

## 10. Jade y Jess

Copyright (c) 2025 Adrián Quiroga Linares Lectura y referencia permitidas; reutilización y plagio prohibidos

### 10.1 Introducción y Arquitectura Conceptual

El objetivo es comprender cómo dotar a un agente de software de capacidad de **razonamiento declarativo**. Mientras que JADE proporciona la infraestructura para la comunicación y el ciclo de vida del agente, Jess actúa como el "cerebro" lógico.

#### 10.1.1 El Modelo de Agente Relacional

Podemos visualizar el agente integrado mediante un modelo cibernetico clásico:

- **Entorno (environment)**: el mundo externo donde vive el agente
- **Sensores (Sensors)**: mecanismos que reciben **perceptos** (entradas, como mensajes ACL)
- **Agente (Agent)**: La entidad computacional. Aquí es donde ocurre la integración:
  - **JADE**: Gestiona la interacción con el entorno (envío/recepción de mensajes)
  - **Jess**: Implementa el módulo de decisión de forma declarativa (reglas lógicas).
- **Actuadores (Actuators)**: mecanismos que ejecutan **acciones** sobre el entorno (ej. enviar una respuesta)

#### 10.1.2 ¿Por qué integrar Jess?

Jess es un motor de reglas ligero y rápido escrito en Java. Permite construir software que "razona" usando conocimiento suministrado en forma de reglas declarativas, en lugar de código imperativo (if-then-else anidados).

- **Programación normal (Java/JADE sin Jess)**: Es como dar órdenes a un soldado. Tienes que decirle paso a paso: "*Si ves un enemigo, levanta el arma. Si está a menos de 50 metros, dispara. Si no, espera*". Si la situación cambia y no prevista en tus instrucciones, el soldado se bloquea.
- **Motor de Inferencia (Jess)**: Es el estratega. Tú no le dices qué hacer paso a paso. Tú le das **Reglas** ("Los enemigos cercanos son peligrosos") y **Hechos** ("Hay un tanque a 20 metros"). El estratega *deduce* por sí mismo: "Debo atacar".

#### Nota

Es como usar un sistema experto de los de IA pero con agentes

## 10.2 Desafíos Técnicos de la Integración

Integrar un motor de inferencia en un agente JADE no es trivial debido a la naturaleza de la concurrencia en ambos sistemas.

### 10.2.1 El Modelo de Hilos

- **JADE:** Un agente JADE es, por principio, **monohilo (single-threaded)**. Esto simplifica la gestión del estado pero impone restricciones de bloqueo.
- **Jess (Clase Rete):** Para embeber Jess, instanciamos la clase `jess.Rete`. El método estándar de ejecución es `Rete.run()`.
- **El Problema:** El método `Rete.run()` ejecuta reglas consecutivamente y **bloquea el hilo de llamada** hasta que no quedan más reglas por disparar.

**Advertencia Crítica:** Si ejecutamos `Rete.run()` sin control dentro de un agente JADE, bloquearemos al agente completo. Si el razonamiento tarda mucho, el agente dejará de percibir mensajes y atender otras tareas.

### 10.2.2 Solución: Ejecución Intercalada

Para evitar el bloqueo total, utilizamos una variante del método `run` que acepta un límite de ciclos: `jess.run(int maxCycles)`. Esto permite que el agente rzone "un poco", atienda otras tareas (como leer mensajes), y vuelva a razonar.

## 10.3 Implementación: El `JessBehaviour`

La arquitectura recomendada encapsula el motor Jess dentro de un comportamiento cíclico de JADE (`CyclicBehaviour`).

### 10.3.1 Estructura del Comportamiento

El `JessBehaviour` debe gestionar la carga del motor y su ciclo de ejecución.

#### 1. Inicialización (Constructor):

- Se crea la instancia de `jess.Rete`.
- Se carga el archivo de reglas (`.clp`) usando un `FileReader` y el parser `jess.Jesp`.

#### 2. Ciclo de Ejecución (Método `action()`):

- Se define una constante `MAX_JESS_PASSES` (ej. 1) para limitar el tiempo de CPU dedicado a razonar en cada vuelta.
- Se invoca `jess.run(MAX_JESS_PASSES)`.
- **Gestión de inactividad:** Se compara el número de pases ejecutados con el máximo permitido. Si `ejecutados < maximo`, significa que el motor se detuvo

porque **no hay más reglas aplicables**. En este caso, debemos bloquear el comportamiento (**block()**) para no consumir CPU inútilmente.

## 10.3.2 Inserción de Hechos (Facts)

Para que el motor razoné, necesita datos. Debemos proveer métodos para insertar hechos y despertar al comportamiento si estaba dormido.

**Ejemplo de método auxiliar `addFact`:**

```
JAVA
boolean addFact(String jessFact) {
    try {
        jess.assertString(jessFact); // Insertar hecho en Jess [cite: 164]
    } catch (JessException je) { return false; }

    if (!isRunnable()) restart(); // Si el comportamiento estaba bloqueado,
despertarlo [cite: 174]
    return true;
}
```

Nota: Es fundamental llamar a `restart()` (o un mecanismo equivalente para desbloquear) después de insertar un hecho, ya que nuevas premisas podrían activar nuevas reglas.

## 10.4 Intercambio de Información: De JADE a Jess

El flujo de información desde el entorno hacia el motor de reglas suele ocurrir a través de mensajes ACL.

### 10.4.1 Estrategia Básica: Traducción Manual

Se puede implementar un comportamiento (ej. `MsgListening`) que reciba mensajes ACL, extraiga su contenido y construya una cadena de texto (String) con formato de hecho Jess para inyectarlo mediante `addFact`.

### 10.4.2 Estrategia Avanzada: Shadow Facts (Java Beans)

Jess permite mapear objetos Java directamente a "Shadow Facts" en su memoria de trabajo, evitando la traducción manual a String.

- Se define un `deftemplate` en Jess apuntando a la clase Java: `(deftemplate ACLMessage (declare (from-class ACLMessage)))`.
- Se añade el objeto con `Rete.add(objeto)`.

**El Problema de `ACLMensaje`:** La clase `ACLMensaje` de JADE no cumple estrictamente con el estándar JavaBeans necesario para Jess. Específicamente, el campo `content` es un `StringBuffer`, pero sus métodos get/set trabajan con `String`. Esto hace que Jess falle al intentar inspeccionar el objeto automáticamente.

**Solución:** Crear una clase envoltorio o extendida, por ejemplo `MyACLMensaje`, que corrija los accesos a los campos para ser compatible con la introspección de Java Beans .

## 10.5 Intercambio de Información: De Jess a JADE (Acciones)

Una vez que Jess ha razonado, el agente debe actuar (generalmente enviando un mensaje). Existen tres niveles de integración:

### 10.5.1 Post-procesamiento (Nivel Bajo)

El agente verifica la memoria de trabajo de Jess después de cada ejecución para ver si hay "hechos de salida" y actúa en consecuencia. Esto es pasivo y poco eficiente.

### 10.5.2 `Userfunction` (Nivel Medio - Recomendado)

Se crea una clase Java que implementa la interfaz `jess.Userfunction` (ej. `JessSend`). Esta clase actúa como un puente.

1. **Implementación:** El método `call` de la clase Java recibe argumentos desde Jess, construye un `ACLMensaje` y usa `myAgent.send()` para enviarlo .
2. **Registro:** Se registra la función en el motor: `jess.addUserfunction(new JessSend(myAgent))` .
3. **Uso:** En el código Jess (reglas), se llama directamente: `(send ...)` .

### 10.5.3 Acceso Directo al Agente (Nivel Alto)

Se puede insertar el propio objeto `Agent` dentro del motor Jess usando `Rete.add(Agent)` . Esto permite invocar cualquier método público del agente (incluyendo `send`) directamente desde el código de las reglas usando la sintaxis de reflexión de Jess.

## 10.6 Conclusión de la Asignatura

Una vez más se demuestra que el grado presenta una dejadez considerable por parte de las "vacas sagradas" de la facultad. Y es que no puede ser: estas personas son reconocidísimas en sus ámbitos, publican 300 artículos, acuden a congresos internacionales... pero cuando llega el momento de pensar en sus alumnos y proporcionarles una bibliografía de calidad con la cual puedan aprender los conceptos de forma intuitiva, te sueltan un PDF de 2001 robado de otra universidad o

presentaciones sin orden lógico alguno, llenas de palabras sueltas y esquemas incomprensibles.

Eso sí, si te quejas, la alternativa que te ofrecen es el libro de turno de 1990 escrito en alemán por un médico francés. Y apáñate tú para conseguirlo y entenderlo.

**La situación es insostenible.** Es frecuente encontrarse con:

- Documentación completamente desactualizada
- Materiales de dudosa procedencia sin adaptar al contexto actual
- Presentaciones caóticas sin estructura pedagógica
- Bibliografía obsoleta, inaccesible o en idiomas que nadie domina

**¿Tan difícil es hacer las cosas bien?** Otras carreras universitarias lo consiguen. No se pide la perfección, pero sí un mínimo de coherencia y actualización en los materiales docentes.

Considero que el grado debería aprender de otras titulaciones y mejorar urgentemente este aspecto. Con la jubilación progresiva de las viejas vacas sagradas de la universidad, espero sinceramente que esta situación mejore con el tiempo y que la nueva generación de profesorado traiga una renovación real en la calidad docente, no solo en la investigadora.