de tareas* y las coloca en la *agenda* en orden primero en entrar-primero en salir.
- *Ejecutor*: toma las acciones de la *agenda* e intenta ejecutarlas. Cuando toma una acción que requiere de confirmación, por ejemplo, un mensaje KQML, la coloca en la *cola de acciones pendientes*. Luego, si el *Dispatcher* determina que una acción pendiente se ha completado con éxito, la elimina de dicha cola y notifica al *Ejecutor*. En caso contrario se produce un error.

Una desventaja de DECAF consiste en que los planes que rigen el comportamiento de los agentes son estáticos, por lo que resulta difícil construir agentes que se adapten a los cambios ocurridos en el mundo. Además, DECAF no permite definir agentes con capacidad de percepción, aprendizaje, *planning on-line* ni movilidad. Tampoco prescribe mecanismos de representación ni manipulación de

los estados mentales. Sin embargo, prescribe la arquitectura de los agentes.

En general, las clases Java que componen el *framework* DECAF son clases concretas que no han sido diseñadas para ser extendidas, a excepción de las que representan a las RJT. Por lo tanto, el *toolkit* no puede ser extendido ni adaptado a nuevos requerimientos con facilidad, aunque permite desarrollar agentes en forma muy rápida y simple.

3.4 FraMaS

FraMaS [6] es un *framework* Java para desarrollar agentes con capacidades de percepción, comunicación, reacción, deliberación, aprendizaje y movilidad. FraMaS está siendo desarrollado con el fin de proveer un conjunto de componentes Java reusables para construir sistemas multi-agente.

Los agentes FraMaS se ejecutan en una plataforma que provee los servicios básicos de identificación, interacción y movilidad. Los servicios de identificación permiten que un agente descubra otros agentes que ejecutan en el mismo ambiente.

Los servicios de movilidad proveen migración débil, es decir que la migración no es transparente desde el punto de vista del desarrollador. Por tal razón, se debe tratar en forma *ad-hoc* la restauración del estado de ejecución de los *threads* que conforman a cada agente. Por otro lado, FraMaS no ofrece ningún mecanismo de seguridad ni protección a los agentes móviles que ejecutan en la plataforma.

Un agente FraMaS está formado por un conjunto de capacidades organizadas en capas. Así, por ejemplo, un agente puede tener una capa que implementa las capacidades de comunicación, otro responsable del razonamiento y otro del aprendizaje. Cada uno de los niveles interfiere el pasaje de mensajes hacia los niveles interiores mediante un protocolo muy simple basado en decoradores [45]. Por ejemplo, el nivel responsable del aprendizaje del agente intercepta los mensajes del nivel de razonamiento para determinar si el mensaje recibido corresponde a alguna situación resuelta en el pasado. En caso contrario, reenvía el mensaje al nivel de razonamiento.

La percepción se realiza mediante un mecanismo de intercepción de eventos mediante el cual es posible percibir eventos generados por los agentes o por aplicaciones que generan eventos en forma explícita para que algún agente FraMaS lo intercepte. Esto significa que un agente no es capaz de percibir cualquier objeto de su medio ambiente, sino que sólo puede percibir aquellos que han sido diseñados para ser observados.

El *framework* define la interfaz abstracta para implementar mecanismos de razonamiento tales como *planning* o razonamiento basado en casos. Utilizando esa interfaz, define un componente genérico para razonamiento basado en casos. Sin embargo, no provee ningún componente concreto para *planning* u otros mecanismos de razonamiento.

Los servicios de comunicación del *framework* no proveen ningún lenguaje de alto nivel tal como KQML o ACL. Sin embargo, FraMaS provee mecanismos genéricos de interacción mediante los cuales sería posible desarrollar comunicación basada en conocimiento. Tampoco define mecanismos para representar conversaciones tales como los de COOL.

FraMaS no define componentes responsables de representar y manipular los estados mentales del agente, ni provee ningún ambiente que facilite o asista al programador en la construcción de agentes.

El *framework* define los componentes de los agentes en forma abstracta y reusable, debido a que uno de los objetivos de FraMaS consiste, justamente, en proveer una infraestructura genérica y adaptable

para construir sistemas multi-agente.

3.5 JAF

JAF [55] define un conjunto de componentes reusables genéricos basados en la tecnología *JavaBeans* para construir agentes. JAF provee componentes para representar estados mentales, comunicar agentes y razonar.

JAF está formado por un conjunto de *JavaBeans* [92] responsables de las capacidades de los agentes. Un *JavaBean* es un componente de software reusable que puede ser manipulado visualmente en una herramienta de construcción compatible con *JavaBeans*.

Cada componente *JavaBean* posee un conjunto de propiedades, las cuales son establecidas en tiempo de desarrollo mediante una herramienta gráfica. Por ejemplo, una propiedad puede indicar los estados mentales del agente en el momento de su creación, o el tipo de mecanismo que el agente utiliza para decidir qué acciones ejecutar.

Cada una de las propiedades de un *JavaBean* está asociada con dos métodos Java para obtener y modificar dicha propiedad. Por ejemplo, un *JavaBean* con una propiedad *color* posee un método `getColor` para obtener el color y un método `setColor` para modificarlo. Así, las propiedades de un *JavaBean* son obtenidas automáticamente de los métodos `get`/`set` de la clase del *JavaBean*.

Un *JavaBean* puede ofrecer un conjunto de servicios a otros componentes. Los servicios están representados por todos los métodos públicos de la clase del *JavaBean*.

Los componentes *JavaBeans* pueden interactuar por medio de mensajes entre objetos o mediante un mecanismo de eventos que permite que un componente notifique a otros acerca de un determinado acontecimiento.

Cada agente JAF posee una descripción de las tareas que puede ejecutar y de las características del mundo en que se encuentra representadas mediante estructuras TÆMS [29]. TÆMS es un formalismo para representar tareas, dependencias entre tareas, tiempos y recursos. Por ejemplo, en la figura 3.2 se muestra una estructura que describe cómo preparar un café. Los triángulos representan recursos, los óvalos objetivos y los rectángulos acciones. Los arcos representan relaciones entre las entidades. Por ejemplo, un arco dirigido desde una acción a un recurso indica que la acción consume el recurso en la cantidad indicada por la etiqueta *consumes* de dicho arco.

En JAF, un agente posee al menos dos componentes: *Control* y *State*. El componente *Control* es responsable de crear el agente, inicializar sus componentes y mantener información interna. *State* es un repositorio que se utiliza para almacenar los estados mentales del agente. Este componente no define mecanismos específicos para representar estados mentales tales como creencias, deseos e intenciones, sino que consiste, simplemente, en una tabla de *hashing* que puede ser utilizada por el programador según las necesidades de cada aplicación.

Otros componentes de JAF son:

- *Communicate*: provee servicios de interacción entre agentes utilizando mensajes KQML. La comunicación puede realizarse entre agentes situados en diferentes sitios de una red.
- *Execute*: es el responsable de ejecutar las acciones del agente.
- *Log*: permite observar el ¤ujo de eventos entre los componentes.

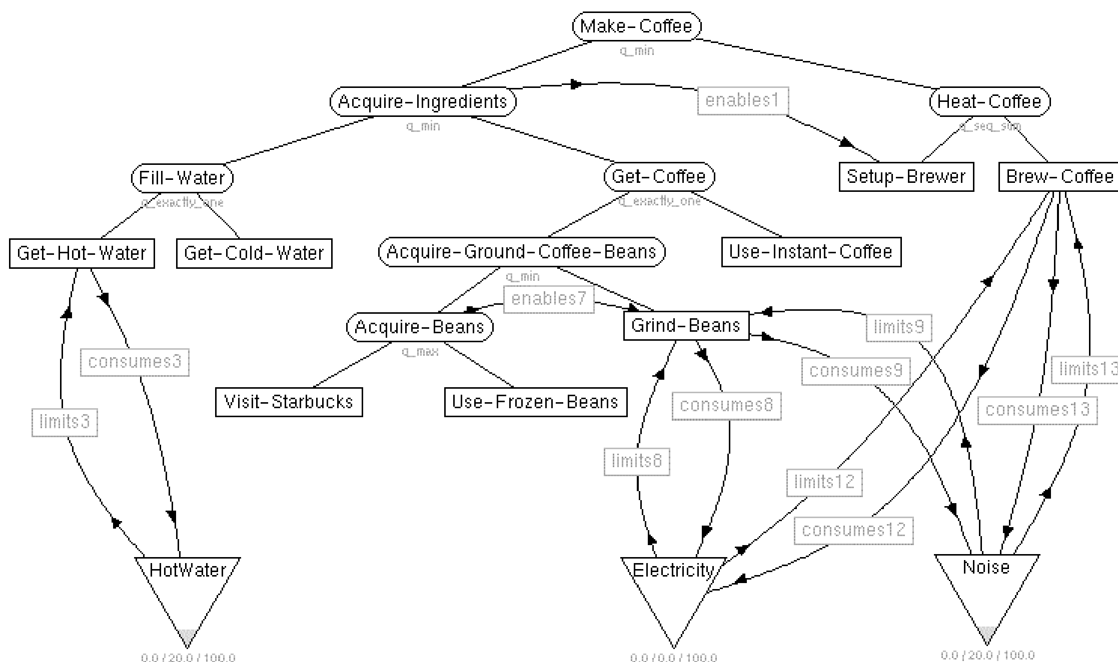


Figura 3.2: Estructura TÆMS describiendo cómo preparar un café

- *ProblemSolver*: es el responsable del comportamiento autónomo y racional del agente. El *ProblemSolver* actúa cuando algún otro componente coloca una estructura TÆMS en el componente *State*. En ese momento, *ProblemSolver* utiliza un algoritmo de *scheduling* independiente del dominio [96] para obtener un *schedule* de las tareas a ejecutar y las coloca nuevamente en el componente *State* para que *Execute* las ejecute.

  La estructura TÆMS con la cual trabaja el *ProblemSolver* representa un plan con las acciones que debe realizar el agente para cumplir con determinados objetivos. Obsérvese que dicha estructura no es construída por el *ProblemSolver*, sino que forma parte de los datos de entrada del mismo. *ProblemSolver* determina el tiempo de ejecución de cada acción de la estructura TÆMS, considerando cursos de acción alternativos, utilización de recursos, con¤ictos, etc.

JAF no provee ninguna herramienta gráfica de desarrollo, sin embargo, los componentes que provee pueden ser incorporados en cualquier herramienta compatible con *JavaBeans*, tal como *BeanBox* de Sun. *BeanBox* es una herramienta para componer *JavaBeans* mediante la cual es posible combinar los componentes de los agentes, interconectarlos mediante eventos, examinar y modificar sus propiedades. Además, permite incorporar componentes desarrollados por terceros.

JAF no define componentes relativos al comportamiento racional dirigido por objetivos, reacciones, aprendizaje y movilidad. Sin embargo, es posible implementar componentes responsables de dicha funcionalidad e incorporarlos a JAF con relativa facilidad. Esto se debe a que JAF está basado en *JavaBeans*, por lo tanto, puede ser usado con cualquier componente que implemente la interfaz *JavaBeans*.

3.6 JAFIMA

JAFIMA (Java Framework for Intelligent and Mobile Agents) [60] es un *framework* orientado a objetos para desarrollar agentes cognitivos con capacidades de percepción, comunicación, movilidad, reacción y comportamiento dirigido por objetivos.

JAFIMA ha sido desarrollado con el objetivo de producir una metodología y una plataforma para construir agentes robustos y mantenibles mediante técnicas de ingeniería de software y orientación a objetos.

El *framework* tiene una arquitectura de capas con las siguientes responsabilidades: percepción, creencias, razonamiento, acciones, colaboración, traducción y movilidad. En la figura 3.3 se muestra un esquema de la arquitectura, indicando las responsabilidades de cada una de las capas. En la derecha se puede observar el ¤ujo de información desde las capas inferiores hacia las superiores, mientras que en la izquierda se observa el ¤ujo de información en el sentido contrario.

El ¤ujo de información desde las capas inferiores hacia las superiores es: las creencias del agente están basadas en las entradas sensoriales. Cuando un agente tiene algún objetivo, razona acerca de sus creencias para determinar qué hacer. Luego, si decide realizar una acción, puede ejecutarla inmediatamente si dicha acción forma parte de sus capacidades, en caso contrario debe solicitar colaboración a otros agentes. Dentro de la capa de colaboración, el agente decide cómo negociar con otros agentes. Una vez que esta capa determina cómo hacerlo, formula un mensaje y lo pasa a la capa de traducción para que lo traduzca, en caso de ser necesario. Finalmente, la capa de movilidad transporta el mensaje al agente destinatario.

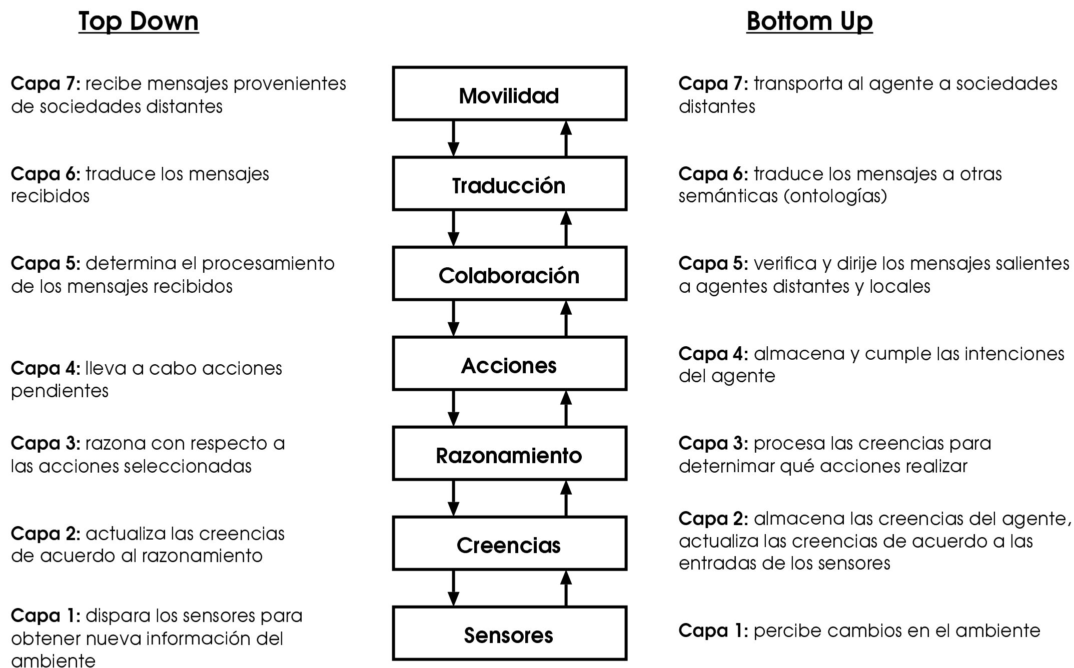


Figura 3.3: Arquitectura de niveles de JAFIMA [60]

El ¤ujo de información desde arriba hacia abajo es el siguiente: la capa de movilidad recibe los mensajes provenientes de otros agentes. Luego, en caso de ser necesario, los mensajes recibidos son

traducidos. La capa de colaboración administra las conversaciones y determina si un agente debe procesar un mensaje. En caso afirmativo, pasa el mensaje a la capa de acciones, y luego a la capa de razonamiento. Dependiendo del tipo de mensaje, el procesamiento puede continuar en la capa de creencias o de sensores.

Los agentes construidos con JAFIMA se comportan según patrones comportamentales estáticos basados en reglas, denominadas *expresiones regulares*. El *framework* no prescribe un mecanismo de razonamiento basado en *planning* o *scheduling*, ni provee medios para manipular los estados mentales del agente (sólo define su representación). Tampoco provee un *toolkit* que asista al programador en la instanciación del *framework*.

JAFIMA posee una gran ¤exibilidad, lo que permite extenderlo y adaptarlo a las necesidades de cada dominio de aplicación.

3.7 JAFMAS

JAFMAS [20, 21] es un *framework* Java para el dominio de sistemas multi-agente. El *framework* provee mecanismos de comunicación basados en KQML y coordinación utilizando un enfoque similar al de COOL.

El principal objetivo de JAFMAS consiste en proveer un conjunto de servicios para reducir el esfuerzo de construcción de agentes que actúan de manera cooperativa y coordinada.

En JAFMAS, un agente es un objeto Java capaz de interactuar mediante mensajes KQML punto a punto o *multicast*. Cada agente puede ofrecer servicios a otros agentes, o utilizar los servicios que otros proveen. Sin embargo, JAFMAS no provee un componente que actúe de facilitador de comunicaciones [40], manteniendo una lista con los agentes y sus localizaciones, y ofreciendo servicios de *brokering*.

Cada agente JAFMAS se define mediante un conjunto de clases de conversación similares a las utilizadas por COOL. La principal diferencia entre ambos consiste en que los agentes JAFMAS son capaces de procesar varias conversaciones en forma simultánea, debido a que pueden tener varios *threads* de ejecución procesando diferentes conversaciones.

JAFMAS sólo define agentes con capacidades de comunicación y coordinación. Por ejemplo, si se desea construir un agente que utilice *planning* para decidir qué acciones realizar, habrá que implementar el algoritmo de *planning*, los mecanismos de representación y manipulación de operadores, la representación del mundo, representación de planes, etc. Algo similar sucede con otros tipos de capacidades, tales como la percepción y movilidad.

El *framework* está basado en KQML, por lo tanto los agentes intercambian información simbólica representando conocimiento, planes, intenciones o creencias, entre otros. Sin embargo, el *framework* no define mecanismos para manipular ni representar esa información.

Adicionalmente, JAFMAS define una metodología genérica para construir agentes utilizando el *framework*. La metodología consta de cinco etapas: (i) identificar los agentes, (ii) identificar las conversaciones, (iii) identificar las reglas conversacionales, (iv) analizar y validar las conversaciones, y (v) implementar el sistema multi-agente.

Así, JAFMAS provee los servicios básicos de interacción y coordinación multi-agente utilizando KQML y conversaciones COOL. Sin embargo, no prescribe varias de las características de agen-

tes presentadas en la sección 2.1, en particular, lo único que define en relación al comportamiento de los agentes es el referido a las comunicaciones.

JAFMAS no provee ningún tipo herramienta para asistir al programador en la instanciación del *framework*. Sin embargo provee un ambiente para monitorear los agentes en ejecución.

3.8 MadKit

MadKit [53] es un *framework* Java para construir agentes con capacidades de comunicación a nivel de conocimiento, reactividad y percepción.

El principal objetivo de MadKit consiste en permitir la construcción de sistemas multi-agente formados por agentes heterogéneos. Por ejemplo, podrían haber agentes reactivos y deliberativos o agentes que se comunican con diferentes lenguajes o protocolos. Básicamente, el enfoque propuesto por MadKit consiste en no prescribir la arquitectura interna de los agentes, especificando sólo un conjunto de mecanismos de interacción y organización.

MadKit está basado en un modelo organizacional que estructura un sistema multi-agente en grupos de agentes con roles asociados. El modelo organizacional permite dividir un sistema multi-agente en grupos de agentes. Un agente puede pertenecer a cero o más grupos. Dentro de un grupo, cada agente tiene uno o más roles. Un rol representa la función de un agente, los servicios que provee o simplemente un identificador dentro de un grupo.

Un agente MadKit es una entidad activa capaz de comunicarse, pertenecer a uno o más grupos y ofrecer servicios. El *framework* no prescribe otras capacidades de agentes, tales como acción, deliberación o representación de estados mentales. Esto permite que el desarrollador defina los agentes de la manera que considere apropiada, aprovechando los mecanismos de comunicación y organización del *framework*.

MadKit posee un sub-*framework* para agentes reactivos que utiliza los mecanismos organizacionales de roles y grupos. Dicho sub-*framework* define agentes que habitan en un medio ambiente compuesto por otros agentes y objetos. Los agentes pueden percibir su medio ambiente, colocar marcas o dejar olores.

En general, MadKit posee un diseño ¤exible que permite que el diseñador extienda y adapte cada uno de sus componentes en función de sus necesidades. Sin embargo, el *framework* sólo prescribe capacidades de comunicación a nivel de conocimiento, reacción y percepción; no prescribe ninguna de las otras capacidades de agentes inteligentes descriptas en la sección 2.1.

3.9 ZEUS

ZEUS [71] es un *toolkit* Java para agentes deliberativos desarrollado por British Telecommunications. El *toolkit* provee componentes para comunicación, conversaciones, representación del conocimiento, reacción, *planning*, *scheduling* y coordinación.

ZEUS fue desarrollado con el objetivo de permitir que programadores con mínimos conocimientos sobre agentes, pudiesen desarrollar sistemas multi-agente con facilidad utilizando la metáfora de ma-

nipulación directa, y a la vez, permitir que los expertos extiendan la biblioteca de componentes. Para esto, ZEUS provee una herramienta gráfica que permite:

- Construir agentes, especificar sus capacidades, mecanismos de coordinación, ontologías, comportamiento y estados mentales.
- Visualizar las interacciones y estados mentales de agentes en ejecución.
- Producir reportes acerca de las interacciones, utilizaciones de recursos, etc.

La coordinación está basada en modelos organizacionales predefinidos, tales como cliente/servidor, superior/subordinado o compañero; además, ZEUS soporta *planning* multi-agente. Las interacciones se realizan mediante KQML.

Los agentes razonan utilizando *planning* y *scheduling*. Se asume que los agentes operan en un ambiente dinámico en el cual la tasa de cambio es, al menos, un orden de magnitud menor que el tiempo de razonamiento. Esto se debe a que si el medio ambiente se modifica mientras el agente decide qué hacer utilizando *planning*, entonces el plan construido podría contener errores o inconsistencias respecto de lo que ocurre en el ambiente.

Es importante destacar que el razonamiento de los agentes sigue un modelo comercial típico, en el cual existen recursos, productores y consumidores. La mayor parte de los diálogos e interacciones son del siguiente tipo: un agente necesita un recurso, entonces determina quién/quienes proveen ese recurso mediante su propio estado mental o consultando con un agente *broker*. Luego, mediante un protocolo similar a *contract-net* obtiene el recurso del proveedor que lo ofrece a menor costo.

ZEUS es presentado por sus autores como un "*toolkit genérico y adaptable*". Esas características son logradas mediante un *framework* Java que soporta al *toolkit*. Sin embargo, las clases de ese *framework* no están diseñadas para ser redefinidas por el programador. Por ejemplo, en la clase que implementa el *planner*/*scheduler*, la mayor parte de los métodos son específicos del algoritmo implementado. Además, no es posible personalizar o adaptar los componentes del agente utilizando otras clases sin realizar modificaciones considerables al conjunto de clases básicas. En general, esto se debe a que el *framework* no define cada uno de los componentes de manera abstracta, sino que únicamente define componentes concretos.

Por otro lado, el *toolkit* de ZEUS sólo permite especificar un conjunto de métodos Java que implementan las capacidades básicas de un agente. Sin embargo, no permite definir dichos métodos en el *toolkit*, ni tampoco posee un formalismo de alto nivel para especificar el comportamiento del agente, tales como RADL o TÆMS.

3.10 Conclusiones

En las secciones anteriores se describieron y analizaron algunas herramientas para construir agentes según los criterios definidos en la sección 3.1. La tabla 3.1 sumariza las capacidades de los agentes que cada herramienta soporta y la ¤exibilidad de la implementación de cada una de esas capacidades, mientras que la tabla 3.2 muestra las características generales de las herramientas.

En la tabla 3.1 pueden observarse dos grupos de herramientas claramente diferenciadas:

- las que proveen una biblioteca de componentes y no definen esos componentes en forma abstracta (AgentBuilder, DECAF y ZEUS): la ¤exibilidad de estas herramientas es limitada, en

	AgentBuilder	DECAF	FraMaS	JAF	JAFIMA	JAFMAS	MadKit	ZEUS
Percepción			+++	++	+++		+++	
Representación de sí mismo	++	++						++
Representación de los estados mentales	++	++		++				++
Manipulación de los estados mentales	++							
Reacción	++			+++	+++		+	++
Comunicación a nivel de conocimiento	++	+++		+++	+++	++	+++	++
Coordinación	++					+++	+++	++
Movilidad			+++		+++		+	
Deliberación	++	++	+++		+++			++
Dirigido por objetivos	++	++			+++	++		++
Planning (on-line)					+++			++
Scheduling		++						++
Aprendizaje			+++					

+ Imposibilidad de adaptación o extensión.
++ Adaptabilidad limitada.
+++ Adaptabilidad y extensibilidad.

Tabla 3.1: Capacidades de agentes soportadas por las herramientas

cuanto a que no es posible extender los diversos componentes o desarrollar otros componentes para reemplazar los provistos por la herramienta.

- las que definen la funcionalidad de sus componentes de forma abstracta (quizás mediante clases abstractas) y proveen componentes concretos basados en esas especificaciones (FraMaS, JAF, JAFIMA, JAFMAS y MadKit): éste grupo de herramientas ofrecen mayor reusabilidad y ¤e-xibilidad que las anteriores. Sin embargo, se observa que sólo soportan un subconjunto de las capacidades de agentes presentadas en la sección 2.1. Además, están orientadas al programador experto, por cuanto no ofrecen *toolkits* con el nivel de sofisticación de AgentBuilder o ZEUS.

Tres de las herramientas analizadas no definen la arquitectura de los agentes (JAF, JAFMAS y MadKit). De esta forma, el programador es responsable de diseñar y materializar una arquitectura para los agentes utilizando el soporte provisto por la herramienta, por lo tanto, posee libertad de definir la arquitectura de los agentes apoyándose en el soporte ofrecido por la herramienta. En las otras herramientas, el desarrollador se ve forzado a utilizar una arquitectura para sus agentes, sin embargo, el esfuerzo involucrado en la construcción de dichos agentes se ve reducido en forma considerable respecto del primer enfoque.

Una herramienta debería encontrarse en el centro del espectro definido por esos enfoques, prescribiendo la arquitectura general de los agentes de forma tal de evitar que el desarrollador deba diseñarla, y a la vez ofreciendo ¤exibilidad y adaptabilidad a los diferentes dominios de aplicación [88].

Con respecto a las capacidades de agentes, hay algunos aspectos que no son tratados por las herramientas:

- *no poseen mecanismos para mantener una representación del ambiente coherente y actualiza-*

	Tipo	Objetivos	Arquitectura de Agentes	Tipo de Agentes	Lenguaje
AgentBuilder	*Toolkit*+código, reglas RADL, clases Java externas.	Facilidad de uso.	Basada en BDI con reglas y conversaciones.	Deliberativos con comportamiento basado en reglas.	Java, RADL.
DECAF	*Toolkit* (sólo planes), clases Java externas, *framework* (planes), componentes concretos.	Facilidad, investigación.	Basada en BDI con planes estáticos.	Deliberativos con comportamiento basado en planes estáticos.	Java, RJT.
FraMaS	*Framework*, componentes concretos.	Reusabilidad	Capacidades combinables organizadas en capas.	Deliberativos (de£nición abstracta), aprendizaje con CBR	Java.
JAF	Biblioteca de componentes, *toolkit* extensible.	Reusabilidad	No especi£ca, componentes *JavaBeans*, eventos.	Deliberativos, scheduling de estructuras TÆMS.	Java, TÆMS.
JAFIMA	*Framework*, componentes concretos.	Metodología, plataforma genérica.	Capas, basado en BDI con planes estáticos y conversaciones.	Móviles, híbridos planes estáticos.	Java.
JAFMAS	*Framework*, componentes concretos.	Servicios para cooperación y coordinación.	No especi£ca.	Deliberativos (sólo prescribe comunicaciones y conversaciones).	Java.
MadKit	*Framework*, componentes concretos.	Plataforma para MAS heterogéneos y organizados	No especi£ca.	No especi£ca.	Java.
ZEUS	*Toolkit*+código, *framework* (clases concretas), componentes concretos, clases Java externas.	Gran facilidad de uso, extensible.	Basada en BDI con colaboración y planning.	Deliberativos con planning y patrones de coordinación.	Java.

Tabla 3.2: Tipo, objetivos de diseño, arquitectura de agentes y lenguaje de cada una de las herramientas

da: la representación del ambiente no debería admitir la existencia de información incoherente, tal como un objeto que se encuentra en dos lugares en forma simultánea o un vehículo que se mueve y está quieto, al mismo tiempo.

La información que un agente posee sobre al ambiente, incluyendo a los agentes que lo rodean, se denominan creencias [104]. Los agentes deliberativos razonan en función de sus estados mentales, lo que incluye a las creencias. Esto implica que un agente que posee creencias incoherentes o desactualizadas respecto de lo que occure realmente en el ambiente podría actuar de manera incorrecta.

Por ejemplo, un agente a_1 que intenta tomar una caja posee un rango de percepción limitado. Dicho agente cree que la caja se encuentra en la posición (x_1, y_1) tal que esa posición está fuera del rango de sus capacidades de percepción. Otro agente a_2 toma la caja, la transporta a otra posición (x_2, y_2) y le informa a a_1 la nueva localización de la caja. El agente a_1 debería actualizar sus creencias, re¤ejando la nueva posición de la caja.

En general, éste aspecto sólo es tratado en forma parcial por las herramientas, representando al ambiente mediante estructuras estáticas o descripciones simbólicas manipuladas por el programador en forma *ad-hoc*. Sin embargo, ninguna de las herramientas provee soporte para construir agentes que poseen una descripción simbólica del ambiente, y a la vez, mantienen esa información actualizada respecto de lo que ocurre en el ambiente utilizando las capacidades de percepción o comunicación, y analizando la coherencia de las mismas.

El problema de determinar si dos creencias son incoherentes es dependiente de la aplicación. Así, por ejemplo, en determinada aplicación podría ser válido creer que dos objetos se encuentran en la misma coordenada (x, y), quizás debido a que existe una coordenada z que no es utilizada por el agente. Por tal razón, es deseable que una herramienta para construir agentes permita que el programador especifique información sobre el dominio que permita a la herramienta mantener la coherencia de las creencias.

- *no consideran la posibilidad de que un agente realice varias actividades concurrentemente* [88]. Por ejemplo, un agente podría construir un plan de acción para alcanzar sus objetivos, percibir mediante sus sensores, reaccionar a los cambios en el ambiente, modificar sus creencias y objetivos, mantener conversaciones con otros agentes, etc.

  Los agentes construidos con las herramientas existentes sólo pueden realizar una actividad a la vez. Por ejemplo, mientras planean analizando cuidadosamente qué acciones ejecutar para lograr sus objetivos, no perciben, no procesan comunicaciones ni son capaces de reaccionar rápidamente ante situaciones predeterminadas, tales como un choque.

- *no permiten que los agentes deliberen utilizando diferentes mecanismos concurrentes*: existen diferentes mecanismos deliberativos para que un agente decida qué acciones realizar. Considerando sólo las herramientas descriptas en el presente capítulo se encuentran: selección de planes estáticos en función de los objetivos (AgentBuilder), *scheduling* (JAF), planes estáticos describiendo conversaciones multi-agente (JAFMAS y AgentBuilder), mecanismos de desición utilizando la experiencia (FraMaS) y *planning on-line* (ZEUS). Una herramienta genérica debería permitir la construcción de agentes que razonan utilizando varios mecanismos deliberativos en forma concurrente [88], dependiendo de la naturaleza de los objetivos, del dominio de aplicación, etc.

  Por ejemplo, un agente planea la forma de caminar hacia determinado lugar de una casa, en

forma simultánea, mantiene una conversación con otro agente acerca de una compra/venta de un producto, analizando cuidadosamente las propuestas y contrapropuestas realizadas por el otro agente; e intenta recordar una experiencia sobre un negocio similar resuelto satisfactoriamente en el pasado.

El primer punto ha sido estudiado por las diversas teorías de revisión de creencias [78], sin embargo, no ha sido incluído en herramientas para desarrollar agentes. Los otros dos puntos, son considerados como unos de los principales requerimientos de las herramientas genéricas para desarrollo de agentes [88] actuales.

CAPÍTULO 4

Brainstorm/J

En el capítulo anterior se describieron y analizaron herramientas existentes para desarrollar agentes inteligentes. Esas herramientas proveen componentes de software que implementan un subconjunto de las capacidades de agentes descriptas en el capítulo 2.

Las herramientas genéricas para construir agentes se enfrentan a dos problemas. En primer lugar, la gran variedad de aplicaciones en las que es posible utilizar agentes dificultan el desarrollo de componentes genéricos responsables de las capacidades de agentes. En segundo lugar, la gran variedad de capacidades de agentes [72] y las diferentes formas de combinar esas capacidades dan origen a numerosos tipos de agentes, por lo que resulta aún más problemático abstraer toda la funcionalidad de los agentes en un herramienta genérica.

Esto implica que una herramienta genérica para construir agentes debería permitir la construcción de agentes de diversos tipos con variadas capacidades, permitiendo que el programador adapte y extienda las capacidades de agentes definidas por la herramienta según los requerimientos de cada aplicación. Así, por ejemplo, el programador podría utilizar una herramienta para construir agentes que utilizan un algoritmo de *planning* predefinido, especificando sólo los aspectos dependientes de la aplicación, tales como las acciones o parámetros del algoritmo. Adicionalmente, la herramienta podría permitir que el programador utilice sus propios algoritmos de *planning* o extienda los algoritmos predefinidos.

De esta manera, si una herramienta define un conjunto de componentes de software responsables de las capacidades de agentes, entonces dicha herramienta podría permitir que el programador utilice dichos componentes, los adapte y redefina según sus necesidades o defina sus propios componentes, ya sea desde cero o basado en los componentes existentes; lo cual permitiría utilizar la herramienta para construir agentes en diversos dominios de aplicación. Los *frameworks* orientados a objetos [59] han sido aplicados con éxito en este tipo de circunstancias.

Como se puede observar en el capítulo anterior, las herramientas existentes permiten construir agentes con un subconjunto de las capacidades descriptas en el capítulo 2, limitando la extensibilidad a sólo algunas de ellas. Por otro lado, no ofrecen soporte para aspectos importantes tales como [88]:

- representación del ambiente, coherencia de dicha representación, actualización en función de los cambios del mismo.
- concurrencia en los procesos internos del agente.
- deliberación utilizando diferentes mecanismos concurrentes.

La propuesta aquí presentada, Brainstorm/J, es un *framework* Java para construir agentes inteligentes basado en la arquitectura Brainstorm. El mismo permite construir agentes inteligentes con las capacidades presentadas en el capítulo 2. Además, provee soporte para los aspectos enumerados anteriormente.

El *framework* especifica componentes de software reusables responsables de las capacidades de agentes, definiendo las interfaces y el ¤ujo de control entre ellos. Esos componentes pueden ser adaptados o extendidos, permitiendo que el programador desarrolle agentes a partir de ellos. Así, por ejemplo, es posible utilizar componentes que definen capacidades de comunicación utilizando KQML o definir componentes para otros lenguajes de comunicación tales como ACL.

La extensibilidad del *framework* permite agregar nuevas capacidades de agentes, especializarlas según el dominio de aplicación o extender su funcionalidad. Esto se logró por medio de los mecanismos de la programación orientada a objetos tales como herencia, métodos *hook* y abstractos.

Adicionalmente, Brainstorm/J permite construir agentes que mantienen una representación del medio ambiente en que se encuentran, asegurando la coherencia de la misma y razonando en función de ella. Los agentes pueden deliberar y, en forma simultánea, percibir, mantener conversaciones con otros agentes, actuar, etc. Además, un agente puede razonar, en forma simultánea, sobre cómo lograr sus objetivos utilizando varios mecanismos deliberativos.

El *framework* ha sido implementado utilizando JavaLog, un lenguaje multi-paradigma basado en Java y Prolog [5], y una extensión Java para meta-objetos, denominada JMOP, basada en Luthier-MOPS [18]. En los apéndices A y B se describen JMOP y JavaLog, respectivamente.

A continuación se describe la organización del capítulo. En la sección 4.1 se describe brevemente la materialización de los componentes arquitectónicos de Brainstorm en el *framework* Brainstorm/J. Luego, la sección 4.2 presenta la materialización del componente *MetaAgent*, el cual es responsable de la creación de los agentes. A continuación, la sección 4.3 describe la representación y utilización de las capacidades básicas de acción. Las secciones 4.4 a 4.9 presentan la materialización de los componentes arquitectónicos responsables de las capacidades de acción, percepción, detección de situaciones, reactividad, movilidad, deliberación y comunicación; respectivamente.

4.1 Materialización de los componentes arquitectónicos

En la figura 4.1 se muestra la materialización de la arquitectura en dos etapas. La primera etapa muestra el mapeo de componentes arquitectónicos a clases. La segunda etapa muestra las principales clases del *framework* y sus relaciones, las cuales constituyen la estructura abstracta del *framework*.

En la figura 4.1(b) se muestran las principales clases del *framework* organizadas de forma similar al esquema de Brainstorm de la figura 4.1(a). Las principales diferencias entre ellos se deben a adaptaciones y extensiones realizadas al *framework* luego de la materialización de la arquitectura para posibilitar movilidad y comunicaciones entre agentes físicamente distribuidos.

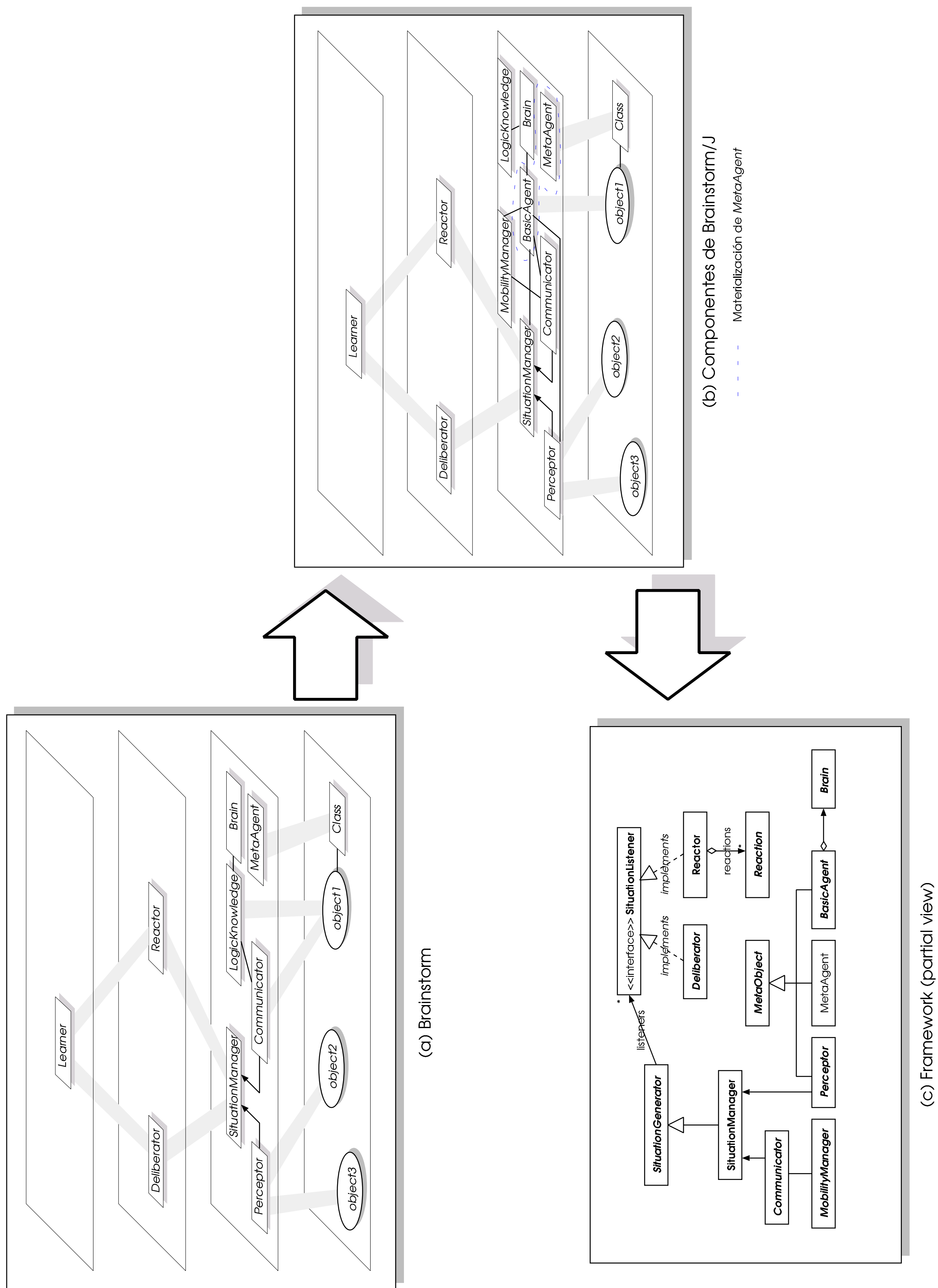


Figura 4.1: Materialización de Brainstorm

La materialización de los componentes de Brainstorm se realizó de la siguiente forma:

- *Perceptor*: permite que un agente observe el ¤ujo de mensajes entre objetos. La materialización de este componente se realizó mediante la clase `Perceptor`. Las instancias de dicha clase son meta-objetos que pueden ser asociados a uno o más objetos con el fin de interceptar los mensajes.

- *Communicator*: es responsable por manejar protocolos de comunicación de alto nivel tales como KQML. Dicho componente es un meta-objeto que actúa cuando el objeto de nivel base recibe un mensaje que pertenece al lenguaje de comunicación del agente. Así, para enviar un mensaje *m* perteneciente al lenguaje de comunicación de un agente cuyo objeto base es *o* se utiliza el mecanismo de mensajes propio de un lenguaje orientado a objetos, por ejemplo *o.m()*. Cuando un agente recibe un mensaje, éste es interceptado por el meta-objeto de comunicación, el cual lo interpreta.

  En Brainstorm/J, la materialización de este componente soporta comunicaciones entre agentes físicamente distribuidos, para lo cual puede utilizar varios medios de transporte para los mensajes entre agentes. Por ejemplo, los mensajes podrían enviarse utilizando correo electrónico, TCP/IP o mensajes entre objetos.

  En Brainstorm/J, las comunicaciones entre agentes por medio de mensajes se realiza invocando el método `sendMessage` del componente de comunicación de cada agente. Dicho método envía un mensaje al *agente* destinatario del mismo utilizando un mecanismo de transporte especificado por el programador. La principal diferencia respecto de Brainstorm consiste en que no es necesario que el componente de comunicación sea un meta-objeto, debido a que el destinatario del mensaje nunca es el objeto de nivel base, sino que es un *agente* con un identificador asociado. Por ejemplo, el identificador puede ser la dirección física del sitio en que se encuentra y un identificador único dentro de ese sitio tal como el número de *port* o nombre de usuario. Nótese que este mecanismo sirve tanto para comunicar agentes situados en diferentes sitios como para agentes de un mismo sitio.

  Este componente se materializó en la clase abstracta `Communicator`.

- *LogicKnowledge* y *Brain*: estos componentes permite utilizar cláusulas lógicas en objetos. La materialización de estos componente se realizó mediante una integración de Prolog y Java denominada JavaLog [5] (apéndice B). Sin embargo, en dicha materialización no se utilizaron meta-objetos, sino que las clases que contienen cláusulas lógicas se convierten, mediante un pre-procesador, en clases Java estándar. La clase `Brain` implementa la funcionalidad del componente *Brain* utilizando JavaLog. Básicamente, el mecanismo de pre-procesamiento es equivalente a utilizar meta-objetos, ya que la funcionalidad de los objetos situados en el nivel base se combina con la de los meta-objetos en tiempo de compilación, mientras que en Brainstorm esto se realiza en tiempo de ejecución.

- *MetaAgent*: es el componente encargado de crear e inicializar agentes. Se materializó en las clases `MetaAgent` y `BasicAgent`. La primera de ellas crea los agentes e inicializa sus componentes colaborando con `BasicAgent`. Además, las instancias de `BasicAgent` son responsables de mantener información sobre una instancia particular de un agente, tales como su identidad y componentes con los que está formado, e interceptar los mensajes recibidos por el objeto de nivel base para modificar su comportamiento.

  Este componente se materializó en dos clases debido a que parte del proceso de instanciación

del agente utiliza conocimiento administrado por los componentes *Brain* y *LogicKnowledge*, por lo que requiere de un objeto de la clase `Brain` perteneciente a un agente en particular. El proceso que no requiere de ese conocimiento, y por lo tanto del objeto `Brain`, es responsabilidad de `MetaAgent`, el resto es responsabilidad de una instancia de `BasicAgent` que posee una referencia al objeto `Brain` del agente.

- *SituationManager*: se materializó mediante la clase concreta `SituationManager`.
- *Reactor* y *Deliberator*: son responsables por el comportamiento reactivo y deliberativo, respectivamente. Están materializados por las clases `Reactor` y `Deliberator`. En Brainstorm, estos componentes son meta-objetos que interceptan el mensaje *newSituation* generado por el *SituationManager* cuando detecta una situación de interés. Dado que el protocolo de interacción entre el administrador de situaciones y los componentes responsables del comportamiento del agente está bien definido y es estable, se optó por materializar dichos componentes como objetos simples.

  La materialización del *SituationManager* y de los componentes de reacción y deliberación constituyen una instancia del patrón observador [45], en el que el administrador de situaciones actúa de sujeto observado y los otros componentes actúan de observadores. Así, se logra la misma funcionalidad que la que se obtendría mediante meta-objetos, es decir, que el administrador de situaciones notifique, en forma indirecta, a los *interesados* en situaciones.

Para soportar movilidad se diseñó un nuevo componente encargado de proveer dicho servicio. El componente *MobilityManager* permite transportar un agente entre diferentes sitios de una LAN. La clase `MobilityManager` implementa esta funcionalidad utilizando el *framework* Aglets [63] para movilidad de objetos Java.

Como observa en la figura 4.1(b), varios de los componentes del *framework* son meta-objetos, por lo que fue necesario desarrollar soporte para meta-objetos para el lenguaje Java. Para tal fin, se implementó un *framework* para meta-objetos, denominado JMOP, basado en LuthierMOPS [18]. En el apéndice A se describe JMOP.

En la figura 4.1(c) se presenta la jerarquía de clases de Brainstorm/J en notación UML[1]. La clase abstracta `MetaObject` define el protocolo de meta-objetos en el método abstracto *handleMessage*. Básicamente, cuando un objeto reflejado recibe un mensaje *m*, su meta-objeto asociado es notificado con el mensaje *handleMessage(m)*. En el *framework*, hay tres subclases de `MetaObject`: `MetaAgent`, `BasicAgent` y `Perceptor`, las cuales redefinen el método *handleMessage* de acuerdo a la funcionalidad que cada una de ellas implementa, como se verá en las siguientes secciones.

La clase `SituationGenerator` y la interfaz `SituationListener` definen un mecanismo de invocación implícita basado en el patrón observador que es utilizado por la clase `SituationManager` para notificar a los interesados en situaciones. Básicamente, `SituationGenerator` posee métodos para agregar, eliminar y notificar a los *listeners* (interesados en situaciones). De esta forma, cuando el administrador de situaciones detecta una nueva situación y se envía a sí mismo el mensaje `newSituation`, todos los objetos subscritos son notificados.

Las clases `Reactor` y `Deliberator` implementan la interfaz `SituationListener`, por lo tanto, las instancias de estas clases pueden subscribirse con el administrador de situaciones para ser notificadas cuando se detecta una situación de interés.

[1] En el apéndice D se describe brevemente UML.

En las secciones siguientes se describe el *framework* con mayor detalle.

4.2 Creación de un agente

Brainstorm prescribe un componente arquitectónico llamado *MetaAgent*, el cual es responsable de crear los agentes. Dicho componente fue materializado en el *framework* por las clases `MetaAgent` y `BasicAgent`.

Para crear un agente, se debe asociar un meta-objeto de la clase `MetaAgent` a una clase que se ubica en el nivel base, como se muestra en la figura 4.2. De esta forma, cada vez que se crea una instancia de la clase de nivel base mediante el mensaje `new` (pasos 1 y 2 de la figura), se intercepta dicho mensaje y actúa el meta-objeto (paso 3). Este meta-objeto crea los meta-objetos y objetos auxiliares necesarios para formar el agente (paso 4), los asocia re¤exivamente con el objeto de nivel base (paso 5) y los inicializa. Finalmente, retorna el control al método constructor de nivel base (paso 6).

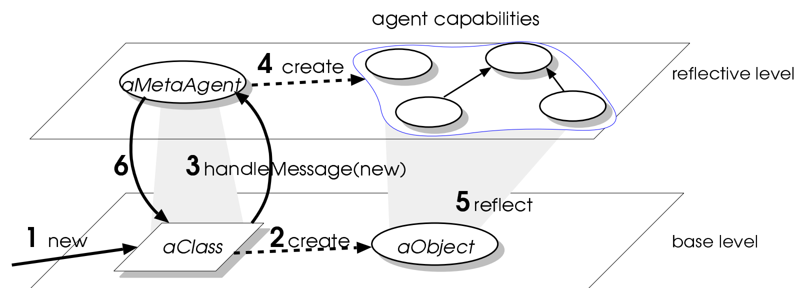


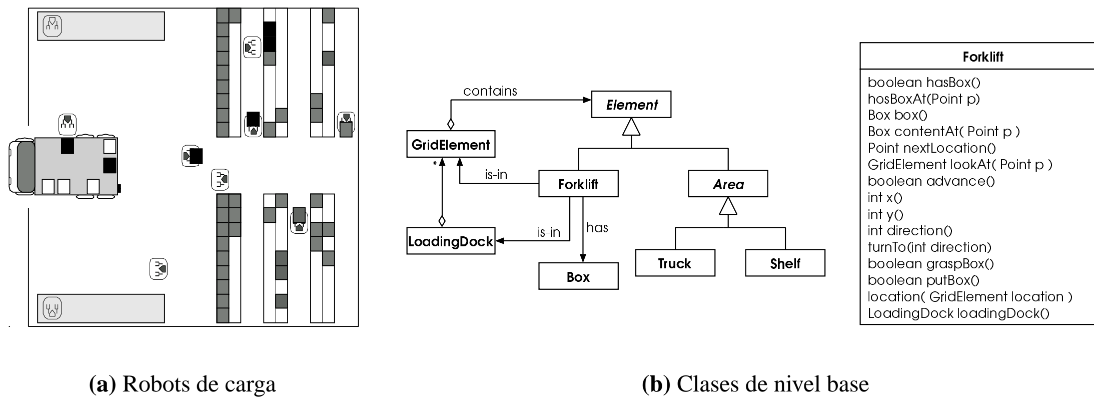
Figura 4.2: Creación de un agente

Con el objetivo de clarificar el mecanismo de creación de agentes de Brainstorm/J, se describirá una aplicación consistente de un conjunto de robots cuyo objetivo es descargar un número de cargas de un camión y colocarlas en zonas de descarga [42]. En la figura 4.3(a) se muestra un diagrama de la aplicación. Consiste de una grilla rectangular dividida en regiones. Una región contiene un camión del cual los robots toman las cargas. Las zonas en gris claro situadas en la parte inferior y superior izquierda de la grilla son los lugares de los cuales parten inicialmente los robots. También hay regiones de descarga (o estanterías) en las cuales es posible depositar las cargas y, finalmente, zonas de libre tránsito.

Cada robot puede llevar una carga a la vez, y sólo puede tomarla o dejarla en una celda inmediatamente en frente de él. Además, un robot puede girar hacia cualquiera de los puntos cardinales y avanzar en el sentido en que se encuentra, es decir, que no puede, por ejemplo, avanzar directamente en diagonal. Los robots están dotados de un sensor simple que les permite percibir y detectar el objeto que se encuentra inmediatamente en frente.

En la figura 4.3(b) se muestra el diagrama de clases de la aplicación. En este modelo sólo se define el comportamiento básico de los robots (instancias de la clase `Forklift`): avanzar, girar, cargar, descargar y mirar. Es decir, el modelo no define las operaciones para caminar hacia el sitio de carga, tomar una carga, caminar al sitio de descarga, etc. Este comportamiento *inteligente* será definido utilizando el *framework*, es decir, que a cado uno de los robots se asociarán capacidades de agentes.

En Brainstorm/J, un agente está compuesto por un objeto de nivel base, y un nivel re¤exivo compuesto de conjunto de meta-objetos y objetos que implementan las capacidades de agentes. En este ejemplo,



(a) Robots de carga

(b) Clases de nivel base

Figura 4.3: La aplicación de robots de carga

el objeto de nivel base es una instancia de la clase `Forklift`. A esa clase se le debe asociar un meta-objeto, instancia de `MetaAgent`, encargado de crear el nivel re¤exivo cada vez que se crea una instancia de la clase de nivel base. Así, para crear un agente con un objeto de nivel base de la clase `Forklift` se asociará un meta-objeto de la clase `MetaAgent` a dicha clase.

Básicamente, el meta-nivel de un agente *forklift* contendrá un objeto `Brain`, uno o más meta-objetos de percepción, un objeto de comunicación, un objeto administrador de situaciones, un objeto de reacción y uno de deliberación. En el diagrama de clases de Brainstorm/J mostrado en la figura 4.1(c) se pudo apreciar que existen clases para cada uno de esos objetos y meta-objetos, con lo que para crear un agente, un objeto `MetaAgent` simplemente crea instancias de esas clases según corresponda. El usuario del *framework* puede especializar alguna de esas clases. Esto significa que un agente, por ejemplo, podría no estar compuesto por instancias de `Deliberator`, sino que podría utilizar instancias de una subclase de la misma, tal como `ForkliftDeliberator`. El *framework* permite que el programador especifique cuáles son las clases que definen las capacidades de un agente.

La clase `MetaAgent` define un método *template* llamado `createAgent` que es invocado para crear el nivel re¤exivo. Dicho método invoca a un conjunto de métodos *factory* responsables de crear los objetos y meta-objetos que forman el meta-nivel. En `MetaAgent` existen un conjunto de métodos que permiten especificar las clases de los objetos y meta-objetos que componen un agente. Por ejemplo, invocando el método `deliberatorClass` es posible especificar la clase del componente deliberativo del agente.

En la figura 4.4 se muestra un diagrama con las principales clases involucradas en la creación de un agente y su especialización para la aplicación de robots de carga.

Cuando se crea un meta-objeto de la clase `MetaAgent`, se asocia re¤exivamente a una clase de nivel base. De esta manera, cada vez que se crea una instancia de dicha clase base se activa el meta-objeto, que le asocia capacidades de agentes utilizando el método `createAgent`.

Varios de los componentes de un agente son opcionales y dependen del tipo y de las capacidades del mismo. Por ejemplo, un agente reactivo no utiliza componente deliberativo, ya que todo su comportamiento es responsabilidad del componente reactivo. La clase `MetaAgent` posee variables de instancia que indican la presencia o ausencia de determinado componente. Además, posee métodos para acceder a dichas variables y especificar las capacidades de un agente. Por ejemplo, el método

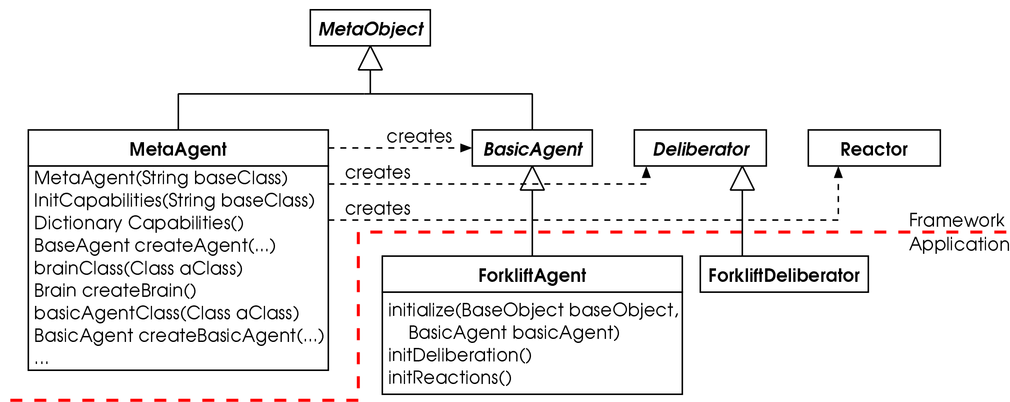


Figura 4.4: Diagrama de clases de la aplicación de robots de carga

`hasReaction` con un parámetro `true` se utiliza para indicar que los agentes poseen capacidad de reacción.

Para crear agentes *forklift* con capacidades de reacción, comunicación y deliberación hay que crear un meta-objeto de la clase `MetaAgent` indicando que la clase base es `Forklift`:

```
// Refleja la clase Forklift con una instancia de MetaAgent
MetaAgent metaAgent=new MetaAgent( "Forklift" );
```

luego, se deben especificar las capacidades del agente invocando a los métodos `hasReaction` y `hasDeliberation`:

```
// El agente tiene reacción y deliberación
metaAgent.hasDeliberation(true); metaAgent.hasReaction(true);
```

finalmente, deben indicarse las clases responsables de las capacidades de reacción y deliberación (si no son las predefinidas por el *framework*), e indicar una subclase de `BasicAgent` responsable de inicializar el agente:

```
// El componente de reacción por defecto es Reactor
// El deliberador es instancia de ForkliftDeliberator
metaAgent.deliberatorClass( ForkliftDeliberator.class );
// El componente BasicAgent se especializó mediante ForkliftAgent
metaAgent.basicAgentClass( ForkliftAgent.class );
```

Obsérvese que no se especifican todas las clases que componen al agente, ya que el *framework* posee definiciones predefinidas para algunas de ellas. Por ejemplo, la clase `Reactor`, por defecto, implementa las capacidades de reacción.

La inicialización de los componentes de un agente son responsabilidad de la clase `BasicAgent`. La misma define varios métodos *hook* que son utilizados para inicializar cada uno de los componentes de un agente. Por ejemplo, el método `initReactions` se utiliza para definir cuáles son las reacciones del agente ante ciertas situaciones.

Una instancia de `BasicAgent` posee referencias a los objetos y meta-objetos que componen un agente. En particular, posee una referencia al objeto `Brain` del agente, el cual es utilizado para inicializar

los estados mentales del agente, sus situaciones, capacidades, etc. Nótese que el objeto `Brain` es necesario en métodos que utilizan el lenguaje JavaLog para inicializar alguno de los componentes del agente. Por ejemplo, las situaciones de interés deben ser especificadas mediante JavaLog; también puede ser necesario inicializar las creencias y objetivos del agente.

En la figura 4.5 se sumarizan las interacciones ocurridas durante la creación de un agente.

Figura 4.5: Diagrama de interacción de la creación de un agente

4.3 Capacidad de acción

En Brainstorm/J, las capacidades básicas de acción de un agente están formadas por algunos de los métodos del objeto base que modifican el medio ambiente.

Las capacidades son representadas por la clase concreta `AgentCapability` mostrada en la figura 4.6. Las instancias de dicha clase están compuestas por un objeto que representa un método público del objeto base del agente. El método `execute`, simplemente invoca dicho método.

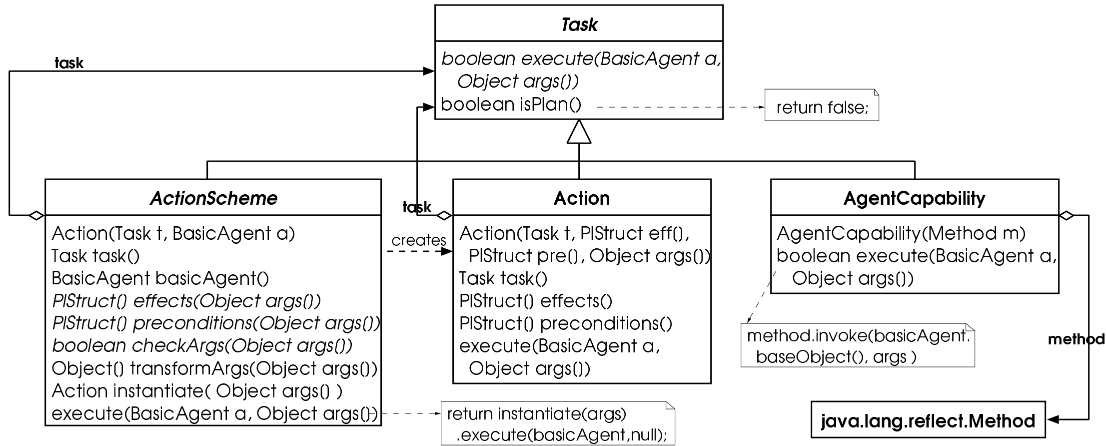


Figura 4.6: Representación de las capacidades de acción

Las capacidades básicas de un agente son especificadas utilizando el método `addCapability` de la clase `MetaAgent`. Por ejemplo, en la aplicación de robots, la capacidad de rotación de cada robot está implementada en el método `turnTo(int dir)`. Para agregar dicho método al conjunto de capacidades de un agente *forklift* debe invocarse el método `addCapability` de `MetaAgent`:

```
addCapability( turnTo, Integer.TYPE )
```

Para que un agente deliberativo pueda utilizar sus capacidades, por ejemplo, para construir un plan, es necesario que posea una descripción acerca de los efectos que dichas capacidades tienen sobre el mundo y las precondiciones de cada una de ellas. A tal fin, el *framework* provee la clase concreta `Action`, la cual describe una tarea[2] en término de sus parámetros, precondiciones, efectos. Éstas dos últimas son representadas mediante predicados Prolog.

En la clase `Action` se define el método *template* `execute` (heredado de `Task`) para que una vez ejecutada una acción, actualice las creencias del agente acerca del ambiente de acuerdo a los efectos de dicha acción. Por ejemplo, un agente posee la siguiente capacidad:

turnTo(*DirI*, *DirF*)

PARÁMETROS: dirección(*DirI*)$\wedge$dirección(*DirF*)$\wedge$*DirF*$\neq$*DirI*

PRECONDICIÓN: mirandoHacia(*DirI*)

EFECTO: mirandoHacia(*DirF*)$\wedge\neg$mirandoHacia(*DirI*)

El agente se encuentra mirando hacia el sur, por lo que se estado mental contiene: mirandoHacia(sur). Luego, ejecuta la acción turnTo(sur, oeste), con lo que sus creencias se modifican, resultando:

[2]Esto incluye a las capacidades de un agente de actuar sobre el mundo y a las capacidades de actuar sobre sí mismo.

mirandoHacia(sur)$\wedge\neg$mirandoHacia(oeste)

En el diagrama de clases de la figura 4.6 se muestran dos clases con características muy similares: `Action` y `ActionScheme`. La primera de ellas representa una acción con sus parámetros instanciados, es decir, que sus parámetros sólo contienen objetos (no variables). La segunda clase, por el contrario, representa una acción con parámetros no instanciados (sólo variables), tal como mover(*R*, *Origen*, *Destino*). Una acción de ese tipo se denomina *esquema de acción* u *operador*. Obsérvese que al instanciar los parámetros de un esquema de acción se obtiene una acción común.

Por ejemplo, en la aplicación de robots, la clase `Forklift` define un método `turnTo(int i)`. El parámetro $i \in [0 \dots 3]$ indica la dirección en que debe girar el robot (0=norte, 1=sur, 2=este, 3=oeste.). Luego, podría definirse una subclase de `ActionScheme` que implemente los métodos apropiados para especificar la precondición, efectos y parámetros de `turnTo`:

```
public PlStruct[] preconditions( Object args[] ) {
  //args={DirI, DirF}
  // retorna: lookingAt(DirI)
  return new PlStruct[] { {%lookingAt(#args[0]#).%} };
}

public PlStruct[] effects( Object args[] ) {
  //args={DirI, DirF}
  // retorna:  not(lookingAt(DirI)), lookingAt(DirF))
  return new PlStruct[] { {%not(lookingAt(#args[0]#)).%},
                          {%lookingAt(#args[1]#).%} };
}

public boolean checkArgs( Object args[] ) {
  return ( args[0]>=0 && args[0]<=3 &&
           args[1]>=0 && args[0]<=3 );
}
```

El texto incluído entre los símbolos {% y %} es parte del lenguaje JavaLog, el cual permite representar y manipular los estados mentales de los agentes utilizando programación lógica combinada con objetos Java [5].

Cada instancia de `MetaAgent` posee un conjunto con las capacidades de cierto tipo de agentes. Por ejemplo, un meta-objeto `MetaAgent` para los agentes *forklift* contiene *turnTo*, *advance*, *lookAt*, *graspBox* y *putBox*.

Cuando se inicializa un meta-objeto `MetaAgent` invocando el método `initialize`, se buscan y crean esquemas de acción para cada una de las capacidades según el algoritmo 1.

4.4 Percepción

Las capacidades de percepción de los agentes son responsabilidad de la clase `Perceptor`. Dicha clase es subclase de `MetaObject`, por lo tanto, sus instancias son meta-objetos.

Require: C conjunto de capacidades, S conjunto de pares (identificador, esquema de acción), CP conjunto de clases

1: $S = \varnothing$
2: **for all** $c_i \in C$ **do**
3: **if** $\exists class \in CP / class.name() == ''\text{Action_}'' + c_i.name() + c_i.nArgs()$ **then**
4: $S = S \cup \{(c_i.name() + c_i.nArgs(), \text{new } ActionScheme(c_i))\}$
5: **end if**
6: **end for**

Algoritmo 1: Inicialización de los esquemas de acción

Un meta-objeto de percepción es capaz de percibir el ¤ujo de mensajes entre objetos de forma no intrusiva y transparente. Esto implica que es posible percibir cualquier objeto sin necesidad de modificar ningún método de su clase. Esto presenta ventajas importantes respecto de enfoques basados en el patrón *observador*, en el cual es necesario que los objetos observados notifiquen a sus observadores [45].

Los meta-objetos de percepción pueden ser asociados a clases, objetos o a todas las instancias de una clase para percibir los mensajes recibidos por las clases, objetos en particular o todas las instancias de una clase, respectivamente.

Cuando un meta-objeto de percepción detecta la llegada de un mensaje a uno de los objetos observados, determina si dicho mensaje es de interés para el agente. En caso afirmativo, se notifica a sí mismo utilizando el mensaje `messagePerceived` y finalmente, notifica al administrador de situaciones invocando `handleMessage`. El método *hook* `messagePerceived`, por defecto no hace nada, sin embargo, las subclases podrían redefinirlo para realizar algún procesamiento extra sobre las percepciones, como se verá luego en esta sección.

El administrador de situaciones es responsable de buscar situaciones de interés utilizando las percepciones, mensajes recibidos (comunicaciones) y el estado mental del agente. De esta forma, el agente puede percibir una gran cantidad de mensajes, sin embargo, el administrador de situaciones es el que determina cuáles son relevantes según su estado mental en un instante dado.

Se considerará la utilización de percepción en el ejemplo de los robots. Se mencionó que cada robot posee un sensor capaz de detectar y distinguir objetos que se encuentran frente a él. Esta funcionalidad ya es parte del objeto base y está implementada en el método `lookAt` de la clase `Forklift`. La percepción puede realizarse cada vez que el robot avanza o gira. La idea consiste en observar qué hay frente al robot cuando éste cambia de posición utilizando el método `lookAt`. Tales capacidades de percepción pueden ser logradas asociando un meta-objeto de percepción al objeto base para que perciba los mensajes correspondientes a las actividades girar (`turnTo`) y avanzar (`advance`).

En el siguiente fragmento de código[3] se asocia un meta-objeto de percepción a una instancia del objeto base `forklift`:

```
...
// Obtiene la instancia de ForkliftAgent que refleja
// al objeto forklift
BasicAgent basicAgent=metaAgent.BasicAgentOf(forklift);
```

[3] es continuación del código de la sección 4.2.

```
// Indica que debe percibirse el mensaje 'advance()'
// en el objeto forklift
basicAgent.perceiveMethodOf(forklift,"advance",null);
// Indica que debe percibirse el mensaje 'turnTo(int)'
basicAgent.perceiveMethodOf(forklift,"turnTo",
  new Class[] { Integer.TYPE });
// Inicia la percepción
basicAgent.startPerceptionOn(forklift);
```

Cada vez que el objeto base `forklift` reciba el mensaje `advance` indicándole que debe avanzar un paso hacia adelante, o el mensaje `turnTo` indicándole que debe rotar, entonces su meta-objeto de percepción será notificado.

En la aplicación de robots de carga cada uno de los robots desconoce el medio ambiente en que se encuentra, y por lo tanto ignora dónde se encuentra el camión, las cargas, las estanterías, otros robots, etc. Básicamente, lo que se hace es utilizar las capacidades de percepción para que cada agente construya un modelo mental del mundo utilizando el repositorio de creencias del componente deliberador. Dicho repositorio puede almacenar un conjunto de cláusulas Prolog que representan las creencias. Además, posee un mecanismo de revisión de creencias para asegurar la coherencia de las mismas. De esta manera se evita, por ejemplo, que un agente crea que un objeto está en dos lugares en forma simultánea.

Para definir la funcionalidad requerida, esto es, construir un modelo mental en base a las percepciones, puede definirse una subclase de `Perceptor` y redefinirse el método `messagePerceived` para que determine qué hay frente al robot mediante el método `lookAt` del objeto base y agregue una creencia según la tabla 4.1. En la sección 4.8.5 se describe con mayor detalle cómo utilizar el repositorio creencias.

Condición	**Creencia**
hasBox()	location(box(aBox), x, y)
isTruckZone()	location(truck(aTruck), x, y)
isShelfRegion()	location(shelf(aShelf), x, y)

Tabla 4.1: Creencias adquiridas durante la percepción

4.5 Situaciones

El administrador de situaciones implementado en la clase `SituationManager` es el responsable de detectar las situaciones de interés para el agente. Realiza esto teniendo en cuenta los eventos percibidos, las comunicaciones y el estado mental del agente.

Para representar las situaciones de interés se utilizan un conjunto de cláusulas Prolog almacenadas en un módulo lógico[4]. De esta forma, es posible expresar en forma declarativa las condiciones que deben cumplirse para detectar una situación.

Por ejemplo, en la aplicación de robots de carga hay tres situaciones de interés: *"hay una caja enfrente"*, *"hay una estantería libre enfrente"* y *"hay un camión enfrente"*. La primer situación, denomi-

[4] Un módulo lógico es un conjunto de cláusulas lógicas.

nada *boxInFront*, se detecta cuando hay una caja en las coordenadas (X,Y), tales que (X,Y) son las coordenadas que tendría el robot (objeto base) si avanza. Esto se expresa mediante la siguiente regla Prolog:

```
situation(boxInFront,Box) :-
  location(box(Box),X,Y), /* Box está en X, Y */
  newInstance('java.awt.Point',[X,Y],Front),  /* Front = (X,Y) */
  baseObject(Base), /* Base es una instancia de Forklift */
  send(Base,nextLocation,[],Front). /* Front está delante del robot */
```

en este fragmento de código, la relación location(box(*Box*), *X*, *Y*) indica que la caja *Box* se encuentra en las coordenadas (X,Y). La relación newInstance('java.awt.Point', [*X*, *Y*], *Front*) expresa que *Front* es un punto (instancia de la clase `java.awt.Point`) de coordenadas (X,Y), mientras que baseObject(*Base*) indica que *Base* es el objeto base del agente, en este caso una instancia de la clase `Forklift`. Así, el predicado situation(boxInFront, *Box*) es verdadero si se cumple que la caja *Box* se encuentra frente al robot.

La clase `SituationManager` mostrada en la figura 4.7 es responsable de detectar situaciones y notificar a los componentes interesados en dichas situaciones. Típicamente, los objetos de reacción y deliberación actúan cuando se detecta una situación, por lo tanto, deben ser notificados acerca de ello. Un objeto interesado en las situaciones debe implementar el método `newSituation` especificado en la interfaz `SituationListener`. Luego, debe subscribirse con el administrador de situaciones enviándole el mensaje `addSituationListener`. Así, cuando se detecta una situación, todos los objetos subscritos son notificados mediante el mensaje `newSituation`.

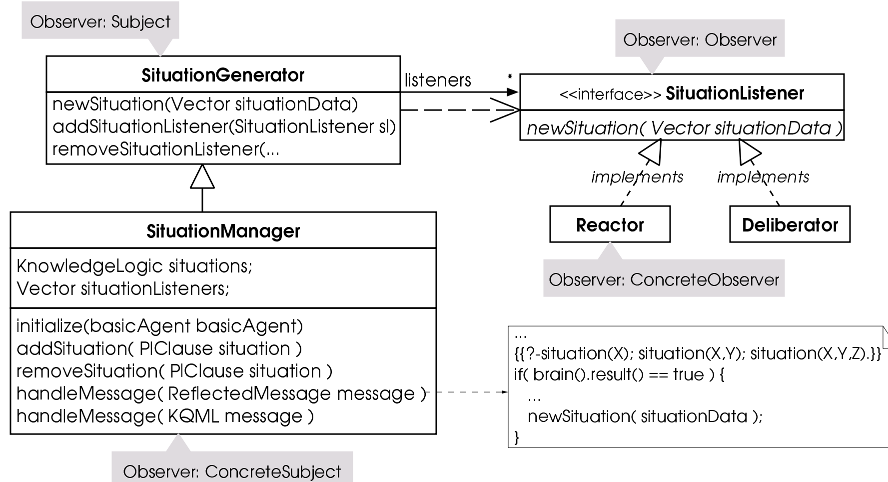


Figura 4.7: Materialización del administrador de situaciones

Para definir las situaciones se utiliza el método `addSituation` de la clase `SituationManager`, el cual recibe como parámetro una cláusula delimitada por {% y %}. Por ejemplo, para la situación *boxInFront*:

```
situationManager().addSituation({% situation(boxInFront,Box) :- ... }%);
```

4.6 Comportamiento reactivo

El componente reactivo de Brainstorm es el responsable de reaccionar ante situaciones detectadas por el administrador de situaciones. En la figura 4.8 se muestra un diagrama con las principales clases involucradas en dicho comportamiento.

Una reacción consta de dos partes: una condición de activación y una tarea a ser ejecutada. La clase `Reaction` representa una reacción. El método abstracto `checkPre` representa la condición de activación, mientras que la tarea a ser ejecutada está representada por el método abstracto `execute`.

La clase `Reactor` implementa la interfaz `SituationListener`, por lo tanto, sus instancias son capaces de recibir información sobre las situaciones detectadas y actuar en función de esto en el método `newSituation`.

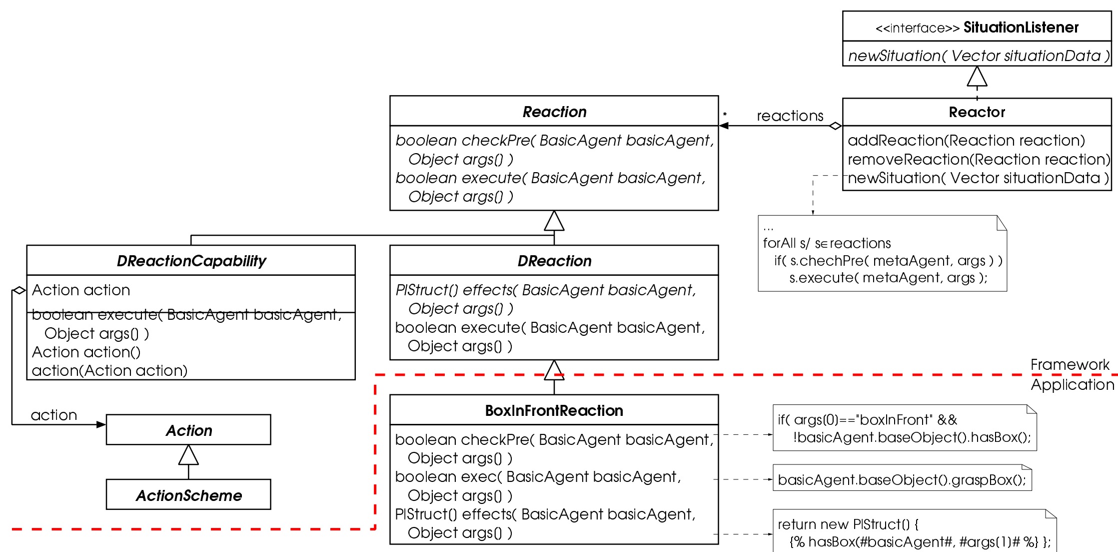


Figura 4.8: Reacciones

El componente reactivo implementado en la clase `Reactor` posee una secuencia de reacciones. Ante una situación de interés busca una reacción tal su condición de activación codificada en el método `checkPre` sea verdadera y la ejecuta enviándole el mensaje `execute`. Si dicho método retorna *false*, indicando la ejecución fallida, entonces el proceso continúa (algoritmo 2).

Require: $R = \{r_1, r_2, \dots, r_n\}$ conjunto de reacciones, s nueva situación
1: **for all** $r_i \in R$ **do**
2:     **if** $r_i.checkPre(s)$ **then**
3:         **if** $r_i.execute()$ **then**
4:             **return**
5:         **end if**
6:     **end if**
7: **end for**

Algoritmo 2: Ejecución de reacciones

Un agente que mantiene un modelo mental del ambiente en el que se encuentra debería, luego de

ejecutar una reacción, actualizar dicho modelo mental utilizando sus capacidades de percepción. Sin embargo, esto no es del todo necesario, debido a que un agente racional *conoce* cuáles son los efectos de sus reacciones. Esto significa que si la reacción se ejecuta, causa los efectos *esperados* sobre el mundo, con lo cual el agente no debería percibir el mundo que lo rodea, sino que simplemente debería actualizar sus creencias acerca del mundo utilizando el efecto que él *cree* que sus reacciones tuvieron. Típicamente, esto resulta de gran utilidad para agentes que mantienen un modelo mental del mundo en el que operan, ya que se elimina la secuencia reacción-percepción que sería necesaria si se desconociera el efecto que las reacciones del agente tienen sobre el mundo.

Para obtener tal funcionalidad, es necesario conocer el efecto de cada reacción. Así, la clase `DReaction` define una reacción que incluye una descripción de sus efectos utilizando una conjunción de predicados Prolog. La ejecución exitosa de este tipo de reacción modifica el estado mental del agente de acuerdo con los efectos de la misma.

En la clase `DReaction` se asume que el ambiente en el que se encuentra el agente es determinístico[5], es decir, que cualquier acción tiene un efecto predecible. Sin embargo, el mundo físico real puede ser no determinístico, y por lo tanto, ser imposible de predecir el efecto exacto que las acciones tienen sobre él [103]. Esto significa, que la clase `DReaction` sólo puede ser utilizada para construir agentes que operan en un medio ambiente determinístico.

En la sección anterior se especificó una situación simple denominada *boxInFront*. Una posible reacción ante esta situación sería intentar tomar la caja mediante el método `graspBox` de la clase `Forklift`, si el robot no lleva otra carga. Así, las precondiciones y efectos de dicha reacción son:

> PRECONDICIÓN: *situation*="boxInFront" && not(hasBox(*Agent*, _))
> EFECTO: hasBox(*Agent*, *Box*)

donde, *aBox* es la caja que se encuentra frente al robot. El predicado not(hasBox(*Agent*, _)) indica que el robot no lleva una carga, por lo tanto podría tomar la que se encuentra frente a él. En la figura 4.8 se muestra la definición de esa reacción en la clase `BoxInFrontReaction`.

Cuando se desean definir reacciones utilizando la capacidad efectora básica de un agente puede especializarse la clase `DReactionCapability`. De esta manera, los agentes puramente reactivos sólo utilizan la clase `Reaction`, mientras que los agentes híbridos utilizan `DReactionCapability` para representar reacciones que involucran sus capacidades o `DReaction` para otros tipos de reacciones.

4.7 Movilidad

El componente de movilidad permite migrar el agente desde un sitio hacia otro durante su ejecución. Dicho componente utiliza un *framework* para movilidad débil de objetos denominado Aglets [63].

Básicamente, la clase `MobilityManager` da al agente la capacidad de migrar a un sitio remoto. Sin embargo, existe una limitación en la máquina virtual de Java que imposibilita la migración del estado de ejecución. Esto se soluciona en forma parcial mediante un mecanismo de eventos que indican al agente el momento de partida y llegada a un sitio.

[5] Si un observador omnisciente puede predecir el estado futuro del ambiente a partir del estado actual del mismo y del conjunto de acciones que pueden ser realizadas sobre él con capacidades de procesamiento £nitas, entonces dicho ambiente es determinístico.

En la figura 4.9 se presenta un ejemplo en el cual se puede observar la sucesión de eventos recibidos por un agente durante su migración. En el instante en que decide migrar, el agente ejecuta la acción *dispatch("sitioA")*. A continuación, recibe una notificación *onDispatch* indicándole que va a ser migrado. Generalmente, dicha notificación es utilizada para salvar el estado de ejecución de los *threads* que componen al agente.

Luego, cuando el agente llega al sitio de destino, el `MobilityManager` restaura el estado de todos los objetos que componen al agente y le envía una notificación *onArrival*. Esa notificación puede ser utilizada para restaurar la ejecución de los *threads*, es decir, realizar las acciones inversas a las realizadas para la notificación *onDispatch*.

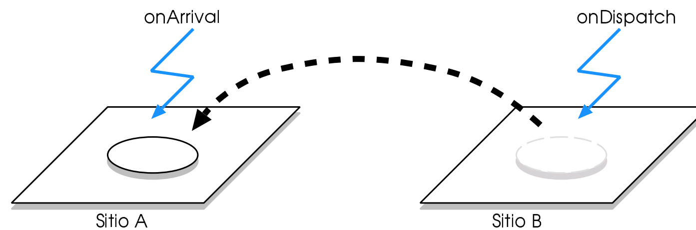


Figura 4.9: Movilidad débil

La clase abstracta `MobilityManager` define métodos abstractos para tratar los eventos *onDispatch* y *onArrival*. Obsérvese que salvar o restaurar el estado de ejecución de un *thread* no es una tarea trivial, debido a que el lenguaje Java no provee ningún soporte para ello. Además, no es posible determinar el valor del contador de programa de un *thread*, ni acceder a la pila de invocaciones, por lo que no es posible salvar ni restaurar el estado de un *thread* en forma automática. Para poder realizar esto, cada uno de los componentes del agente es responsable de salvar el estado de sus propios *threads*. Así, por ejemplo, si el componente de deliberación ejecuta en un *thread* independiente, entonces él es responsable de salvar y restaurar su estado cuando reciba las notificaciones *onDispatch* y *onArrival*.

Hay otros dos factores que dificultan la utilización de movilidad: el número de objetos que forman al agente y la utilización de recursos que el mismo realiza. En primer lugar, el tiempo de migración de un agente es proporcional al número de objetos que lo componen. En segundo lugar, si el agente utiliza recursos locales no migrables tales como una ventana, un archivo o una conexión de red, entonces habrá que administrar de manera especial cada uno de ellos para que la migración no destruya los *bindings* con estos recursos. Sin embargo, los mecanismos para tratar estos problema dependen exclusivamente de soporte provisto por el lenguaje [44]. En particular, en Java, no existe soporte alguno para mantener los *bindings* entre el agente y los recursos que utiliza, por lo que no es posible definir mecanismos genéricos responsables de dicha funcionalidad. De esta manera, es responsabilidad de quien instancia el *framework* codificar los métodos para mantener los bindings. Esto puede ser realizado como parte del tratamiento de los eventos *onDispatch* y *onArrival*, lo cual es responsabilidad de quien instancia el *framework*.

Una forma de reducir los problemas mencionados consiste en minimizar el número de objetos, *threads* y recursos que un agente utiliza. En general, esto no es posible. Una alternativa consiste en utilizar agentes *esclavos*. Un agente esclavo es un agente que sólo posee la mínima funcionalidad para realizar tareas simples en sitios remotos, migrar de manera eficiente y comunicarse con otros agentes. Los agentes esclavos son creados por otros agentes (que quizás no pueden migrar) para realizar tareas en sitios remotos.

Brainstorm/J provee una clase abstracta `Slave`, que puede ser especializada para construir agentes esclavos con movilidad. Sin embargo, debe destacarse que estos agentes no poseen la arquitectura prescripta por Brainstorm, debido a que se intentó minimizar la cantidad de objetos que lo componen.

Básicamente, una instancia de la clase `Slave` es capaz de migrar de un sitio a otro siguiendo un itinerario predefinido. En cada sitio ejecuta un acción predefinida en su método `onArrival`. Si uno de los sitios de su itinerario no se encuentra, lo ignora y continúa. Si se produce una situación inesperada, notifica inmediatamente a su creador. Cuando termina el itinerario, retorna al sitio de origen y entrega un informe sobre las tareas realizadas a su creador.

4.8 Deliberación

La deliberación puede definirse como el proceso por el cual un agente razona acerca de sus estados mentales y las relaciones entre ellos para decidir qué hacer [80]. En dicho proceso, el agente podría tener que cumplir con varios, y posiblemente con¤ictivos, objetivos, para lo cual debería tomar decisiones acerca de cómo lograrlos para obtener el efecto esperado.

Las tres nociones cognitivas básicas de un agente racional –creencias, deseos (objetivos) e intenciones (BDI)– tienen un rol fundamental en la formación y determinación de acciones de un agente [48, 81]. Según Shoham [86] "*lo que hace a cualquier componente de hardware o software un agente es precisamente el hecho de que actúa en función de esas actitudes mentales*".

El deliberador debe decidir qué tareas realizar para cumplir con los objetivos, cómo llevarlas a cabo, y cuándo ejecutarlas. Los medios para tomar esas decisiones pueden ser variados. Así, por ejemplo, el deliberador podría utilizar *planning* [98], razonamiento basado en casos [61], razonamiento probabilístico [62] o *scheduling* [109], entre otros.

En la figura 4.10 se muestra un esquema del componente deliberativo del *framework*. En cada instante de tiempo, el razonador debe seleccionar cuáles son los objetivos a cumplir. Luego, seleccionar o construir un plan para intentar hacer verdaderos los objetivos seleccionados. Finalmente, debe ejecutar las acciones de dicho plan.

Los planes Brainstorm/J tienen la particularidad de tratar con dos tipos de acciones: *simples* y *deliberativas*. Una acción *simple* representa una actividad que puede ser realizada por el agente de forma inmediata, es decir, sin necesidad de razonar acerca de cómo llevarla a cabo. Las acciones *deliberativas* representan una descripción sobre cómo alcanzar un conjunto de objetivos. Así, por ejemplo, una acción deliberativa podría indicar al agente que debe utilizar un algoritmo de *planning*. El resultado de una acción deliberativa, es un plan *concreto* compuesto por acciones simples, construído mediante un proceso o mecanismo deliberativo. Dichos planes son ejecutados por el componente *Executor*.

El deliberador ejecuta en un *thread* independiente, permitiendo que el agente continúe percibiendo, comunicándose y actuando. Además, en el deliberador pueden haber varios *sub-threads* ejecutando concurrentemente. Por ejemplo, mientras se construye un plan para cumplir un conjunto de objetivos, se podría estar analizando la mejor manera de responder una comunicación a otro agente que intenta vender un producto.

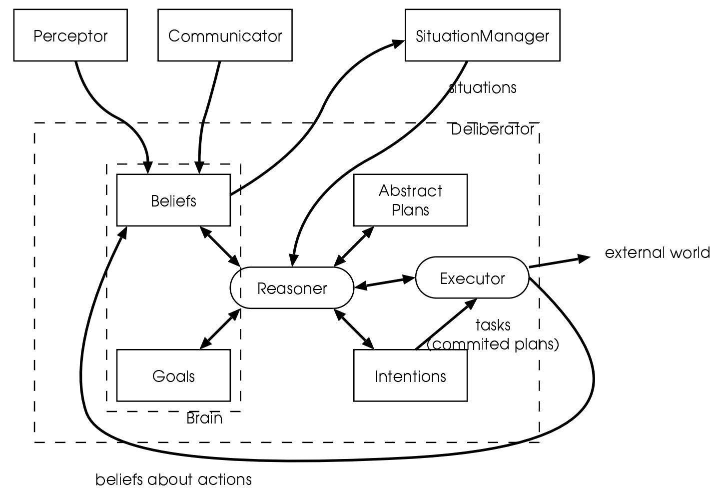


Figura 4.10: Estructura del deliberador

4.8.1 Planes abstractos

Cada agente puede tener un conjunto de planes *abstractos* predefinidos que permiten que actúe frente a situaciones conocidas o siga un comportamiento dirijido por objetivos. Esencialmente, un plan abstracto es una *receta* que debe ser seguida por el agente en respuesta a un evento o para cumplir con sus objetivos. Este tipo de plan predefinido no debe confundirse con los planes que se construyen, por ejemplo, mediante un algoritmo de *planning* [98]. En particular, los planes abstractos, también llamados *conocimiento procedural* o *know-how* [100], están compuestos de dos tipos de acciones: *acciones simples* y *acciones deliberativas*. Las acciones simples son tareas que el agente es capaz de ejecutar de forma inmediata. Una acción deliberativa hace que el agente razone acerca de cómo llevar a cabo esa acción.

Podría pensarse en la posibilidad de construir los planes en forma *on-line*, en función de las situaciones con las que el agente se enfrenta. Sin embargo, esta solución presenta varios problemas. Por ejemplo, no es posible determinar en forma automática la mejor forma de resolver los subobjetivos (*planning*, CBR, etc). Incluso decidir acerca de la conveniencia o no de dividir un problema compuesto por un conjunto de objetivos en varios subproblemas, y luego integrar las soluciones parciales, tiene una dificultad considerable [108]. Así, los planes abstractos ayudan a combinar las ventajas del *planning on-line* y *off-line*. Brainstorm/J permite utilizar ambos tipos de planes.

En Brainstorm/J los planes abstractos son representados por un grafo de transición de estados. Cada estado del grafo representa una acción (simple o deliberativa). Las transiciones entre los estados contienen una condición sobre el estado mental del agente, situaciones, eventos internos o comunicaciones recibidas.

Por ejemplo, en la aplicación de robots de carga, cuando un robot desea descargar un camión se activa el plan abstracto de la figura 4.11. Dicho plan, define la secuencia acciones que el agente debe llevar a cabo para transportar una caja desde el camión hasta los depósitos. Las cuatro acciones de dicho plan son deliberativas. Por ejemplo, el estado S_0 representa la actividad de llegar al camión. El agente permanece en dicho estado hasta que se produce la situación *truckInFront* o *boxInFront*, indicando que el robot se encuentra frente al camión o a una caja, respectivamente.

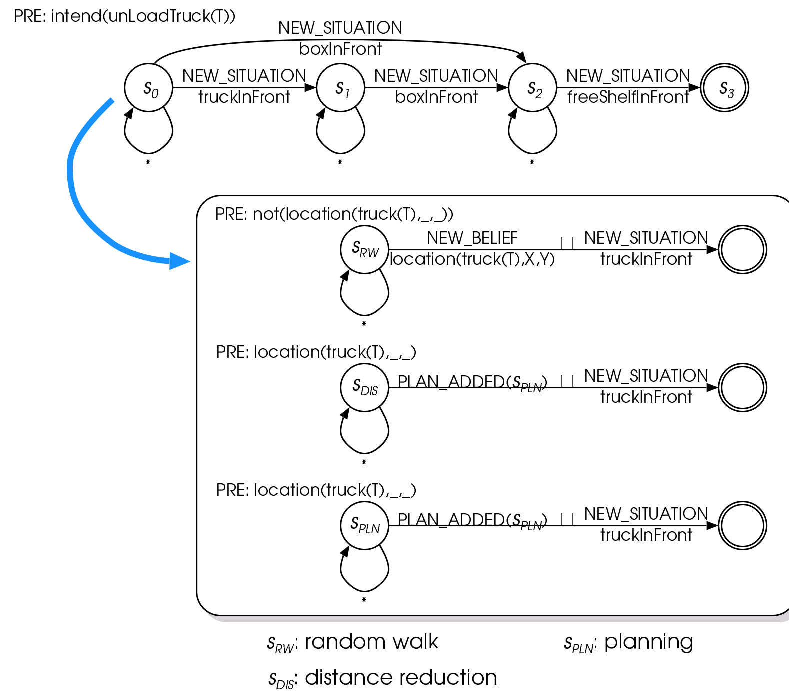


Figura 4.11: Plan abstracto para descargar un camión

Cuando un robot decide ir al camión, se enfrenta al siguiente problema: *£cómo ir al camión?*. En este punto pueden observarse dos casos. En primer lugar, el robot podría desconocer la localización del camión, en cuyo caso no tendría más opción que explorar el medio ambiente. Por otro lado, si conoce dónde se encuentra el camión, la situación podría resolverse mediante *planning*. Para utilizar esta solución habría que considerar los siguientes aspectos:

- Mientras se genera el plan £el robot se *congela* o explora el terreno?
- Si el robot explora y planea simultáneamente £qué sucede con los obstáculos que descubre durante la exploración? £se replanea o se repara el plan?

En la figura 4.11, el estado S_0 representa la acción para ir al camión (posiblemente desconociendo su localización). Si el robot tiene éxito en la ejecución de esta acción, entonces se encontrará junto al camión, por lo que se producirá la situación *truckInFront*. Para lograr alcanzar ese objetivo, el robot realiza lo siguiente:

- si no sabe dónde se encuentra el camión, entonces camina aleatoriamente,
- en caso contrario:
  - comienza a construir un plan.
  - en forma simultánea intenta reducir la distancia que lo separa del camión. Así, mientras construye el plan, acorta la distancia que lo separa del camión.

En el instante en que finaliza la construcción del plan, el mismo se ejecuta, suspendiendo el comportamiento de reducción de distancia. Nótese, que el plan construido podría ser inconsistente con la realidad, debido a que dicho plan se construía y en forma simultánea, se avanzaba hacia el camión.

Por ejemplo, podrían haber acciones innecesarias en dicho plan, por haber sido realizadas durante el acercamiento al camión. Esto se soluciona mediante un algoritmo de reparación de planes que elimina del plan aquellas acciones redundantes o innecesarias. Por otro lado, se agregan acciones cuyos efectos incluyen proposiciones que el agente considera actualmente como falsas (consideradas verdaderas mientras se construía el plan). En la sección 4.8.7 se describe la reparación de planes con mayor detalle.

El estado compuesto S_0 contiene tres planes. El plan cuyo estado inicial es S_{RW} se ejecuta si el robot desconoce la localización del camión y no se encuentra frente a él. Básicamante, mientras el agente permanece en dicho estado camina aleatoriamente.

Concurrentemente con el plan de caminata aleatoria se ejecutan otros dos planes. El primero, cuyo estado inicial es S_{DIS}, se ejecuta si el agente conoce la localización del camión, no se agregó un plan y no se encuentra frente al camión. En dicho estado, el agente intenta reducir la distancia que lo separa del camión. El plan abstracto que comienza en el estado S_{PLN} construye un plan para caminar hacia el camión considerando los obstáculos del camino.

Como resultado de esto, el agente camina aleatoriamente hasta tanto conozca la posición del camión. En ese instante el planner comienza con la construcción del plan. En forma simultánea, dado que conoce dónde se encuentra el camión, intenta reducir la distancia que lo separa de éste. En general, éste último comportamiento sería suficiente para que el agente llegue hasta el camión (sin necesidad de planear). Sin embargo, para configuraciones complejas del terreno, tales como un laberinto, la simple reducción de distancia puede encerrar al robot en zonas sin salida.

Nótese que los planes abstractos pueden ser utilizados para especificar conversaciones multi-agente similares a las de COOL. Esto se logra especificando condiciones lógicas acerca de un mensaje recibido en los arcos de transición de estados del grafo y produciendo mensajes en dichos arcos como se verá en la sección 4.9.

4.8.2 Arquitectura

El deliberador utiliza un *blackboard* [27] en el que se construyen planes concretos para alcanzar los objetivos del agente (figura 4.12). Las fuentes de conocimiento (FC) son las acciones de los planes abstractos (representados por estados de los autómatas), las cuales construyen gradualmente los planes concretos. El componente de ejecución también es una fuente de conocimiento, aunque se encarga de ejecutar y adaptar los planes concretos construídos por las otras fuentes de conocimiento. El componente de control es responsable de seleccionar los objetivos, crear fuentes de conocimiento y administrar los repositorios de creencias, objetivos y planes abstractos.

Un plan abstracto es representado por un autómata finito recursivo determinístico (AFDR). Cada estado del autómata puede generar una o más fuentes de conocimiento, incluyendo otros autómatas representando planes abstractos. Básicamente, una fuente de conocimiento es capaz de generar acciones para que el agente las ejecute, modificar las creencias, percibir eventos tales como nuevas creencias, ejecución de una acción/plan, nuevas situaciones, nuevos objetivos, nuevos planes, etc. Por ejemplo, en el plan abstracto para descargar un camión, el comportamiento de caminata aleatoria se logra mediante una fuente de conocimiento que, en forma aleatoria, selecciona una dirección. Luego, coloca en el *blackboard* un plan formado por las acciones: *turnTo(dir)*→*advance()* y espera a que ese plan sea ejecutado por la fuente de conocimiento *Executor*. Esto se repite hasta que la fuente de

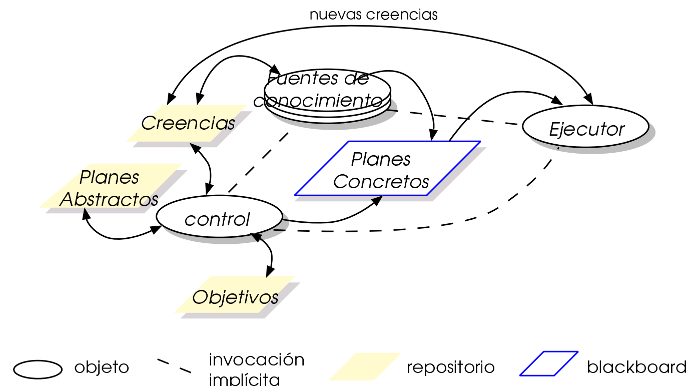


Figura 4.12: Diagrama del componente deliberativo

conocimiento es eliminada. A su vez, la fuente de conocimiento *Executor* actúa cuando se coloca un plan en el *blackboard*.

En la figura 4.13 se muestra un diagrama con las principales clases utilizadas para representar los planes abstractos. La clase abstracta `AbsPlan` representa un AFDR. Cada uno de los estados de dicho autómata puede dar origen a una o más fuentes de conocimiento, por lo que cada estado está relacionado con una subclase de `DelibKS`. Por ejemplo, en el plan de la figura 4.11 el estado S_{RW} está relacionado con la clase `RandomWalkerKS`. Así, cuando la ejecución del plan entra en estado S_{RW}, se crea una instancia de la clase `RandomWalkerKS` que produce acciones para que el agente camine en forma aleatoria.

Las transiciones de estados están representadas por la clase abstracta `DTransition`. Para cada estado s_i existe una secuencia de transiciones de estado $\left\langle (t_{i1}, s_{j_{i1}}), (t_{i2}, s_{j_{i2}}), \ldots, (t_{im_i}, s_{j_{im_i}}) \right\rangle$, que pueden producirse a partir del estado s_i. Cada par $(t_{ik}, s_{j_{ik}})$ indica que si se cumple la condición de transición de t_{ik}, entonces, el autómata pasa al estado $s_{j_{ik}}$. La clase abstracta `DTransition` define un método abstracto `check` que debe ser definido por las subclases para especificar una condición de transición. Además de la condición de transición, la clase `DTransition` define un mecanismo para indicar cuáles son los eventos que podrían causar la activación de la condición, y la consiguiente transición de estado. Por ejemplo, en la clase `TruckInFrontTr` mostrada en la figura 4.13 se redefine el método `check`, indicando que la condición de transición es la situación *truckInFront*. Obsérvese que en el constructor de dicha clase se indica que el único evento de interés es NEW_SITUATION.

Básicamente, para definir un plan abstracto hay que especializar la clase `AbsPlan`. Esto incluye la definición del autómata mediante su matriz de transición de estados, las condiciones de transición, las fuentes de conocimiento asociadas a cada estado y la condición de activación del plan (objetivos que se alcanzan cuando se ejecuta el plan). Para definir estas propiedades se utilizan los métodos `nextState`, `tr`, `ksClasses` y `checkPre`, respectivamente. Por ejemplo, para construir el plan abstracto de figura 4.14 puede definirse una subclase de `AbsPlan`. A continuación se muestra el constructor de dicha clase:

```
super( deliberator );
ksClasses(new Class[] {RandomWalkerKS.class, RandomWalkerKS.class,
  RandomWalkerKS.class});
nextState(new int[][] {{1,2,0}, {2,1}, {3,2} });
```

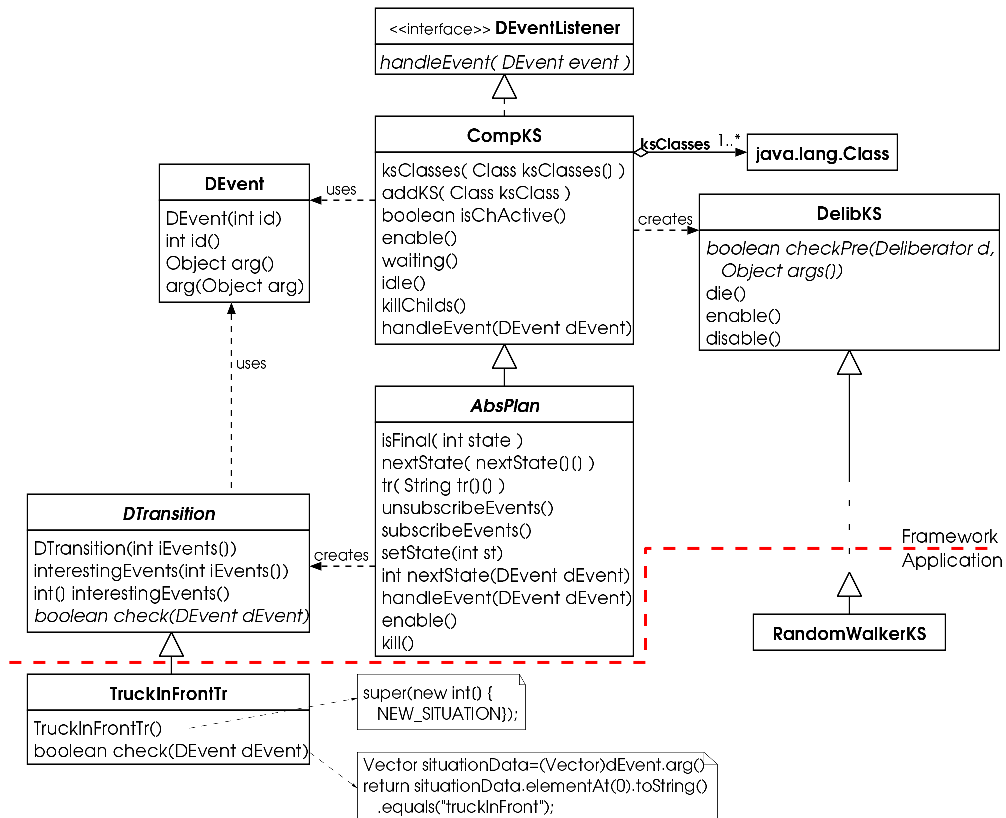


Figura 4.13: Planes abstractos

```
tr(new Class[][] {
  {TruckInFrontTr.class, BoxInFrontTr.class, DTrueTransition.class},
  {BoxInFrontTr.class, DTrueTransition.class},
  {FreeShelfInFrontTr.class, DTrueTransition.class} });
final(3);
```

En este fragmento de código se invoca al método `ksClasses` con un arreglo de clases. Así, se asocia el comportamiento de caminata aleatoria implementado en la clase `RandomWalkerKW` a tres de los cuatro estados del autómata. Cada uno de los estados del autómata es identificado por un número entero, comenzando por 0 para el estado inicial. El método `nextState` inicializa la matriz de transición de estados del autómata. Luego, mediante el método `tr` se asocia una condición a dichas transiciones.

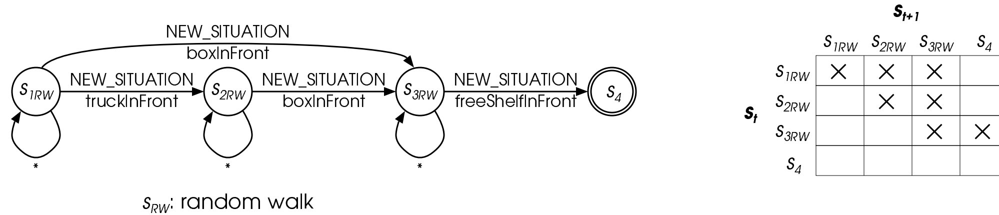


Figura 4.14: Plan abstracto para descargar un camión (sólo caminata aleatoria)

La clase `DTrueTransition` define una condición de transición que siempre se cumple, por lo que se utiliza para indicar una transición que se activa por defecto cuando todas las otras fallan.

Finalmente, para terminar de definir el plan abstracto, debe definirse el método `checkPre` indicando la condición de activación de dicho plan. En este caso, el plan se activa cuando el robot intenta descargar un camión, lo cual es representado mediante una consulta JavaLog:

```
public boolean checkPre( Object args[] ) {
  ?-intend(unloadTruck(_)).
}
```

4.8.3 Fuentes de conocimiento

En la figura 4.15, la jerarquía de clases que comienza en `DelibKS` define diversos tipos de fuentes de conocimiento. La interacción entre las fuentes de conocimiento se realiza mediante un mecanismo de invocación implícita basado en eventos. Las FC pueden suscribirse a eventos de interés tales como cambios en los planes, logro o abandono de objetivos, nuevas situaciones, nuevas creencias, etc. La clase `Deliberator` es responsable de despachar eventos; activar/desactivar/crear fuentes de conocimiento; administrar objetivos y creencias, seleccionar los planes abstractos para lograr un conjunto de objetivos, etc.

Cuando se crea una fuente de conocimiento, se verifica su condición de activación mediante el método `checkPre`. En caso de ser verdadera, dicha FC se activa, se subscribe a los eventos que le interesan y se le da acceso al *blackboard*. Típicamente, lo que ocurre cuando la condición de activación de una FC es verdadera se codifica en el método `enable`.

Cuando se crea una FC, se suscribe al evento KS_KILL. Cuando recibe un evento KS_KILL con una referencia a sí misma, invoca al método `kill` y finalmente al método `die`. El método `kill`, puede ser

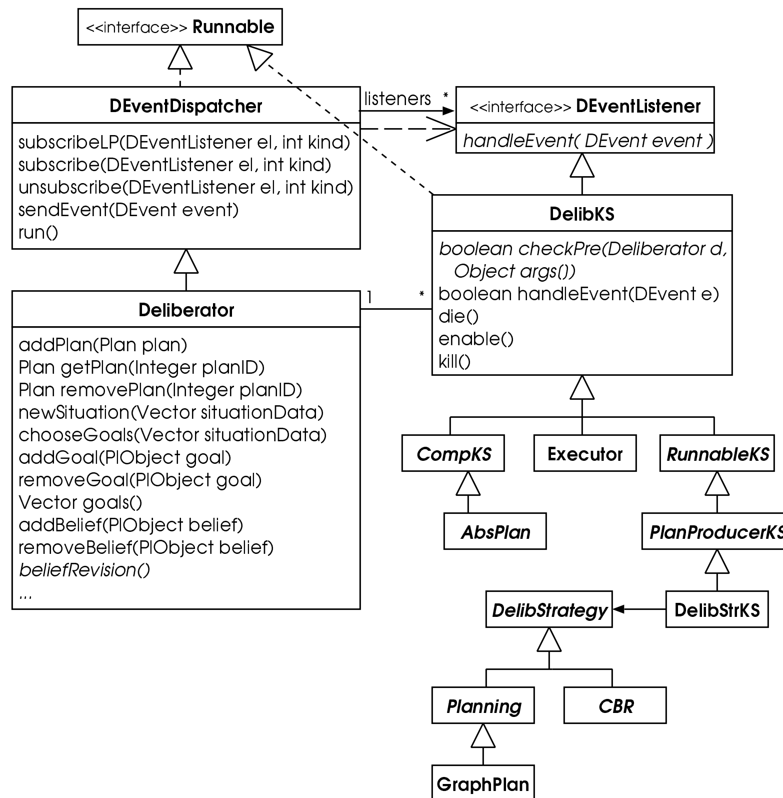


Figura 4.15: Principales clases del componente de deliberación

utilizado para realizar acciones previas a la destrucción de una fuente de conocimiento. Por ejemplo, esperar a que se complete la ejecución de un plan. El método `die` libera recursos y se de-suscribe de eventos. Finalmente, cuando una FC se destruye señala el evento KS_DESTROY.

Una FC puede trabajar en forma sincrónica, en el mismo *thread* que todo el deliberador, o en forma asincrónica, en su propio *thread*. Esto permite, que el agente razone sobre diferentes cosas al mismo tiempo o realice tareas que insumen un tiempo considerable sin bloquear el resto de los componentes del deliberador. El método `inThread` de la clase `RunnableKS` se utiliza para determinar si una FC corre en el mismo *thread* que el deliberador o en otro. Típicamente, cuando una `RunnableKS` ejecuta en un *thread* propio realiza lo siguiente:

```
while(!isTimeToDie())
   execute();
die();
```

El método abstracto `execute` es el que debe implementar la funcionalidad que corre en el *thread* independiente mientras el método `isTimeToDie` retorne falso.

Existe la posibilidad de que una FC que corre en su propio *thread* reciba el evento KS_KILL mientras se encuentra realizando cierto proceso. En este caso, no debería destruirse inmediatamente, sino que debería finalizar el proceso que se encantraba realizando. Para tal fin, la clase `RunnableKS` define el método `busy` para señalar que la FC se encuentra realizando algún proceso, y el método `idle` que indica que se encuentra desocupada. Luego, si se invoca el método `busy` y la FC recibe el evento KS_KILL, no procesa dicho evento hasta que se invoque el método `idle`. La FC recuerda la recepción del evento KS_KILL con el fin de procesarlo cuando se encuentre desocupada.

La clase `PlanProducerKS` especializa `RunnableKS` definiendo una fuente de conocimiento capaz de ejecutar en su propio *thread*, y producir planes. Una FC de esta clase, produce un plan invocando al método abstracto `getPlan()`, luego lo coloca en el *blackboard* y espera a recibir el evento PLAN_EXECUTED o PLAN_FAILED, indicando la ejecución satisfactoria o fallida del plan.

Por ejemplo, el estado S_o del plan de la figura 4.11 se representa por las siguientes fuentes de conocimiento: comportamiento aleatorio, planner y reducción de distancia. La FC para reducción de distancia es activada sólo si el robot conoce la localización del camión. Básicamente, esta FC coloca en el blackboard (`addPlan`) un plan formado por dos acciones: girar y avanzar, que reduce la distancia que lo separa del camión. Esto lo realiza en el momento de su activación, y cada vez que le informan que ese plan fue ejecutado mediante el evento PLAN_EXECUTED.

Las FC compuestas (`CompKS`) pueden generar cero o más FC *hijas*. Típicamente una FC compuesta espera a que sus FC hijas completen su trabajo o sean destruidas. Los planes abstractos son un tipo especial de fuente de conocimiento compuesta capaz de generar una fuente de conocimiento por cada activación de un estado.

Las fuentes de conocimiento definidas por la clase `DelibStrKS` utilizan un algoritmo o estrategia para obtener un plan que permita al agente lograr un conjunto de sus objetivos. La principal característica de ellas consiste en que utilizan una estrategia genérica para construir los planes. Por ejemplo, pueden utilizar un algoritmo de *planning* tal como UCPOP [98] o GraphPlan [8].

La clase abstracta `Planner` define una interfaz común para los diversos algoritmos de *planning*. La entrada de un algoritmo de *planning* consiste de un conjunto de esquemas de acción, un conjunto de objetivos y un conjunto de creencias sobre el mundo. Estos conjuntos, en particular los esquemas y las creencias, son subconjunto de las capacidades básicas del agente y de sus creencias, respectivamente.

Esto se debe a que, por razones de eficiencia temporal y espacial, es importante que los algoritmos de *planning* utilicen una cantidad mínima de esquemas de acción y creencias para resolver un problema dado [108].

Cuando una `DelibStrKS` es activada, debe determinar, de manera *ad-hoc*, qué capacidades del agente son aplicables al problema a resolver, y cuáles son las creencias relevantes. Luego, delega la contrucción del plan al *planner* asociado.

Durante la contrucción de un plan el agente puede abandonar un objetivo, lograrlo de otra manera, modificar sus creencias, etc. Esto significa que, un plan en construcción podría contener errores originados por la dinamicidad del mundo en el cual se encuentra el agente, o por el hecho de que el agente realiza varias cosas en forma simultánea. Algo similar puede ocurrir durante la ejecución de un plan. En la sección 4.8.7 se analizan varios de estos casos, y las estrategias para evitar o reducir las inconsistencias en los planes.

Por último, la fuente de conocimiento `Executor` es la encargada de tomar los planes *completos* construidos por otras FC y ejecutarlos. Un plan está *completo* cuando no existe ninguna fuente de conocimiento, exceptuando al ejecutor, esperando por él. Así, el ejecutor toma todos los planes completos, los secuencia e intenta ejecutar las acciones. Luego, señala la ejecución exitosa o fallida de un plan mediante los eventos PLAN_EXECUTED o PLAN_FAILED, respectivamente. Además, el `Executor` es responsable de asegurar la integridad de los planes en ejecución con respecto a las creencias y objetivos de un agente. Esto se analiza en la sección 4.8.7.

4.8.4 Eventos

Todas las interacciones entre las fuentes de conocimiento del componente deliberador de Brainstorm/J se realizan mediante un mecanismo de eventos [15].

Las FC pueden suscribirse a diversos eventos de interés sobre los que desean ser informados. Algunos de los eventos más importantes son:

- PLAN_ADDED(*ks*, *plan*): indica que la fuente de conocimiento *ks* ha agregado un plan al blackboard.
- PLAN_MODIFIED(*ks*, *planId*): indica que la fuente de conocimiento *ks* ha modificado un plan del blackboard.
- PLAN_EXECUTED(*planId*): indica que un plan ha sido ejecutado con éxito.
- PLAN_FAILED(*planId*): indica que la ejecución de un plan ha fracasado.
- KS_FAILED(*ks*): indica que *ks* no pudo cumplir sus objetivos.
- KS_SUCCESS (*ks*): indica que *ks* cumplió sus objetivos.
- KS_DESTROYED(*ks*): indica que la fuente de conocimiento *ks* ha sido destruida. Generalmente, las fuentes de conocimiento se auto-destruyen cuando completan su trabajo o fracasan. Este evento se genera en respuesta al evento KS_KILL.
- KS_KILL(*ks*): indica a una fuente de conocimiento que debe finalizar.
- GOAL_ACHIEVED(*goals*): indica que los objetivos *goals* ha sido alcanzado. Esto puede ser originado por la ejecución exitosa de un plan, por una situación o por cambios en las creencias.

- GOAL_DROPPED(*goals*): indica que un conjunto de objetivos han sido abandonados.
- NEW_GOAL(*goals*): se produce cuando el agente adopta un conjunto de objetivos.
- NEW_SITUATION(*situationData*): indica una situación generada por el administrador de situaciones.
- NEW_COMMUNICATION(*msg*): indica la recepción de un mensaje.
- NEW_CONVERSATION(*conv*): indica que se ha iniciado una conversación multi-agentes.
- END_CONVERSATION(*msg*): indica que se ha concluído una conversación multi-agentes.
- NEW_BELIEF(*bel*): indica que el agente tiene una nueva creencia.

4.8.5 Creencias

Tradicionalmente, la representación y manipulación de creencias ha sido objeto de numerosos estudios filosóficos y lógicos, entre otros. Básicamente, existen dos formas de interpretar las creencias [78]:

- El enfoque comportamental interpreta a las creencias como *disposiciones*. De esta forma, creer que una proposición p es verdadera es estar *dispuesto* a actuar como si lo fuera.
- El enfoque mentalista sugiere tratar a las creencias como *estados*. De acuerdo a este enfoque, creer en una proposición p es estar en un cierto *estado* que persiste mientras el agente mantiene la creencia.

En Brainstorm/J, se adoptó el enfoque mentalista, debido a que es el más utilizado en agentes inteligentes [78], además de haber innnumerables formalizaciones del mismo.

El componente deliberador contiene un repositorio en el cual se almacenan las creencias del agente. Desde el punto de vista mentalista, el repositorio es la representación del *estado* de las creencias. En Brainstorm/J, las creencias se representan mediante cláusulas Prolog. Por ejemplo, en la aplicación de robots, location(box(aBox), 10, 20) representa la creencia de que la caja *aBox* se encuentra en la posición $(10,20)$ del ambiente.

Básicamente, es posible realizar tres operaciones sobre el repositorio de creencias:

- agregar una creencia: método `addBelief`.
- remover una creencia: método `removeBelief`.
- verificar la presencia de una creencia: realizando una consulta Prolog del tipo $?-p.$, donde p es la creencia.

La incorporación de nuevas creencias podría introducir contradicciones respecto de las creencias existentes. Por ejemplo, un agente que cree que la caja *box1* se encuentra en $(10,20)$ percibe que esa misma caja se encuentra en $(4,8)$. Obviamente, esto plantea una inconsistencia, la cual podría resolverse eliminando la primer creencia.

La *revisión de creencias* es el proceso por el cual un agente modifica sus creencias para eliminar las contradicciones e inconsistencias. Desde el punto de vista mentalista, la revisión de creencias es una transformación de un estado a otro, originado por un cambio en dicho estado que introdujo una inconsistencia.

Los filósofos han discutido dos teorías de creencias (verdad) [78]: la teoría *fundacional de verdad* y la teoría de la *coherencia*. En base a estas teorías, se han desarrollado teorías de revisión de creencias:

- *teoría fundacional*: para cada creencia existe una secuencia finita y no circular de justificaciones o es auto-evidente. La revisión TFRC[6] consiste en eliminar las creencias que no tienen justificación o añadir nuevas creencias que no necesitan justificación o están justificadas por otras creencias justificadas.
- *teoría de coherencia*: lo que justifica una creencia es su coherencia dentro un conjunto de creencias. La revisión TCRC[7] consiste en realizar cambios mínimos en el conjunto de creencias para mantener la coherencia o eliminar la incoherencia. Por ejemplo, creer que un objeto físico se encuentra en dos posiciones del espacio en forma simultánea es incoherente; la revisión eliminaría una de las creencias incoherentes.
- *teorías híbridas*: combinan las teorías anteriores.

Básicamente, lo que distingue a las teorías de revisión de creencias es el tratamiento de las justificaciones. La teoría de la coherencia sostiene que *ninguna de las creencias del agente necesitan ser justificadas*, mientras que según la teoría fundacional, *todas las creencias de un agente deben estar justificadas*. Esto significa, que TFRC requiere que cada vez que se realiza una modificación en el conjunto de creencias, se analicen *todas* las creencias para verificar si están o no justificadas. Por otro lado, en TCRC, sólo es necesario analizar la coherencia de la nueva creencia respecto de las existentes, lo que obviamente, es más simple que TFRC. En [69] puede encontrarse un análisis muy detallado sobre la complejidad temporal de varios mecanismos de revisión de creencias.

La TCRC contiene las características escenciales de todos los mecanismos de revisión de creencias [78]. Así, por ejemplo, es posible obtener TFRC a partir de TCRC eliminando las creencias no justificadas. Por estas razones, TCRC resulta de sumo interés para un *framework*.

Brainstorm/J provee un mecanismo de revisión de creencias basado en TCRC que puede ser utilizado sin modificaciones, o para construir otros mecanismos tales como TFRC. Dicho mecanismo está implementado en el método `beliefRevision` de la clase `Deliberator`. Ese método es utilizado cada vez que se agrega una creencia, aunque esto puede redefinirse fácilmente.

El mecanismo de revisión de creencias utilizado en el *framework* distingue dos tipos de coherencia:

- Coherencia lógica: sea p una proposición y Σ un conjunto de proposiciones atómicas, la coherencia lógica débil (CLD) se define como:

$$\mathrm{CLD}(p,\Sigma) = \begin{cases} \mathit{true} & \Leftrightarrow \quad \neg p \notin \Sigma \\ \mathit{false} & \Leftrightarrow \quad \neg p \in \Sigma \end{cases}$$

- Coherencia semántica: dos proposiciones p_1 y p_2 son semánticamente coherentes si no existe un axioma de incoherencia $\mathrm{SI}(p_1, p_2)$. Un axioma de incoherencia especifica que en cierto dominio, dos proposiciones son semánticamente incoherentes. Un axioma de incoherencia tiene la siguiente forma: $\mathrm{SI}(p_1, p_2) \leftarrow c_1 \vee c_2 \vee \ldots \vee c_n$, donde c_i es una fórmula del cálculo de predicados.

[6]Teoría fundacional de revisión de creencias.
[7]Teoría de coherencia de revisión de creencias.

Intuitivamente, SI(p_1, p_2) es verdadero si es incoherente creer en p_1 y en p_2. Por ejemplo, un axioma denotando que es incoherente creer que un objeto se encuentra en dos lugares al mismo tiempo es:

$$\text{SI}(location(R,X_1,Y_1), location(R,X_2,Y_2)) \leftarrow X_1 \neq X_2 \vee Y_1 \neq Y_2$$

Básicamente, el mecanismo de revisión de creencias verifica la coherencia lógica y semántica de las nuevas creencias respecto de las existentes mediante el algoritmo 3.

Require: $B_t = \{b_1, b_2, \ldots, b_n\}$ conjunto de creencias, b nueva creencia
1: $B_{t+1} = B_t$
2: **if** $\neg\text{CLD}(b, B_{t+1})$ **then**
3: $B_{t+1} = B_{t+1} \cup \{p\} - \{\neg p\}$
4: **else if** $\exists q \in B_{t+1} / \text{SI}(p,q) == true$ **then**
5: $B_{t+1} = B_{t+1} \cup \{p\} - \{q\}$
6: **else**
7: $B_{t+1} = B_{t+1} \cup \{p\}$
8: **end if**
9: **return** B_{t+1}

Algoritmo 3: Revisión de creencias

Obsérvese que la relación SI(p,q) es dependiente del dominio, y por lo tanto, tiene que ser especificada por quien instancia el *framework* en el método `beliefRevisionD` de la clase `Deliberator`. Por ejemplo, en la aplicación de robots de carga, dicho método contiene:

```
public void beliefRevisionD() {
  {{...
    si(location(R,X1,Y),location(R,X2,Y)) :- X1 \\== X2.
    si(location(R,X,Y1),location(R,X,Y2)) :- Y1 \\== Y2.
    si(location(R,X1,Y1),location(R,X2,Y2)) :- X1 \\== X2, Y1 \\== Y2.}};
}
```

Obsérvese que las relaciones deben ser especificadas utilizando Prolog en un módulo lógico. El método `beliefRevision` invoca a `beliefRevisionD` para utilizar el conocimiento definido en dicho método.

4.8.6 Objetivos

El componente deliberador posee un repositorio con los objetivos (también llamados deseos) del agente. Los objetivos contribuyen y dan origen a las formación de *intenciones* [26].

Un objetivo representa una actitud motivacional acerca de lo que lo que el agente *desearía* hacer. Nótese que esto no implica que el agente deba hacer lo que desea, sino que si el agente decide actuar o razonar, entonces, ese acto deberá tener como fin alcanzar alguno de sus objetivos.

El componente deliberador selecciona un conjunto de objetivos cada vez que se produce una nueva situación de interés. Luego, intenta alcanzar esos objetivos seleccionando uno o más planes abstractos.

Un objetivo consiste de un par (o_i, c_i), donde o_i es un identificador del objetivo, mientras que c_i es la condición de selección o activación del mismo. Por ejemplo, para la aplicación de robots de carga, uno de los objetivos del agente *forklift* es:

$$
\begin{aligned}
o_1 &= \text{unloadTruck}(T) \\
c_1 &= \text{truck}(T) \wedge \neg\text{empty}(T)
\end{aligned}
$$

esto representa el objetivo de descargar el camión T. El objetivo se selecciona cuando el agente cree que existe algún camión (en particular T) y cree que T está vacío. En general, las condiciones que rigen la activación del objetivo sólo contienen predicados acerca de las creencias del agente. Sin embargo, esto no está restringido, sino que podrían utilizarse fórmulas con otro tipo de condiciones.

Un objetivo o_i se selecciona si su condición de activación c_i es verdadera, ni ese objetivo ni su negación están seleccionados, y el agente no cree que o_i es verdadero.

Cuando se selecciona un objetivo, se busca un plan abstracto para alcanzar ese objetivo y se inicia la ejecución de dicho plan (si existe). Luego, se informa a las fuentes de conocimiento acerca de la selección del objetivo. De esta forma, un objetivo podría ser alcanzado al ejecutar un plan abstracto, o al activar una fuente de conocimiento capaz de lograr dicho objetivo. Por ejemplo, una fuente de conocimiento capaz de generar un plan para caminar hacia un objeto situado en el espacio, sería capaz de lograr todos los objetivos de la forma goto(*Object*), siendo *Object* un objeto cuya posición es conocida para el agente. Ante la selección de un objetivo goto(box), la FC se activaría y construiría un plan para llegar a *box*.

4.8.7 Reparación de planes

4.8.7.1 Planes en construcción

Supóngase que uno de los robots de carga descriptos previamente intenta llegar al camión, para lo cual, simplemente se dirige hacia él evitando los obstáculos que encuentra en el camino. Si el medio ambiente es lo suficientemente complejo (ej. un laberinto), entonces, no bastará con *caminar hacia el camión*, sino que habrá que determinar el camino a seguir. Esto podría realizarse utilizando un algoritmo de *planning*.

Típicamente, los algoritmos de *planning* insumen un tiempo considerable, por lo que el robot estaría inactivo mientras planea cómo llegar al camión. Esto podría mejorarse, si, además de planear, camina en la dirección en la que se encuentra el camión en forma simultáneamente. Así, durante esa caminata podría reducir la distancia que lo separa de éste.

Sin embagro, se producirán cambios en las creencias del robot que podrían afectar el plan en construcción en la medida en que recorre el medio ambiente. Por ejemplo, el robot podría encontrar un obstáculo que era desconocido en el momento de comenzar a construir el plan. Además, el simple hecho de que el robot se mueva hacia el camión también constituye un cambio en sus creencias, que, seguramente, tendrían algún efecto sobre el plan en construcción.

Básicamente, un cambio en las creencias puede afectar a un plan de las siguientes maneras:

1. una creencia se hace falsa y esa creencia forma parte de las condiciones iniciales del plan.
2. una creencia que es efecto de una acción se hace verdadera.

En ambos casos, un plan puede ser modificado o reparado. En el caso 1), la reparación del plan tendrá como objetivo hacer que la creencia falsa sea verdadera. En la figura 4.16 se muestra un ejemplo en el que la proposición p_3, incluída dentro de las condiciones iniciales del plan, pasa a ser considerada falsa por el agente. La reparación del plan involucra la adición de una nueva acción a_s que tiene como efecto a p_3.

En general, la reparación de un plan P con condiciones iniciales W, cuando el agente cree que en la falsedad de una proposición $p \in W$ puede realizarse de la siguiente forma:

1. Construir un plan P' con condiciones iniciales W, conteniendo a p como efecto (estado objetivo).
2. El plan reparado es $P'' = P' \mid P$, donde P'' resulta de la concatenación de los planes P' y P.

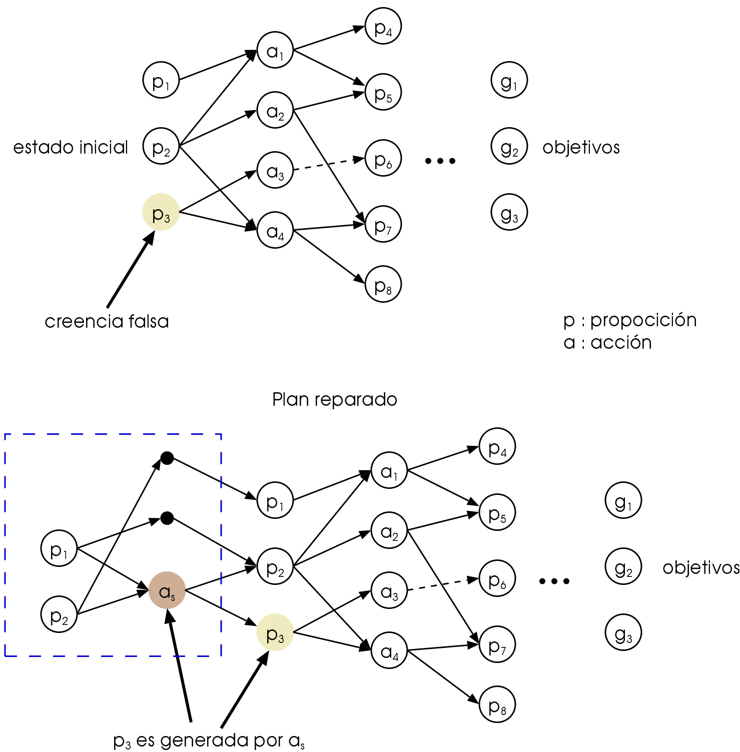


Figura 4.16: Reparación de un plan (creencia falsa)

En el caso 2), el plan contendrá acciones innecesarias debido a que en el momento de iniciarse su construcción se creía en la falsedad de cierta proposición, por lo que habrá una o más acciones innecesarias. En la figura 4.17 se muestra un plan en el que aparece una proposición p_4 que es verdadera según las creencias del agente. Esto hace que la acción a_1 sea innecesaria, por lo que el plan se repara eliminando a_1.

En general, la reparación de un plan P con condiciones iniciales W, cuando el agente cree que en una proposición $p \in W \wedge p \in GP_t$, donde GP_t denota el nivel proposicional válido en tiempo t del grafo de *planning* puede realizarse de la siguiente forma:

1. Para cada acción a, tal que $p \in EFFECTS(a) \wedge \nexists q \in EFFECTS(a) / q \in GP_t$, entonces eliminar a del plan.

2. Para cada acción a eliminada, y para cada $q \in PRE(a)/(\nexists b \in GA_{t-1}/q \in POST(b))$ eliminar q del plan.

3. Eliminar las acciones y proposiciones redundantes aplicando los pasos 1) y 2) en los niveles anteriores del grafo.

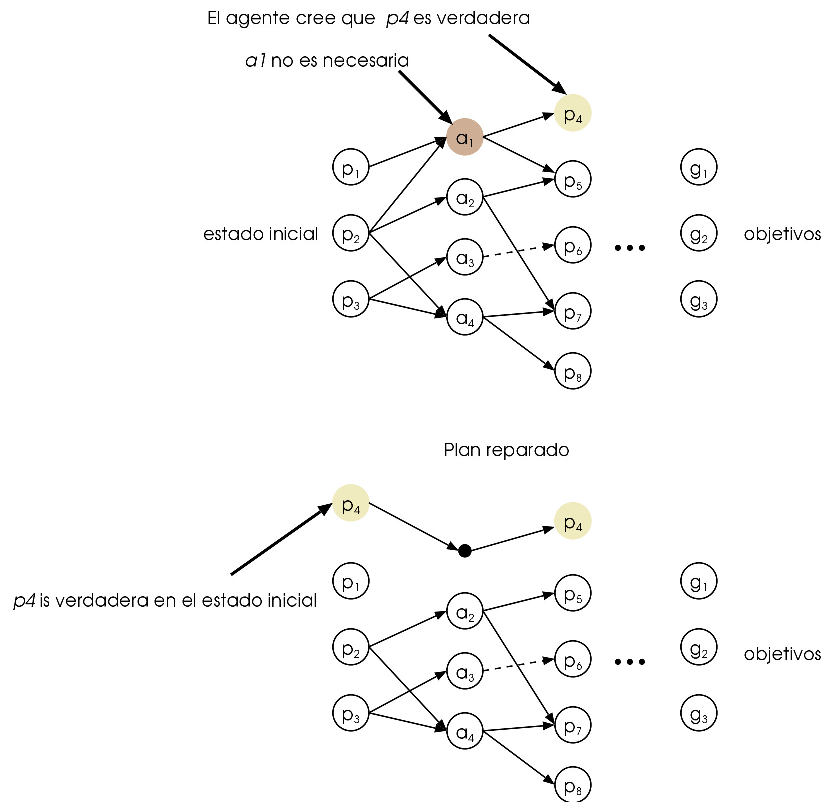


Figura 4.17: Reparación de un plan (acción redundante)

El *framework* Brainstorm/J define una clase que implementa los mecanismos de reparación de planes arriba descriptos. La clase `PlanFixer` define dos métodos: `fixPlanRmBel` y `fixPlanNewBel`. El primero se utiliza para reparar un plan, instancia de la clase `Plan`, cuando una creencia del agente se hace falsa y esa creencia forma parte de las condiciones iniciales del plan. El segundo método se utiliza para eliminar las partes redundantes de un plan cuando un efecto de una acción pasa a formar parte de las creencias del agente.

La clase `PlanFixer` puede ser utilizada por cualquier fuente de conocimiento que construya planes, si existe la posibilidad de que se modifiquen las creencias del agente durante el proceso de construcción de dichos planes. El *framework* no define cuándo se debe reparar un plan. De esta forma, cada fuente de conocimiento es responsable de utilizar o no los servicios provistos por la clase `PlanFixer`, y en caso afirmativo, es responsable de determinar cuándo se reparan los planes.

4.8.7.2 Planes en ejecución

Durante la ejecución de un plan es posible que se modifiquen las creencias del agente. Esto podría hacer fallar la ejecución del plan. En la figura 4.18 se muestra un plan en ejecución en el que se

han añadido arcos que denotan las relaciones entre las acciones y proposiciones (links causales). Las acciones a_1, a_2 y a_3 (en ese orden) ya han sido ejecutadas, por lo que p_5, p_6 y p_7 son verdaderas. Sin embargo, en $t = 1$ no es seguro que la proposición p_1 sea verdadera debido a que el agente no realizó ninguna acción con p_1 como efecto. Mientras se ejecuta a_2 y a_3 el agente percibe que p_1 es falsa. Luego, cuando intenta ejecutar a_5 descubre que esa acción sólo puede ser realizada si p_1 y p_5 son verdaderas, sin embargo p_1 no lo es. Podría considerarse que el intento fallido de ejecutar una acción hace fallar todo el plan, con lo cual se debería construir otro plan que considere a $\neg p_1$en los estados iniciales. Sin embargo esto no sería bueno debido a que se descartaría todo el plan.

Una mejor estrategia sería intentar reparar el plan para considerar las nuevas creencias. Así, en el ejemplo de la figura 4.18 el plan podría ser reparado haciendo verdadero p_1. En general, la reparación del plan se realiza construyendo un nuevo plan que posee como condición inicial todas las proposiciones válidas en el instante de falla y la nueva creencia. Los objetivos sólo difieren de las condiciones iniciales en que incluyen la negación de la nueva creencia.

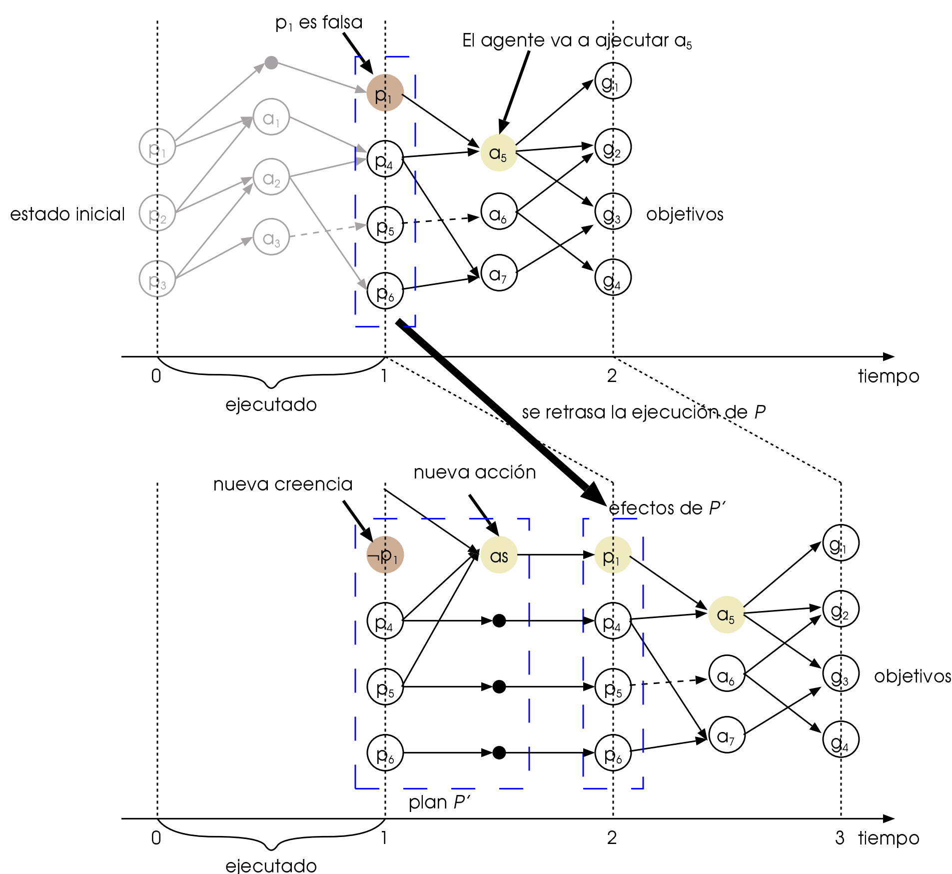


Figura 4.18: Reparación de un plan (creencia falsa)

Por otro lado, podría suceder lo contrario, es decir, que podría surgir una nueva creencia que hiciera innecesaria la ejecución de una acción del plan. En este caso, el mecanismo de adaptación es similar el utilizado durante la construcción del plan.

Brainstorm/J provee una fuente de conocimiento para ejecutar planes completos, y en forma simultánea, adaptar dichos planes a las condiciones dinámicas de las creencias. A tal fin provee la clase

`AdaptableExecutor`, la cual es subclase de `Executor`.

4.9 Comunicación

El componente de comunicación de Brainstorm se extendió para soportar comunicaciones entre agentes físicamente distibuidos y conversaciones similares a las de COOL.

La clase abstracta `Communicator` mostrada en la figura 4.19 es responsable de las capacidades de comunicación de los agentes. Básicamente, el método abstracto `sendMessage` de dicha clase define la interfaz para enviar un mensaje a un agente conociendo su dirección o identificador. Una variación de este método no necesita del destinatario del mensaje. Esto puede ser utilizado cuando la dirección del destinatario se obtiene del mensaje (ej. el campo `:sender` de KQML), o cuando se envían mensajes *broadcast*.

El método *template* `receiveMessage` invoca al método abstracto `processMessage` con el mensaje recibido como parámetro y notifica al administrador de situaciones sobre la recepción de un mensaje.

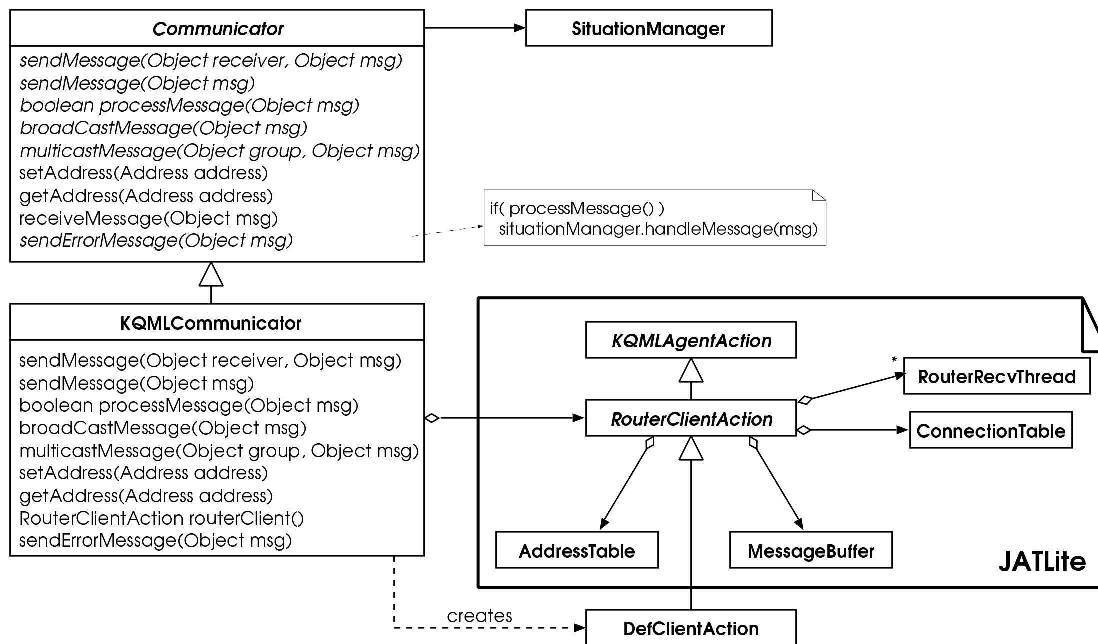


Figura 4.19: Diagrama de clases del componente de comunicación y su especialización para KQML

Brainstorm/J define una clase concreta para comunicar agentes utilizando KQML. La clase concreta `KQMLCommunicator` extiende `Communicator` para enviar y recibir mensajes KQML utilizando TCP/IP, o cualquier protocolo basado en éste, tales como SMTP/POP (e-Mail), FTP o HTTP. Es importante destacar que mediante dicha clase es posible comunicar agentes situados en el mismo sitio, en diferentes máquinas virtuales Java dentro del mismo sitio, o en diferentes sitios.

La clase `KQMLCommunicator` utiliza el *framework* JATLite [75], el cual define mecanismos para comunicar agentes distribuidos utilizando mensajes textuales simples y KQML sobre redes TCP/IP. JATlite puede ser extendido con facilidad para utilizar otros lenguajes de comunicación tales como ACL. Sin

embargo, asume que todas las comunicaciones se realizan sobre redes TCP/IP. En términos prácticos esto no representa ningún problema, debido a la masiva difusión de dicho protocolo.

En la clase `KQMLCommunicator` se define el método `processMessage` para notificar al deliberador acerca de la recepción de un mensaje. El deliberador decide si responde o no a las comunicaciones, y en caso afirmativo cómo las responde.

Un agente con capacidades de comunicación posee un objeto de la clase `Communicator` que le asigna un identificador único dentro de la red denominado *identificador físico*. Generalmente, ese identificador consiste de dos partes: un identificador del sitio en que se encuentra el agente, y un identificador dentro de ese sitio. Por ejemplo, en Internet, el agente será identificado con la dirección IP del sitio en que se encuentra, y una dirección de *port* que lo identifica dentro de ese sitio.

El identificador de un agente es utilizado en todas las operaciones de comunicación entre pares. Así, si se desea enviar un mensaje KQML a un agente, se debe conocer su identificador.

Esto plantea ciertos problemas, debido a que típicamente, el identificador tiene una estrecha relación con el protocolo de red que se utiliza para transportar los mensajes (en este caso TCP/IP). Por otro lado, dicho identificador depende del sitio en que se encuentra el agente. Por supuesto que esto presenta varios inconvenientes. Por ejemplo, £qué sucede si el agente migra hacia otro sitio de la red? o £si se cambia el protocolo de red y cambian las direcciones de sitios?

Para resolver esto, se asigna a cada agente un *identificador lógico*, por ejemplo su nombre. Luego, en el momento de creación del agente se registra el par (*identificador lógico*, *identificador físico*) con un *facilitador*. Un facilitador es un componente que mantiene un registro de los agentes presentes en un sistema multi-agente y su correspondiente identificador físico. De esta forma, cualquier agente puede obtener el identificador físico de otros agentes a partir de un identificador lógico. Así, cuando un agente desea enviar un mensaje KQML a otro agente, utiliza el identificador lógico para obtener el identificador físico del agente destinatario y envía el mensaje.

JATLite permite utilizar el facilitador como ruteador de mensajes, es decir, que todos los mensajes son enviados al facilitador, el cual se encarga de obtener el identificador físico y enviar los mensajes. Esto permite que los agentes operen con conexiones intermitentes. Por ejemplo, un agente puede enviar mensajes a otro agente que no está conectado a la red. Dicho mensaje es almacenado por el ruteador hasta que el agente destinatario se conecte y obtenga sus mensajes. El concepto es similar al e-Mail, y resulta de gran utilidad con agentes móviles o que actúan en forma intermitente.

En Brainstorm/J, las comunicaciones KQML siempre se realizan utilizando un facilitador que actúa de ruteador de mensages. Cuando se crea un agente con capacidades de comunicación, éste registra su nombre y dirección física con el facilitador (figura 4.20). Luego, todo el intercambio de mensajes entre agentes se realiza utilizando al facilitador como intermediario. Así, para cada grupo de agentes que deseen comunicarse utilizando KQML, debe existir un facilitador. Dicho facilitador puede residir en cualquier sitio de la red, por lo tanto, cada uno de los agentes debe conocer la dirección física del mismo.

El facilitador de Brainstorm/J utiliza KQML para manejar las operaciones tales como registración o búsqueda de la dirección de un agente dado su identificador lógico. Por lo tanto, debe ser utilizado con agentes capaces de comunicarse mediante KQML. Esta restricción se debe, básicamente, a que se utiliza el facilitador definido por JATLite.

En la figura 4.19, la clase `RouterClientAction` representa un cliente del facilitador de comunicaciones. Cuando un cliente desea enviar o recibir mensajes, debe conectarse con el facilitador (método

connect), registrarse (método register), luego enviar y recibir los mensajes de la forma usual, y desconectarse (método disconnect). La clase KQMLCommunicator utiliza RouterClientAction para enviar/recibir mensajes utilizando el facilitador. Además, administra la conexión y desconexión en forma automática.

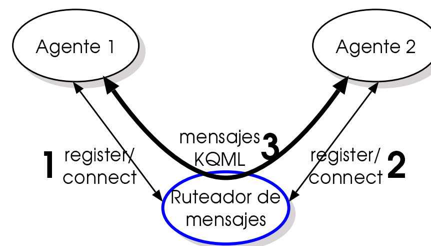


Figura 4.20: Facilitador de comunicaciones (adaptado de [75])

El facilitador, además de proveer servicios de nombres (también llamados *páginas blancas*), provee servicios de *brokering* o *páginas amarillas*, mediante los cuales es posible comunicar agentes en base a los servicios que ofrecen. En la figura 4.21(b) se ejemplifican los servicios básicos del facilitador.

Brainstorm/J define un conjunto de acciones básicas para tratar las performativas KQML, las cuales pueden ser especializadas según las necesidades de cada aplicación. Por ejemplo, para la performativa *ask-one* el comportamiento por defecto verifica si el contenido del mensaje forma parte del conocimiento o creencias del agente, y luego responde utilizando un mensaje con performativa *tell*. Existe una clase concreta para cada performativa: AskOneHandler, AskAllHandler, TellHandler, etc.

Por ejemplo, en la aplicación de robots, podría mejorarse la performance del sistema si los agentes se comunican cada vez que encuentran una caja. Así, podrían utilizar la performativa *tell* para enviar un predicado indicando la localización de una caja a otros agentes:

```
sendMessage(new KQMLmessage("(tell
  :sender forklift2 :receiver forklift1
  :ontology forks :languaje Javalog
  :content location(box("+box.name()+"),"+x+","+y+").)");
```

El comportamiento por defecto definido en la clase TellHandler, consiste en que el agente receptor de un mensaje con performativa *tell* incorpore a su estado mental una creencia con el campo *content* del mensaje. Así, para que un robot pueda entender *tell*, debe ejecutar:

```
deliberator.addConvHandlerForOntology("forks",TellHandler.class);
```

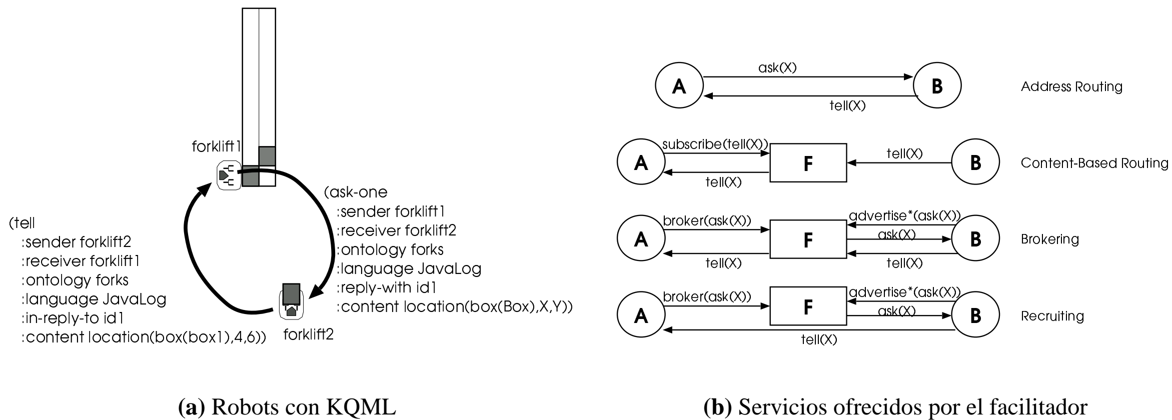
con lo que se agrega una clase de conversación para todo mensaje con performativa *tell* perteneciente a la ontología *forks*[8].

Cuando un robot deja una carga en su lugar, podría preguntar a otros si saben dónde hay una caja utilizando un mensaje con la performativa *ask-one* (figura 4.21(a)):

```
sendMessage(new KQMLmessage("(ask-one
  :sender forklift1 :receiver forklift2
  :ontology forks :languaje Javalog
  :content location(box(Box),X,Y).)");
```

[8]Obsérvese que todos los handlers prede£nidos utilizan JavaLog como lenguaje contenido.

donde `Box`, `X` e `Y` son variables no instanciadas que representan las incógnitas o cosas que el agente desea conocer.



(a) Robots con KQML

(b) Servicios ofrecidos por el facilitador

Figura 4.21: Comunicación con KQML

Brainstorm/J soporta la construcción de agentes que interactúan mediante conversaciones similares a las de COOL [7]. Una clase de conversación es un descripción de las interacciones y acciones que un agente realiza durante una conversación para alcanzar sus objetivos. Una clase de conversación se define mediante un autómata finito determinístico que representa los posibles cursos de acción que podrían realizarse durante un diálogo con otros agentes.

En Brainstorm/J las clases de conversación son similares a los planes abstractos, por lo que se representan de forma similar (figura 4.22). Las clases de conversación estás representadas por la clase `ConvHandler`.

Las reglas conversacionales (transiciones de estados) son representadas por la clase `ConvRule`, la cual especializa `DTransition`. Una regla conversacional relaciona dos estados (e_i, e_j). La transición se produce si se cumple la condición de activación especificada en el método `check`, que en este caso consta de:

- una condición sobre el mensaje recibido (método `messageMatch`). Indica las propiedades que debe tener el mensaje recibido para que se produzca la transición.
- una condición extra sobre los estados mentales del agente (método `suchThat`).

Si ambas condiciones son verdaderas, entonces se ejecutan el método `doBefore`, se envía un mensaje KQML y se ejecuta el método `doAfter`. En la sección 5.2 se presentan ejemplos sobre la utilización de conversaciones en Brainstorm/J.

Cuando un agente recibe un mensaje KQML con el campo `:conversation` conteniendo una cadena de caracteres, se delega el tratamiento de dicho mensaje al componente deliberador. Dicho componente determina si existe una conversación para tratar dicho mensaje. En caso afirmativo, delega el tratamiento del mensaje en dicha conversación. En caso contrario, verifica si el agente posee una clase de conversación (representada por `ConvHandler`) para iniciar una conversación con el mensaje recibido. Si no existe, se enviá un error al emisor del mensaje.

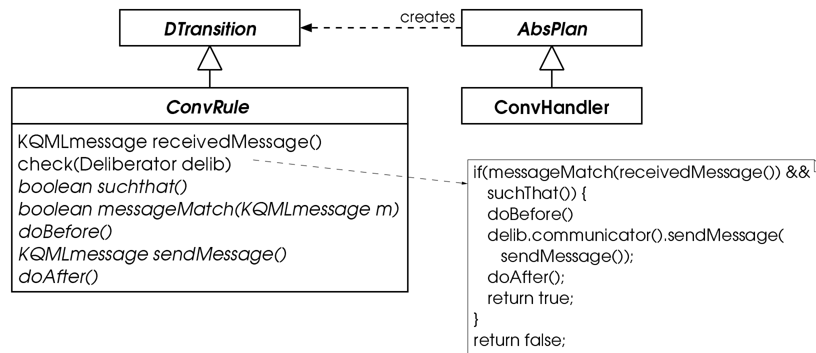


Figura 4.22: Clases de conversaciones representadas como planes abstractos

4.10 Conclusiones

En este capítulo se describió el *framework* de agentes Brainstorm/J. Brainstorm/J ha sido desarrollado a partir de la arquitectura Brainstorm, la cual prescribe agentes capaces de manipular estados mentales, percibir, actuar, comunicarse, reaccionar y deliberar.

La materialización básica de la arquitectura ha sido extendida para soportar comunicaciones entre agentes físicamente distribuidos, conversaciones COOL, multi-*threading* en los procesos internos del agente y revisión de creencias.

En el siguiente capítulo se presentan dos sistemas multi-agente desarrollados con el *framework*.

CAPÍTULO 5

Instanciación de Brainstorm/J

Para construir agentes utilizando Brainstorm/J es necesario especializar e instanciar clases definidas por el *framework*. En el presente capítulo se describe este proceso a través de dos sistemas multi-agente. Además, se analizan algunos experimentos y comparaciones realizadas con estos SMA.

A continuación se describe la organización del capítulo: en la sección 5.1 se presenta la implementación del SMA *Forklifts*; luego, en la sección 5.2 se describe la implementación de una solución multi-agente al problema de las n reinas.

5.1 Forklifts

En esta sección se presenta un sistema multi-agente basado en FORKS [42] desarrollado con el *framework* Brainstorm/J. FORKS consiste de un conjunto de robots (agentes) cuyo objetivo es descargar un número de cargas de un camión y colocarlas en zonas de descarga. En la figura 5.1 se muestra un diagrama de la aplicación. Consiste de una grilla rectangular dividida en regiones. Una región contiene un camión del cual los robots toman las cargas. Las zonas en gris claro situadas en la parte inferior y superior izquierda de la grilla son los lugares de los cuales parten inicialmente los robots. También hay regiones de descarga (o estanterías) en las cuales es posible depositar las cargas y, finalmente, zonas de libre tránsito.

Cada robot puede llevar una carga a la vez y sólo puede tomarla o dejarla en una celda inmediatamente en frente de él. Además, un robot puede girar hacia cualquiera de los puntos cardinales y avanzar en el sentido en que se encuentra, es decir, que no puede, por ejemplo, avanzar directamente en diagonal. Los robots están dotados de un sensor simple que les permite percibir y detectar el objeto que se encuentra inmediatamente en frente.

En las siguientes secciones se describe la implementación del sistema multi-agente.

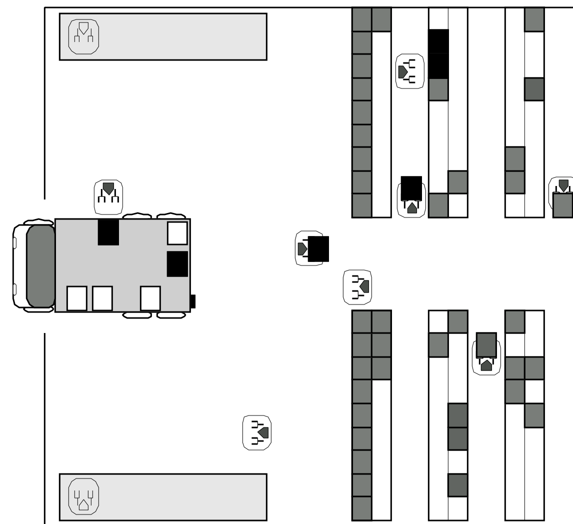


Figura 5.1: Sistema multi-agente FORKS

5.1.1 Nivel base

El primer paso en el desarrollo del un sistema multi-agente utilizando Brainstorm/J consiste en definir las responsabilidades del nivel base y del nivel re¤exivo.

En esta aplicación, el nivel base contiene objetos que representan a los robots sin capacidades de agentes. Cada uno de esos objetos es responsable de la capacidad efectora básica de los agentes, entre otras cosas. Por ejemplo, tomar una caja, dejar una caja, girar, avanzar y percibir. Estas capacidades se representan, respectivamente, en los métodos `graspBox`, `putBox`, `turnTo`, `advance` y `lookAt` de la clase `Forklift` mostrada en la figura 5.2.

En la figura 5.2 también se muestra un diagrama con las clases que representan el ambiente en el cual se mueven los robots (`LoadingDock`) y cada uno de los elementos presentes en el ambiente (`Element`): áreas (`Area`), camiones (`Truck`) y estanterías (`Shelf`).

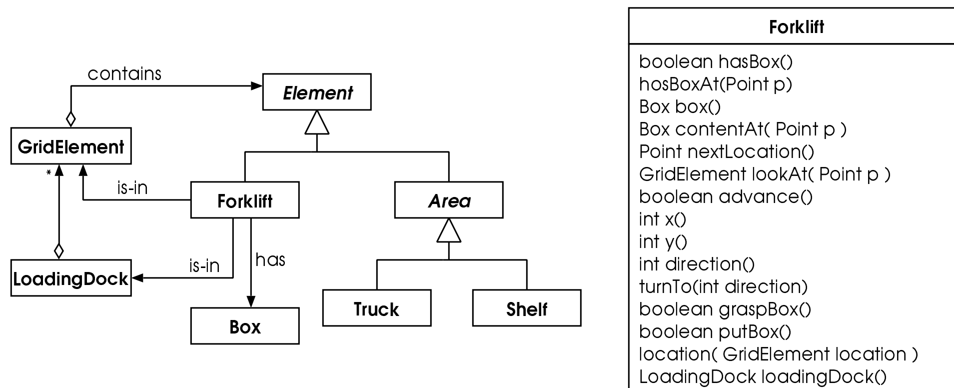


Figura 5.2: Clases de los objetos situados en el nivel base

5.1.2 Creación de un agente

Cada agente *forklift* está formado por un objeto de la clase `Forklift` (objeto situado en el nivel base) al cual se le asocian un conjunto de objetos y meta-objetos responsables de las capacidades de agentes: percepción, reacción, deliberación, manipulación de estados mentales y detección de situaciones.

Para crear agentes *forklift* con capacidades de reacción, comunicación y deliberación hay que crear un meta-objeto de la clase `MetaAgent` indicando que la clase de los objetos situados en el nivel base es `Forklift`:

```
MetaAgent metaAgent=new MetaAgent( "Forklift" );
```

Esto, además de crear el meta-objeto responsable de crear los agentes, asocia ese meta-objeto con la clase `Forklift` para interceptar el mensaje `new`. De esta forma, cada vez que se crean objetos de la clase `Forklift` se activa el meta-objeto que crea y asocia capacidades de agentes a dicho objeto.

Luego, se debe indicar al meta-objeto que los agentes *forklift* poseen capacidad de reacción y deliberación invocando a los métodos `hasReaction` y `hasDeliberation` (por defecto, todos los agentes poseen capacidad de manipular estados mentales y detectar situaciones):

```
// El agente tiene reacción y deliberación
metaAgent.hasDeliberation(true);
metaAgent.hasReaction(true);
```

finalmente, deben indicarse las clases que implementan las capacidades de reacción y deliberación (si no son las predefinidas por el *framework*), e indicar una subclase de `BasicAgent` responsable de inicializar el agente:

```
// El componente de reacción por defecto es Reactor
// El deliberador es instancia de ForkliftDeliberator
metaAgent.deliberatorClass( ForkliftDeliberator.class );
// La clase BasicAgent se especializó mediante ForkliftAgent
metaAgent.basicAgentClass( ForkliftAgent.class );
```

En la figura 5.3 se muestran las principales clases involucradas en la creación de los agentes *forklift*.

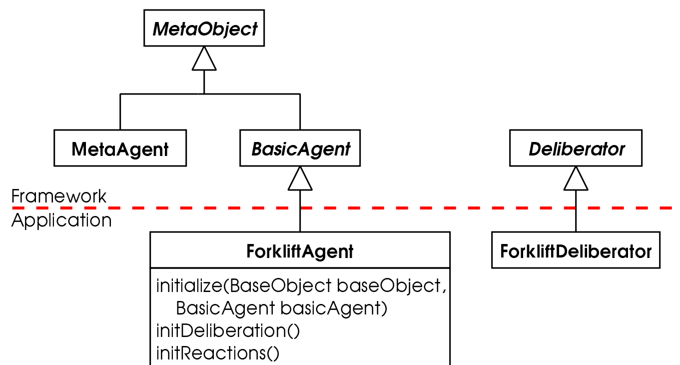


Figura 5.3: Principales clases involucradas en la creación de los agentes *forklift*

5.1.3 Capacidad de acción

Las capacidades básicas de acción de los agentes están representadas en los métodos `graspBox`, `putBox`, `turnTo`, `advance` y `lookAt` de la clase `Forklift`. Para que el agente conozca cuáles son sus capacidades básicas, se debe indicar al `MetaAgent` responsable de crear los agentes cuáles son los métodos que implementan cada una de las capacidades:

```
metaAgent.addCapability("turnTo",new Class[] {Integer.TYPE});
metaAgent.addCapability("graspBox",null);
metaAgent.addCapability("advance",null);
metaAgent.addCapability("putBox",null);
```

Luego, para que un agente pueda utilizar sus capacidades, por ejemplo, para construir un plan, debe conocer las precondiciones y efectos de cada una de esas capacidades. A continuación se definen las capacidades *advance* y *turnTo* en base a sus parámetros, precondiciones, efectos y capacidad básica asociada:

advance(*Dir*, *X*, *Y*, *NewX*, *NewY*)

PARÁMETROS: direction(*Dir*)$\wedge$isValid(*X*, *Y*)$\wedge$isValid(*NewX*, *NewY*) $\wedge$nextLocation(*Dir*, *X*, *Y*, *NewX*, *NewY*)

PRECONDICIÓN: lookingAt(*Dir*)$\wedge$location(forklift(aForklift), *X*, *Y*)

EFECTO: lookingAt(*DirF*)$\wedge\neg$lookingAt(*DirI*)

CAPACIDAD BÁSICA ASOCIADA: advance()

turnTo(*DirI*, *DirF*)

PARÁMETROS: direction(*DirI*)$\wedge$direction(*DirF*)$\wedge$*DirF*$\neq$*DirI*

PRECONDICIÓN: lookingAt(*DirI*)

EFECTO: lookingAt(*DirF*)$\wedge\neg$lookingAt(*DirI*)

CAPACIDAD BÁSICA ASOCIADA: turnTo(*DirF*)

Estas capacidades se especifican en las clases `Action_advance0` y `Action_turnTo1`, respectivamente, mostradas en la figura 5.4. Las capacidades de tomar y dejar cajas se especifican en las clases `Action_graspBox` y `Action_putBox0`.

5.1.4 Percepción

Cada robot posee un sensor capaz de detectar y distinguir objetos que se encuentran frente a él. Esta funcionalidad es parte del objeto base y está implementada en el método `lookAt` de la clase `Forklift`. La percepción puede realizarse cada vez que el robot avanza o gira, de forma tal de observar qué hay frente a él (mediante el método `lookAt`) cuando cambia de posición. Tales capacidades de percepción pueden ser logradas asociando un meta-objeto de percepción al objeto base (instancia de `Forklift`) para que perciba los mensajes correspondientes a las actividades girar (`turnTo`) y avanzar (`advance`).

En el siguiente fragmento de código[1] se asocia un meta-objeto de percepción a una instancia del objeto base `forklift`:

```
...
// Obtiene la instancia de ForkliftAgent que refleja al objeto forklift
```

[1] es continuación del código de la sección 5.1.2.

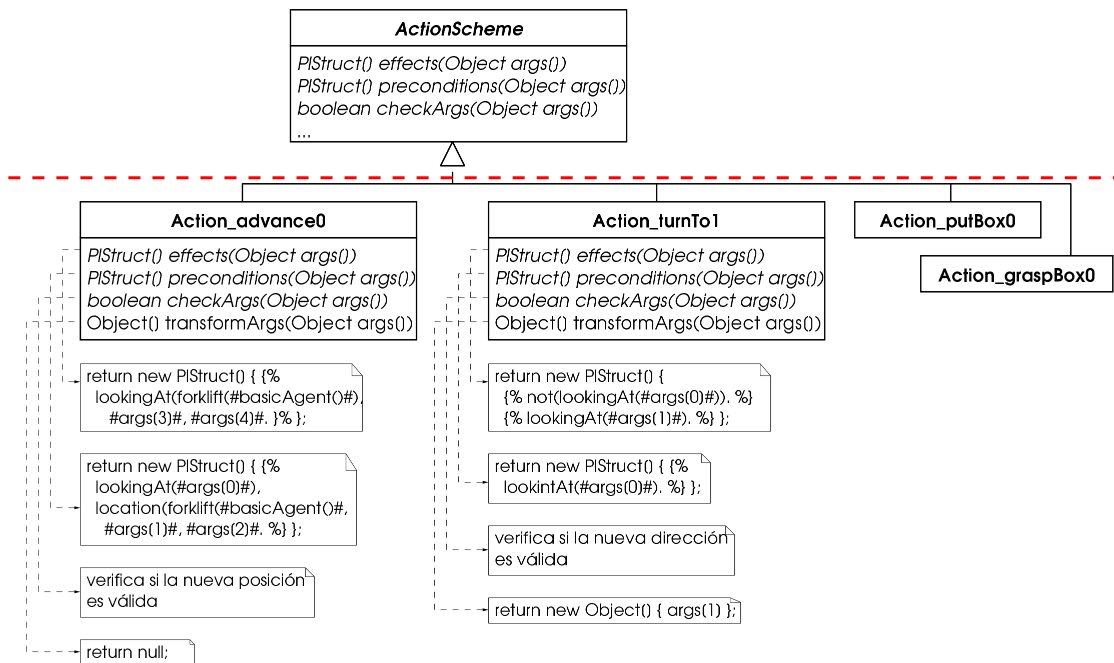


Figura 5.4: Capacidad de acción de los agentes

```
BasicAgent basicAgent=metaAgent.BasicAgentOf(forklift);
// Indica que debe percibirse el mensaje 'advance()'
// en el objeto forklift
basicAgent.perceiveMethodOf(forklift,"advance",null);
// Indica que debe percibirse el mensaje 'turnTo(int)'
basicAgent.perceiveMethodOf(forklift,"turnTo",
   new Class[] { Integer.TYPE });
// Inicia la percepción
basicAgent.startPerceptionOn(forklift);
```

Cada vez que el objeto base `forklift` reciba el mensaje `advance` indicándole que debe avanzar un paso hacia adelante, o el mensaje `turnTo` indicándole que debe rotar, entonces su meta-objeto de percepción será notificado.

En la aplicación de robots de carga cada uno de los robots desconoce el medio ambiente en que se encuentra, y por lo tanto ignora dónde se encuentra el camión, las cargas, las estanterías, otros robots, etc. Básicamente, lo que se hace es utilizar las capacidades de percepción para que cada agente construya un modelo mental del ambiente utilizando el repositorio de creencias del componente deliberador.

Para definir la funcionalidad requerida, se extendió la clase `Perceptor` en `ForkliftPerceptor`. En dicha clase se redefine el método `messagePerceived` para que agregue una creencia al estado mental del agente si se percibe un camión, una caja o una estantería:

```
Forklift forklift = (Forklift)agent.baseObject();
GridElement ge = forklift.lookAt(forklift.nextLocation());
Deliberator deliberator = agent.deliberator();
```

```
if( ge != null ) {
  Point location = forklift.nextLocation();
  if( ge.isShelfRegion() || ge.isTruckZone() ) {
    Element elem = ge.content();
    if( elem.hasBoxAt( location ) )
      deliberator.addBelief( {%
        location(box(#elem.contentAt(location)#),
        #new Integer(ge.x())#,
        #new Integer(ge.y())#).%} );
    if( ge.isTruckZone() ) {
      deliberator.addBelief( {%
        location(truck(#elem#),
        #new Integer(ge.x())#,
        #new Integer(ge.y())#).%} );
    else if( ge.isShelfRegion() )
      deliberator.addBelief( {%
        location(shelf(#elem#),
        #new Integer(ge.x())#,
        #new Integer(ge.y())#).%} );
    situationManager().handleMessage( reflectedMessage );
  }
}
reflectedMessage.send();
```

5.1.5 Situaciones

En la aplicación hay tres situaciones de interés: *"hay una caja enfrente"*, *"hay una estantería libre enfrente"* y *"hay un camión enfrente"*. La primer situación, denominada *boxInFront*, se detecta cuando hay una caja en las coordenadas (X,Y), tales que (X,Y) son las coordenadas que tendría el robot (objeto base) si avanza. Esto se expresa mediante la siguiente regla Prolog:

```
situation(boxInFront,Box) :-
  location(box(Box),X,Y), /* Box está en X, Y */
  newInstance('java.awt.Point',[X,Y],Front),  /* Front = (X,Y) */
  baseObject(Base), /* Base es una instancia de Forklift */
  send(Base,nextLocation,[],Front). /* Front está delante del robot */
```

en este fragmento de código, la relación location(box(*Box*), *X*, *Y*) indica que la caja *Box* se encuentra en las coordenadas (X,Y). La relación newInstance('java.awt.Point', [*X*, *Y*], *Front*) expresa que *Front* es un punto (instancia de la clase `java.awt.Point`) de coordenadas (X,Y), mientras que baseObject(*Base*) indica que *Base* es el objeto base del agente, en este caso una instancia de la clase `Forklift`. Así, el predicado situation(boxInFront, *Box*) es verdadero si se cumple que la caja *Box* se encuentra frente al robot.

La situación *"hay una estantería libre enfrente"*, denominada *freeShelfInFront*, se detecta cuando hay una estantería libre en las coordenadas (X,Y), tales que (X,Y) son las coordenadas que tendría el robot (objeto base) si avanza:

```
situation(freeShelfInFront,Shelf) :-
  location(shelf(Shelf),X,Y), /* Shelf está en X, Y */
  newInstance('java.awt.Point',[X,Y],Front), /* Front = (X,Y) */
  not(send(Shelf,hasBoxAt,[Front])), /* Shelf está libre */
  baseObject(Base),
  send(Base,nextLocation,[],Front).
```

En forma similar, la situación *"hay un camión enfrente"* se expresa de la siguiente forma:

```
situation(truckInFront,ShelfInTruck) :-
  location(truck(ShelfInTruck),X,Y),
  newInstance('java.awt.Point',[X,Y],Front), /* Front = (X,Y) */
  baseObject(Base),
  send(Base,nextLocation,[],Front).
```

5.1.6 Comportamiento reactivo

En la sección anterior se especificó una situación simple denominada *boxInFront*. Una posible reacción ante esta situación sería intentar tomar la caja que se encuentra en frente. La capacidad de un robot de tomar una caja fue definida anteriormente en la clase `Action_graspBox0`, por lo tanto, la reacción puede ser definida de la siguiente manera:

PRECONDICIÓN: *situation*="boxInFront" && not(hasBox(*Agent*, _))
ACCIÓN: `Action_graspBox0`

El predicado not(hasBox(*Agent*, _)) indica que el robot no lleva una carga, por lo tanto podría tomar la que se encuentra frente a él. En la figura 4.8 se muestra la definición de esa reacción en la clase `BoxInFrontReaction`.

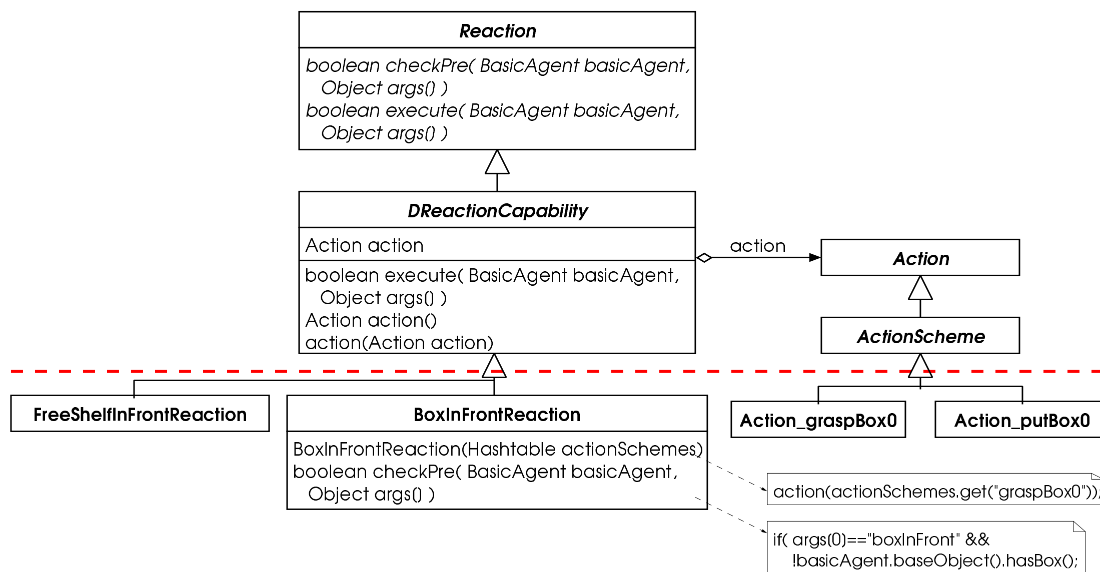


Figura 5.5: Reacciones de los agentes *forklift*

De manera similar, es posible definir una reacción para la situación *freeShelfInFront*:

PRECONDICIÓN: *situation=”freeShelfInFront”* && hasBox(*Agent*, _)
ACCIÓN: `Action_putBox0`

En la figura 4.8 se muestra la definición de esa reacción en la clase `FreeShelfInFrontReaction`.

5.1.7 Deliberación

Para utilizar el deliberador, deben definirse las reglas de inconsistencia semántica de creencias, objetivos, las fuentes de conocimiento que producen el comportamiento de los agentes y los planes abstractos. En las siguientes secciones se detalla cada una de estos aspectos.

5.1.7.1 Creencias

Básicamente, un agente no puede creer, en forma simultánea, en:

- estar observando en dos direcciones diferentes.
- un objeto se encuentra en dos lugares.
- una caja que está siendo transportada por un agente, está en otro lugar.

Estas reglas deben ser especificadas en el método `beliefRevisionD` de la clase `Deliberator`:

```
public void beliefRevisionD() {
  {{
    si(lookingAt(L1),lookingAt(L2)) :- L1 \\== L2.
    si(location(R,X1,Y),location(R,X2,Y)) :- X1 \\== X2.
    si(location(R,X,Y1),location(R,X,Y2)) :- Y1 \\== Y2.
    si(location(R,X1,Y1),location(R,X2,Y2)) :- X1 \\== X2, Y1 \\== Y2.
    si(hasBox(_,Box),location(box(Box),_,_)).
    si(location(box(Box),_,_),hasBox(_,Box)).
  }};
}
```

5.1.7.2 Objetivos

Los agentes sólo poseen un objetivo: descargar el camión. Brainstorm/J representa los objetivos mediante cláusulas Prolog. En este caso, el objetivo *“descargar un camión”* se define como:

```
goal(unloadTruck(Truck)) :-
  truck(Truck),
  not(send(Truck,empty,[])).
```

el predicado truck(*Truck*) se cumple si el agente cree que *Truck* es un camión; el predicado not(send(*Truck*, empty, [])) se cumple si *Truck* no está vacío. Así, el agente podría intentar descargar un camión si dicho camión posee cargas.

5.1.7.3 Fuentes de conocimiento

Se definieron los siguientes tipos de fuentes de conocimiento:

- `RandomWalkerKS`: produce un plan compuesto por dos acciones: girar en una dirección aleatoria y avanzar, lo coloca en el *blackboard* y espera a que se ejecute. Esta clase es subclase de `MultiPlanProducerKS`, la cual define fuentes de conocimiento que generan un plan (método abstracto `getPlan`), lo colocan en el *blackboard* y esperan a que se ejecute. Luego, repiten este proceso hasta que son destruidas.
- `DistanceReductionKS`: es similar a la anterior, aunque la dirección no es generada de manera aleatoria, sino que elige una dirección para reducir la distancia entre el agente y un objeto del ambiente.
- `BorderKS`: genera acciones para que un agente explore en forma sistemática los alrededores de un objeto. Por ejemplo, cuando agente llega a un camión, podría crearse una fuente de conocimiento `BorderKS` para que el agente busque una caja en el camión.

5.1.7.4 Planes abstractos

Para definir un plan abstracto hay que especializar la clase `AbsPlan`, definiendo el autómata que representa un plan abstracto mediante su matriz de transición de estados, las condiciones de transición, las fuentes de conocimiento asociadas a cada estado y la condición de activación del plan (objetivos que se alcanzan cuando se ejecuta el plan). Para definir esto se utilizan los métodos `nextState`, `tr`, `ksClasses` y `checkPre` de la clase `AbsPlan`, respectivamente. Por ejemplo, para construir el plan abstracto de figura 5.6 puede definirse una subclase de `AbsPlan` con el siguiente constructor:

```
super( deliberator );
ksClasses(new Class[] {RandomWalkerKS.class, RandomWalkerKS.class,
  RandomWalkerKS.class});
nextState(new int[][] {{1,2,0}, {2,1}, {3,2} });
tr(new Class[][] {
  {TruckInFrontTr.class, BoxInFrontTr.class, DTrueTransition.class},
  {BoxInFrontTr.class, DTrueTransition.class},
  {FreeShelfInFrontTr.class, DTrueTransition.class} });
final(3);
```

En este fragmento de código se invoca al método `ksClasses` con un arreglo de clases. Así, se asocia el comportamiento de caminata aleatoria implementado en la clase `RandomWalkerKW` a tres de los cuatro estados del autómata. Cada uno de los estados del autómata es identificado por un número entero, comenzando por 0 para el estado inicial. El método `nextState` inicializa la matriz de transición de estados del autómata. Luego, mediante el método `tr` se asocia una condición a dichas transiciones.

La clase `DTrueTransition` define una condición de transición que siempre se cumple, por lo que se utiliza para indicar una transición que se activa por defecto cuando todas las otras fallan.

Finalmente, para terminar de definir el plan abstracto, debe definirse el método `checkPre` indicando la condición de activación de dicho plan. En este caso, el plan se activa cuando el robot intenta descargar un camión, lo cual es representado mediante una consulta JavaLog:

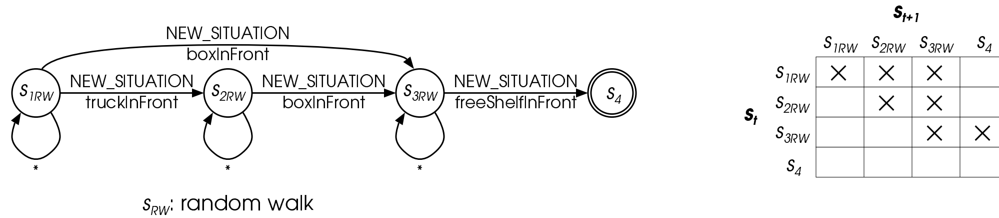


Figura 5.6: Plan abstracto para descargar un camión (sólo caminata aleatoria)

```
public boolean checkPre( Object args[] ) {
  ?-intend(unloadTruck(_)).
}
```

En forma similar, es posible ampliar el plan abstracto de la figura 5.6, asociando a cada estado del mismo las fuentes de conocimiento de reducción de distancia y de exploración definidas previamente, con el fin de obtener comportamiento más *inteligente*.

5.2 *n*-Queens

Para mostrar la utilización de Brainstorm/J con agentes con capacidades de comunicación, y con el fin de comparar Brainstorm/J con JAFMAS [20], se desarrolló una solución multi-agente al problema de las *n*-reinas [20]. Dicho problema consiste en posicionar n reinas en un tablero de ajedrez de $n \times n$ de forma tal que ninguna de las reinas se encuentre amenazada. Cada una de las reinas se representó mediante un agente capaz de desplazarse sobre el eje y del tablero, con una coordenada x fija.

Las reinas poseen conocimiento limitado acerca de las ubicaciones de las otras reinas. Además, cada reina sólo puede comunicarse con las reinas vecinas situadas a su derecha e izquierda. La reina del borde derecho del tablero sólo se comunica con la reina de su izquierda, mientras que la reina del extremo izquierdo sólo se comunica con la reina situada a su derecha. Se asume que dos reinas no pueden ocupar una misma columna del tablero.

Las reinas interactúan mediante mensajes KQML con tres performativas:

- *Propose*: cuando una reina se coloca en una posición del tablero envía un mensaje *propose* a la reina situada a su derecha.
- *Accept*: cuando una reina encuentra una posición que no amenaza a ninguna de las reinas situadas a su izquierda envía un mensaje *accept* a la reina situada a su izquierda.
- *Reject*: cuando una reina no puede encontrar una posición que no amenace a ninguna de las reinas situadas a su izquierda envía un mensaje *reject* a la reina situada a su izquierda.

El contenido de los mensajes, para una reina i situada en la columna i es la lista de posiciones de todas las reinas situadas a su izquierda. Así, las reinas sólo conocen las ubicaciones de las reinas de su izquierda. Luego, si la reina del extremo derecho del tablero encuentra una posición, entonces todas las reinas habrán encontrado una posición segura. Por otro lado, si la reina del extremo izquierdo del tablero no encuentra un posición segura, significa que no hay más soluciones posibles.

Resumiento, las reinas intercambian los siguientes mensajes:

- Propuesta: `(propose :sender` q_i `:receiver` q_{i+1} `:content` $(y_1, y_2, \ldots, y_i)$`)`
- Aceptación: `(accept :sender` q_i `:receiver` q_{i-1} `:content` $(y_1, y_2, \ldots, y_i)$`)`
- Rechazo: `(reject :sender` q_i `:receiver` q_{i-1} `:content` $(y_1, y_2, \ldots, y_{i-1})$`)`

La conversación que cada agente realiza depende de su posición x en el tablero. Así, es posible distinguir tres clases de conversaciones en la aplicación:

- *FirstQueenConv*: modela las conversaciones de la reina situada en el extremo izquierdo del tablero (figura 5.7).
- *MiddleQueenConv*: modela las conversaciones de las reinas situadas en las columnas del centro del tablero (figura 5.8).
- *LastQueenConv*: modela las conversaciones de la reina situada en el extremo derecho del tablero (figura 5.9).

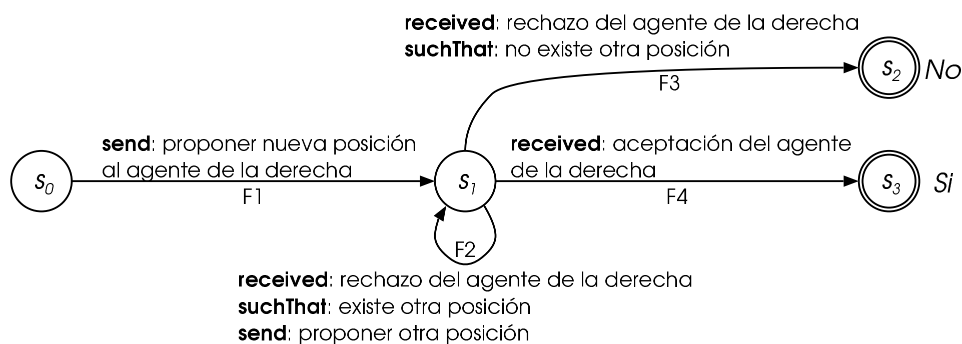


Figura 5.7: Representación de *FirstQueenConv*

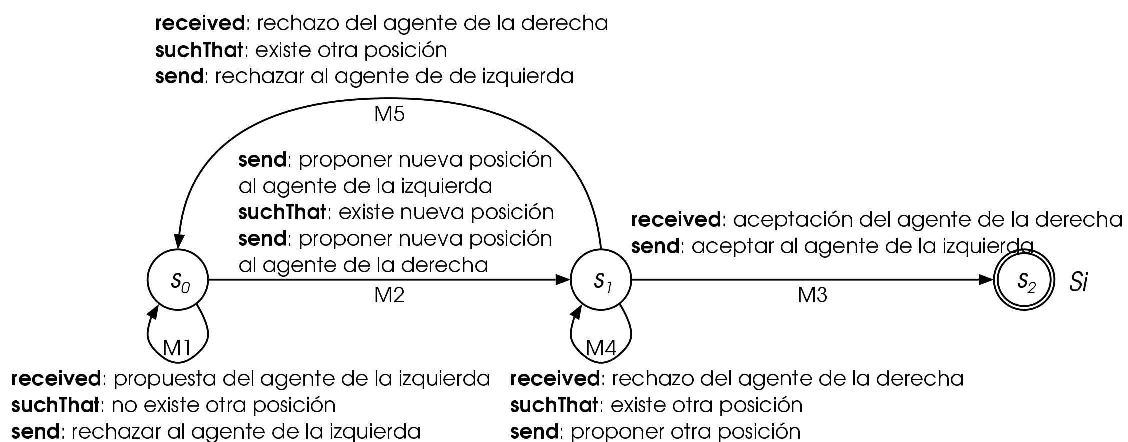


Figura 5.8: Representación de *MiddleQueenConv*

5.2.1 Implementación

En la figura 5.10 se muestran los principales objetos y meta-objetos que componen cada agente. El nivel base está formado por dos objetos: un objeto de la clase `Queen`, el cual representa a la reina

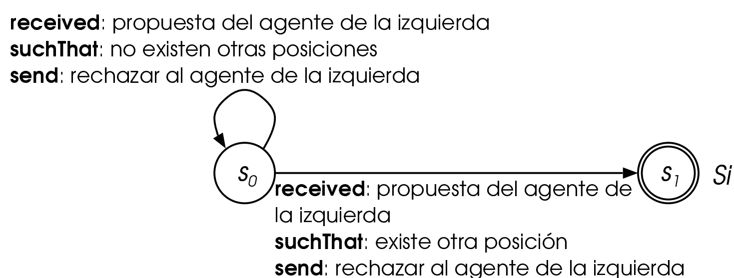


Figura 5.9: Representación de *LastQueenConv*

de ajedrez sin capacidades de agentes, y un objeto de la clase `Board` que representa al tablero. El nivel meta de cada agente sólo posee los objetos y meta-objetos responsables de las capacidades de comunicación, deliberación, representación y manipulación de los estados mentales.

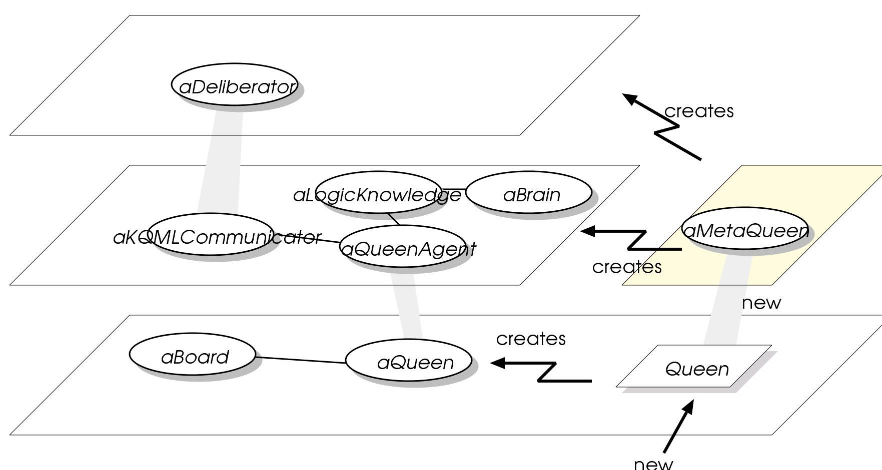


Figura 5.10: Objetos y meta-objetos de un agente reina

En general, la instanciación de Brainstorm/J para construir los agentes que representan a las reinas es más simple que en la aplicación de la sección 5.1, por lo que no se describe todo el proceso, sino que se sólo detallan los aspectos de comunicación y conversaciones presentadas en el capítulo anterior (sección 4.9).

Los agentes interactúan por medio de mensajes KQML, por lo tanto, utilizan un objeto de la clase `KQMLCommunicator`. El componente deliberativo es utilizado para manejar las conversaciones multi-agente.

Cada uno de los agentes es capaz de conversar según una clase de conversación que depende de la columna que ocupa en el tablero. Así, por ejemplo, el agente que ocupa la columna 0 del tablero (la del extremo izquierdo) posee una representación de la clase de conversación *FirstQueenConv*.

Brainstorm/J posee dos clases que permiten representar clases de conversaciones: `ConvHandler` y `ConvRule`. La clase `ConvHandler` representa una clase de conversación mediante un autómata finito determinístico. Cada transición del autómata, denominada regla conversacional [7], es representada mediante la clase `ConvRule`.

Para implementar la clase de conversación *MiddleQueenConv* se especializó `ConvHandler`, definien-

do la clase `MiddleQueenConv` como se muestra en la figura 5.11. Cada una de las transiciones del autómata de la figura 5.8 se representó mediante las subclases de `ConvRule`: `M1`, `M2`, `M3`, `M4` y `M5`.

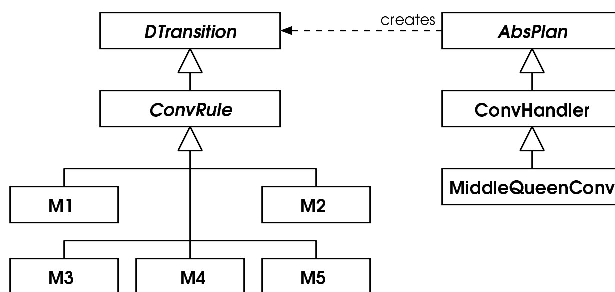


Figura 5.11: Materialización de *MiddleQueenConv* utilizando Brainstorm/J

La clase `MiddleQueenConv` sólo redefine el constructor de `ConvHandler`. En el mismo, se indica cuál es el estado final, y especifican las clases de las fuentes de conocimiento asociadas a cada estado no final, que en este caso son nulas:

```
isFinal( 2 );
ksClasses(new Class[] {null,null});
```

Luego, se especifica la matriz de transición de estados del autómata que representa la clase de conversación:

```
nextState(new int[][] { {0, 1},
                        {0, 1, 2} });
```

Por último, se asocia a cada una de esas transiciones una clase representando la regla conversacional:

```
tr(new Class[][] { {M1.class, M2.class },
                   {M5.class, M4.class, M3,class} });
```

Cada una de las reglas conversacionales especifica las condiciones que se deben cumplir para que se efectúe la transición de estados, e indica una acción que es ejecutada si la transición se produce.

Cada una de las reglas de la clase *MiddleQueenConv* se representó mediante una clase con el mismo nombre que la regla. Por ejemplo, para representar la regla *M1* (figura 5.8) se implementó una subclase de `ConvRule` llamada `M1`. La regla *M1* se cumple si:

- se recive una propuesta (*propose*) del agente de la izquierda.
- no existe otra posición segura a la que desplazarse.

Esto se representa mediante los métodos `messageMatch` y `suchThat` de la clase `M1`, respectivamente:

```
public boolean messageMatch( KQMLmessage receivedMessage ) {
  // (propose :sender q_i ....
  return receivedMessage.getValue("performative").equals("propose") &&
         receivedMessage.getValue("sender").equals(
           ((Queen)deliberator.basicAgent().baseObject())
               .leftQueen().name() );
```

```
}

public boolean suchThat() {
  // (propose ...... :content (y1, y2, ..., yi)
  // Obtiene el objeto del nivel base
  Queen q = (Queen)deliberator.basicAgent().baseObject();
  // actualiza el tablero con las posiciones recibidas
  q.board().setBoard(
    new takeOffList(receivedMessage().getValue("content")));
  // verifica que no exista una posición segura
  return q.newPositionExists() == -1;
}
```

Si esas condiciones se cumplen, se envía un mensaje *reject* al agente de la izquierda:

```
public KQMLmessage sendMessage() {
  KQMLmessage msg = new KQMLmessage();
  msg.addFieldValuePair( "performative", "reject" );
  msg.addFieldValuePair( "receiver", msg.getValue("sender") );
  msg.addFieldValuePair( "content", ((Queen)deliberator.basicAgent()
    .baseObject()).boardAtLeft() );
  return msg;
}
```

Cuando se crea el componente deliberativo de cada agente, se deben agregar al mismo las clases de conversaciones para que el agente sea capaz de utilizarlas. La clase `QueenDeliberator` especializa `Deliberator` definiendo algunas particularidades de la aplicación. Así, por ejemplo, en el método `initialize` de `QueenDeliberator` se preparan las clases de conversaciones:

```
Queen q = (Queen)basicAgent.baseObject();
if( q.x() == 0 )
  convHandlerFactory().addConvHandler("queens",
    FirstQueenConv.class);
else if( q.x() == q.board().size() - 1 )
  convHandlerFactory().addConvHandler("queens",
    LastQueenConv.class);
else
  convHandlerFactory().addConvHandler("queens",
    MiddleQueenConv.class);
```

Para correr la aplicación es necesario efectuar los siguientes pasos: en primer lugar debe ejecutarse el facilidador de comunicaciones KQML provisto por Brainstorm/J; luego, deben crearse *n* instancias del los agentes. Opcionalmente, los agentes pueden ejecutar en varias máquinas de una red.

Cada uno de los agentes presenta una ventana como la mostrada en la figura 5.12. La parte izquierda de la ventana muestra una representación del conocimiento parcial del agente sobre las posiciones de las otras reinas. La caja de texto de la derecha presenta una lista con los mensajes enviados y recibidos. El botón *Step* se utiliza para indicar a un agente que reciba o envíe un mensaje, mientras que el botón *Run* hace que todos los agentes encuentren una solución.

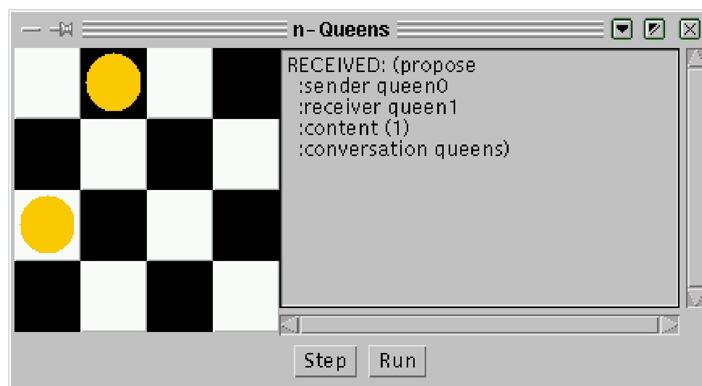


Figura 5.12: Ventana presentada por cada reina

5.2.2 Comparación

La implementación de la aplicación de las reinas se comparó con una implementación realizada con JAFMAS [20] del mismo problema.

Para obtener una medida comparativa de la complejidad y cantidad de código de las dos implementaciones del problema de las reinas se calcularon varias métricas clásicas. En la tabla 5.1 se muestran las siguientes métricas: cantidad de clases y métodos, números de sentencias (NCSS, *non commenting source statements*), métodos por clase, NCSS por clase, NCSS por método y complejidad ciclomática[2] [67]. En la columna *diferencia* se observa que en la mayoría de las métricas, la implementación realizada con Brainstorm/J posee valores menores. Esto indica que la implementación del problema de las reinas realizado con Brainstorm/J posee menos código fuente, lo que podría indicar una mayor reusabilidad y menos esfuerzo de desarrollo para dicha aplicación.

	JAFMAS	Brainstorm/J	Diferencia
Clases	18	21	14.28%
Métodos	92	87	-5.43%
NCSS	625	491	-21.44%
Métodos por clase	5.11	4.14	-18.98%
NCSS por clase	34.72	23.38	-32.66%
NCSS por método	6.79	5.64	-16.93%
CCN promedio por método	1.57	1.51	-3.82%

Tabla 5.1: Métricas de clases, métodos y complejidad ciclomática de las dos implementaciones del problema de las reinas

Para analizar la performance de las aplicaciones se realizaron varias corridas utilizando entre dos y ocho agentes. Se analizó la performance corriendo la aplicación en una sóla máquina y en dos máquinas con la mitad de los agentes en cada una.

La performance de las dos implementaciones resultó ser muy similar (en promedio sólo difieren en $\mp 3\%$). Esto se debe a que los agentes son relativamente simples, por lo que el principal factor que

[2] Es una medida de la complejidad de un algoritmo en un método.

afecta el tiempo de ejecución es el tiempo de envío de los mensajes KQML mediante TCP/IP.

Un aspecto importante a tener en cuenta es que JAFMAS sólo define componentes reusables para construir agentes con capacidades de comunicación, mientras que Brainstorm/J provee componentes adaptables para una variedad de capacidades de agentes.

5.3 Conclusiones

En este capítulo se describió, principalmente, la implementación de dos sistemas multi-agente utilizando el *framework* Brainstorm/J. En ambos casos se pudo observar que la implementación de dichos MAS se realizó con relativa facilidad, debido que el *framework* define la funcionalidad común de los agentes, por lo que sólo fue necesario implementar la funcionalidad particular de cada agente.

CAPÍTULO 6

Especi£cación en Object-Z

En los capítulos previos se describió el *framework* Brainstorm/J y la arquitectura en la cual está basado. En el presente capítulo se formalizan las principales clases del dicho *framework* utilizando el lenguaje de especificación formal Object-Z. En el apéndice C se describe dicho lenguaje.

6.1 JMOP

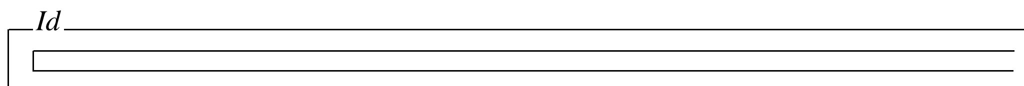
El *framework* JMOP permite asociar meta-objetos a objetos y clases Java. Un meta-objeto puede ser asociado a uno o más objetos o clases en forma simultánea. En el apéndice A se presenta JMOP con mayor detalle.

JMOP depende de varias características presentes en todos los lenguajes orientado a objetos, tales como las clases, objetos, métodos y mensajes. Por lo tanto, la especificación de JMOP debe ser realizada a partir de la definición formal de dichos conceptos.

Para modelar las construcciones básicas del lenguaje Java se utilizaron diferentes clases definidas en Object-Z. Cada una de ellas representa una construcción de Java. Así, por ejemplo, un método Java se modeló mediante una clase *Method* definida en Object-Z. El enfoque adoptado en esta sección es similar a la utilizada en [33].

6.1.1 Identi£cadores

Algunas de las construcciones Java modeladas en la presente sección poseen un identificador. Por ejemplo el nombre de una clase, método o variable. La siguiente clase denota todos los posibles identificadores de Java:



Los identificadores son modelados por una clase para posibilitar la utilización de los operadores de composición de objetos [32] de Object-Z, los cuales no son compatibles con los tipos ni esquemas estándar de Z.

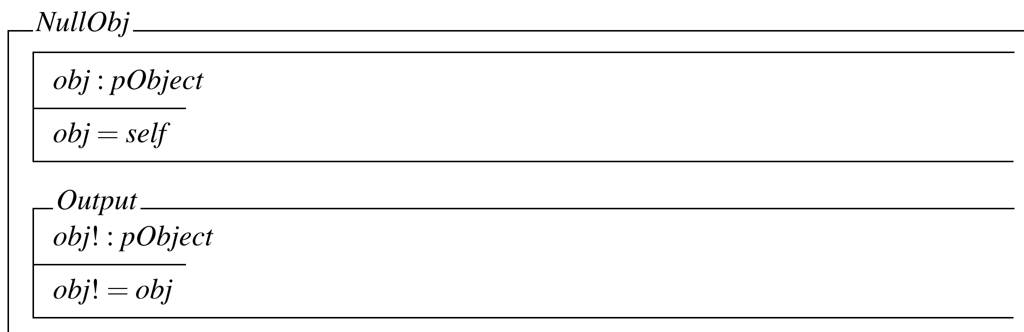
6.1.2 Objetos

En Java, a diferencia de SmallTalk, algunas entidades del lenguaje no son objetos. Dichas entidades son denominados *objetos primitivos*. Los objetos primitivos son instanciados a partir de los tipos primitivos `boolean`, `int`, `long`, `short`, `char`, `byte`, `float` o `double`. El resto de las entidades del lenguaje son objetos pertenecientes a una clase. Dichos objetos poseen un estado interno compuesto por un conjunto de referencias a objetos (atributos), incluyendo objetos primitivos.

En Java, todos los objetos no primitivos son creados mediante la sentencia `new` a partir de una clase. Cuando se crea un objeto, sus atributos son inicializados con un objeto sin valor, notado por `null`. En este modelo, las clases Object-Z *InstObj*, *NullObj*, *BooleanObj*, *IntObj*, etc., definen los tipos de objetos Java: objeto instanciado, nulo, booleano, entero, etc., respectivamente. Los objetos en general, se definen como una unión de clases:

$$pObject \mathrel{\widehat=} InstObj \cup NullObj \cup BooleanObj \cup IntObj \cup LongObj \cup \ldots$$

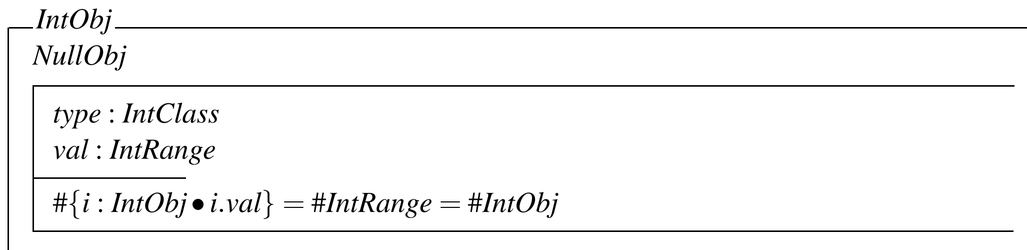
La siguiente clase modela el objeto `null` como una clase, siguiendo el enfoque de [33]:



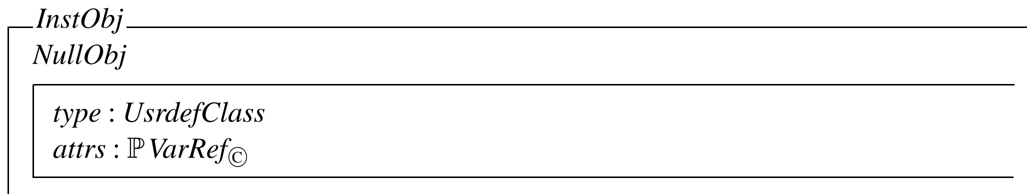
La definición de las clases Object-Z para todos los objetos primitivos es muy similar, por lo que soló se presenta la definición de *IntObj*. El rango de los valores enteros de Java es:

$$IntRange == -2^{31} .. (2^{31} - 1)$$

Cada entero es representado como una instancia de la clase *IntObj*:



La clase *InstObj* define los objetos no primitivos (objetos instanciados) como una subclase de *NullObj*:



El atributo *type* especifica que el tipo de un objeto instanciado es la clase a la cual pertenece. El atributo *attrs* especifica que un objeto instanciado posee un conjunto de variables referencia (*VarRef*). La notación © indica que cada objeto instanciado posee sus propias variables referencias [90]. Formalmente, esto significa que:

$$\forall o_1, o_2 : InstObj \bullet o_1 \neq o_2 \Rightarrow o_1.attrs \cap o_2.attrs = \varnothing$$

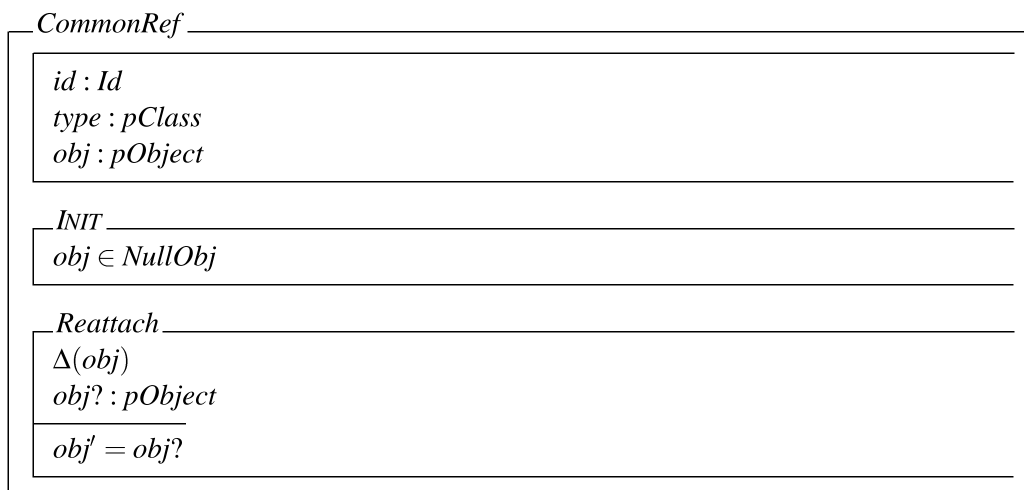
En las secciones 6.1.3 y 6.1.7 se definen las clases *VarRef* y *UsrdefClass*, respectivamente.

6.1.3 Variables referencia

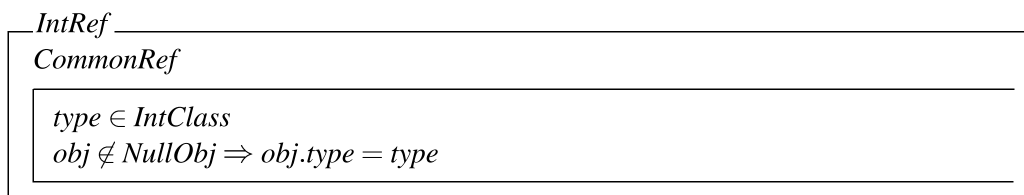
Las variables referencia a objetos pueden referenciar objetos instanciados u objetos primitivos tales como enteros, booleanos, etc. Así, una variable referencia se define como una unión de clases que representan los diferentes tipos de referencias:

$$VarRef \mathrel{\widehat{=}} InstRef \cup BooleanRef \cup IntRef \cup LongRef \cup \ldots$$

Una variable que referencia a un objeto primitivo posee propiedades similares a una variable que referencia a un objeto instanciado. Por ejemplo, ambas tienen un indentificador, un tipo declarado (clase) y un puntero a un objeto. Además, ambas poseen el mismo comportamiento, es decir, obtener el objeto al cual referencian, o reasignar la referencia a otro objeto. Así, la clase *CommonRef* representa la estructura común de todas las referencias. En la definición de esta clase, se considera que las variables son inicializadas con `null`.

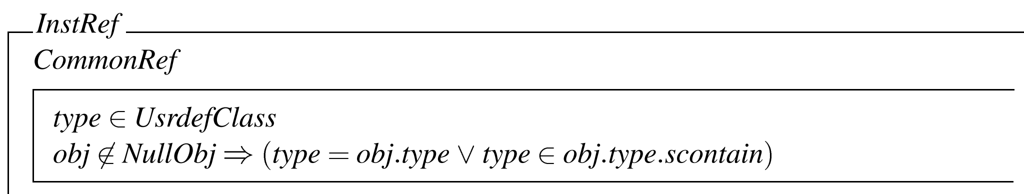


En forma similar a lo que ocurría con los objetos de tipos predefinidos, sólo se presenta la definición de la clase que modela las variables referencia a objetos enteros, ya que las demás son similares:



El invariante de la clase especifica que una variable *IntRef* sólo puede referenciar a objetos cuya clase sea *IntClass*.

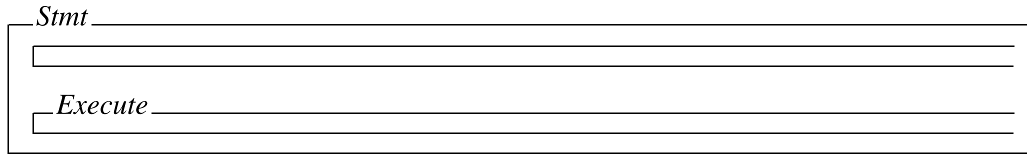
De forma similar, la clase *InstRef* representa una referencia a un objeto instanciado:



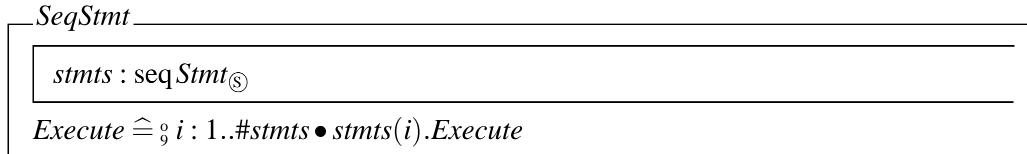
En este caso, el invariante de clase especifica que las variables *InstRef* pueden referenciar a objetos cuya clase es del mismo tipo que la variable referencia. Además, el invariante permite que una variable *InstRef* apunte a un objeto de cualquier subclase de la clase declarada (polimorfismo). El término *obj.type.scontain* nota al conjunto de subclases (directas y transitivas) de la clase del objeto [32].

6.1.4 Sentencias

Los métodos Java están compuestos por una secuencia de sentencias. La clase *Stmt* modela una sentencia, sin especificar sus atributos, ni la semántica de sus operaciones. Esto se debe a que a los fines de la especificación de JMOP no es necesario contar con la definición precisa de sentencia.

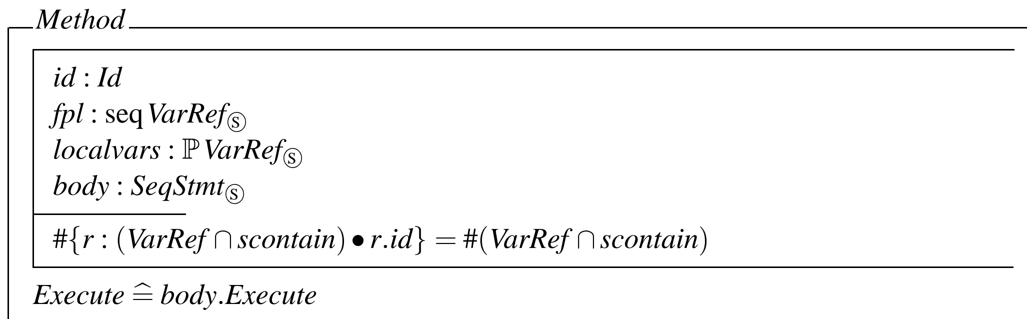


Una secuencia de sentencias consiste de una lista de instancias de *Stmt*:



6.1.5 Métodos

Un método es una operación definida en una clase. El estado interno de un objeto de la clase a la que pertenece el método puede ser modificado por la ejecución de dicho método. Un método consiste de un nombre, una lista de parámetros formales (*fpl*), un conjunto de variables locales y un cuerpo formado por un conjunto de sentencias.

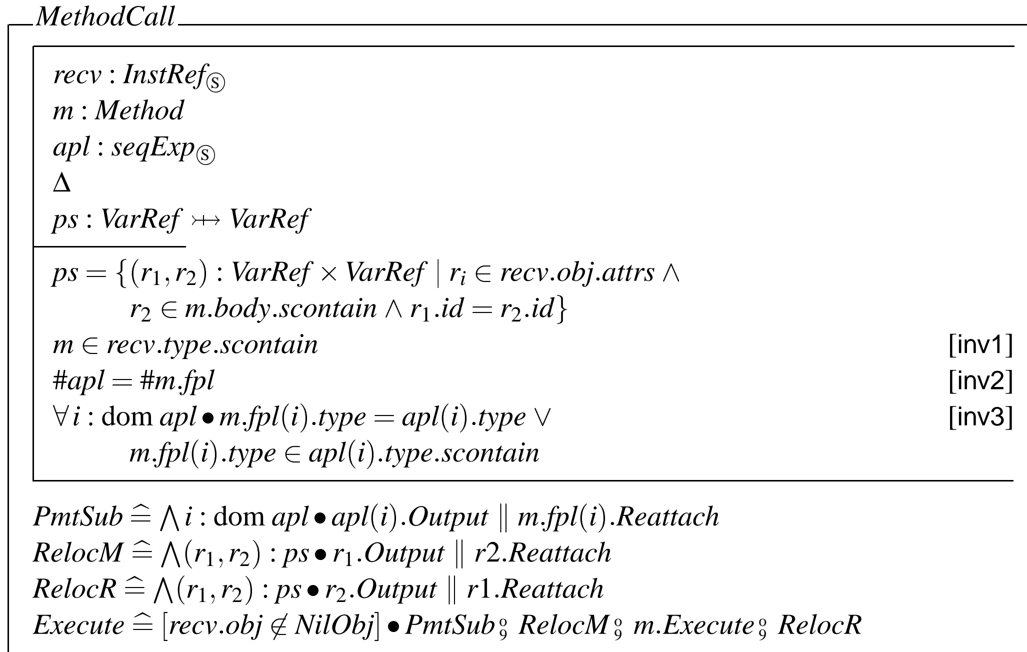


El invariante de clase especifica que en un método, dos variables con el mismo identificador deben ser la misma variable. La ejecución de un método se define como la ejecución de las sentencias del cuerpo del mismo. Además, el símbolo Ⓢ indica que pueden haber dos métodos distintos con el mismo identificador en clases diferentes [32].

6.1.6 Invocaciones a métodos

Una invocación a método consiste de un receptor, un método y una lista de parámetros actuales (*apl*). La lista de parámetros actuales es una secuencia de expresiones. Cada expresión (*Exp*) denota un objeto. Por ejemplo, una variable referencia (`point`), una expresión componente (`point.x`) o una expresión aritmético-lógica (`x+2`).

La clase *MethodCall* especifica una invocación a método de la forma $revc.m(apl_1, apl_2, \ldots, apl_n)$:

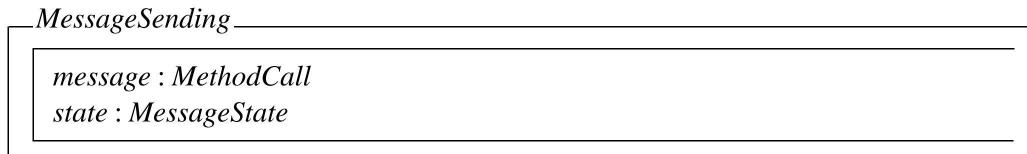


El atributo dependiente *ps* asocia variables referencia del método con las variables del objeto del mismo nombre.

El invariante de clase inv1 especifica que el método invocado debe estar definido en clase del objeto receptor. El invariante inv2 especifica que el número de parámetros actuales de la invocación debe ser igual al número de parámetros formales del método. El invariante inv3 especifica la regla de compatibilidad de tipos para los parámetros formales/actuales.

La operación *PmtSub* vincula las variables referencias de los parámetros actuales con los parámetros formales. La operación *RelocM* asocia las variables referencia del método invocado con los atributos del objeto receptor, para permitir que el método modifique dichos atributos. La operación *RelocR* especifica el proceso inverso a *RelocM*. Finalmente, la operación *Execute* hace uso de esas tres operaciones para ejecutar el método invocado, modificando el estado interno del objeto receptor.

La clase *MessageSending* definida a a continuación representa los mensajes en tiempo de ejecución. Como se puede observar, tienen una variable mostrando el estado del mensaje.



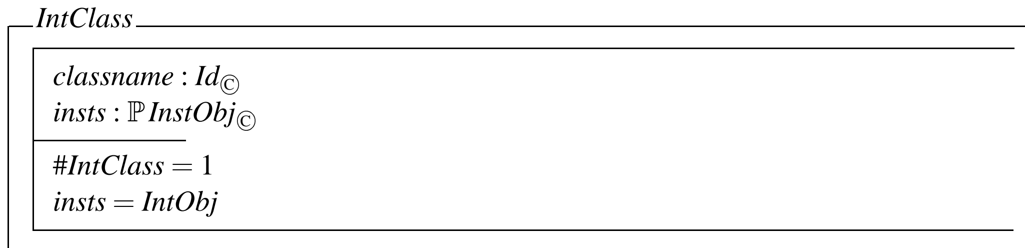
$MessageState = sending \mid analysing \mid executing \mid noActive$

6.1.7 Clases

Una clase es un molde para objetos. Cada objeto de una clase posee un estado que conforma la especificación dada por su clase. El estado de un objeto puede ser accedido o modificado invocando

los métodos de su clase.

En Java, existen un conjunto de tipos primitivos. Dichos tipos pueden ser modelados mediante clases Object-Z. Por ejemplo, el tipo `int` es representado mediante la clase *IntClass*:



El invariante de clase especifica que sólo existe una única clase *IntClass*. El segundo invariante especifica que todos los enteros Java posibles están contenidos en la clase *IntClass*. De manera similar, es posible especificar los tipos `boolean`, `float`, etc.

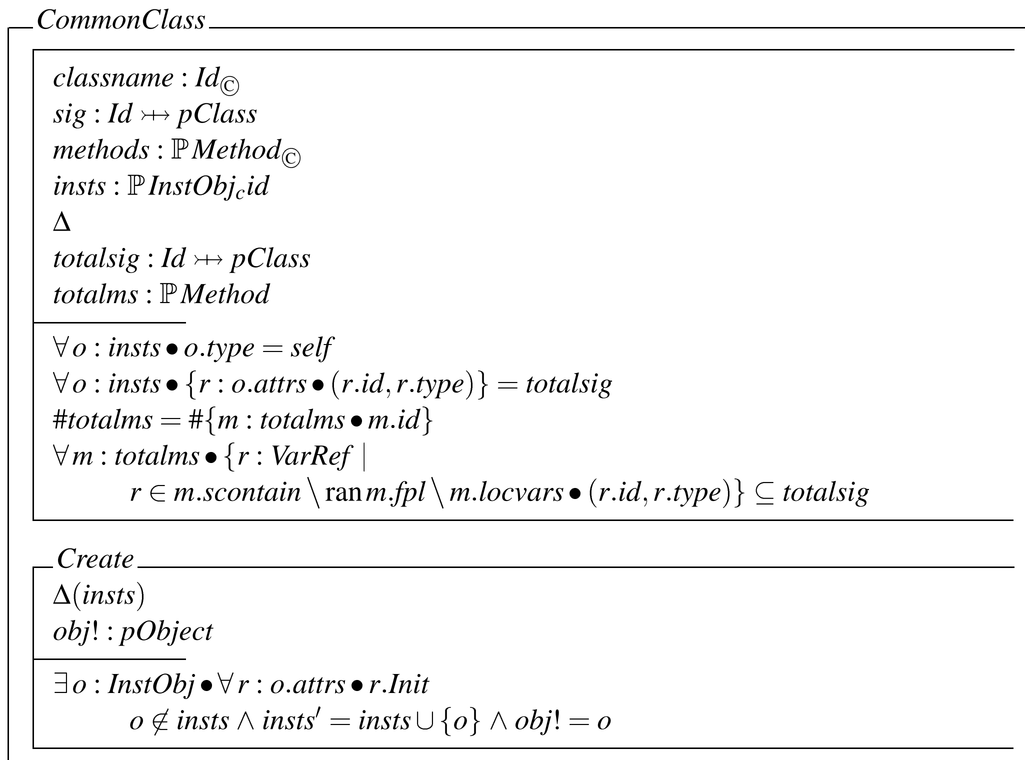
Desde el punto de vista de la herencia, hay dos tipos de clases: las clases *raíz* y las clases *derivadas*:

$UsrdefClass \mathrel{\widehat=} RootClass \cup DerivedClass$

Similarmente, las clases en general, pueden definirse como una unión de clases:

$pClass \mathrel{\widehat=} IntClass \cup BooleanClass \cup \ldots \cup DerivedClass$

Los aspectos comunes de las clases raíz y derivadas están definidas por *CommonClass*:

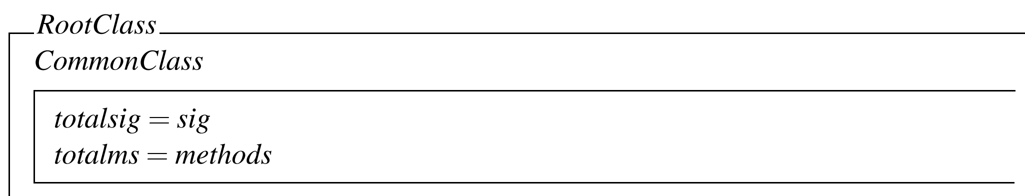


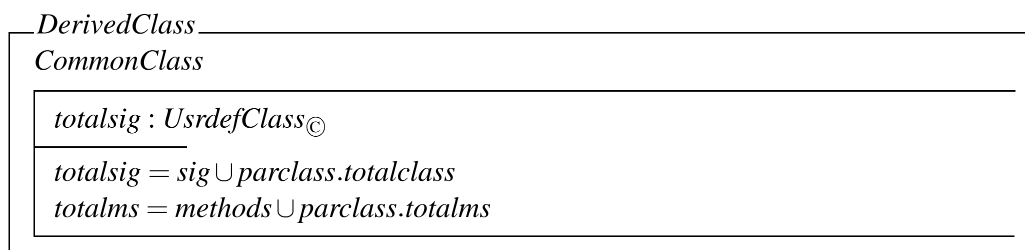
En *CommonClass*, *sig* denota las declaraciones de los atributos declarados en una clase. El atributo dependiente *totalsig* denota todos los atributos accesibles desde una clase, incluyendo a los atributos heredados. En forma similar, *methods* denota los métodos definidos en una clase, mientras que *totalms* denota tanto a los métodos definidos en una clase y a los métodos heredados de las superclases. Todos los objetos de una clase dada son denotados por *insts*.

Los invariantes de clase especifican que todos los objetos deben pertenecer a una clase; todos los objetos deben poseer los mismos atributos que los especificados por su clase. Además, cada método de una clase debe tener un nombre diferente.

Las variables referencias accedidas en un método (excepto los parámetros formales y variables locales) deben estar declaradas en la clase.

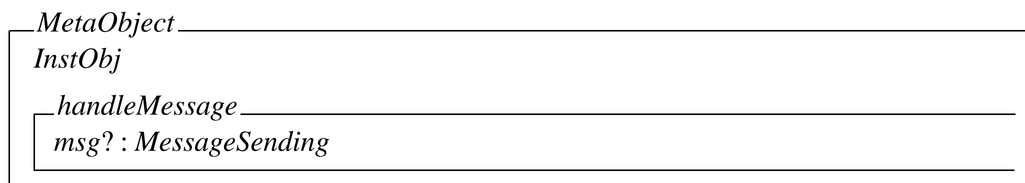
A continuación se especifican las clases *RootClass* y *DerivedClass*:



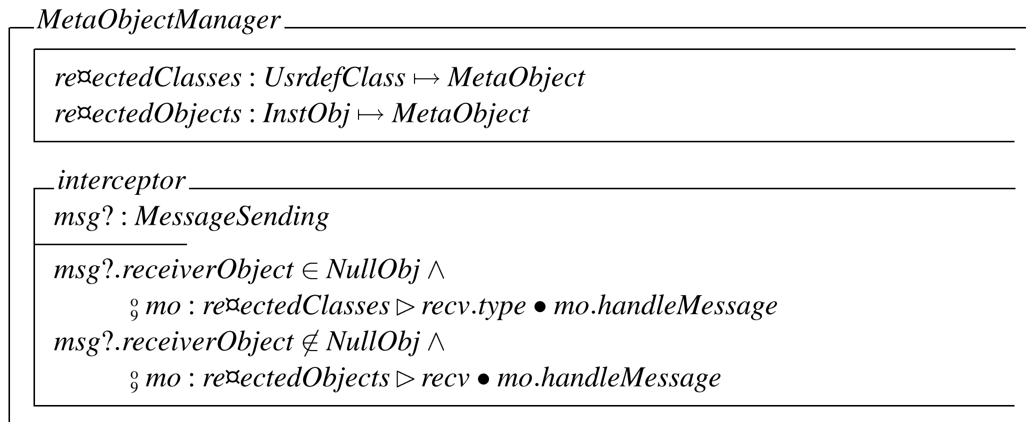


6.1.8 Meta-objetos

Un meta-objeto actúa cuando alguno de los objetos base a los que se encuentra asociado recibe un mensaje, es decir, que la activación de un meta-objeto lleva asociado una instancia de *MethodCall*.



La conexión entre los objetos y meta-objetos es administrada por el *MetaObjectManager*:

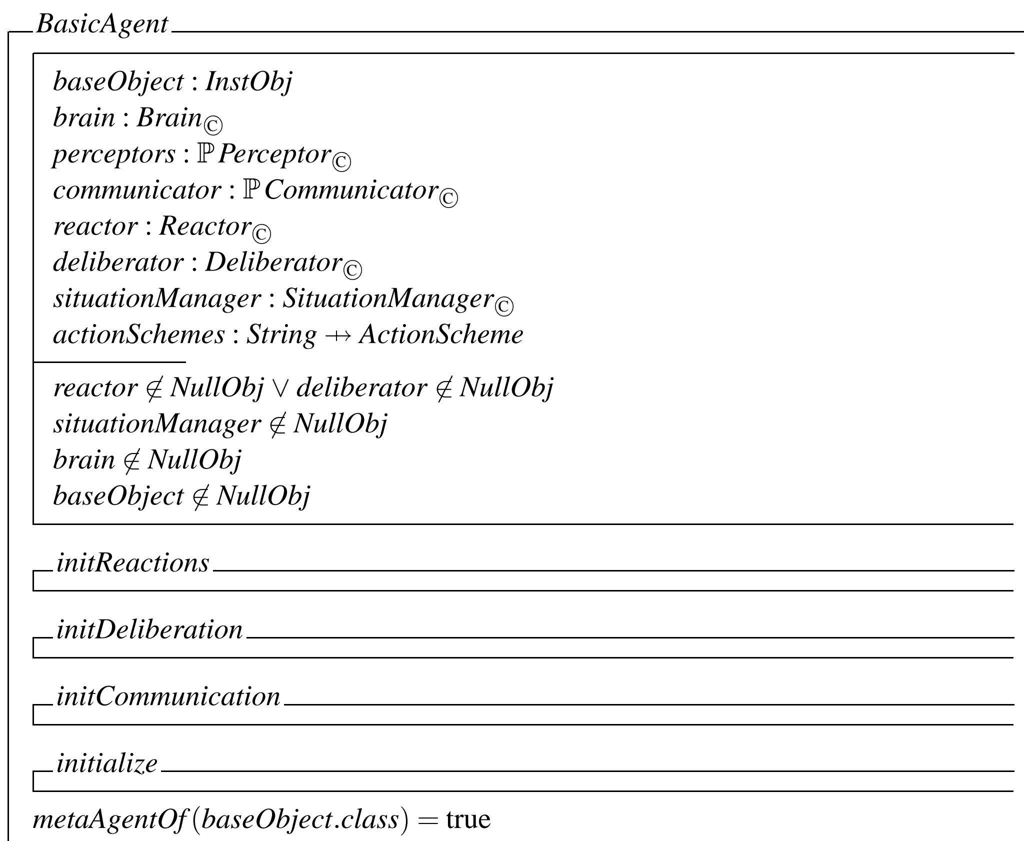


El método interceptor es invocado cada vez que se produce una invocación a un método por el mecanismo de intercepción de mensajes del *framework* JMOP. Básicamente, dicho mecanismo consiste en modificar los métodos de las clases de los objetos a los cuales se desean asociar meta-objetos, para que invoquen el método interceptor del *MetaObjectManager*.

6.2 Brainstorm/J

Un sistema multi-agente construído con Brainstorm/J está formado por agentes y objetos:

Cada agente está formado por un objeto situado en el nivel base y un conjunto de objetos y meta-objetos situados en los meta-niveles responsables de las capacidades de agentes:



La clase *BasicAgent* mantiene referencias a los principales componentes de un agente. Además, define métodos asbtractos para inicializar dichos componentes. Esos métodos son invocados por el meta-objeto *MetaAgent* asociado a la clase de los objetos base de un agente.

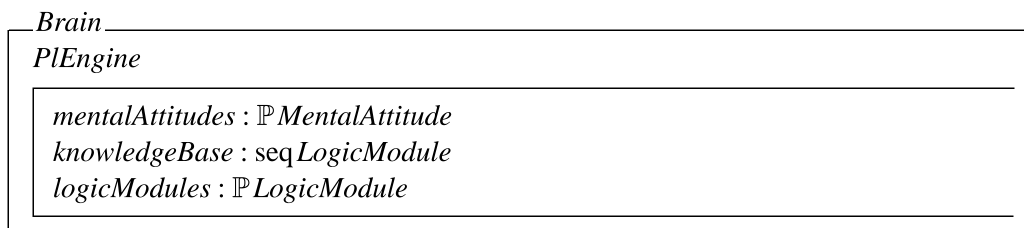
6.2.1 Brain

El componente Brain de Brainstorm/J es el responsable de proveer servicios para representar y manipular actitudes mentales. El conjunto *MentalAttitude* está formado por las actitudes mentales tales como objetivos, intenciones, creencias, etc. El conjunto *Clause* está compuesto por cláusulas lógicas:

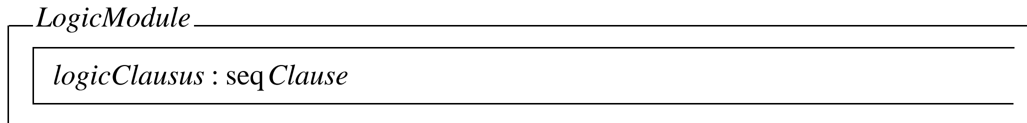
$[MentalAttitude]$

$[Clause]$

La clase *Brain* es subclase de la clase *PlEngine* del lenguaje JavaLog:



Un módulo lógico se define como una secuencia de cláusulas lógicas:

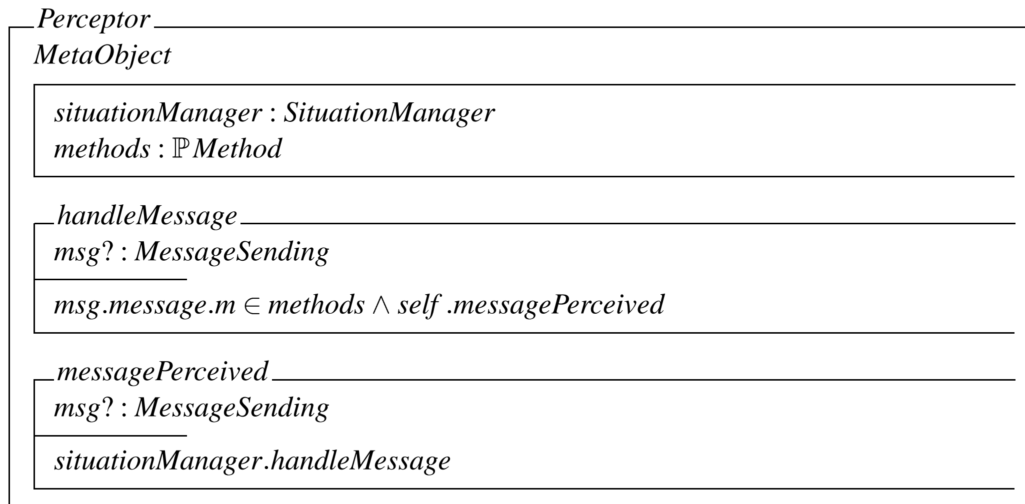


6.2.2 MetaAgent

La clase *MetaAgent* es la responsable de crear los agentes cuando se crea una instancia de la clase del objeto base del agente.

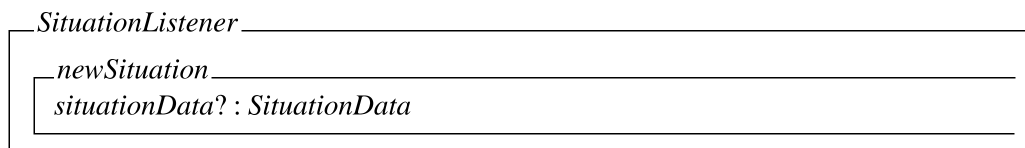
6.2.3 Perceptor

Cada meta-objeto de percepción posee una referencia al administrador de situaciones y un conjunto de métodos de interés. Cuando uno de los objetos asociados a un meta-objeto de percepción recibe un mensaje, el meta-objeto recibe *handleMessage*, conteniendo como argumanto el mensaje recibido por el objeto de nivel base, luego analiza si se trata de un mensaje de interés y, en caso afirmativo, se envía a sí mismo el mensaje *messagePerceived*. Por defecto, este método notifica al administrador de situaciones sobre la percepción de un mensaje.



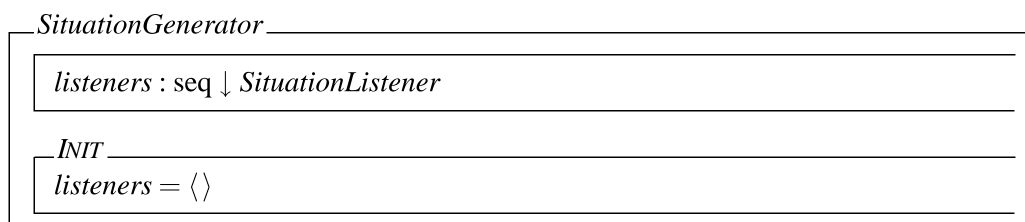
6.2.4 SituationListener

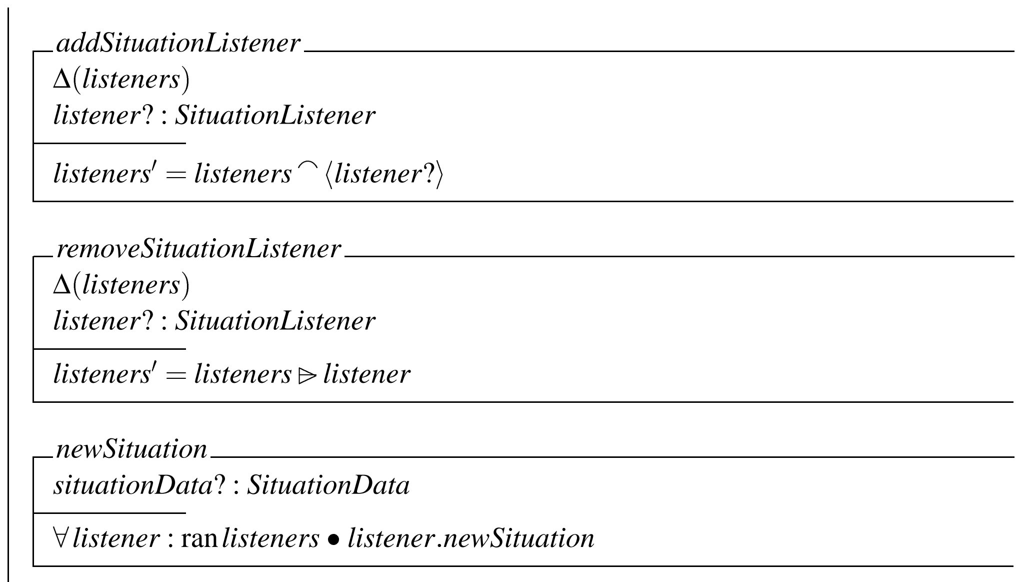
La clase SituationListener especifica en forma abstracta un componente capaz de suscribirse con el administración de situaciones para ser notificado acerca de la ocurrencia de una situación de interés para el agente:



6.2.5 SituationGenerator

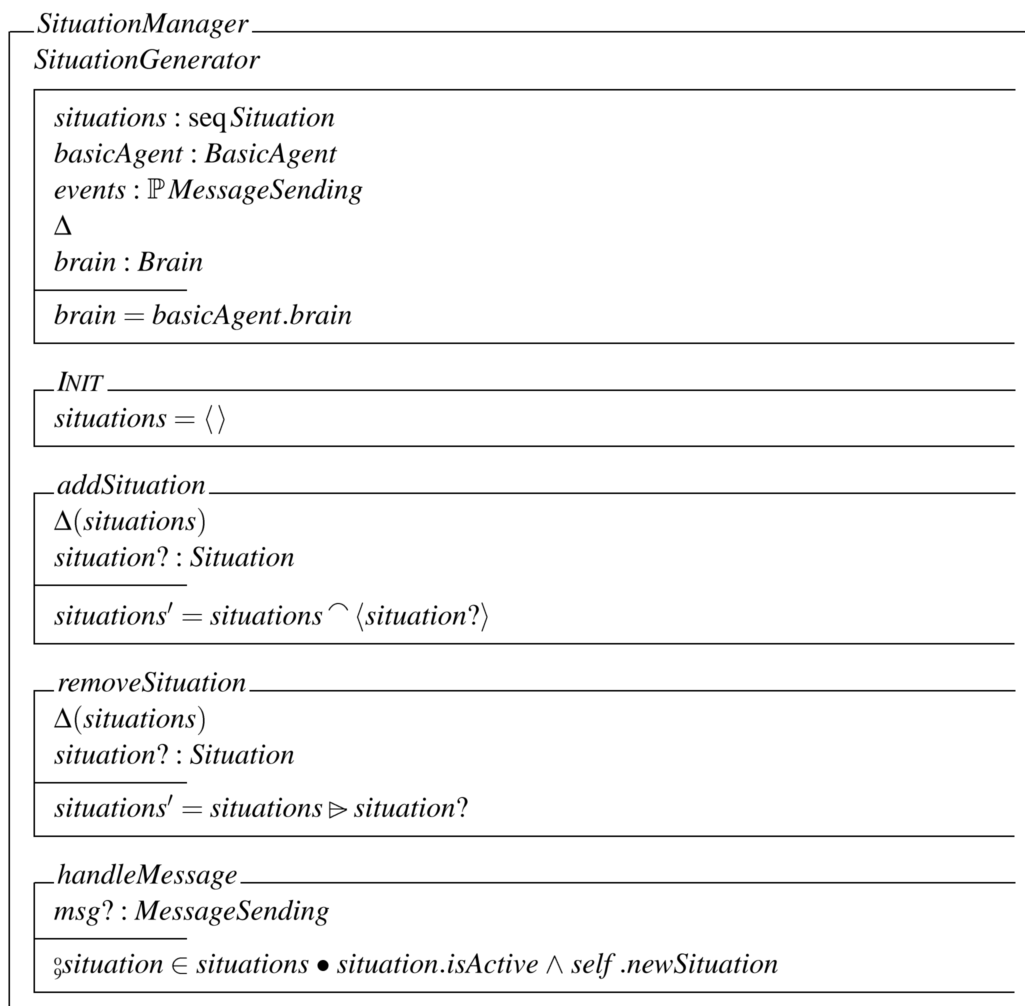
La siguiente clase define el comportamiento de un objeto capaz de notificar a un conjunto de objetos interesados en las situaciones de interés:



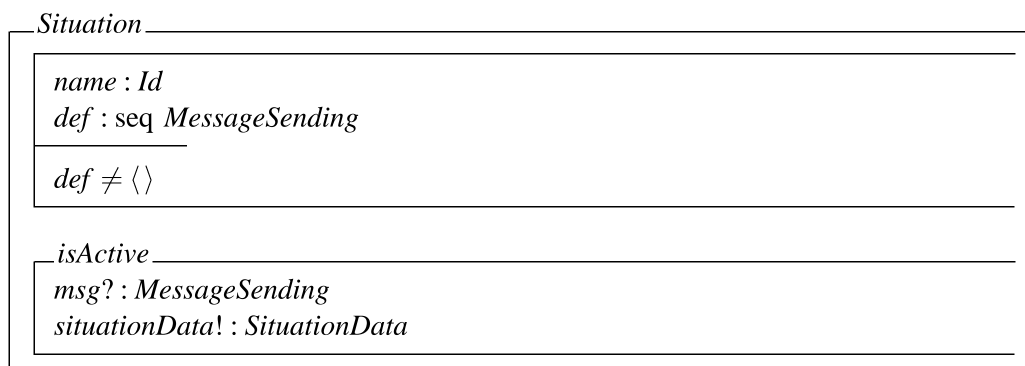


6.2.6 SituationManager

El administrador de situaciones el el responsable de detectar las situaciones de interés y notificar a los objetos interesados en tales sucesos. Básicamente, cuando el agente percibe un mensaje, notifica al administrador de situaciones enviándole *handleMessage*. Luego, el administrador de situaciones busca las situaciones para las cuales la condición de activación es verdadera. Cada vez que detecta una situación se envía a sí mismo el mensaje *newSituation* con la información sobre la situación. Luego, el método *newSituation* definido en *SituationManager* notifica a todos los objetos interesados en las



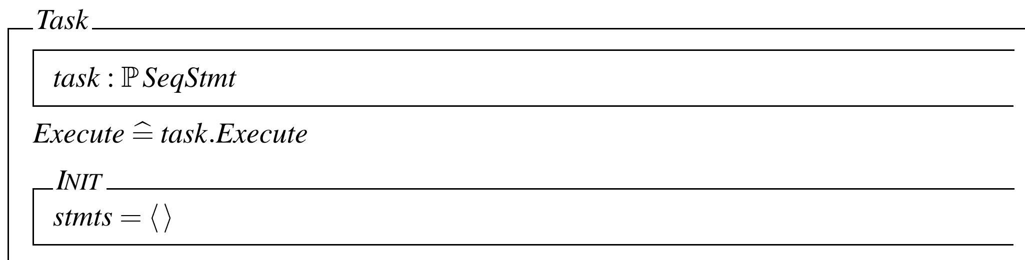
La clase *Situation* mostrada a continuación define una situación de interés:



6.2.7 Task

La siguiente clase define, en forma abstracta, una tarea como un conjunto de sentencias Java. En el *framework* esta clase se utiliza para definir acciones más complejas, tales como planes, reacciones,

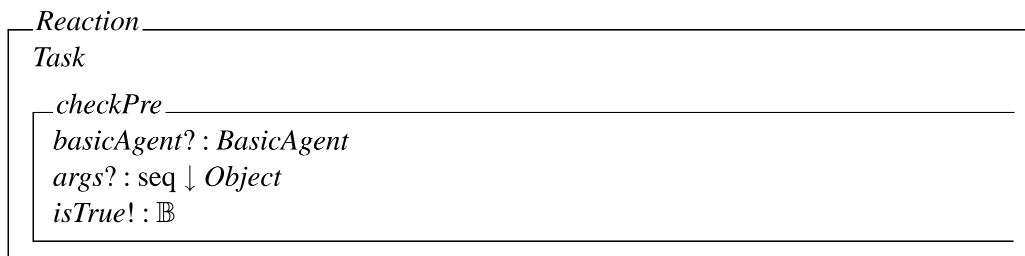
etc.



Las clases *Plan* y *PlanLevel*, extienden *Task* definiendo planes formados por acciones parcialmente ordenadas.

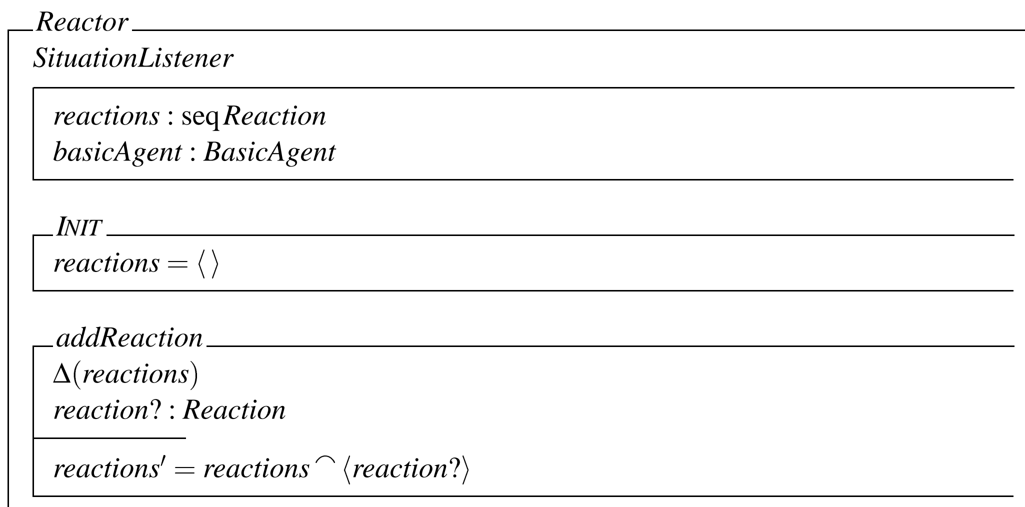
6.2.8 Reaction

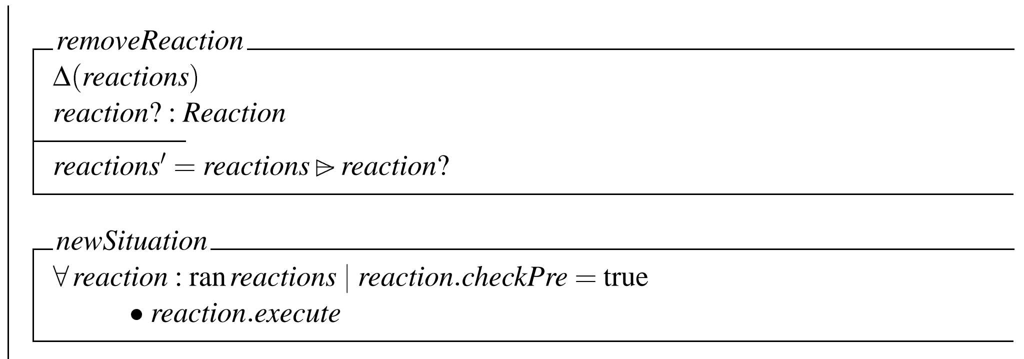
Una reacción se define como una tarea con una condición asociada. Dicha condición debe ser verdadera para que la reacción sea ejecutada.



6.2.9 Reactor

La siguiente clase define el componente reactivo del *framework* como una subclase de *SituationListener*:





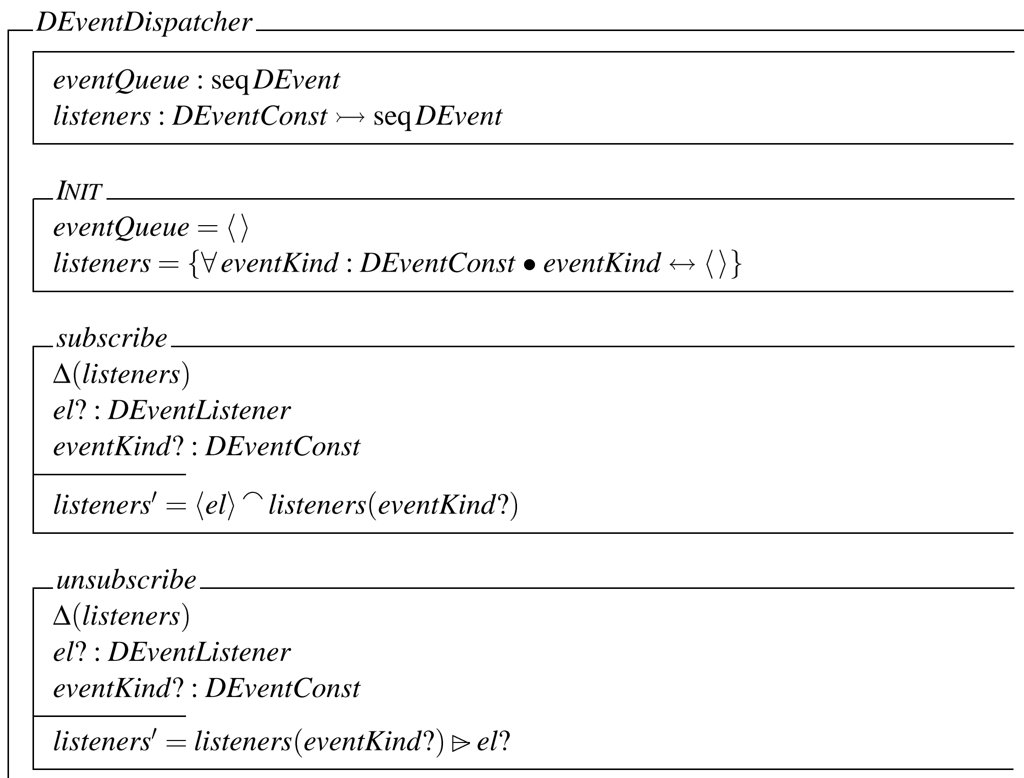
En el método *newSituation* se analiza la existencia de una reacción cuya precondición sea verdadera y se ejecuta.

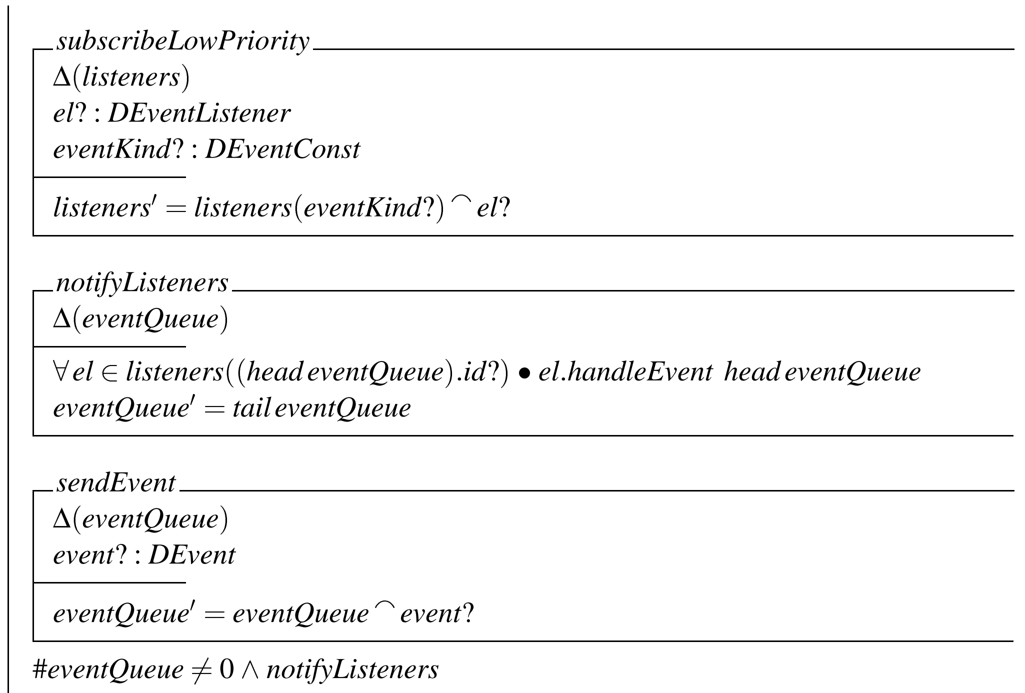
6.2.10 DEventDispatcher

La clase *DEventDispatcher* define un manejador de eventos al cual pueden subscribirse objetos que deseen ser notificados sobre el suceso de un determinado tipo de evento.

El manejador de eventos utiliza un vector de secuencias para almacenar los objetos subscriptos al mismo (variable de instancia *listeners*). En cada una de las secuencias almacena los objetos subscriptos a un determinado tipo de evento.

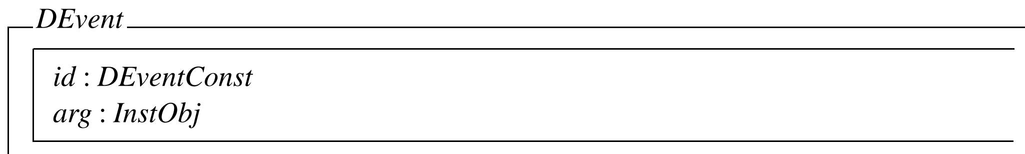
A continuación se define la clase *DEventDispatcher*:





6.2.11 DEvent

La clase *DEvent* define la base de la jerarquía de eventos del componente deliberador:



El tipo *DEventConst* define los posibles eventos del deliberador:

$$DEventConst = PLAN_ADDED \mid PLAN_MODIFIED \mid PLAN_EXECUTED \mid \\ PLAN_FAILED \mid KS_FAILED \mid KS_SUCCESS \mid KS_DESTROYED \mid KS_KILL \mid \ldots$$

En el capítulo 4 se describen los eventos más importantes.

6.2.12 Deliberator

Deliberator

DEventDispatcher
SItuationListener

$topKS : \mathbb{P}\, DelibKS$
$activeKS : \mathbb{P}\, DelibKS$
$plans : \mathbb{P}\, Plan$
$conversations : \mathbb{P}(String \pfun ConvHandler)$
$convHandlerFactory : ConvHandlerFactory$

INIT

handleMessage
$msg? : MessageSending$

newSituation

addPlan

record
$kqmlMsg? : KQMLmessage$

chooseGoals
$situationData?$

addGoal

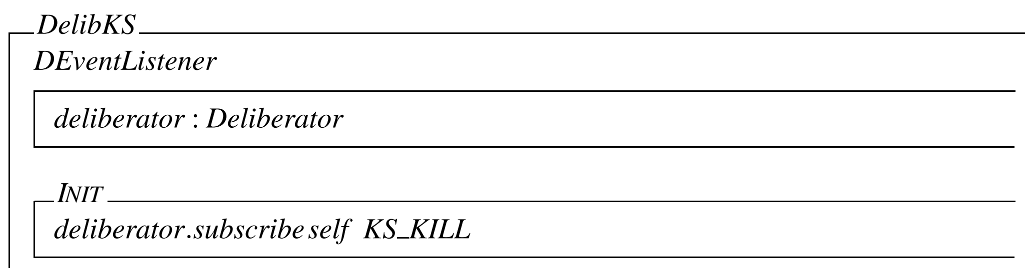
removeGoal

addBelief

removeBelief

6.2.13 DelibKS

La clase *DelibKS* es el tope de la jerarquía de fuentes de conocimiento:

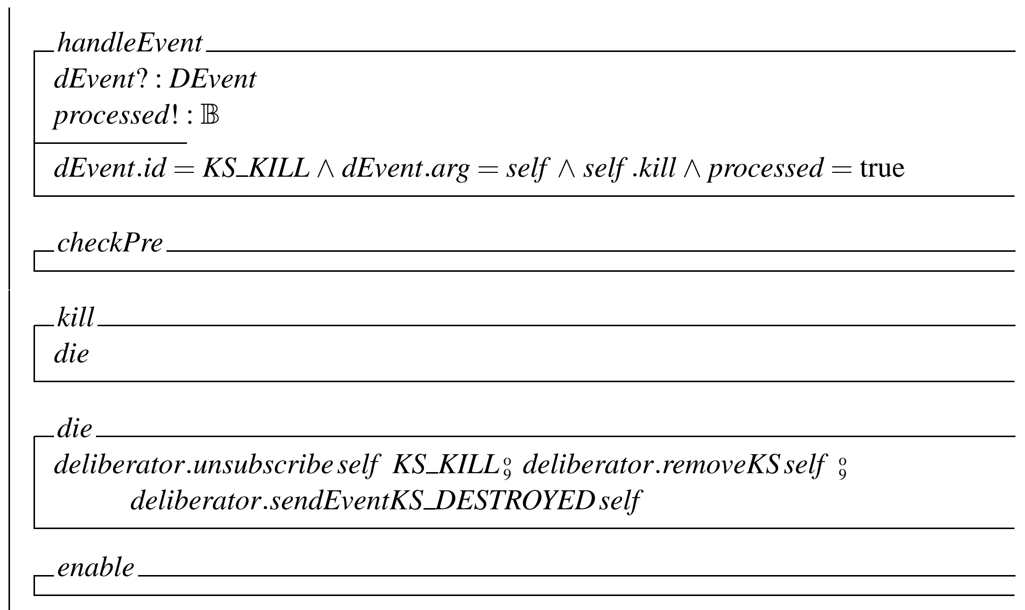


DelibKS

DEventListener

$deliberator : Deliberator$

INIT
$deliberator.subscribe\, self\ \ KS_KILL$


handleEvent
dEvent? : DEvent
processed! : 𝔹
dEvent.id = KS_KILL ∧ dEvent.arg = self ∧ self.kill ∧ processed = true
checkPre
kill
die
die
deliberator.unsubscribe self KS_KILL ⨾ deliberator.removeKS self ⨾
deliberator.sendEvent KS_DESTROYED self
enable

CAPÍTULO 7

Conclusiones y trabajos futuros

En los capítulos previos se presentó un *framework* orientado a objetos para construir agentes. El *framework* está basado en la arquitectura Brainstorm, la cual prescribe agentes capaces de representar y manipular estados mentales, percibir, reaccionar, deliberar, comunicarse y transportarse entre las computadoras de una red.

La arquitectura Brainstorm fue materializada mediante el *framework* Brainstorm/J. Para realizar dicha materialización se utilizó un lenguaje multi-paradigma, denominado JavaLog y se desarrolló un *framework* para soportar meta-objetos en Java.

El lenguaje JavaLog integra la programación orientada a objetos y la programación lógica. De esta forma, es posible representar y manipular los estados mentales de los agentes mediante cláusulas lógicas en un entorno orientado a objetos.

El soporte de meta-objetos desarrollado permite asociar capacidades de agentes a objetos simples en forma transparente, es decir, sin modificar los métodos de las clases. Esto resultó de gran utilidad para implementar las capacidades de percepción de los agentes, ya que permite a los agentes observar cualquier objeto del sistema en forma transparente y no intrusiva.

La materialización básica de Brainstorm fue extendida para soportar agentes móviles y comunicaciones entre agentes físicamente distribuidos. De esta forma, el *framework* permite construir agentes que migran entre diversos sitios de una red y se comunican utilizando protocolos de mensajes con varios mecanismos de transporte (TCP/IP, e-Mail, FTP, etc.).

Brainstorm/J define la funcionalidad común de los agentes en forma reusable y extensible, evitando al programador construir los agentes desde cero, y a la vez, permitiéndole que extienda las capacidades del *framework* según las necesidades de cada aplicación. Esto resulta de gran importancia si se considera la gran variedad capacidades y tipos de agentes, sumado a las innumerables aplicaciones de los mismos.

A continuación se presentan las principales contribuciones del trabajo, las limitaciones del mismo y las posibles extensiones o trabajos futuros.

7.1 Contribuciones

El trabajo presenta varias contribuciones, las cuales se resumen a continuación:

- Desarrollo de un *framework* para meta-objetos para el lenguaje Java que permite re¤ejar aplicaciones sin modificar su código fuente y sin requerir de una máquina virtual extendida.
- Desarrollo de un *framework* basado en la arquitectura Brainstorm que provee una infraestructura adaptable y extensible para desarrollar agentes con diversas capacidades en diversos dominios de aplicación.
  - Definición e implementación de mecanismos que permiten desarrollar agentes que adquieren y actualizan sus creencias sobre el medio ambiente mediante sus capacidades de percepción, revisando y eliminando creencias incoherentes, y actuando en función de ellas.
  - Diseño e implementación de mecanismos que permiten desarrollar agentes que realizan varias actividades en forma concurrente (percepción, comunicación, deliberación, etc).
  - Diseño e implementación de mecanismos que permiten desarrollar agentes que razonan en forma concurrente sobre cómo alcanzar sus objetivos, utilizando diferentes mecanismos deliberativos.

7.2 Limitaciones

El *framework* fue implementado utilizando el lenguaje multi-paradigma JavaLog. Esto facilita la tarea del programador, por cuando le permite manipular cláusulas lógicas Prolog. Sin embargo, la excesiva utilización de dicho lenguaje por parte del programador puede afectar en forma negativa la performance de los agentes desarrollados con Brainstorm/J, debido a que JavaLog utiliza un intérprete de Prolog implementado en Java.

Por otro lado, el *framework* de meta-objetos también puede impactar negativamente en la performance de los agentes, principalmente si se utilizan muchos meta-objetos de percepción.

El componente deliberador de Brainstorm/J hace uso extensivo de *threads*. En Java, los *threads* dependen del soporte ofrecido por el sistema operativo, lo que puede ocasionar leves diferencias en la ejecución de un mismo MAS en diferentes plataformas o en diferentes máquinas virtuales Java.

Otra limitación relacionada con ésta última es la referida al soporte de movilidad del *framework*. En particular, las limitaciones propias de la máquina virtual de Java por las cuales no es posible acceder al estado de un *thread* en ejecución, impiden el poder salvar y restaurar el estado de los *threads* en forma automática. Por lo tanto, esto es responsabilidad de programador. En la práctica esto hace que el programador no utilice movilidad con agentes que poseen varios *threads*, limitando su utilización a agentes simples.

7.3 Trabajos futuros

El *framework* Brainstorm/J puede ser utilizado para desarrollar agentes de diversos tipos en variados dominios de aplicación. Por tales razones, existen varias extensiones posibles al mismo.

A continuación se describen posibles trabajos futuros sobre Brainstorm/J:

- Soporte para aprendizaje

  Para integrar capacidades de aprendizaje en el *framework* será necesario estudiar en profundidad la utilización de dicha capacidad en dos componentes:

  - *Deliberator*: habrá que analizar los puntos del deliberador susceptibles de ser interceptados por un meta-objeto de aprendizaje. En tal sentido, existen numerosas posibilidad. Por ejemplo, si sólo se interceptan las fuentes de conocimiento que utilizan *planning*, habría que analizar las diferentes estrategias para almacenar y recuperar planes para resolver situaciones similares en el futuro. Además, la gran variedad de fuentes de conocimiento y maneras en que interactúan sería otro punto de estudio que podría dar resultados positivos para aplicar técnicas de aprendizaje.
  - *Communicator*: un componente de aprendizaje observando y modificando las interacciones multi-agentes podría ser utilizado para soportar aprendizaje multi-agente.

- Biblioteca de componentes concretos

  En la mayoría de los casos, la implementación de agentes con Brainstorm/J se realiza especializando clases definidas por el mismo. En algunos casos, no es necesario especializar, sino que sólo hay que instanciar clases y componerlas, lo que resulta mucho más fácil para el programador debido a que no debe conocer la estructura interna del *framework* [38].

  Por tal motivo, sería bueno contar con un conjunto de componentes concretos basados en Brainstorm/J que permitan que el programador instancie y componga componentes de software para construir agentes.

  Sin embargo, el diseño e implementación de componentes genéricos y reusables puede ser una tarea sumamente difícil, dado el amplio dominio de aplicación de los agentes, sumado a los diferentes y variables requerimientos que esto implica.

- Coordinación

  El *framework* no provee soporte para agentes que razonan y actúan en función de lo que otros agentes realizan. En tal sentido, el *framework* podría proveer mecanismos de coordinación simples basados en estilos de interacción tales como proveedor/consumidor, *contract-net* o amo/esclavo.

  Por otra parte, sería posible que los agentes razonen acerca de las acciones de los otros agentes, por ejemplo, basándose en la teoría de los *speech-acts* y sus formalizaciones.

- Herramienta para instanciar Brainstorm/J

  Un aspecto que puede dificultar la construcción de agentes con Brainstorm/J es su complejidad en cuanto al gran número de clases y métodos que lo componen. Para solucionar tal deficiencia, se está desarrollando una herramienta gráfica que asiste al programador en la instanciación del

framework, reduciendo la cantidad de código necesaria para programar agentes y, en consecuencia, el esfuerzo del programador.

APÉNDICE A

Meta-Objetos en Java

Para desarrollar Brainstorm/J se utilizó un *framework* para meta-objetos basado LuthierMOPS [18] llamado JMOP. Dicho *framework* permite asociar meta-objetos a objetos[1], clases o métodos. Un meta-objeto puede ser asociado a uno o más objetos de la misma o de distintas clases. Además, un objeto puede tener cero o más meta-objetos asociados.

A diferencia de otras extensiones Java para meta-objetos tales como OpenJava [24] o JavaAssist [23], JMOP posee la particularidad de permitir asociar meta-objetos a objetos sin requerir de modificaciones en el código fuente de las clases de dichos objetos. Esto permite utilizar JMOP con paquetes Java en los que no es posible acceder al código fuente.

Por otro lado, a diferencia de metaXa [51], JMOP no requiere de una máquina virtual Java extendida, por lo que puede ser ejecutado en cualquier plataforma y en cualquier máquina virtual Java.

En la figura A.1 se muestra un diagrama de clases del *framework*. Las principales clases de JMOP son: `ReflectedMessage`, `MetaObjectManager`, `MetaObject` y `ReflectedClass`.

Cuando un objeto re¤ejado recibe un mensaje, es interceptado por un mecanismo de re¤exión. Un mensaje interceptado es representado por una instancia de la clase `ReflectedMessage`. Dicha clase contiene, entre otras cosas, información acerca del método asociado al mensaje, objeto emisor y receptor, parámetros del mensaje y *thread* desde el que se envió dicho mensaje.

Los mensajes re¤ejados son enviados al `MetaObjectManager` asociado a la clase del objeto re¤ejado mediante el mensaje `interceptor`. La clase `MetaObjectManager` es responsable por determinar qué meta-objetos deben ser activados cuando un objeto base recibe un mensaje. Para determinar qué meta-objetos activar utiliza el método `findMetaObjectsFor`, luego los activa invocando al método `activateMetaObjects`. Los meta-objetos activados reciben el mensaje `handleMessage` con una instancia de la clase `ReflectedMessage` como parámetro. Un meta-objeto puede, opcionalmente, ejecutar el método original enviando el mensaje `send` al mensaje re¤ejado.

[1] Esto incluye a los meta-objetos, es decir, que un meta-objeto puede tener asociado otro meta-objeto.

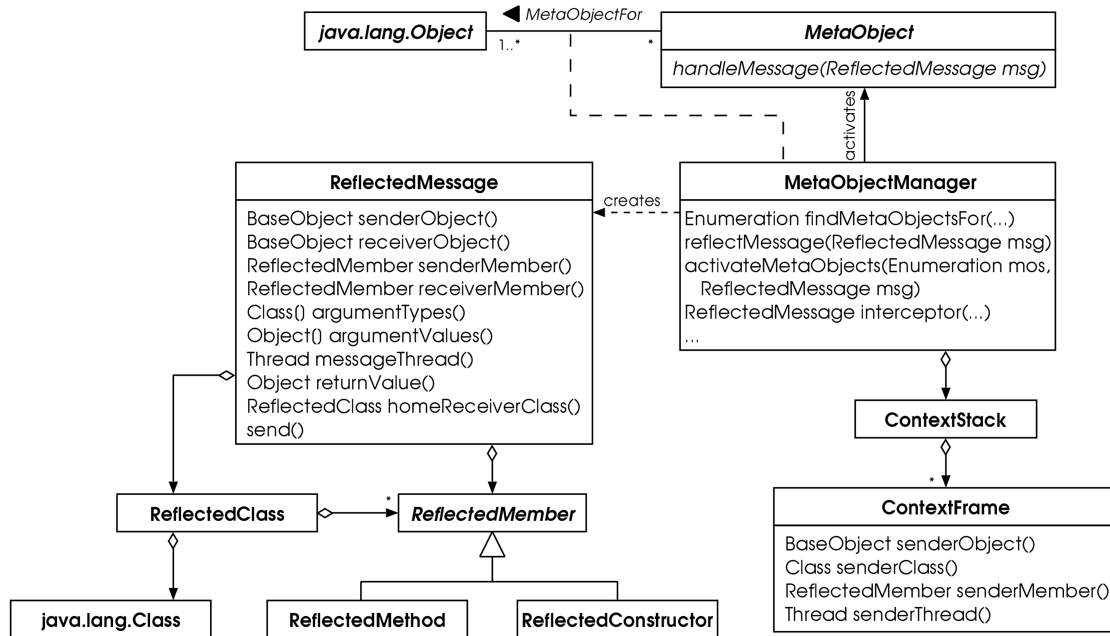


Figura A.1: Principales de clases del *framework* JMOP

La primera vez que se asocia un meta-objeto a objetos de una clase, se crea una instancia de la clase `ReflectedClass`. Dicha clase contiene información sobre una clase re¤ejada y los métodos re¤ejados, representados por las clases `ReflectedMethod` y `ReflectedConstructor`.

La clase `MetaObject` define el método abstracto `handleMessage`, el cual debe ser implementado por las subclases para procesar los mensajes re¤ejados.

Java, a diferencia de lenguajes como SmallTalk o CLOS, no posee mecanismos para modificar los métodos de las clases en tiempo de ejecución, por lo tanto no es posible insertar las invocaciones al meta-nivel de manera directa.

Para poder interceptar los mensajes de los objetos re¤ejados existen dos opciones[2]. La primera consiste en modificar el código fuente de los métodos interceptados para invocar al meta-nivel. Sin embargo, para realizar esto es necesario contar con el código fuente de las clases, lo que, en muchos casos no es posible.

La segunda opción consiste en modificar el código objeto (*bytecode*) de las clases para invocar al meta-nivel. Esto puede realizarse modificando las clases en forma *off-line* u *on-line*. En el primer caso, deben modificarse todas las clases de los objetos que se deseen re¤ejar en forma previa a ejecutar una aplicación. En el segundo caso, esto no es necesario, justamente por el hecho de que las clases se modifican en tiempo de ejecución.

En JMOP, las clases se re¤ejan en forma *on-line*, dando mayor libertad al programador acerca de cuándo re¤ejar las clases. Esto resulta útil con aplicaciones tales como depuradores, herramientas de visualización o, incluso, Brainstorm/J. En general, esto permite utilizar JMOP en aplicaciones en las cuales no el posible conocer las clases de los objetos a re¤ejar en forma previa a la ejecución.

[2]Además, sería posible modi£car la máquina virtual Java para soportar meta-objetos. Sin embargo, esto tendría problemas de portabilidad.

La máquina virtual Java carga el *bytecode* de las clases por demanda utilizando el `ClassLoader`. Esto significa que la primera vez que se crea una instancia de una clase o se referencia una clase, se invoca el método `loadClass` del `ClassLoader` actual. Este método obtiene el *bytecode* de la clase a ser cargada del sistema de archivos o de la red. Una vez realizado esto, se cargan todas las clases referenciadas por la clase cargada y se repite este proceso para cada una de ellas.

Para modificar las clases en tiempo de ejecución se creó una subclase de `ClassLoader` llamada `JMopClassLoader` (figura A.2). Dicha clase utiliza una instancia de `Reflector`, la cual es responsable de modificar el *bytecode* de las clases, introduciendo invocaciones al meta-nivel en cada uno de los métodos interceptados. La clase `Reflector` utiliza el paquete JavaClass [28] para manipular el formato *bytecode* de las clases Java en forma sencilla, independizándose de los detalles de bajo nivel de la máquina virtual Java.

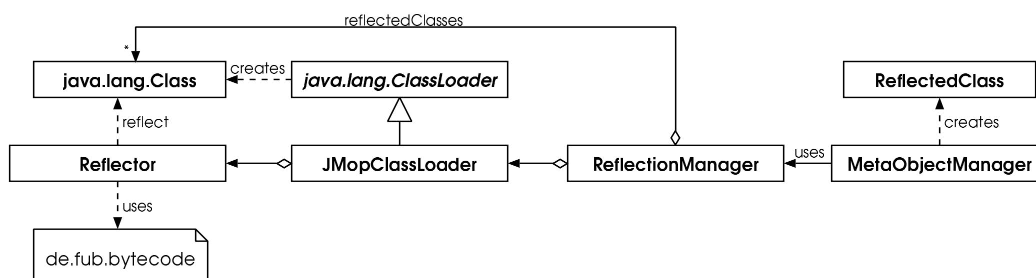


Figura A.2: Modi£cación de las clases en tiempo de ejecución

APÉNDICE B

JavaLog

JavaLog [5] es un lenguaje multi-paradigma que integra la programación orientada a objetos y la programación lógica a través de los lenguajes Java y Prolog.

El lenguaje está basado, principalmente, en el hecho de que un agente puede ser considerado un objeto [87], donde sus métodos representan las capacidades de acción del agentes y las variables de instancia representan los estados mentales.

En las siguientes secciones se describe brevemente JavaLog.

B.1 Módulos

El principal componente manipulable por medio del lenguaje JavaLog es el *módulo*. Básicamente, el lenguaje considera cada método Java como un módulo que encapsula comportamiento, los objetos son considerados módulos que encapsulan estado, mientras que los elementos manipulados por el paradigma lógico encapsulan estados mentales.

En particular, este último tipo de módulo recibe el nombre de *módulo lógico* [73]. Un módulo lógico es definido por un conjunto de cláusulas de Horn con las operaciones de unión y redefinición [14].

La figura B.1 muestra un esquema de las diferentes formas de utilización de módulos lógicos. Los módulos lógicos se representan con círculos grises. Las clases definen parte de sus métodos mediante módulos lógicos. Los objetos definen módulos privados en variables de instancia. Un objeto simple que utiliza módulos lógicos debe estar relacionado con un objeto *brain* que representa un intérprete lógico.

Cada objeto que utiliza módulos lógicos es una instancia de una clase que puede definir parte de sus métodos en Java y parte en Prolog. Así, la definición de los métodos de una clase puede estar compuesta por varios módulos lógicos. Una instancia de una clase de este tipo presenta una composición de módulos lógicos definidos en métodos y módulos lógicos referenciados por variables de instancia.

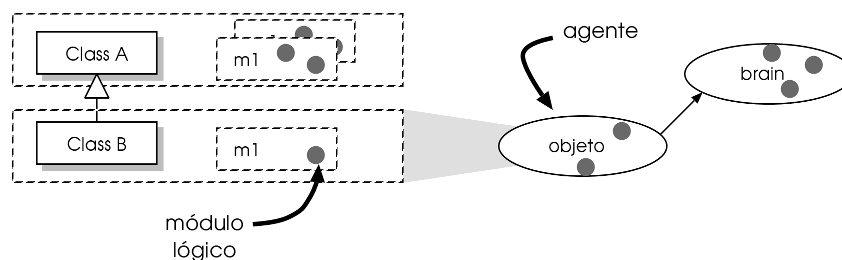


Figura B.1: Módulos lógicos en clases, métodos y objetos

Para realizar esta composición de módulos lógicos y objetos, JavaLog provee un intérprete Prolog basado en el concepto de módulo y una extensión Java que permite utilizar módulos lógicos y cláusulas en Java. En las secciones siguientes se describen las distintas formas de integrar objetos y lógica con JavaLog.

B.1.1 Módulos lógicos en variables de instancia y en métodos

Los módulos lógicos pueden ser utilizados en variables o en métodos Java. Así, un agente puede ser diseñado como un objeto con conocimiento lógico privado representado en módulos lógicos referenciados por variables de instancia. Las capacidades de acción del agente son representadas por métodos que pueden acceder a los módulos lógicos.

La figura B.2 presenta un ejemplo de un agente que utiliza módulos lógicos en las variables de instancia `context1`, `context2` y `context3`. El agente representa un asistente personal. Los módulos lógicos referenciados por las variables de instancia representan diferentes preferencias del usuario sobre pedidos de citas.

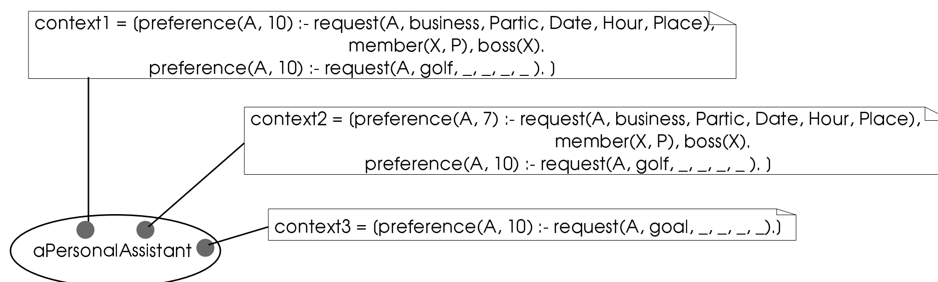


Figura B.2: Módulos lógicos privados de un agente

Cuando un agente utiliza varios módulos lógicos, esto no significa que todas las cláusulas de esos módulos lógicos estén disponibles en las siguientes consultas. Los únicos módulos lógicos cuyas cláusulas forman parte del conocimiento del agente son aquellos que fueron enviados al objeto *brain*.

A continuación se ejemplifica la utilización de módulos lógicos en un método:

```
//definición de variables locales
A1 = ...;
...
userPreferences(context1);
```

```
...
{{ preference(A,5) :- ...
   ?- preference(#A1#,X). }}
return brain.goal().stateOf("X");
```

En éste método se utilizan las preferencias del usuario almacenadas en el módulo referenciado por `context1`. En forma similar a Java, los métodos JavaLog poseen variables locales y sentencias Java. Adicionalmente, pueden poseer módulos lógicos.

En el método, la invocación a `userPreferences` activa las cláusulas del módulo lógico enviado como argumento. Luego, se define un módulo lógico utilizando la sintaxis {{<cláusula | consulta>+}}.

En la consulta `?- preference(#A1#,X)` se utiliza la variable local Java `A1`. Luego, se obtiene el valor de la variable `X` utilizada en la consulta que contiene un número indicando la preferencia del usuario respecto de una reunión.

Los objetos simples Java no tiene la posibilidad de utilizar conocimiento en formato lógico. Para lograrlo, se debe asociar un intérprete de un lenguaje lógico (por ejemplo *brain*) a los objetos con capacidad de usar módulos lógicos. De esta forma, un objeto puede adquirir la capacidad de representar conocimiento en variables y en métodos.

B.1.2 Objectos como cláusulas

Los objetos Java pueden ser convertidos en cláusulas manipulables por el intérprete lógico. Por ejemplo, si un objeto de la clase `Persona` se convierte en cláusula, se obtiene:

```
persona(this, 'Ann', 33, ingeniero).
```

El nombre de la cláusula es igual al nombre de la clase a la que pertenece el objeto convertido en cláusula. El primer argumento es el objeto; los otros argumentos son los contenidos de las variables de instancia del objeto.

B.1.3 Cláusulas con objetos

Cualquier cláusula puede usar objetos como argumentos. Esto permite utilizar objetos, junto con su comportamiento asociado dentro de cláusulas lógicas. JavaLog permite enviar mensajes a objetos desde código lógico mediante la cláusula `send`. Así, por ejemplo, `send(unaPersona, edad, [], A)` en el cuerpo de una cláusula envía el mensaje `age` al objeto `unaPersona` sin ningún argumento e instancia la variable `A` con el múmero retornado por ese mensaje.

B.1.4 Composición de módulos

Los módulos referenciados por variables pueden ser combinados directamente utilizando dos métodos predefinidos: unión y redefinición. Además, los módulos lógicos definidos en métodos pueden ser combinados por medio de la herencia.

Por ejemplo, el asistente personal descripto previamente tiene tres variables de instancia: `context1`, `context2` y `context3` para registrar diferentes formas de evaluar propuestas de reuniones. De esta

forma, frente a cierta propuesta se podrían utilizar los módulos lógicos referenciados por las variables, o combinaciones de los mismos.

Los siguientes operadores permiten combinar módulos lógicos:

- redefinición: dados dos módulos lógicos a y b, $a.reWrite(b)$ define un módulo lógico que contiene todas las cláusulas definidas en b y las cláusulas de a tal que no existe una cláusula en b con el mismo nombre de cabeza.
- unión: dados dos módulos lógicos a y b, $a.union(b)$ define un módulo lógico que contiene todas las cláusulas de a y de b.

Por otro lado, para módulos lógicos definidos en métodos, la redefinición y unión se realiza por medio de la herencia. En la figura B.3 se muestra un ejemplo de la operación de combinación.

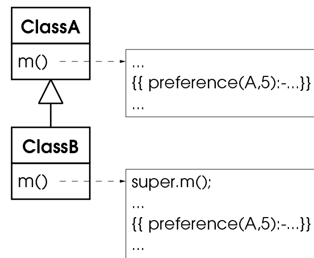


Figura B.3: Combinación de módulos lógicos mediante herencia

APÉNDICE C

Object-Z

En este capítulo se describe el lenguaje de especificación Object-Z, en el cual se especificó Brainstorm/J.

Object-Z [34] es una extensión del lenguaje de especificación Z [107, 91]. Por lo tanto, la primera sección de este apéndice describe brevemente Z. Luego, en la sección C.2 se presenta el lenguaje Object-Z.

C.1 El lenguaje Z

En esta sección se describen las principales construcciones del lenguaje Z [107, 91, 12].

Una especificación Z está formada por texto matemático (formal), aclaraciones informales y diagramas. El texto informal facilita la interpretación del texto formal.

C.1.1 Tipos de datos

Los tipos de datos Z están basados en la teoría de conjuntos tipados. Z pe rmite introducir tipos *dados* arbitrarios mediante la notación $[tipo]$. Por ejemplo:

$$[Pesona, Clave, Cuenta]$$

introduce los conjuntos de personas, claves y cuentas.

Z posee varios tipos predefinidos, tales como el conjunto de enteros $\mathbb{Z}$, naturales $\mathbb{N}$ y booleanos $\mathbb{B}$, con las operaciones usuales. Además, el lenguaje permite especificar nuevos tipos utilizando conjuntos potencia '$\mathbb{P}$', producto cartesiano '$\times$' o *esquemas* (se describen en párrafos posteriores).

Z también posee símbolos para denotar tipos usuales. Por ejemplo, el símbolo relación '$\leftrightarrow$' aplicado como en $r : A \leftrightarrow B$, donde A y B son tipos arbitrarios, denota el conjunto de pares pertenecientes

al conjunto $\mathbb{P}(A \times B)$. El lenguaje posee símbolos que denotan relaciones especiales, tales como las funciones parciales '$\pfun$', funciones parciales finitas '$\ffun$', funciones totales '$\fun$', inyecciones parciales '$\pinj$', biyecciones '$\bij$', etc. Z provee símbolos que denotan operaciones típicas sobre relaciones, por ejemplo, dominio 'dom', rango 'ran', restricción de dominio '$\lhd$', sustracción de dominio '$\ndres$', restricción de rango '$\rhd$' y sustracción de rango '$\nrres$', composición '$\comp$', etc.

En Z, una secuencia $s : \mathrm{seq}\, T$, denotando que s es una secuencia de objetos de tipo T, es formalizada como una función con dominio contiguo en $\mathbb{N}$, desde 1 hasta la longitud de la secuencia $\#s$. La declaración de s es equivalente a:

$$\mathrm{seq}\, T == \{s : \mathbb{N}_1 \pfun X \mid \exists n : \mathbb{N}_1 \bullet \mathrm{dom}\, s = 1 \ldots n\}$$

donde '$==$' denota *equivalencia sintáctica*, $\mathbb{N}_1$ denota el conjunto de naturales mayores que cero, '$|$' y '$\bullet$' denotan *tal que* y 'dom' abrevia *dominio de*. Así, si $t_1, t_2, \ldots \in T$, entonces $\{(1, t_1), (2, t_2), \ldots\} \in \mathrm{seq}\, T$. Z posee una notación más conveniente para las secuencias, por ejemplo, s puede ser denotada por $\langle t_1, t_2, \ldots \rangle$. Obsérvese que los elementos de una secuencia no tienen que ser distintos. La secuencia vacía es denotada por $\langle\rangle$. Luego, si s no es vacía, las funciones '*head*' y '*tail*' significan $head\, s = s(1)$ y $s = \langle head\, s \rangle \frown tail\, s$, respectivamente, donde '$\frown$' denota concatenación de secuencias.

Una secuencia, también es un conjunto y una función, por lo tanto las operaciones usuales de conjuntos ($\in$, $\cap$, $\cup$, $\subseteq$, #, etc.) y funciones se pueden aplicar a secuencias.

C.1.2 Esquemas de estado

Un esquema de estado consiste de declaraciones de variables junto con predicados que restringen los valores de las variables. Por ejemplo, el siguiente esquema define coordenadas enteras situadas en el primer cuadrante del eje. El predicado especifica que las variables de *Coord* nunca son negativas:

Coord

$x : \mathbb{Z}$
$y : \mathbb{Z}$

$x \geq 0 \land y \geq 0$

Los esquemas pueden ser usados como tipos. Las variables componentes se acceden usando notación de punto. Por ejemplo: $c : Coord$, $c.x$ denota el componente x de c.

El esquema *Origen* es una restricción de *Coord* que agrega un predicado adicional:

Origen

Coord

$x = 0 \land y = 0$

El esquema *Origen* es equivalente a:

Origen

$$
\begin{array}{|l}
x : \mathbb{Z} \\
y : \mathbb{Z} \\
\hline
x \geq 0 \land y \geq 0 \\
x = 0 \land y = 0
\end{array}
$$

Otra notación para notar los esquemas es la siguiente:

$$Origen \mathrel{\widehat{=}} [\, Coord \mid x = 0 \land y = 0 \,]$$

C.1.3 Esquemas de operaciones

Un esquema de operación modela una transición de estados relacionando valores de variables antes de la operación (valores-pre) y después de la operación (valores-post). Los valores post son notados mediante ' para distinguirlos de los valores-pre. Las operaciones pueden tener entradas '?'. Por ejemplo, la siguiente operación, suma las coordenadas de entrada $x?$, $y?$ a x, y, respectivamente:

SumaCoord

$$
\begin{array}{|l}
\Delta Coord \\
x?, y? : \mathbb{Z} \\
\hline
x' = x + x? \\
y' = y + y?
\end{array}
$$

Las operaciones también pueden tener salidas, las cuales se identifican con '!'. En la operación *SumaCoord*, $\Delta Coord$ indica que se incluyan las variables del esquema como valores-pre y valores-post. Así, el esquema *SumaCoord* también puede ser notado como:

SumaCoord

$$
\begin{array}{|l}
x, x' : \mathbb{Z} \\
y, y' : \mathbb{Z} \\
x?, y? : \mathbb{Z} \\
\hline
x \geq 0 \land y \geq 0 \\
x' \geq 0 \land y' \geq 0 \\
x' = x + x? \\
y' = y + y?
\end{array}
$$

C.1.4 Cálculo de esquemas

Z permite modificar, combinar o definir nuevos esquemas a partir de esquemas existentes.

Si A y B son esquemas, la conjunción de esquemas $A \land B$ es un esquema que contiene las declaraciones de A y de B y la conjunción de sus predicados. Por ejemplo, en el esquema *SumaCoord*, $\Delta Coord \mathrel{\widehat{=}} Coord \land Coord'$. Otras operaciones con esquemas son la negación, disyunción, implicación y equivalencia.

C.2 El lenguaje Object-Z

Object-Z [34, 35] es una extensión del lenguaje de especificación formal Z que facilita la especificación utilizando un estilo orientado a objetos.

La construcción más importante de Object-Z es la *clase*. Una clase encapsula tipos, constantes y variables describiendo el estado de un conjunto de objetos en el sistema, sus estados iniciales y operaciones aplicables. Las clases pueden ser definidas en forma incremental, heredando definiciones de clases existentes.

Una clase define un tipo cuyas instancias son referencias a objetos de dicha clase. Por ejemplo, dada una clase A, la declaración $a : A$ declara una referencia a un objeto de la clase A, y $a :\downarrow A$ declara una referencia a objetos de la clase A o cualquier clase derivada de A mediante herencia.

En forma similar a lo que sucede con los esquemas Z, las constantes y variables de un objeto referenciado pueden ser accedidos utilizando notación de punto, por ejemplo, $a.x$ denota la constante o variable x del objeto referenciado por a. La notación de punto también puede ser utilizada para especificar que un objeto está en su estado inicial, por ejemplo, $a.\mathrm{INIT}$, y para aplicar una operación a un objeto, por ejemplo, $a.op$, donde op es el nombre de una operación de A.

Una clase Object-Z es representada sintácticamente en forma semejante a los esquemas, conteniendo definiciones de tipos locales y constantes, un esquema de estado, un esquema de estado inicial, y cero o más esquemas de operaciones. Además de esas definiciones explícitas, una clase puede heredar la definición de una o más clases. También puede tener parámetros genéricos, una lista de visibilidad restringiendo la forma en que los objetos de la clase pueden ser usados y un invariante de historia.

ClassName

clases heredadas
declaraciones de tipos locales y definiciones de constantes
esquema de estado
esquema de estado inicial
operaciones

Una clase se designa por su nombre y una lista de renombramiento opcional. Dicha lista permite cambiar los nombres de constantes, variables de estado y operaciones. Por ejemplo, dada una clase A con variables de estado x e y y operaciones op_1 y op_2, el siguiente designador de clase renombra y a z y op_2 a op_3:

$$A[z/y, op_3/op_2]$$

Cuando una clase es heredada, sus tipos locales y constantes quedan disponibles en la subclase. Todos los tipos o constantes con el mismo nombre en ambas clases deben tener definiciones compatibles. El esquema de estado y esquema de estado inicial de la clase heredada son combinados mediante $\wedge$. Las operaciones heredadas quedan disponibles en la subclase, excepto cuando existe otro método con el mismo nombre. En este caso, las operaciones son combinadas con $\wedge$.

Las definiciones de tipos locales y constantes de una clase tienen la misma sintaxis que las respectivas definiciones globales en Z. Sin embargo, su alcance está limitado a la clase en la cual está la declara-

ción. Una constante es asociada con un valor fijo que no puede ser modificado por ninguna operación de la clase. Sin embargo, el valor de las constantes, puede diferir para diferentes objetos de la clase.

El esquema de estado no posee nombre. Las declaraciones son particionadas mediante 'Δ' en variables *primarias* y *secundarias*. Las variables secundarias se definen a partir de las primarias. Por ejemplo, el esquema de estado de una clase *Cuadrado* podría tener una variable primaria *lado*, y variables secundarias *área* y *perímetro*:

$$
\begin{array}{|l}
\hline
lado : \mathbb{N} \\
\Delta \\
area : \mathbb{N} \\
perimetro : \mathbb{N} \\
\hline
area = area * area \\
perimetro = 4 * lado \\
\hline
\end{array}
$$

El esquema de estado inicial lleva el nombre INIT y sólo posee predicados. Dicho esquema incluye las declaraciones y predicados del esquema de estado.

Las operaciones se definen mediante *esquemas de operación* o *expresiones de operación*. Las mismas son interpretadas en el ambiente local de la clase, enriquecido con las declaraciones y predicados del esquema de estado con el símbolo ' y sin él. Así, si se declara $a : A$ en el esquema de estado de una clase, $a.op$ es una operación válida si op es una operación de A.

Un esquema de operación extiende la noción de esquema en Z, añadiendo al mismo una *lista-delta*. La lista-delta es una lista de las variables primarias que pueden ser modificadas por la operación cuando es aplicada a un objeto de la clase; todas las otras variables permanecen sin cambio. Cuando dos o más operaciones son combinadas para definir una nueva operación, sus listas-delta son unidas para que la nueva operación pueda modificar cualquier variable que pudiese ser alterada por una de sus operaciones componentes.

Por ejemplo, dadas las operaciones *Inc_x* e *Inc_y*:

$$
\begin{array}{|l}
\hline
Inc_x \\
\Delta(x) \\
\hline
x' = x + 1 \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline
Inc_y \\
\Delta(y) \\
\hline
y' = y + 1 \\
\hline
\end{array}
$$

la operación $Inc_x \wedge Inc_y$ es equivalente a:

$$
\begin{array}{|l}
\hline
Inc_x \wedge Inc_y \\
\Delta(x, y) \\
\hline
x' = x + 1 \\
y' = y + 1 \\
\hline
\end{array}
$$

Además de la operación de conjunción de esquemas '$\wedge$', Object-Z define: paralelo '$\parallel$', selección '$[]$', enriquecimiento '$\bullet$', ocultamiento '$\setminus$', renombre y secuencia '$\fatsemi$'.

El operador paralelo '$\|$' permite comunicar objetos. Básicamente, '$\|$' identifica e iguala las variables de entrada de cualquiera de las dos operaciones con las de salida con el mismo nombre base (omitiendo ! y ?). Las variables de entrada se ocultan en la operación resultante, las variables de salida no se ocultan, por lo tanto pueden servir como entrada para otras composiciones paralelas siguientes.

El operador de selección ' $[]$ ' permite especificar selección nodeterminística entre operaciones. El significado de $op_1 \; [] \; op_2$ es que ocurre op_1 u op_2, pero no ambas. Por ejemplo, $Inc_x \; [] \; Inc_y$ es equivalente a:

$Inc_x \; [] \; Inc_y$

$\Delta(x, y)$

$(x' = x + 1 \land y' = y)$
$\lor$
$(y' = y + 1 \land x' = x)$

El operador de enriquecimiento '$\bullet$' permite que las operaciones sean interpretadas en el alcance del ambiente de la clase enriquecida con las declaraciones y predicados de otra operación o esquema. Es particularmente útil cuando se especifican operaciones que ocurren en objetos particulares de un conjunto de objetos: las referencias a los objetos seleccionados aparecen en las declaraciones de la primer operación y pueden ser usadas en la definición de la segunda. Por ejemplo, dada la declaración $s : \mathbb{P} A$ en el esquema de estado de una clase, la operación que involucra la composición paralela de dos objetos distintos referenciados por s que realizan la operación op (una operación de A) es:

$$[a_1, a_2 : s \mid a_1 \neq a_2] \bullet a_1.op \| a_{2.op}$$

El operador de composición secuencial '$\fatsemi$' permite definir una operación cuyo efecto es equivalente al de dos o más operaciones ejecutadas en secuencia. Dicho operador identifica y conecta las salidas de la primera operación con las entradas de la segunda con el mismo nombre base.

APÉNDICE D

Notación UML

El Lenguaje Unificado de Modelamiento (**U**nified **M**odeling **L**anguaje) [82, 83] es un lenguaje visual de modelamiento de propósito general diseñado para especificar y documentar los elementos de un sistema de software.

En este capítulo se describe brevemente la notación gráfica UML.

D.1 Correspondencia tipo-instancia

Un objetivo del modelamiento es la preparación de descripciones genéricas que describen muchos ítems particulares. Esto, frecuentemente se conoce como la *dicotomía tipo-instancia*. Muchos de los conceptos de UML poseen este carácter dual, y son modelados por dos elementos unidos, uno de los cuales representa el descriptor genérico y el otro, uno de los ítems individuales. Por ejemplo: Clase-Objeto, Asociación-Link, Parámetro-Valor, Operación-Invocación, etc.

Si bien los diagramas para elementos como tipos e instancias no son exactamente los mismos, poseen ciertos puntos en común. Además, es conveniente que la notación para cada par de elementos tipo-instancia muestre la distinción entre ambos en forma clara. En UML la distinción tipo-instancia se muestra empleando el mismo símbolo geométrico para cada par de elementos y subrayando el nombre de un elemento instancia como se muestra en la figura D.1.

D.2 Diagramas de estructura estática

Los diagramas de clase muestran la estructura estática, en particular, las entidades que *existen* (como las clases y tipos), su estructura interna y sus relaciones con otras entidades. Los diagramas de clases no muestran información temporal. Un diagrama de objetos muestra instancias compatibles con un diagrama de clases en particular.

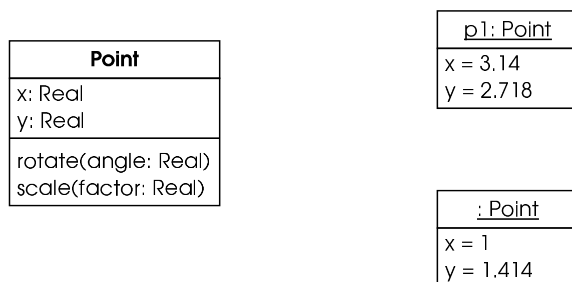


Figura D.1: Clases y objetos

En esta sección se describe la notación para clases y sus variantes, incluyendo *templates* y clases instanciadas, y las relaciones entre clases: asociación y generalización. Se incluye también el contenido de las clases: atributos y operaciones.

D.2.1 Diagrama de clases

Un *diagrama de clases* es un grafo de elementos *Clasificadores* (*Clase*, *TipoDeDato* o *Interfaz*) conectados por relaciones estáticas. Un diagrama de clases puede contener interfaces, paquetes, relaciones e instancias, por lo que también se conoce como *diagrama de estructura estática*. Los diagramas de clases pueden estar organizados en *paquetes*.

D.2.2 Diagrama de objetos

Un *diagrama de objetos* es un grafo de instancias, incluyendo objetos y valores de datos. Un diagrama de objetos estático es una instancia de un diagrama de clases; ésto muestra una *instantánea* del estado detallado de un sistema en un punto del tiempo. El uso de diagramas de objetos es bastante limitado (se usa principalmente para mostrar ejemplos de estructuras de datos).

D.2.3 Clases

Una *clase* es el descriptor para un conjunto de objetos con estructura similar, comportamiento y relaciones. UML provee una notación para declarar clases y especificar sus propiedades. Algunos elementos de modelamiento similares, en forma, a las clases (interfaces o tipos) se notan con palabras clave en los símbolos de clase.

Una clase representa un concepto dentro del sistema modelado. Las clases tienen estructura de dato, comportamiento y relaciones a otros elementos.

Una clase posee un nombre, cuyo alcance o ámbito está limitado al paquete en el cual es definida. El nombre debe ser único (entre nombres de clases) dentro de ese paquete.

Una clase se dibuja con un rectángulo de línea llena con 3 componentes separados por líneas horizontales. El compartimento superior contiene el nombre de clase y otras propiedades generales de la clase; el compartimento del medio contiene una lista de atributos; en la lista inferior se listan las operaciones.

Por defecto, se asume que las clases mostradas en un paquete están definidas en ese paquete. Para mostrar una referencia a una clase de otro paquete, se usa la sintaxis:

NombreDePaquete::NombredeClase

Los nombres de clases (concretas) se muestran centrados y en negrita, mientras que los nombres de clases abstractas se muestran en negrita e itálica. Las operaciones abstractas también se muestran en itálica (figura D.2).

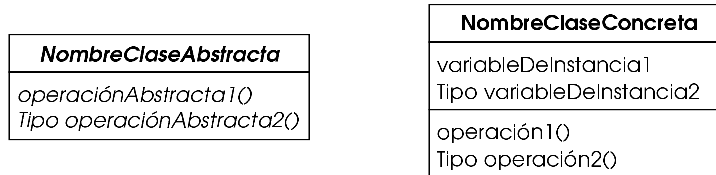


Figura D.2: Notación de clases abstractas y concretas

D.2.4 Interfaces

Una interfaz es un especificador de las operaciones externamente visibles de una clase, componente u otra entidad, sin especificación de la estructura interna. Típicamente, cada interfaz especifica sólo una parte limitada del comportamiento de una clase. Las interfaces no tienen implementación, ni atributos, estados o asociaciones; sólo poseen operaciones. Las interfaces pueden tener relaciones de generalización. Una interfaz es equivalente a una clase abstracta sin atributos ni métodos y sólo operaciones abstractas.

Una interfaz se nota igual que una clase, pero se incluye la palabra *interface* junto al nombre. El compartimento de operaciones debe contener una lista de las operaciones soportadas por la interfaz (figura D.3).

Una interfaz también puede notarse con un pequeño círculo sobre el cual se sitúa el nombre de la misma. El círculo puede estar unido por una línea sólida a una clase que la soporta. Esto indica que la clase provee todas las operaciones especificadas por la interfaz (y posiblemente más). La notación de círculo no muestra las operaciones.

Una clase que usa o requiere de las operaciones provistas por una interfaz puede ser unida al círculo con una línea de trazos apuntando al círculo. La línea de trazos indica que la clase requiere una o más de las operaciones especificadas en la interfaz; la clase cliente puede no usar todas las operaciones.

D.2.5 Clases parametrizadas (*templates*)

Un *template* es un descriptor para una clase con uno o más parámetros formales no ligados. El mismo define una familia de clases, cada clase especificada por un *binding* de los parámetros a los valores actuales. Típicamente, los parámetros representan tipos, pero también pueden representar enteros, otros tipos u operaciones. Los atributos y operaciones dentro del *template* se definen en términos de los parámetros formales, así, ellos se ligan cuando el *template* se liga a los valores actuales.

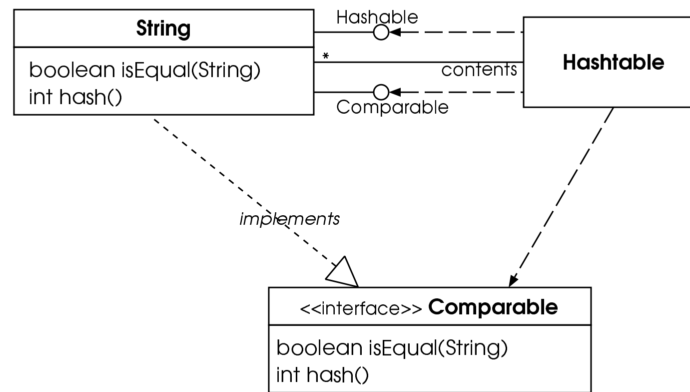


Figura D.3: Notación para las interfaces en los diagramas de clases

Un *template* no es una clase directamente utilizable, debido a que posee parámetros no ligados. Sus parámetros deben ser ligados a valores actuales para crear una clase utilizable desde la clase *template*. Un *template* puede ser subclase de cualquier clase ordinaria; esto implica que todas las clases formadas, al ligarla, son subclases de la superclase dada.

En las clases parametrizadas se coloca un pequeño rectángulo en la esquina superior derecha del símbolo de clase (u otro símbolo). El rectángulo de línea de trazos contiene la lista de los parámetros formales para la clase y sus tipos de implementación (figura D.4). Dicha lista no debe estar vacía, aunque puede ser suprimida en una representación resumida. El nombre, atributos y operaciones de la clase parametrizada aparecen como en las clases concretas, pero pueden incluir ocurrencias de los parámetros formales. Las ocurrencias de los parámetros formales pueden apareces dentro del contexto de la clase, por ejemplo, para mostrar una clase relacionada identificada por uno de los parámetros.

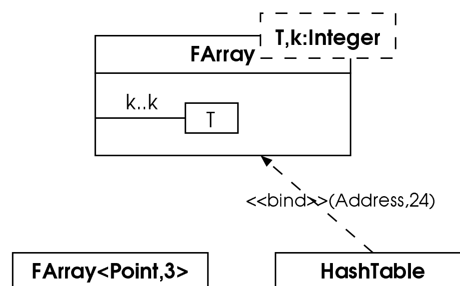


Figura D.4: Notación de clases parametrizadas

D.2.6 Elemento ligado

Un *template* no puede ser usado directamente en una relación ordinaria como la generalización o asociación, debido a que posee un parámetro libre que no posee ningún significado fuera del ámbito que declara el parámetro. Para ser utilizado, los parámetros del *template* deben ser ligados a los valores actuales. El valor actual para cada parámetro es una expresión definida dentro del ámbito de uso. Si el ámbito referenciado es un *template*, luego los parámetros del *template* referenciado pueden ser usados como valores actuales en el binding del *template* referenciado, pero no puede asumirse una

correspondencia entre los nombres de parámetros en los dos *templates*, ya que ellos no tienen alcance fuera de sus respectivos *templates*.

Un elemento ligado se indica con una cadena de texto con la misma sintaxis que otros elementos:

Template-name '<' value-list '>'

donde *value-list* es una lista de elementos delimitados por coma, de expresiones de valor, y *Template-name* es idéntico al nombre del *template*. Por ejemplo, *VArray<Point,3>* designa una clase descripta por el *template VArray* (figura D.4).

El número y los tipos de los valores deben coincidir con el número y tipos de los parámetros *template* para el *template* con el nombre dado.

El nombre del elemento ligado puede ser utilizado en cualquier parte en donde el nombre de elemento de la clase parametrizada sea útil. Por ejemplo, un nombre de clase ligado podría ser usado dentro de un símbolo de clase en un diagrama de clases, como un tipo de atributo, o como parte de la declaración de una operación.

Note que un elemento ligado queda totalmente especificado por su *template*, además, su contenido no puede ser extendido; no está permitida la declaración de nuevos atributos u operaciones, por ejemplo; pero una clase ligada podría ser subclasificada y la subclase extendida en la forma usual.

La relación entre el elemento ligado y su *template* puede notarse mediante una relación de dependencia con la palabra clave *bind*. Los argumentos se especifican entre paréntesis, a continuación de dicha palabra clave.

D.2.7 Objetos

Un objeto representa una instancia particular de una clase. Posee una identidad y valores de atributos. La misma notación también representa un rol dentro de una colaboración, debido a que los roles poseen características de instancias.

La notación de objetos deriva de la notación de clases, como se vió en la sección D.1. Así, un objeto se representa por un rectángulo con compartimentos. El compartimento superior muestra el nombre del objeto y su clase, ambos subrayados usando la sintaxis:

objectname: classname

Si es necesario, el nombre de clase puede incluir el nombre de camino completo del paquete al que pertenece. Los nombres de paquetes preceden al nombre de clase y son separados por "::". Por ejemplo:

display_window: WindowingSystem::GraphicWindows::Window

Para mostrar la presencia de un objeto en un estado particular, se usa la sintaxis:

objectname: classname '[' statename-list ']'

La lista debe ser estar conpuesta por nombres de estados separados por comas.

D.2.8 Objeto compuesto

Un objeto compuesto representa un objeto de alto nivel formado por partes muy relacionadas. Un objeto compuesto es similar (pero más simple y restricto que) una colaboración, pero se define completamente por la composición en un modelo estático.

Un objeto compuesto se nota con un símbolo objeto. El nombre del string del objeto compuesto se coloca en un compartimento en la parte superior del rectángulo. El compartimento inferior contiene las partes del objeto compuesto, en lugar que la lista de atributos.

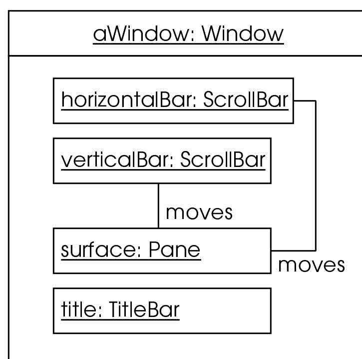


Figura D.5: Objeto compuesto

D.2.9 Asociación

Las asociaciones binarias se notan con líneas que conectan dos símbolos de clase. Las líneas pueden poseen una variedad de adornos para mostrar diversas propiedades. Las asociaciones ternarias y de mayor orden se muestran con rombos conectados a los símbolos de clases con líneas.

D.2.10 Asociación binaria

Una asociación binaria es una asociación entre dos clases (incluyendo la posibilidad de una asociación re¤exiva de la clase consigo misma).

La asociación binaria se representa con un camino sólido que conecta dos símbolos de clase (ambos extremos pueden estar conectados a la misma clase, pero los dos extremos son distintos). El camino puede consistir de uno o más segmentos conectados.

El camino puede tener adornos gráficos asociados. Esos elementos indican propiedades de la asociación. Los siguientes adornos pueden ser parte de una asociación:

- Nombre de la asociación: Designa el nombre opcional de la asociación.
- Símbolo de asociación de clase: Designa una asociación que tiene propiedades similares a una clase, como atributos, operaciones, y otras asociaciones. Este símbolo está presente sólo si la asociación es una clase asociación.

Las restricciones OR indican una situación en la cual sólo una de varias asociaciones potenciales puede ser instanciada a la vez para cualquier objeto. Esto se indica con una línea de trazos conectando

dos o más asociaciones, las que deben tener una clase en común, con la etiqueta "*{or}*" sobre la línea de trazos (figura D.6).

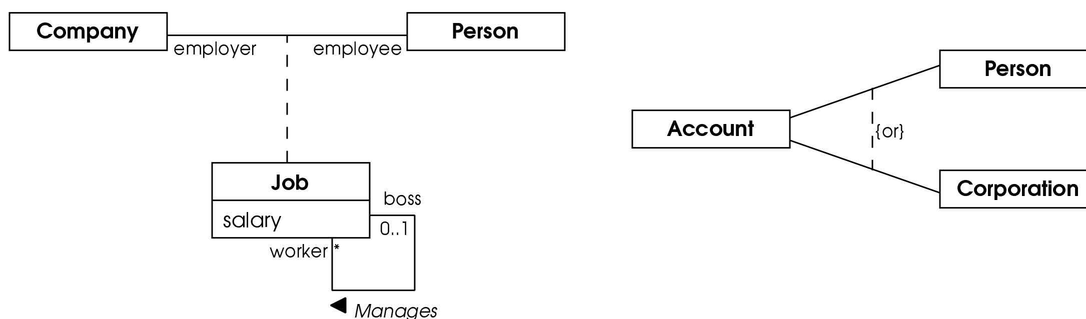


Figura D.6: Notación de las asociaciones

D.2.11 Extremo £nal de la asociación

El extremo final de una asociación el lugar donde un asociación se conecta con una clase.

La línea de una asociación puede poseer adornos gráficos en sus extremos. Esos adornos indican propiedades de la asociación relacionadas con la clase. Los adornos son parte del símbolo de asociación, no parte del símbolo de clase. Los siguientes elementos pueden ser colocados en los extremos de una asociación:

- Multiplicidad: se especifica en sintaxis textual.
- Orden: si la multiplicidad es mayor que uno, luego el conjunto de elementos relacionados pueden ser ordenados o desordenados. Si no se indica nada, entonces es desordenado (los elementos de un conjunto). Varios tipos de orden pueden ser especificados como una restricción en el extremo de la asociación. La declaración no especifica cómo se establece o mantiene el orden; las operaciones que insertan nuevos elementos deben poseer la capacidad de especificar una posición, ya sea en forma implícita como explícita. Los valores posibles incluyen:
  - Desordenado: los elementos forman un conjunto desordenado. Esta es la opción por defecto y no necesita ser indicada explícitamente.
  - Ordenado: los elementos del conjunto están ordenados en una lista. No se permiten elementos duplicados. Esto puede especificarse con la palabra clave *{ordered}*.
- Navegabilidad: una ¤echa en el extremo de la asociación indica que la navegación es soportada hacia la clase a la que apunta la ¤echa. Las ¤echas pueden estar en 0, 1 o 2 extremos de la asociación.
- Indicador de agregación: un rombo vacío se coloca en el extremo de la línea de asociación para indicar agregación. El rombo no puede estar en los dos extremos de la línea, pero puede estar ausente. El rombo va junto a la clase que es el agregado. La agregación es opcional, pero no suprimible. Si el rombo se llena, indica una forma fuerte de agregación conocida como *composición*.
- Nombre de rol: una etiqueta cerca de los extremo de la línea de asociación indica el rol que juega la clase. El rol es opcional, pero no es suprimible.

- Especificador de interfaz: el nombre de un clasificador con sintaxis “*:classifiername*” indica el comportamiento esperado de un objeto asociado por el objeto relacionado. En otras palabras, el especificador de interfaz especifica el comportamiento requerido para permitir la asociación. En este caso, la clase actual sólo provee más funcionalidad que la requerida para la asociación en particular. El uso de un nombre de rol y un especificador de interfaz es equivalente a la creación de una colaboración que con sólo una asociación con dos roles, cuya estructura está definida por el nombre de rol y el clasificador de rol de la asociación original. La asociación y las clases originales son, además, un uso de la colaboración. La clase original debe ser compatible con el especificador de interfaz. Si se omite el especificador de interfaz, entonces la asociación puede ser usada para obtener acceso total a la clase asociada.
- Visibilidad: se especifica mediante ’+’, ’#’, ’-’ o una palabra clave como {*public*}, {*protected*} o {*private*}, respectivamente, delande de un nombre de rol. Especifica la visibilidad de la asociación en la dirección del nombre de rol.

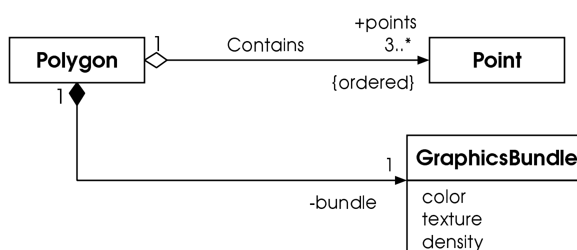


Figura D.7: Asociaciones

D.2.12 Multiplicidad

Un ítem de multiplicidad especifica el rango de cardinalidades válidas que un conjunto puede asumir. Las especificaciones de multiplicidad pueden darse para roles dentro de asociaciones, partes de objetos compuestos, repeticiones u otros usos. Esencialmente, una especificación de multiplicidad es un subconjunto del conjunto abierto de enteros no negativos.

La especificación de multiplicidad se realiza mediante un rango de enteros en el formato *límite-inferior* .. *límite-superior*, donde *límite-inferior* y *límite-superior* son enteros que especifican el rango cerrado de enteros desde el límite inferior hasta el límite superior. Además, el carácter ’*’ puede ser usado para el límite superior, denotando un número ilimitado. En un contexto parametrizado (como un *template*) los límites pueden ser expresiones, pero deben evaluar a valores enteros.

D.2.13 Cali£cador

Un calificador es un atributo o lista de atributos cuyo valor sirve para particionar un conjunto de objetos asociados con un objeto en una asociación. Los calificadores son atributos de la asociación.

Un calificador se dibuja como un rectángulo conectado al extremo de una línea de asociación (figura D.8). Un objeto de la clase origen con un valor del calificador selecciona una partición en el conjunto de objetos de las clases destino en el otro extremo de la asociación.

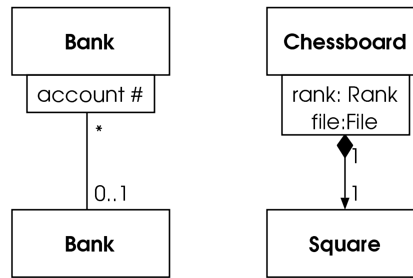


Figura D.8: Asociaciones cali£cadas

D.2.14 Clase asociación

Una clase asociación es una asociación que posee las propiedades de una clase (o una clase que tiene propiedades de asociación). Una clase asociación se dibuja como un símbolo de clase (Job en figura D.9).

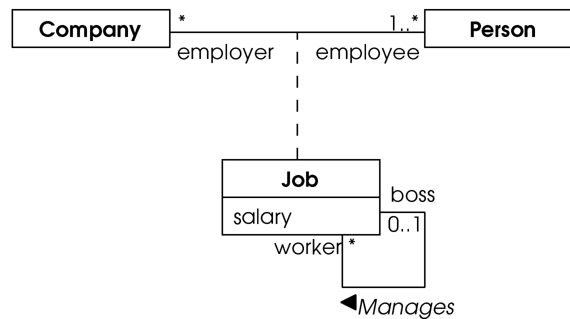


Figura D.9: Clase asociación

D.2.15 Asociación n-aria

Una asociación n-aria es una asociación entre 3 o más clases (una clase puede aparecer más de una vez). Cada instancia de la asociación en una n-tupla de valores de las clases respectivas. Una asociación binaria es un caso especial con su propia notación.

La multiplicidad para las asociaciones n-arias puede ser especificada, pero es menos obvia que la multiplicidad binaria. La multiplicidad en un rol representa el número potencial de instancias de tuplas en la asociación cuando cuando los otros $N-1$ valores son fijos.

Una asociación n-aria no puede contener el símbolo de agregación o ningún rol.

Una asociación n-aria se nota con un rombo grande con una línea desde el rombo a cada clase participante (figura D.10). El nombre de la asociación (si posee) se muestra cerca del rombo. Los adornos de rol pueden aparecer en cada línea de asociación como es usual. Puede indicarse multiplicidad, sin embargo, no se permiten calificadores ni agregación.

Una símbolo de clase asociación puede estar conectado al rombo por una línea de trazos. Esto representa una asociación n-aria con atributos, operaciones o asociaciones.

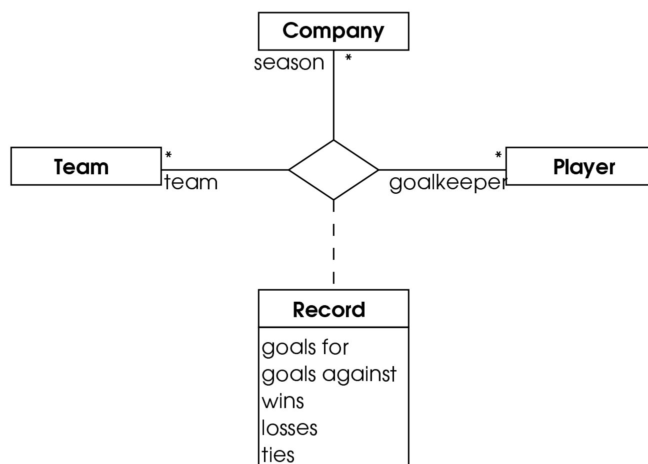


Figura D.10: Asociación ternaria en una clase asociación

D.2.16 Composición

La composición es una forma de agregación con propiedad más estricta y tiempo de vida coincidente entre las partes de un todo. La multiplicidad del agregado no puede exceder uno.

Las partes de una composición pueden incluir clases y asociaciones. El significado de una asociación en una composición es que cualquier tupla de objetos conectado por un único link debe pertenecer al mismo objeto contenedor.

La composición puede ser notada por un rombo sólido en un extremo de una línea de asociación. Alternativamente, UML provee una forma anidada que en algunos casos resulta más conveniente (figura D.11).

En lugar de usar asociaciones binarias con el símbolo de agregación, la composición puede ser notada anidando gráficamente los símbolos de los elementos de las partes dentro del símbolo del elemento para el todo.

Alternativamente, la composición puede notarse mediante un rombo relleno en el extremo final de una línea de asociación.

D.2.17 Links

Un link es una tupla de referencias a objetos. Comunmente, es un par de referencias a objetos. Es una instancia de una asociación.

Un link binario se nota con una línea entre dos objeto (figura D.12). Al final del link se puede indicar un nombre de rol. Un nombre de asociación puede ser mostrado cerca de la línea; si está presente, se subraya para indicar una instancia. Los links no poseen nombre de instancia; toma su identidad de los objetos a los que pertenecen. La multiplicidad no se muestra en los links debido a que son instancias. Otros símbolos como agregaciones, composición o agregación pueden ser mostrados en los links.

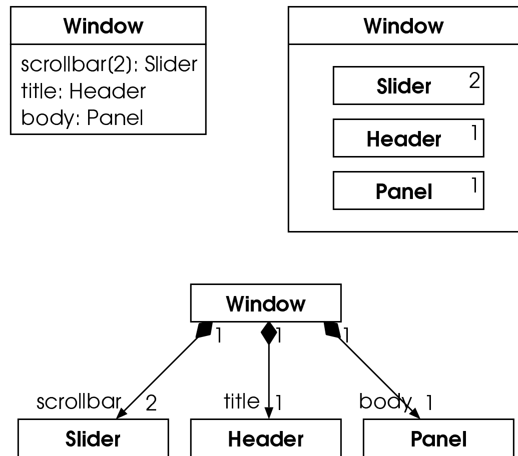


Figura D.11: Diferentes formas de mostrar la composición

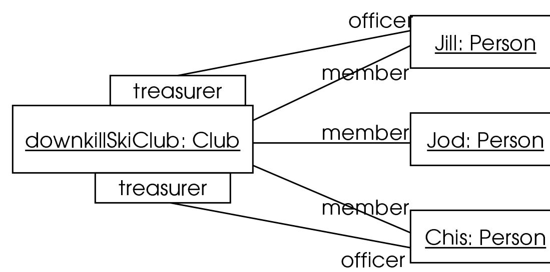


Figura D.12: Links

D.2.18 Generalización

La generalización es la relación entre un elemento más general y un más específico consistente con el primero, que añade información adicional. Se utiliza para clases, paquetes, y otros elementos.

La generalización se nota con una línea sólida desde el elemánto más específico (como una subclase) al más general (como la superclase), con un triángulo en el extremo final de la línea junto al elemento más general (figura D.13).

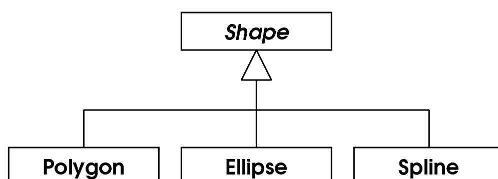


Figura D.13: Generalización

D.2.19 Dependencia

Una dependencia indica una relación semántica entre dos o más elementos del modelo. Indica situaciones en las que un cambio en la situación del elemento destino puede requerir un cambio del elemento origen de la dependencia (figura D.13).

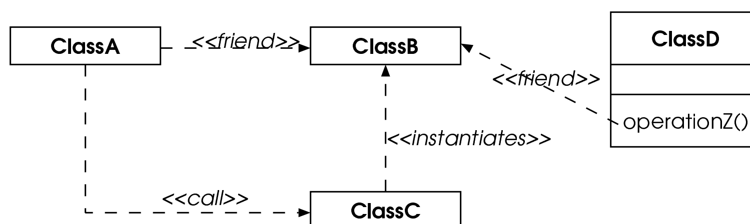


Figura D.14: Dependencias entre clases

D.3 Diagrama de interacción

Un diagrama de interación muestra una interacción organizada en una secuencia en el tiemp. En particular, muestra los objetos participantes en la interacción por sus *líneas de vida* y los mensajes que intercambian organizados en una secuencia temporal.

Un diagrama de interacción (figura D.15) tiene dos dimensiones: la dimensión vertical representa el tiempo, y la horizontal representa objetos. Normalmente, el tiempo avanza hacia la parte inferior de la página. Usualmente, sólo interesa la secuencia temporal de acontecimientos, pero en aplicaciones de tiempo real, el eje temporal puede medir tiempo. No hay importancia en el orden de horizontal (de los objetos).

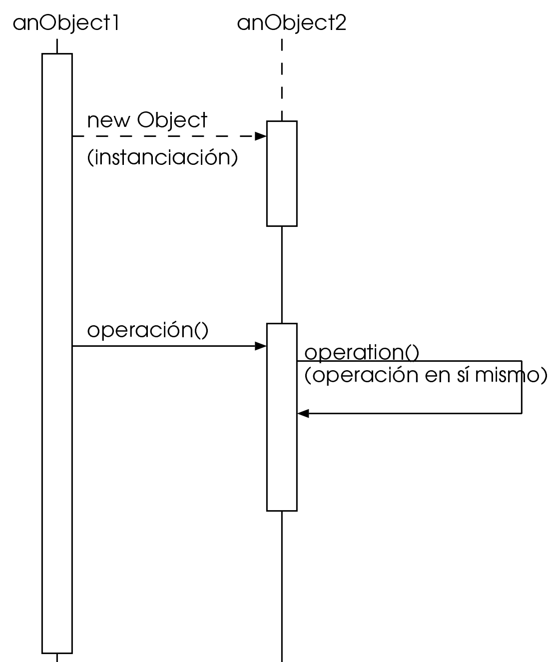


Figura D.15: Notación de los diagramas de interacción

Bibliografía

[1] AMANDI, A., ITURREGUI, R., AND ZUNINO, A. Object-agent oriented programming. In *Argentinian Symposium on Object Orientation (JAIIO)* (Buenos Aires, Argentina, Sept. 1998), SADIO. 1

[2] AMANDI, A., AND PRICE, A. Building intelligent agents from objects. In *Proceedings of the Argentine Symposium on Object Orientation* (Buenos Aires, Argentina, Aug. 1997), SADIO. 2.4

[3] AMANDI, A., AND PRICE, A. Object-oriented agent programming through the brainstorm system. In *PAAM'97 (Practical Applications of Intelligent Agents and Multi-Agents)* (London, Apr. 1997). 2.4, 2.4.2

[4] AMANDI, A., AND PRICE, A. Building object-agents from a software meta-architecture. In *Lecture Notes in Artificial Intelligence* (1998), vol. LNAI 1515, Springer-Verlag, pp. 21–30. 1, 2.3, 2.4

[5] AMANDI, A., ZUNINO, A., AND ITURREGUI, R. Multi-paradigm languages supporting multi-agent development. In *Multi-Agent System Engineering, 9th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, MAAMAW'99* (Valencia, Spain, June 1999), F. J. Garijo and M. Boman, Eds., vol. LNAI 1647 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, pp. 128–139. 1, 2.4, 2.4.2, 4, 4.1, 5, B

[6] AVANCINI, H., AND AMANDI, A. Towards a framework for multi-agent systems. In *Proc. of the Argentinian Symposium on Artificial Intelligence (28th JAIIO)* (Buenos Aires, Argentina, Sept. 1999), SADIO. 1, 3.4

[7] BARBUCEANU, M., AND FOX, M. The design of a coordination language for multi-agent systems. In *Proceedings of the ECAI'96 Workshop on Agent Theories, Architectures, and Languages: Intelligent Agents III* (Berlin, Aug. 12–13 1999), J. P. Müller, M. J. Wooldridge, and N. R. Jennings, Eds., vol. 1193 of *LNAI*, Springer, pp. 341–356. 1, 2.1.3, 2.1.3.2, 11, 5.2.1

[8] BLUM, A. L., AND FURST, M. L. Fast planning through planning graph analysis. *Artificial Intelligence 90*, 1–2 (1997), 279–298. 2.1.6.1, 4.8.3

[9] BRADSHAW, J. M. *Software Agents*. AAAI Press, Menlo Park, USA, 1997. (document), 1, 2.1, 2.1, 40, 87, 99

[10] BRATMAN, M. E., ISRAEL, D. J., AND POLLACK, M. E. Plans and resource-bounded practical reasoning. *Computational Intelligence 4*, 4 (1988), 349–355. 2

[11] BRAZIER, F., DUNIN-KEPLICZ, B., TREUR, J., AND VERBRUGGE, R. Modelling internal dynamic behaviour of bdi agents. In *Proceedings of the Third International Workshop on Formal Models of Agents, MODELAGE'97* (1999), vol. Lecture Notes in AI, Springer Verlag. 2

[12] BRIEN, S. M., AND NICHOLLS, J. E. Z base standard: Version 1. Tech. Rep. PRG-107, Oxford University Computing Laboratory, 1992. C.1

[13] BROOKS, R. A. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation 2* (Apr. 1986), 14–23. (document), 1, 2.1.5

[14] BUGLIESI, M., LAMMA, E., AND MELLO, P. Modularity in logic programming. *The Journal of Logic Programming 19 & 20* (May 1994), 443–502. B.1

[15] BUSCHMANN, F., MEUNIER, R., ROHNERT, H., SOMMERLAD, P., AND STAL, M. Blackboard. In *Pattern-Oriented software architecture*. John Wiley & Sons, England, 1996, ch. 6, pp. 71–95. 4.8.4

[16] CAMPO, M., AND AMANDI, A. Architectural issues in framework design. In *OOSA'98 Object-Oriented Software Architectural Workshop,* 12th *Europen Conference on Object-Oriented Programming* (Bruselas, Belgica, July 1998). (document), 1, 2.7, 2.3

[17] CAMPO, M., AND PRICE, T. Meta-object manager: A framework for customizable meta-object support for smalltalk-80. In *Proc. of the Brazilian Symposium of Languages* (Brazil, Sept. 1996). 1

[18] CAMPO, M., AND PRICE, T. Luthier - a framework for building program analysis tools. In *Implementing Applications Frameworks: Object Oriented Frameworks at Work*, M. Fayad, D. C. Schmidt, and R. Johnson, Eds. Wiley & Sons, 1999. 4, 4.1, A

[19] CARBONELL, J. G., KNOBLOCK, C., AND MINTON, S. Prodigy: An integrated architecture for planning and learning. In *Architectures for Intelligence*, K. V. Lehn, Ed. Lawrence Erlbaum Associates, 1990. 2.1.6.1

[20] CHAUHAN, D. *JAFMAS: A Java-based Agent Framework for Multiagent Systems Development and Implementation*. PhD thesis, ECECS Department, University of Cincinnati, 1997. http://www.ececs.uc.edu/~dchauhan/. 3.7, 5.2, 5.2.2

[21] CHAUHAN, D., AND BAKER, A. Jafmas: A multiagent application development system. In *Proceedings of the* 2nd *International Conference on Autonomous Agents (Agents'98)* (Minneapolis, St. Paul, May 1998). ACM Press. 1, 3.7

[22] CHESS, D., GROSOF, B., HARRISON, C., LEVINE, D., PARIS, C., AND TSUDIK, G. Itinerant agents for mobile computing. IBM Research Report RC 20010, IBM, Mar. 1995. 2.1.4

[23] CHIBA, S. Javassist - re¤ection-based programming wizard for java. In *Proceedings of the ACM OOPSLA'98 Workshop on Re¤ective Programming in C++ and Java* (Oct. 1998). 14

[24] CHIBA, S., AND TATSUBORI, M. Yet another java.lang.Class. In *ECOOP'98 Workshop on Re¤ective Object-Oriented Programming and Systems* (Brussels, Belgium, July 1998). 14

[25] CODENIE, W., HONDT, K. D., STEYAERT, P., AND VERCAMMEN, A. From custom applications to domain-specific framworks. *Communications of the ACM 40*, 10 (Oct. 1997), 71–77. 1, 2.3

[26] COHEN, P. R., AND LEVESQUE, H. J. Intention is choice with commitment. *Artificial Intelligence, AI 42*, 2–3 (Mar. 1990), 213–261. 4.8.6

[27] CORKILL, D. D. Blackboard systems. *AI Expert 6*, 9 (Sept. 1991), 40–47. 4.8.2

[28] DAHM, M. Byte code engineering. In *Proceedings of the Java-Information-Days* (Germany, 1999). 15

[29] DECKER, K. TÆMS: A framework for environment centered analysis & design of coordination mechanisms. In *Foundations of Distributed Artificial Intelligence*, G. O'Hare and N. Jennings, Eds. Wiley Inter-Science, 1996, ch. 16, pp. 429–448. 1, 3.5

[30] DEMAZEAU, Y., AND MÜLLER, J.-P., Eds. *Decentralized AI - Proceedings of the First European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW-89)* (1990), Elsevier Science B. V.: Amsterdam, Netherland. 1

[31] DEMAZEAU, Y., AND MÜLLER, J.-P. From reactive to intentional agents. In *Decentralized A.I. 2 — Proceedings of the Second European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW-90)* (Amsterdam, The Netherlands, 1991), Y. Demazeau and J.-P. Müller, Eds., Elsevier Science Publishers B.V., pp. 3–10. 2.1

[32] DONG, J. S., AND DUKE, R. The geometry of object containment. *Object-Oriented Systems 2*, 1 (Mar. 1995), 41–63. 6.1.1, 6.1.3, 6.1.5

[33] DONG, J. S., AND DUKE, R. Using Object-Z to specify object-oriented programming languages. Tech. Rep. 96-02, Software Verification Research Centre - Departament of Computer Science, University of Queensland, Feb. 1996. 6.1, 6.1.2

[34] DUKE, R., KING, P., ROSE, G., AND SMITH, G. The Object-Z specification langauge, version 1. Tech. Rep. 91-1, Software Verification Research Centre, Department of Computer Science, The University of Queensland, Australia, 1991. C, C.2

[35] DUKE, R., ROSE, G., AND SMITH, G. Object-Z: A specification language advocated for the description of standards. *Computer Standards & Interfaces 17*, 5–6 (Sept. 1995), 511–533. C.2

[36] ETZIONI, O., AND WELD, D. S. A softbot-based interface to the internet. *CACM 37*, 7 (1994), 72–76. (document), 1

[37] ETZIONI, O., AND WELD, D. S. Intelligent agents on the internet: Fact, fiction, and forecast. *IEEE Expert 10*, 4 (Aug. 1995), 44–49. 2.1

[38] FAYAD, M. E., AND SCHMIDT, D. C. Object-oriented application frameworks (special issue introduction). *Communications of the ACM 40*, 10 (Oct. 1997), 39–42. 1, 2.2, 7.3

[39] FIKES, R. E., AND NILSSON, N. J. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence Journal 2* (1971), 189–208. 2.1.6.1

[40] FININ, T., LABROU, Y., AND MAYFIELD, J. KQML as an agent communication language. In *Software Agents* [9]. 2.1.3, 2.1.3.1, 2.4, 3.7

[41] FISCHER, K. Knowledge-based reactive scheduling in a ¤exible manufacturing system. In *Proc. of the IFIP TC5/WG5.7 Workshop on Knowledge-Based Reactive Scheduling* (Feb. 1994), R. M. Kerr and E. Szelke, Eds., Elsevier Science B.V., pp. 1–18. (document), 1

[42] FISCHER, K., MÜLLER, J. P., AND PISCHEL, M. Unifying control in a layered agent architecture. Technical Memo TM-94-05, Deutsches Forschungszentrum für Künstliche Intelligenz, Deutsches Forschungszentrum für Künstliche Intelligenz GmbH, 1994. 1, 2.3, 4.2, 5.1

[43] FRANKLIN, S., AND GRAESSER, A. Is it an agent, or just a program?: A taxonomy for autonomous agents. In *Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages (ATAL'96)* (New York, 1996), Springer-Verlag. 2.1

[44] FUGGETTA, A., PICCO, G. P., AND VIGNA, G. Understanding code mobility. *IEEE Transactions on Software Engineering 24*, 5 (May 1998), 342–361. 2.1, 2.1.4, 4.7

[45] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison Wesley, Reading, MA, 1995. 1, 2.3, 3.4, 4.1, 4.4

[46] GARLAN, D. What is style? In *Proceedings of the Dagstuhl Workshop on Software Architecture* (Saarbruecken, Germany, February 1995). 1, 2.3

[47] GENESERETH, M. R., AND KETCHPEL, S. P. Software agents. *Communications of the ACM 37*, 7 (1994), 48–53. 2.1.3

[48] GEORGEFF, M. P. Situated reasoning and rational behaviour. In *Pacific Rim Intenrnational Conference on Artificial Intelligence* (1990). 4.8

[49] GEORGEFF, M. P., AND INGRAND, F. F. Decision-making in an embedded reasoning system. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence* (Detroit, MI, USA, Aug. 1989), N. S. Sridharan, Ed., Morgan Kaufmann, pp. 972–978. (document), 1, 2, 2.3

[50] GIANPAOLO CUGOLA, CARLO GHEZZI, G. P. P., AND VIGNA, G. A characterization of mobility and state distribution in mobile code languages. In 2nd *ECOOP Workshop on Mobile Object Systems* (Linz, Austria, July 1996), pp. 10–19. 2.1.4

[51] GOLM, M. metaXa and the future of re¤ection. In *OOPSLA'98 Workshop on Re¤ective Programming in C++ and Java* (Vancouver, British Columbia, Oct. 1998). 14

[52] GRAHAM, J., AND DECKER, K. Towards distributed, environment centred agent framework. In *Proc. of Agent Theory and Languages (ATAL) '99* (July 1999). (document), 1, 3.3, 3.1

[53] GUTKNECHT, O., AND FERBER, J. Madkit: Organizing heterogeneity with groups in a platform for multiple multi-agent systems. Tech. Rep. RR97188, Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier. Université Montpellier II C 09928, France, Dec. 1997. 1, 3.8

[54] HAMMOND, K. J. *Case-based planning*. Academic Press, 1989. 2.1.7

[55] HORLING, B. A reusable component architecture for agent construction. Master's thesis, University of Massachusetts/Amherst CMPSCI, May 1998. http://mas.cs.umass.edu/research/jaf/. 1, 3.5

[56] JENNINGS, N. R., SYCARA, K., AND WOOLDRIDGE, M. A roadmap of agent research and development. *Journal of Autonomous Agents and Multi-Agent Systems 1*, 1 (1998), 7–38. 1

[57] JOHANSEN, D., VAN RENESSE, R., AND SCHNEIDER, F. B. An introduction to the TACOMA distributed system. Tech. Rep. 95-23, Department of Computer Science, University of Tromsø, Tromsø, Norway, June 1995. 2.1.4

[58] JOHNSON, R. E. Frameworks = (components + patterns). *Communications of the ACM 40*, 10 (Oct. 1997), 39–42. 2.2

[59] JOHNSON, R. E., AND RUSSO, V. F. Reusing object-oriented designs. Tech. Rep. DCS 91-1696, UIUC, May 1991. 1, 2.2, 3.1, 4

[60] KENDALL, E. A., KRISHNA, P. V. M., PATHAK, C. V., AND SURESH, C. B. A framework for agent system. In *Implementing Applications Frameworks: Object Oriented Frameworks at Work*, M. Fayad, D. C. Schmidt, and R. Johnson, Eds. Wiley & Sons, 1999. (document), 1, 3.6, 3.3

[61] KOLODNER, J. *Case-Based Reasoning*. Morgan Kaufmann, San Mateo, California, 1993. 1, 4.8

[62] KUSHMERICK, N., HANKS, S., AND WELD, D. An algorithm for probabilistic least-commitment planning. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)* (Seattle, Washington, USA, Aug. 1994), vol. 2, AAAI Press/MIT Press, pp. 1073–1078. 2.1.6, 4.8

[63] LANGE, D. B., AND OSHIMA, M. *Programming and Deploying Mobile Agents with Java Aglets*. Addison-Wesley, Reading, MA, USA, Sept. 1998. 4.1, 4.7

[64] LOGAN, B. Classifying agent systems. In *AAAI-98 Workshop on Software Tools for Developing Agents* (Madison, Wisconsin, USA, July 1998). 2.1, 2.1.1

[65] LONG, D., AND FOX, M. Efficient implementation of the plan graph in stan. *Journal of Artificial Intelligence Research 10* (1999), 87–115. 2.1.6.1

[66] MAES, P. Concepts and experiments in computational re¤ection. *ACM SIGPLAN Notices 22*, 12 (Dec. 1987), 147–155. 1, 2.4

[67] MCCABE, T. J. A complexity measure. In *Proceedings of the 2nd International Conference on Software Engineering* (1976), IEEE Computer Society Press, p. 407. 13

[68] MITCHELL, T., CARUNA, R., FRIETAG, D., MCDERMOTT, J., AND ZABOWSKI, D. Experience with a learning personal assistant. *Communications of the ACM, Special Issue on Intelligent Agents 37*, 7 (July 1994), 80–91. (document), 1

[69] NEBEL, B. How hard is it to revise a belief base? In *Handbook of Defeasible Reasoning and Uncertainty Management Systems*, D. Dubois and H. Prade, Eds., vol. 3 - Belief Change. Kluwer Academic, Dordrecht, The Netherlands, 1998, pp. 77–145. 4.8.5

[70] NILSSON, N. J. Introduction to machine learning. Draft, Sept. 1996. 2.1.7

[71] NWANA, H., NDUMU, D., LEE, L., AND COLLIS, J. Zeus: A tool-kit for building distributed multi-agent systems. *Applied Artifical Intelligence Journal 13*, 1 (1999), 129–186. http://www.labs.bt.com/projects/agents/zeus/index.htm. 1, 3.9

[72] NWANA, H. S. Software agents: An overview. *Knowledge Engineering Review 11*, 3 (Sept. 1996), 205–244. (document), 1, 2.1, 2.1.1, 4

[73] O'KEEFE, R. A. Towards an algebra for constructing logic programs. In *Proceedings of the International Symposium on Logic Programming* (July 1985), IEEE Computer Society, Technical Committee on Computer Languages, The Computer Society Press, pp. 152–161. B.1

[74] ORTIGOSA, A., CAMPO, M., AND MORIYÓN, R. Enhancing framework usability through smart documentation. In *Proc. of the Argentinian Symposium on Object Orientation (28th JAIIO)* (Buenos Aires, Argentina, Sept. 1999), SADIO, pp. 103–117. (document), 1

[75] PETRIE, C. Agent-based engineering, the web, and intelligence. *IEEE Expert 11*, 6 (Dec. 1996), 24–29. http://cdr.stanford.edu/NextLink/Expert.html. (document), 4.9, 4.20

[76] PICCO, G. P. *Understanding, Evaluating, Formalizing and Exploiting Code Mobility*. PhD thesis, Politecnico di Torino, Dipartimento di Automatica e Informatica, 1998. 2.1.4, 2.1.4

[77] PICCO, G. P., CARZANIGA, A., AND VIGNA, G. Designing distributed applications with mobile code paradigms. In *Proceedings of the 19th International Conference on Software Engineering (ICSE'97)* (Boston (MA, USA), 1997), R. Taylor, Ed., ACM Press, pp. 22–32. 2.1.4

[78] RAO, A. S. Dynamics of belief systems: A philosophical, logical, and ai perspective. Tech. Rep. 02, Australian Artificial Intelligence Institute, Melbourne, Australia, July 1989. 1, 3.10, 4.8.5, 4.8.5

[79] RAO, A. S. AgentSpeak(L) : BDI agents speak out in a logical computable language. In *Proceedings of the 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World* (Berlin, Jan. 22–25 1996), W. V. de Velde and J. W. Perram, Eds., vol. 1038 of *LNAI*, Springer Verlag, pp. 42–55. 2, 3.1

[80] RAO, A. S., AND GEORGEFF, M. P. Deliberation and the formation of intentions. Tech. Rep. 10, Australian AI Institute, Carlton, Australia, 1990. 2, 2.1.6, 4.8

[81] RAO, A. S., AND GEORGEFF, M. P. Modeling rational agents within a BDI-architecture. In *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning* (San Mateo, CA, USA, Apr. 1991), J. Allen, R. Fikes, and E. Sandewall, Eds., Morgan Kaufmann Publishers, pp. 473–484. 1, 2, 2.3, 4.8

[82] RATIONAL SOFTWARE. *UML Notation Guide*. Rational Software, 1997. D

[83] RATIONAL SOFTWARE. *UML Semantics*. Rational Software, 1997. D

[84] RETICULAR SYSTEMS INC. AgentBuilder: An integrated toolkit for constructing intelligent software agents. White Paper, Feb. 1999. http://www.agentbuilder.com. 1, 3.2

[85] SHAW, M., AND GARLAN, D. *Software Architecture: Perspective on an Emerging Discipline*. Prentice-Hall, New Jersey, 1996. 1, 2.3

[86] SHOHAM, Y. Agent-oriented programming. *Artificial Intelligence 60* (Mar. 1993), 51–92. 1, 2, 2.3, 3.1, 3.2, 4.8

[87] SHOHAM, Y. An overview of agent-oriented programming. In *Software Agents* [9]. 2.1, B

[88] SLOMAN, A. What's and AI toolkit for? In *AAAI-98 Workshop on Software Tools for Developing Agents* (Madison, Wisconsin, USA, July 1998). (document), 1, 3, 3.10, 4

[89] SMITH, D. E., AND WELD, D. S. Incremental graphplan. Tech. Rep. UW-CSE-98-09-06, University of Washington, Sept. 1998. 2.1.6.1

[90] SMITH, G. A logic for Object-Z (additional rules). Tech. Rep. 95-26, Software Verification Research Centre - Departament of Computer Science, University of Queensland, Sept. 1995. 6.1.2

[91] SPIVEY, J. M. *The ZÑotation*, 2 ed. Prentice Hall International, UK, 1992. C, C.1

[92] SUN MICROSYSTEMS. *Java Beans(TM)*, July 1997. Graham Hamilton (ed.). Version 1.0.1. 3.5

[93] THE FOUNDATION FOR INTELLIGENT PHYSICAL AGENTS. The FIPA'97 specification. `http://drogo.cselt.it/fipa/spec/fipa97/fipa97.htm`, 1997. 2.1.3

[94] THOMAS, S. The placa agent programming language. In *ECAI-94 Workshop of Agent Theories, Architectures and Languages* (Aug. 1994), pp. 355–370. 3.1, 3.2

[95] VELOSO, M. Nonlinear problem solving using intelligent casual-commitment. Tech. Rep. CMU-CS-89-210, Carnegie Mellon University, Dec. 1989. (document), 2.4

[96] WAGNER, T., GARVEY, A., AND LESSER, V. Criteria-directed heuristic task scheduling. Technical Report UM-CS-1997-059, University of Massachusetts, Amherst, Computer Science, Oct., 1997. 3.5

[97] WEISS, G. Adaptation and learning in multi-agent systems - some remarks and a bibliography. In *Adaptation and Learning in Multi-Agent Systems*, G. Weiss and S. Sen, Eds., vol. 1042 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1996, ch. 1, pp. 1–21. 2.1.7

[98] WELD, D. S. An introduction to least commitment planning. *AI Magazine 15*, 4 (1994), 27–61. 1, 2.1.6, 2.1.6.1, 2.1.6.1, 4.8, 4.8.1, 4.8.3

[99] WHITE, J. E. Telescript technology: Mobile agents. In *Software Agents* [9]. 2.1, 2.1.4

[100] WOOLDRIDGE, M. Practical reasoning with procedural knowledge. In *Proceedings of the International Conference on Formal and Applied Practical Reasoning (FAPR-96)* (Berlin, June 3–7 1996), D. M. Gabbay and H. J. Ohlbach, Eds., vol. 1085 of *LNAI*, Springer, pp. 663–678. 2, 4.8.1

[101] WOOLDRIDGE, M. Agent-based software engineering. *IEE Proceedings Software Engineering 144*, 1 (1997), 26–37. 2.2

[102] WOOLDRIDGE, M. Agents and software engineering. *AI*IA Notizie XI*, 3 (Sept. 1998), 31–37. (document), 1, 2.2

[103] WOOLDRIDGE, M. Intelligent agents. In *Multiagent Systems*, G. Weiss, Ed. The MIT Press, 1999, ch. 1. 2, 2.1.2, 2.1.5, 8

[104] WOOLDRIDGE, M., AND JENNINGS, N. R. Intelligent agents: Theory and practice. *Knowledge Engineering Review 10*, 2 (Jan. 1995). 1, 2.1, 2.3, 3.10

[105] WOOLDRIDGE, M., AND JENNINGS, N. R. Software agents. *IEE Review* (Jan. 1996), 17–20. 1, 2.1

[106] WOOLDRIDGE, M., AND JENNINGS, N. R. Pitfalls of agent-oriented development. In *Proceedings of the 2nd International Conference on Autonomous Agents (AGENTS-98)* (New York, May 9–13 1998), K. P. Sycara and M. Wooldridge, Eds., ACM Press, pp. 385–391. (document), 1, 3

[107] WORDSWORTH, J. B. *Software Development With Z - A Practical Approach to Formal Methods in Software Engineering*. International Computer Science Series. Addison-Wesley, 1992. C, C.1

[108] YANG, Q. *Intelligent Planning: A Decomposition and Abstraction Based Approach*. Artificial Intelligence. Springer-Verlag, New York, 1997. 4.8.1, 4.8.3

[109] ZWEBEN, M., AND FOX, M. S. *Intelligent Scheduling*. Morgan Kaufmann, 1994. 1, 2.1.6, 4.8