

Versión  
7 de abril de 2009

# LÓGICA PRÁCTICA Y APRENDIZAJE COMPUTACIONAL

Jacinto A. Dávila Q.

UNIVERSIDAD DE LOS ANDES  
MÉRIDA, VENEZUELA  
Mérida, Julio 2009



# Índice general

---

<b>Índice general</b>	<b>VI</b>
<b>Licencia de uso</b>	<b>VII</b>
<b>1. Lógica Práctica</b>	<b>3</b>
1.1. Introducción . . . . .	3
1.2. La semántica de la lógica práctica . . . . .	3
1.2.1. Sobre la validez de un argumento . . . . .	5
1.3. Una realización computacional. . . . .	7
1.4. Lógica y Agentes . . . . .	8
1.4.1. Una Teoría de Agentes en Lógica . . . . .	8
1.4.2. Una Práctica de Agentes en Lógica . . . . .	9
<b>2. Simulación con Agentes y Lógica</b>	<b>25</b>
2.1. Directo al grano . . . . .	25
2.2. El primer modelo computacional . . . . .	25
2.2.1. Nodos: componentes del sistema a simular . . . . .	26
2.2.2. El esqueleto de un modelo Galatea en Java . . . . .	28
2.2.3. El método principal del simulador . . . . .	29
2.2.4. Cómo simular con Galatea, primera aproximación . . . . .	31
2.3. El primer modelo computacional multi-agente . . . . .	33
2.3.1. Un problema de sistemas multi-agentes . . . . .	33
2.3.2. Conceptos básicos de modelado multi-agente . . . . .	34
2.3.3. El modelo del ambiente . . . . .	35
2.3.4. El modelo de cada agente . . . . .	39
2.3.5. La interfaz Galatea: primera aproximación . . . . .	41
2.3.6. El programa principal del simulador multi-agente . . . . .	42
2.3.7. Cómo simular con Galatea. Un juego multi-agente . . . . .	43
2.4. Asuntos pendientes . . . . .	43
<b>Referencias</b>	<b>45</b>
<b>A. De la mano con Java</b>	<b>47</b>
A.1. Java desde Cero . . . . .	47
A.1.1. Cómo instalar <code>java</code> . . . . .	47
A.1.2. Cómo instalar <code>galatea</code> . . . . .	48
A.1.3. Hola mundo empaquetado . . . . .	48
A.1.4. Inducción rápida a la orientación por objetos . . . . .	49

A.1.5.	Qué es un objeto (de software)	50
A.1.6.	Qué es una clase	50
A.1.7.	Qué es una subclase	50
A.1.8.	Qué es una superclase	51
A.1.9.	Qué es poliformismo	52
A.1.10.	Qué es un agente	53
A.1.11.	Qué relación hay entre un objeto y un agente	53
A.1.12.	Por qué objetos	54
A.2.	Java en 24 horas	54
A.2.1.	Java de bolsillo	54
A.2.2.	Los paquetes Java	56
A.2.3.	Java Libre	57
A.2.4.	Un applet	57
A.2.5.	Anatomía de un applet	58
A.2.6.	Parametrizando un applet	59
A.2.7.	Objetos URL	59
A.2.8.	Gráficos: colores	60
A.2.9.	Ejecutando el applet	60
A.2.10.	Capturando una imagen	61
A.2.11.	Eventos	61
A.2.12.	<code>paint()</code> :Pintando el applet	63
A.2.13.	El AppletMonteCarlo completo	63

# Licencia de uso

---

Este libro contiene un cúmulo de información referencial sobre los sistemas Galatea<sup>1</sup>, Gloria<sup>2</sup> y Bioinformantes<sup>3</sup>.

Copyright © ©2009 Jacinto Dávila. Universidad de Los Andes, Venezuela.

Se concede permiso de copiar, distribuir o modificar este documento bajo los términos establecidos por la licencia de documentación de GNU, GFDL, Version 1.2 publicada por la Free Software Foundation en los Estados Unidos, siempre que se mantengan invariables los títulos de secciones y se preserven los nombres en las portadas (explicando los cambios respecto al original). Una copia de esta licencia se incluye al final del documento en el capítulo “GNU Free Documentation License”. Nos apegaremos a esta licencia siempre que no contradiga los términos establecidos en la legislación correspondiente de la República Bolivariana de Venezuela.

Según establece GFDL, se permite a cualquier modificar y redistribuir este material y los autores originales confiamos que otros crean apropiado y provechoso hacerlo. Esto incluye traducciones, bien a otros lenguajes naturales o a otros medios electrónicos o no.

En nuestro entender de GFDL, cualquiera puede extraer fragmentos de este texto y usarlos en un nuevo documento, siempre que el nuevo documento se acoja también a GFDL y sólo si se mantienen los créditos correspondientes a los autores originales (tal como establece la licencia).

---

<sup>1</sup><http://galatea.sourceforge.net>

<sup>2</sup><http://gloria.sourceforge.net>

<sup>3</sup><http://simulants.wiki.sourceforge.net>

Opt



# Lógica Práctica

---

## 1.1. Introducción

Para decidir si la lógica es útil o no uno tendría que producir un **argumento** (en favor o en contra). Entre lógicos se dice, a modo de broma, que esta es la mejor **prueba** de la utilidad de la lógica. Muchas veces, sin embargo, no ocurre que se dude de la utilidad de la lógica en sí misma, sino que las formas de lógica disponibles tienen tal nivel de complejidad que su utilización **práctica** (distinta de su utilización en la especulación teórica) es muy limitada.

La lógica que se propone en este texto tiene 2 características que nos permitimos ofrecer para justificarle el atributo de práctica:

1. Una **semántica** relativamente simple.
2. Una realización computacional.

## 1.2. La semántica de la lógica práctica

En la **lógica matemática** clásica, el mundo (el universo entero y toda otra posibilidad) se ve reducido a un conjunto de objetos y a un conjunto de relaciones (entre los mismos objetos o entre las mismas relaciones). Eso es todo lo que existe (o puede existir): objetos y relaciones.

Esta **ontología** degenerada (respecto a la nuestra) de la lógica clásica es frecuentemente criticada por **reduccionista**. Los **argumentos** para esas críticas, muchos ampliamente aceptados como válidos, son tolerados (o mejor, ignorados) por los lógicos en virtud de una gran ventaja que ofrece esa ontología: Es menos compleja y, por tanto, es más fácil pensar con ella (o respecto a ella).

En definitiva, para darle significados a los símbolos en lógica contamos con:

**OBJETOS** designados por sus nombres, llamados también **términos** del lenguaje y

**RELACIONES** cuyos símbolos correspondientes son los nombres de predicados o, simplemente, predicados.

Por ejemplo, el símbolo **ama(romeo, julieta)** es un átomo constituido sintácticamente por el predicado **ama** y sus dos argumentos, los términos **romeo** y **julieta**.

Para saber qué significa esos términos y ese predicado, uno tiene que proveerse de una definición de la **semántica** de ese lenguaje. Una forma de hacerlo es usar (referirse a) una historia (como la que sugieren esos nombres) o, un poco más matemáticamente, definirlos:

`romeo` se refiere al jovencito hijo de los Montesco, en la Novela de William Shakespeare.

`julieta` se refiere a la jovencita, hija de los Capuletos, en la misma Novela.

Hay muchas formas de hacer esa definición <sup>(1)</sup>. En cada caso, quien la hace tiene en mente su propia **interpretación**. Uno podría, por ejemplo, decir:

`romeo` es el perrito de mi vecina.

y no habría razón para decir que esa interpretación es incorrecta.

Esas definiciones dan cuenta del nombre, pero no del predicado. Este se define también, pero de una forma diferente. Al predicado le corresponde una **relación**. Si hacemos la interpretación que pretende Shakespeare, por ejemplo, la relación correspondiente es aquella en la que el objeto en el primer argumento ama al objeto en el segundo. Es decir, los matemáticos definen la relación *amar* *a* como la colección de todas las duplas en las que X ama a Y. Los computistas, siguiéndoles la corriente, dicen que eso es muy parecido a tener una **tabla** en una **base de datos** que describe todo lo que es **verdad**.

Para saber si Romeo, (el de la historia) realmente ama a Julieta (la de la historia), uno tendría que revisar la base de datos (correspondiente a la historia) y verificar que contiene la tupla correspondiente. Noten, por favor, una sutileza. Esa no es una base de datos de palabras o términos electrónicos. Es, de hecho (dicen los lógicos clásicos), una reunión (¿imaginaria?) de **objetos reales**.

Por simplicidad (y sin olvidar lo que acabo de decir), uno puede representarla así:

```
AMA amante amado
  romeo julieta
  julieta romeo
  bolívar colombia
  manuela bolívar
  ...      ...
```

y, colapsando todo en palabras (en la sintáxis), así:

```
1 ama(romeo, julieta)
2 ama(julieta, romeo).
3 ama(bolivar, colombia).
4 ama(manuela, bolivar).
5 ...
```

(noten que uno puede anotar esa información de muchas maneras diferentes, e.g. *romeo ama a julieta*).

Esta explicación dispensa con algunos elementos básicos de la semántica de la lógica. Pero no con todos. Faltan los más útiles. Las llamadas **palabras lógicas**.

En la lógica, uno puede crear **oraciones compuestas** a partir de oraciones simples (como `ama(romeo, julieta)`).

Digamos que usamos los símbolos **y**, **o**, **no**, **si** (siempre acompañado de **entonces**) y **si y solo si**. (los símbolos empleados son también una decisión del diseñador del lenguaje. Algunas veces se usan  $\wedge$ ,  $\vee$ ,  $\neg$ ,  $\rightarrow$ ,  $\leftrightarrow$  y muchos otros).

Sabemos que `ama(romeo, julieta)` es verdadera al revisar la *base de datos de la interpretación* que se haya seleccionado. Pero, ¿cómo sabemos que `ama(romeo, julieta)` y `ama(julieta, romeo)` es verdadera?

p	q	p y q
v	v	v
f	v	f
v	f	f
f	f	f

Cuadro 1.1: La semántica de **y**

La respuesta está codificada en la **tabla de verdad** de la palabra **y**:

Si podemos decir que  $p = \text{ama}(\text{romeo}, \text{julieta}) = \text{verdad}$  y  $q = \text{ama}(\text{julieta}, \text{romeo}) = \text{verdad}$ , entonces la **conjunción** es también verdad **en esa interpretación**.

Esas tablas de verdad son útiles porque nos permiten decidir si un discurso, visto como una (gran) fórmula lógica, es cierto o falso. Esto es importante para verificar si tal discurso es coherente o, dicho de otra manera, está razonablemente estructurado. En lógica, cuando el discurso es verdadero en alguna interpretación decimos que es un **discurso consistente**.

Un discurso o teoría es inconsistente si permite derivar una proposición y su opuesta: e.g. te amo y no te amo.

### 1.2.1. Sobre la validez de un argumento

Esos ejercicios de análisis de significados con tablas de verdad tienen otro sentido práctico. Se les puede usar para evaluar la validez de un argumento.

Un argumento es válido si siempre que sus condiciones son ciertas, sus conclusiones también son ciertas.

**Argumento** es uno de los conceptos trascendentales en lógica. Se refiere a toda secuencia de fórmulas lógicas, conectadas entre sí (por relaciones lógicas), que establece (**prueba**) una conclusión. Un ejemplo muy conocido de argumento es este:

Todo humano es mortal. Sócrates es humano. Por lo tanto, Sócrates es mortal.

que puede ser codificado y usado por un computador en esta forma:

```
mortal(X) :- humano(X).
humano(socrates).
```

```
?mortal(X)
X = socrates
Yes
```

¿Ve lo que ocurre en el ejemplo?

Para verificar que un argumento es válido basta verificar que **CUANDOQUIERA** que sus condiciones son todas ciertas, sus conclusiones también (son todas ciertas). Vale la pena tomar un tiempo para aclarar el concepto. Algunas personas tienen dificultad para aceptar que un argumento puede ser válido (aún) si:

<sup>1</sup>vean [http://es.wikipedia.org/wiki/Romeo\\_y\\_Julieta](http://es.wikipedia.org/wiki/Romeo_y_Julieta)

sus condiciones son falsas y sus conclusiones son ciertas o  
sus condiciones son falsas y sus conclusiones son falsas también.

De hecho, lo único que **NO** puede ocurrir es que

sus condiciones sean ciertas y sus conclusiones falsas.

Recapitulando, un argumento es válido si cuandoquiera que sus premisas (condiciones) son conjuntamente ciertas, su conclusión (o conclusiones) también son ciertas.

Esta definición, como suele pasar en matemática, sirve para describir, pero no para producir o justificar un argumento válido. El que podamos construir un argumento válido va a depender, siempre, de que podamos establecer la conexión adecuada entre sus premisas y sus conclusiones.

En particular, la validez de un argumento **NO DEPENDE** de los valores de verdad de sus componentes. Eso es lo que verificamos con los siguientes ejemplos (basados en los ejemplos de libro de Irving Copi y Carl Cohen)[1].

Todas las ballenas son mamíferos. Todos los mamíferos tienen pulmones. Por lo tanto, todas las ballenas tienen pulmones.

Este es un argumento válido, con condición cierta y conclusión cierta.

Todas las arañas tienen diez patas. Todas las criaturas de diez patas tienen alas. Por lo tanto, todas las arañas tienen alas.

Este **también** es un argumento válido, con condiciones falsas y conclusión falsa.

Noten que la falsedad de la primera condición y de conclusión es atribuida por nuestro conocimiento de biología (*Animal Planet*, decimos en clase). Decir que la segunda condición es falsa es un poco temerario, puesto que no conocemos criaturas de diez patas (o ¿sí?), pero realmente no importa porque la conjunción se falsifica con la primera condición falsa.

Si yo tuviera todo el dinero de Bill Gates, sería rico. No tengo todo el dinero de Bill Gates. Por lo tanto, no soy rico.

Este es un argumento **inválido** (es decir, no válido) con condiciones ciertas y conclusión cierta (y le juro que es absolutamente cierta si habla del autor).

¿Por qué es inválido?. Piénselo.

Si Bin Laden tuviera todo el dinero de Bill Gates, sería rico. Bin Laden no tiene todo el dinero de Bill Gates. Por lo tanto, Bin Laden no es rico.

Este también es **inválido**, pero sus condiciones son ciertas (suponiendo que Bill es más adinerado) y su conclusión es falsa (por lo que sabemos del origen acomodado de Bin Laden. Alguien podría opinar lo contrario).

¿Será este argumento inválido por la misma razón que el anterior?. Piénselo.

Todos los peces son mamíferos. Todas las ballenas son peces. Por lo tanto, todas las ballenas son mamíferos.

Este es válido, con condiciones (ambas) falsas y conclusión cierta (esta última es otra cosa que sabemos gracias a *Animal Planet*).

¿Podemos llegar a la conclusión de validez, olvidándonos de peces, ballenas y mamíferos?  
Piénselo.

Todos los mamíferos tienen alas. Todas las ballenas tienen alas. Por lo tanto, todas las ballenas son mamíferos.

Este es inválido, con condiciones falsas y condición cierta.

Todos los mamíferos tienen alas. Todas las ballenas tienen alas. Por lo tanto, todos los mamíferos son ballenas.

Este es inválido, con condiciones falsas y conclusión también falsa.

¿Qué podemos decir de esos 7 argumentos?. ¿En qué nos basamos para el análisis?. ¿Qué ocurre con la 8va. posibilidad?. ¿Cuál es la combinación faltante?. ¿Cuál es la relación entre la validez de un argumento y la tabla de verdad de las fórmulas **si .. entonces**?.

Si este es un libro guía, es razonable que el lector reclame las respuestas a estas preguntas. Descubrir las por sí solo o sola, sin embargo, será mucho más productivo. Veamos la otra ventaja interesante de la lógica que se describe en el texto.

### 1.3. Una realización computacional.

En lógica clásica se cultivan separadamente esas estructuras semánticas que hemos discutido en la sección anterior, bajo el nombre de **Teorías de Modelos**. Un **modelo** es una interpretación lógica que hace cierta a una fórmula. Por ejemplo, **p** y **q** es satisfecha si **p = verdad** y **q = verdad**. Por tanto, **esa asignación** es un modelo.

La otra parte de la lógica matemática clásica la constituyen las llamadas **Teorías de Prueba** (*Proof Theories*). En lugar de girar en torno a la noción de **DEDUCCION**, como se hace al analizar argumentos y tablas de verdad, las teorías de prueba se concentran en describir ejercicios y estrategias de **DERIVACION**.

Derivar es obtener una fórmula a partir de otras a través de transformaciones sintácticas. También se usa el término **INFERIR**.

X es P y P es M

entonces

X es M

Una realización computacional de un razonador lógico es un programa que hace justo eso: inferir. Deriva unas fórmulas a partir de otras. Así, para el computista lógico, derivar es computar. El razonamiento lógico es una forma de computación (si bien, no la única).

A los razonadores lógicos se les llama de muchas maneras: motores de inferencia, probadores de teoremas, máquinas lógicas, método de prueba. En todo esos casos, el razonador tiene sus propias reglas para saber que hacer con las fórmulas lógicas (que también suelen ser reglas). A las reglas de un motor de inferencia se les llama: reglas de inferencia. He aquí una muy popular que hasta tiene nombre propio: **modus ponens**

Con una fórmula como:

Si A entonces B

y otra fórmula como:

A

Derive

B

¿Puede ver que se trata de una regla para procesar reglas?. Por eso, algunas veces se dice que es una **meta-regla**.

Las teorías de modelos y las teorías de prueba normalmente caracterizan a la lógica clásica (a una lógica o a cada lógica, si uno prefiere hablar de cada realización por separado). Han sido objeto de intenso debate durante largo tiempo y mucho antes de que existieran los actuales computadores. La lógica computacional, sin embargo, ha abierto un nuevo espacio para describir el propio lenguaje.

En la siguiente sección conoceremos de un primer intento por explicar el alcance expresivo de la lógica computacional y conoceremos también una lista de ejemplos de agentes modelados con y en lógica.

## 1.4. Lógica y Agentes

### 1.4.1. Una Teoría de Agentes en Lógica

Hace más de una década, en la escuela de la **programación lógica**, emprendimos un proyecto colectivo para caracterizar a la nueva noción de Agente que irrumpió en varios escenarios de la Inteligencia Artificial. Robert Kowalski es uno de los mejores exponentes de ese proyecto, con su libro *Cómo ser artificialmente inteligente*<sup>2</sup>. El objetivo de Kowalski es explorar el uso de la lógica para modelar agentes. Eso lo ha llevado a bosquejar una teoría para explicar qué es un agente y cómo se le puede implementar para efectos de computación. Kowalski ha tratado de aproximar esa misma noción de agente que se ha hecho muy popular en el mundo tecnológico en las últimas décadas.

Resumiendo esa teoría, uno puede decir que un **agente** es una pieza de software para controlar un dispositivo capaz de interactuar con su entorno, percibiendo y produciendo cambios en ese entorno. La **especificación lógica** es más general que esa visión resumida y orientada a la máquina. Un agente, en esa teoría lógica, es un proceso auto sostenido y auto dirigido de intercambios entre un estado interno y un estado externo. El proceso es modelado como un programa lógico que puede convertirse en código ejecutable sobre un hardware o en otros casos, decimos nosotros, sobre **bioware**. En este sentido, la conceptualización alcanza a describir aspectos de la conducta autónoma de los humanos y otros seres en los que es posible o conveniente distinguir un estado interno de uno externo. Es el ahora clásico ejercicio de *postura intencional*[2], según la cual la condición de agente es determinada por el ojo del observador.

La teoría renuncia a toda pretensión totalizante: Un agente no es solamente lo que allí se describe y, más aún, nunca se pretende que los humanos (y otros seres conscientes) seamos eso únicamente (pueden verse declaraciones al respecto al principio de los capítulos 1, 6 y 7 del texto de Kowalski).

Sin embargo, esa teoría sí pretende restaurar aquella intención originaria de la lógica: *La lógica simbólica fue originalmente desarrollada como un herramienta para mejorar el razonamiento humano*. Ese es el sentido práctico, de utilidad, que se pretende rescatar con la teoría y que, necesariamente, debe estar asociado a una práctica.

---

<sup>2</sup>publicado libremente en Internet y disponible en español en <http://webdelprofesor.ula.ve/ingenieria/jacinto/kowalski/logica-de-agentes.html> y como anexo de este texto

```

1 Si alguien me pide algo entonces yo sigo el procedimiento.
2
3 Si alguien me pide algo y es algo muy importante,
4     resuelvo inmediatamente.
5
6 Para seguir el procedimiento, consulto el manual si existe
7     o invento un manual si no existe.
8
9 Para inventar un manual de procedimiento,
10     solicite una carta de alguna autoridad y
11     haga lentamente lo que allí dice
12     (para evitar errores).
13
14 Todo procedimiento se sigue lentamente
15     para evitar errores y trampas.
16
17 Algo es muy importante si se lo menciona en las leyes.
18
19 Para las cosas importantes
20     existen manuales de procedimiento.

```

Figura 1.1: Burocratín

```

1 Si necesito información de burocratín
2     entonces se la pido apropiadamente
3
4 Para pedir información apropiadamente (a burocratín),
5     se lo pido y le explico que es muy importante.
6
7 Para explicarle que es muy importante,
8     antes encuentro una referencias del asunto
9     en alguna de las leyes y
10    la menciono con pasión.

```

Figura 1.2: El Usuario frente a Burocratín

### 1.4.2. Una Práctica de Agentes en Lógica

Inspirados por la teoría de Kowalski y por su afán de demostrar que trasciende los símbolos y las operaciones matemáticas, hemos venido recolectando ejemplos particulares de especificaciones, siempre parciales, de agentes en lógica. Cada uno de los ejemplos trata de capturar antes que un agente completo, el rol (papel, partitura) que desempeña cierto agente en cierta circunstancia de interés. El objetivo es, siempre, explicar a un agente y a su forma de pensar sin que la representación matemática interfiera (demasiado).

#### Burocratín

En la figura 1.1 se muestran las reglas que cierto empleado público en Venezuela, llamado Burocratín, emplearía para decidir cómo atender a un usuario de su servicio:

La situación problema es una en la que Ud necesita que Burocratín le suministre información con cierta urgencia. Debemos explicar las reglas de conducta seguiríamos para hacer que Burocratín le responda en el menor tiempo posible. Algo como la figura 1.2:

Con estas reglas, el usuario (uno de nosotros), de *necesitar información de burocratín*, ubicaría

1	Si se acerca un ciudadano en su vehículo
2	entonces lo interpele.
3	
4	Para interpele a un ciudadano,
5	debo pedirle su cédula, su licencia,
6	su certificado médico, el carnet de circulación y
7	le pido que explique de donde viene y adonde va.
8	
9	Si el ciudadano se pone nervioso, le retiro sus
10	documentos por un tiempo,
11	examino con cuidado el vehículo y
12	los documentos y luego lo interpele.
13	
14	Si logro establecer una falta o violación de la ley,
15	le explico el castigo y le permito negociar.
16	
17	Si no negocia, ejecuto el castigo.
18	
19	Si negociando, el ciudadano propone una ayuda mutua,
20	la acepto y le dejo pasar.

Figura 1.3: Matraquín

*antes una referencia al tema en alguna de las leyes, le pediría la información a burocratín y mencionaría con pasión la referencia legal. Eso obligaría a Burocratín a resolver inmediatamente, una acción, por cierto, que no está definida en sus reglas (y que, por tanto, puede significar más problemas. Buena Suerte)*

### Matraquín

Matraquín es un agente que vive en un Universo similar al nuestro en donde sirve como fiscal de tránsito, mientras porta un arma. Sus reglas de conducta incluyen las que se muestran en la figura 1.3:

¿Qué pasaría si Ud, conduciendo y desarmado, se encuentra con Matraquín, tiene una falta evidente y Ud es de quienes creen que no se puede negociar desarmado con quienes portan armas?.

Digamos que esto es lo que me ocurre a mí: 1) (En cualquier caso) Matraquín me somete a la primera interpeleación, al observar que me acerco. 2) Yo observo el arma y, siguiendo la mencionada creencia, no discuto (entiendo que eso significa que no negocio), pero (supongamos) tampoco me pongo nervioso. 3) Matraquín observa mi evidente falla y procede a explicarme el castigo y me deja negociar. 4) Yo ya había decidido no negociar, así que guardo silencio. 5) Matraquín observa mi silencio y procede a ejecutar el castigo. El significado de esto último queda para la imaginación del lector.

Obviamente esta no es la única forma de **desenlazar la historia**. Todo depende de cuáles otras suposiciones hagamos sobre Matraquín y nosotros mismos. Pero parece que ya hay razones suficientes para preocuparse por esta clase de sistema multi-agentes, ¿verdad?. Considere ahora este otro escenario:

¿Qué pasaría si Ud, conduciendo, se encuentra con Matraquín, tiene una falta evidente y tiene Ud mucha prisa?.

```

1 METAS:
2 si hay hembras cerca , establece tu autoridad .
3
4 si alguna hembra necesita ayuda , sondéala .
5
6 si la hembra que necesita ayuda se muestra receptiva ,
7     lánzate .
8
9 si se aproxima una hembra , abórdala .
10
11 si luego de abordar a una hembra se muestra receptiva ,
12     lánzate .
13
14 si se quiebra , suéltaselo .
15
16 CREENCIAS:
17
18 Para establecer tu autoridad , haz una bravuconada .
19
20 Para abordar a una hembra ,
21     convérsale sobre cualquier cosa .
22
23 Para sondearla , abórdala .
24
25 Para lanzarte , invítala a salir .
26
27 Para soltárselo , invítala a ....

```

Figura 1.4: Machotín

1) Matraquín me somete a la interpelación (de rigor). 2) Esta vez no tengo aquella regla que me impide negociar con una persona armada. Supongamos que trato de explicar mi situación y, supongamos también, que tengo suerte de que Matraquín no interprete que estoy nervioso. 3) Matraquín observa mi evidente falla y procede a explicarme el castigo y me deja negociar. 4) Yo le explico (supongamos que calmadamente) que necesito continuar mi viaje y le pido que ayude!. 5) Matraquín entiende que estoy negociando y me propone que lo ayude a ayudarme. 6) Yo me muestro dispuesto a la ayuda mutua. 7) y 8) (*estas acciones deben ser censuradas*) . 9) Matraquín acepta mi parte de la ayuda mutua y me deja pasar.

¿Cuál es la MORALEja de la historia?

### Machotín

Considere la siguiente especificación parcial e informal de un agente que llamamos Machotín en figura 1.4:

Imagine que queremos escribir una historia en la que participe Machotín y en la que se activen, cada una en algún momento, todas sus **metas de mantenimiento**. Nos piden que expliquemos, en esa historia, qué está pensando, observando y haciendo Machotín en cada momento.

Considere la siguiente respuesta:

“Machotin es un agente en busca de una hembra, bien sea para invitarla a salir o para lanzarse en una aventura con ella. Machotin anda por la calle y observa que hay hembras cerca y establece su autoridad. Para establecer su autoridad, hace una bravuconada con su

pinta de sobrado. Mientras camina con su actitud altiva, observa que una hembra necesita ayuda. La sondea, abordándola y sacándole conversación mientras le ofrece ayuda. La hembra en cuestión se muestra receptiva. Entonces, Machotin aprovecha la oportunidad de lanzarse con una invitación a salir. Luego de cuadrada la cita, Machotin continua su recorrido y observa que se aproxima una hembra. La aborda. Conversa con ella y observa que la hembra es receptiva. Luego se lanza con otra de sus cotorras para invitarla a salir. Mientras sigue con su labia, nota que la hembra se quiebra y le suelta una invitación a su casa para que le lave la ropa aprovechando que ella hará todo lo que él le pida.”<sup>3</sup>

¿Ve cómo y cuando se **dispara** cada regla?

### Mataquín

Cierto agente llamado Mataquín está contemplando la posibilidad de cometer un crimen. Quiere vengarse de cierto enemigo suyo pero quiere, desde luego, cometer el crimen perfecto: venganza plena, impunidad total y mantener su imagen pública cubriendo bien sus huellas. Mataquín ha observado que los policías de su ciudad no tienen mucha capacidad de respuesta ante un crimen. No suelen atender sino emergencias extremas. No tienen recursos para investigaciones complejas que involucren a muchas personas, muchos lugares o que impliquen análisis técnicos de cierta sofisticación. Tienen una pésima memoria organizacional, pues llevan todos los registros en papel y los guardan en lugares inseguros. Además, los policías son muy mal pagados y se conocen casos de sobornos, especialmente en crímenes muy complejos o en los que los investigadores se arriesgan mucho y por mucho tiempo. Los ciudadanos de esa ciudad tienen tal desconfianza en su policía que nunca les ayudan, aún cuando tengan información sobre un crimen.

Suponga ahora que, con esas consideraciones en mente, se nos pide proponer una especificación (informal, pero tan completa como sea posible a partir de la información dada) de las reglas de conducta que debería seguir Mataquín para alcanzar su meta.

Considere la respuesta en la figura 1.5 <sup>4</sup>:

La historia resultante, aparte de repulsiva, es bastante obvia, ¿verdad?

### Halcones, Burgueses y Palomas: Sobre las aplicaciones de la Teoría de Decisiones en lógica

Para ilustrar una posible aplicación de la teoría de Decisiones, un desarrollo matemático estrechamente asociado con los agentes y los juegos (ver en el capítulo 8 de Kowalski), consideren el siguiente ejemplo tomado de un examen (originalmente en [3]). Considere los siguiente tres tipos de personas en una sociedad en la que los individuos compiten por recursos que siempre son de alguien:

La **paloma** nunca trata de hacerse con las posesiones de otros, sino que espera a que sean abandonadas y ella misma abandona un recurso propio tan pronto es atacada. Si dos de ellas compiten por el mismo recurso, entonces una de ellas lo obtendrá (por suerte o paciencia) con la misma probabilidad para cada una. El **halcón** siempre trata de apoderarse de los recursos de otros por medio de la agresión y se rinde sólo si recibe graves lesiones. El **burgués** nunca trata de hacerse con las posesiones de otros, sino que espera hasta que son abandonadas, pero defiende su posesión contraatacando hasta que tiene éxito o es derrotado[4].

Cuando dos individuos se encuentran tienen idénticas probabilidades de ganar o perder y de ser, al inicio, dueños ó no del recurso. Por ejemplo, si dos halcones se encuentran, uno de ellos obtendrá el recurso con utilidad  $U$ , mientras el otro tendrá graves lesiones, con costo  $-C_{pelea}$ . Como ambos tienen la misma oportunidad de ganar, la utilidad esperada para cada uno es  $(U - C_{pelea})/2$ .

<sup>3</sup>Joskally Carrero, 2007

<sup>4</sup>Basada en una propuesta de Victor Malavé, 2007

```
1 METAS
2
3 si enemigo se acerca entonces establece contacto.
4
5 si contacto establecido y policía cercano ,
6     crea distracción .
7
8 si contacto establecido y policía distraída ,
9     conduce a enemigo a piso más alto
10    de edificio cercano.
11
12 si enemigo en piso alto de edificio cercano , asesínalo .
13
14 si durante asesinato un policía observó , sobórnalo .
15
16 CREENCIAS
17 Para establecer contacto , invítalo a lugar público .
18
19 Para crear distracción , provoca emergencia extrema .
20
21 Para provocar emergencia extrema ,
22     contrata lugareños
23     que incendien estación de servicio cercana .
24
25 Para conducir enemigo a piso alto de
26     edificio cercano , propónle un negocio atractivo .
27
28 Para asesinarlo ,
29     provoca caída mortal desde ese piso alto .
30
31 Para sobornar policía , entrégale mucho dinero .
```

Figura 1.5: Mataquín

¿Cuáles son las utilidades esperadas correspondientes cuando se enfrentan una paloma y un halcón?. Explique.

Dadas esas reglas de conducta, la paloma nunca gana contra un halcón (y el halcón nunca pierde contra una paloma). Así que la utilidad para la paloma es 0 y para el halcón es  $U$ . Noten que  $U$  es el valor *puro* del recurso a conquistar (otra simplificación muy gruesa).

¿Cuáles son las utilidades esperadas correspondientes cuando se enfrentan una paloma y un burgués?. Explique <sup>5</sup>

“Cuando un burgués encuentra a otro individuo, cada uno de ellos puede ser el dueño lícito del recurso rivalizado. Por ejemplo, si una paloma se encuentra con un burgués y ambos compiten por el mismo recurso, entonces tenemos dos posibilidades igualmente probables:

- Caso1: Si el burgués es el dueño legal del recurso, entonces conserva el recurso (*Beneficio*= $U$ ), y la paloma no se lleva nada (*Beneficio*=0).
- Caso2: Si la paloma es la dueña legal del recurso, ambos deben esperar hasta que alguno renuncie (lo cuál les cuesta digamos *Cespera*. *Beneficio* =  $-Cespera$ ) y, entonces, cada uno de ellos tiene igual probabilidad de lograr el recurso. Así que el resultado local esperado es  $U/2 - Cespera$  para cada uno.

De esta manera, el resultado global es  $\frac{U+(\frac{U}{2}-Cespera)}{2}$  para el burgués y  $\frac{0+(\frac{U}{2}-Cespera)}{2}$  para la paloma”.

Noten que la utilidad global (para cada agente) es igual a  $Caso1 * Prob1 + Caso2 * Prob2$ , donde  $Prob1 = Prob2 = 1/2$ , pues se nos dice que ambos casos son igualmente probables.

En el caso2, además, juega un papel la acción de esperar a que el recurso sea abandonado (sí, esperar también es una acción posible). Cómo no se sabe cuál de los dos va a renunciar primero en esa esperar, se dice que su probabilidad local asociada es  $1/2$  y como el beneficio en disputa es  $U$ , la utilidad esperada local es de  $U/2$ , faltando por descontar el precio de esperar que ambos pagan igualmente (*Cespera*).

Este ejemplo es particularmente interesante porque muestra el uso del concepto de la utilidad esperada *anidada* (cálculo de utilidad esperada sobre otra utilidad esperada).

¿Cómo sería la utilidad global para ambos agentes?.

## Agentes, Ontologías y la Web Semántica

Los ejemplos anteriores se referieren todos a descripciones de agentes en ciertos roles muy puntuales. Pareciera que para tener un agente completo y funcional se requiere de un esfuerzo mucho mayor. Lo cierto es que esto último, incluso para efectos de prestación de servicios, no necesariamente es el caso.

Lo que queremos es mostrar a continuación son dos ejemplos de modelos de agentes que implementan otras dos nociones que se han vuelto fundamentales en la Web: Los **metadatos** y el **razonamiento no monótono**.

Una de las primeras aplicaciones naturales de la tecnología de la **Web Semántica** es la búsqueda de documentos. Con recursos como repositorios **RDF** y **OWL** en la Web, uno puede imaginar un agente que nos ayude a ubicar documentos a partir del tipo de conocimiento que contienen (y que buscamos). Por ejemplo, para responder preguntas como *¿Quiénes han escrito sobre el legado*

```

1 si me preguntan ¿Quien Opina sobre Tema? y
2   Quien es una variante gramatical de Quienes y
3   Opina es una variante semántica de opinar y
4 el análisis del Tema arroja estos Descriptores entonces
5   Rastreo la Web buscando todos los Autores de documentos
6 con esos Descriptores .

```

Figura 1.6: Agente Web Semántica

*epistolar del Libertador de Venezuela?*, uno puede pensar en un agente con una META<sup>6</sup> como la que se muestra en la figura 1.6.

Esta meta requerirá, desde luego, mecanismos de soporte (que podrían convertirse en creencias del agente) para realizar el análisis del Tema que produce los Descriptores y el Rastreo de la Web.

En la tarea de **análisis del tema**, sería muy útil contar con un mecanismo que le permita al agente, en el caso particular de nuestra pregunta, traducir *legado epistolar* en todos sus **sinónimos** e **hipónimos**. Así el agente sabría que quien quiera que declare haber escrito sobre *las cartas del Libertador* es un autor a revisar. Algo similar habría que hacer con *Libertador de Venezuela* para que la máquina entienda que se trata de Simón Bolívar. Esto último requeriría, desde luego, un mecanismo adecuado para el caso *venezolano-colombiano-ecuatoriano-peruano-boliviano*.

Esta clase de relaciones de **sinonimia**, tanto universales como locales, son posibles con sistemas como **WordNET**<sup>7</sup>, una especie de diccionario y tesoro automático. WordNET codifica una **ontología** que le permite relacionar los términos. Desafortunadamente, esa ontología no está escrita en ninguno de los lenguajes ontológicos de la Web Semántica, pero aún así puede ser útil.

La segunda tarea es también muy interesante. Requiere que nuestro agente disponga de medios para identificar entre la enorme cantidad de documentos contenidos en la Web, los autores y los descriptores de esos documentos. Esa información que refiere el contenido y otros atributos de los datos es conocida como **metadata** y es fundamental para que las máquinas puedan contribuir a la gestión de los **repositorios de conocimiento**.

Hay muchos esfuerzos en la dirección de proveernos de un standard de metadatos en la forma de una ontología que explique que son y que contienen los documentos formales. El Dublin Core[5] es quizás el más avanzado de esos esfuerzos.

Podemos ver, en ese primer ejemplo de agente para la Web Semántica, como se incorporaría el manejo de significado en un agente que atienda consultas en la Web.

Hay, sin embargo, un siguiente nivel de complejidad en el manejo de significados para atender consultas. Algo que es bien conocido en Bases de Datos desde hace años: El manejo de datos temporales y el razonamiento no monótono.

Se llama **razonamiento no monótono** a aquel que cambia de opinión. Típicamente, un agente con cierto cúmulo de creencias alcanza ciertas conclusiones, pero si esas creencias cambian (debido quizás, a un cambio en el mundo), las conclusiones también podrían cambiar.

Los lectores del libro del Profesor Kowalski podrán asociar esa forma de razonar con esos lenguajes lógicos que estudiaron en el capítulo sobre el cambiante mundo (**lógicas modales**, el **cálculo de situaciones** y el **cálculo de eventos**). Permítanme, sin embargo, un último ejemplo con un agente que implementa una forma de razonamiento no monótono indispensable para la aplicación a su cargo:

<sup>5</sup>Cito la explicación dada en Simulación para las Ciencias Sociales de Nigel Gilbert y Klaus Troitzsch

<sup>6</sup>Agradecemos a Icaro Alzuru por motivar este ejemplo con su trabajo de maestría en Alejandría Inteligente: Un experimento en la Web Semántica con un sistema de gestión de documentos <http://webdelprofesor.ula.ve/ingenieria/jacinto/tesis/2007-feb-msc-icaro-alzuru.pdf>

<sup>7</sup><http://wordnet.princeton.edu/>, <http://multiwordnet.itc.it/english/home.php>

```

1 | si me solicitan los Propietarios de terrenos en un Área
2 |     entonces reportar Propietarios del Área.
3 |
4 | Para reportar Propietarios del Área haga
5 |     transforme la descripción del Área
6 |         en un Área Geográfica y
7 |     Los Propietarios son los sujetos para quienes
8 |         se cumple (ahora) que poseen
9 |         propiedades contenidas en el Área Geográfica.

```

Figura 1.7: Gea: El Agente Catastral

Imaginen un agente que maneja un repositorio de información catastral elemental<sup>8</sup>. Entre muchos otros tipos de consulta, este agente debe poder explicar quienes son los propietarios de los terrenos en un área dada. La meta de mantenimiento y creencia correspondiente para este agente es simple se muestra en fig 1.7.

La última sub-meta (*Los Propietarios.*) es una consulta que se podría responder con una versión extendida del cálculo de situaciones, o el de eventos, configurada para lidiar con propiedades y áreas geográficas. Lo importante en este caso es que si a este agente se le hace la consulta en un momento dado, su respuesta bien puede variar respecto a otro momento si han ocurrido eventos que cambien las relaciones propiedad-propietario en ese área geográfica.

Desde luego, este agente tendría que operar con conocimiento que le permita asociar cualquier descripción de un área (por ejemplo, en términos políticos: parroquia, municipio, país, etc) con un conjunto de coordenadas standard. Más importante aún, el agente requeriría de un registro sistemático de los eventos de compra y venta de propiedades que incluya los detalles de cada propietario y propiedad (incluyendo la ubicación de cada una, desde luego). Todo este conocimiento bien puede estar almacenado en un repositorio asociado con una ontología de registro catastral (por ejemplo, un archivo, con marcaciones en **OWL** específicas para el problema).

Esos dos ejemplos, informales y muy simples, pueden servir para ilustrar la extraordinaria riqueza de posibilidades en este encuentro entre la Web, la lógica y los agentes.

### Kally: Agente para atención a usuarios y enseñanza isocéntrica

Los agentes Web Semántica y Gea Catastral dejan ver someramente la importancia del procesamiento del lenguaje natural en la Inteligencia Artificial. Es, todavía, una de las fronteras más activas de investigación y promete grandes avances, siempre que logremos contener el entusiasmo que produce la posibilidad de interactuar con el computador en los mismos términos (lenguaje) que usamos con otros humanos.

Kally<sup>9</sup> es un pequeño agente diseñado para asistir a los humanos en el uso de una herramienta Ofimática, OpenOffice, interactúan en una forma controlada de lenguaje natural. En fig 1.8, 1.9 y 1.10 se muestra el código de Kally en una versión preliminar de Gloria (ver capítulo ??).

Kally tiene sólo un par de reglas simples, pero tiene también una gramática interconstruida (el predicado *pregunta*) que le permite entender ciertas preguntas simples en Español y establecer “su significado” (es decir, lo que debe hacer al respecto). La gramática está formalizada usando un lenguaje conocido como DCG, *Definite Clause Grammar*.

<sup>8</sup>Agradecemos a Nelcy Patricia Piña por motivar este ejemplo con su trabajo de maestría en Una Ontología para Manejo de Información Catastral <http://webdelprofesor.ula.ve/ingenieria/jacinto/tesis/2006-mayo-msc-nelcy-pina.pdf>

<sup>9</sup><http://kally.sourceforge.net>

```

1 si me_pregunta(Preg),
2   significa(Preg, Sig),
3   not(Sig=[no, entiendo|_]) entonces respondo_a(Sig).
4 si me_pregunta(Preg),
5   significa(Preg, Sig),
6   Sig=[no, entiendo|_] entonces disculpas(Sig).
7
8 para respondo_a(S) haga
9   rastrear(S),
10  mostrar(S)

```

Figura 1.8: Kally

```

1  %-----
2  % Reglas gramaticales
3  %-----
4  % que es X
5  pregunta([V|S]) →
6    pro_interrog, v_atributivo(V), s_atributivo(S).
7  % como puedo instalar Y
8  pregunta([V|S]) →
9    adv_interrog, s_verbal(V), s_atributivo(S).
10 % puedo instalar Y
11 pregunta([V|S]) → s_verbal(V), s_atributivo(S).
12 % esta es la via de escape cuando no entiende
13 pregunta([no, entiendo, tu, pregunta, sobre|S]) →
14   atributo(S).
15 %
16 s_verbal(V) → v_modal, v_infinitivo(V).
17 s_verbal(V) → v_infinitivo(V).
18 s_verbal(V) → v_conjugado(V).
19 s_atributivo(S) → especificador, atributo(S).
20 s_atributivo(S) → atributo(S).

```

Figura 1.9: La Gramática Española de Kally

```

1  %-----
2  % Reglas de inserción léxica
3  %-----
4  pro_interrog -> ['qué'];[que];[cual];[cuales];
5  ['cuáles']; [cuantos];[cuantas].
6  especificador -> [el];[la];[lo];[los];[las];[un];
7  [una];[unos];[unas];[mi];[mis].
8  v_atributivo(es) -> [es];[son];[significa].
9  adv_interrog -> ['cómo'];[como];[cuando];
10 [donde];[por, que]; [por, 'qué'].
11 v_modal -> [puedo];[puede];[podemos].
12 v_infinitivo(utilizar) -> [utilizar].
13 v_infinitivo(abrir) -> [abrir].
14 v_infinitivo(salvar) -> [guardar].
15 v_infinitivo(salvar) -> [salvar].
16 v_infinitivo(crear) -> [crear].
17 v_infinitivo(instalar) -> [instalar].
18 v_infinitivo(instalar) -> [reinstalar].
19 v_infinitivo(definir) -> [definir].
20 v_infinitivo(realizar) -> [realizar].
21 v_infinitivo(agregar) -> [agregar].
22 v_infinitivo(agregar) -> [añadir].
23 v_infinitivo(exportar) -> [exportar].
24 v_infinitivo(insertar) -> [insertar].
25 v_conjugado(guardar) -> [guarda].
26 v_conjugado(desinstalar) -> [desinstalo].
27 v_conjugado(ubicar) -> [existe].
28 v_conjugado(usar) -> [uso].
29 v_conjugado(cambiar) -> [cambio].
30 v_conjugado(instalar) -> [instalo].
31 v_conjugado(crear) -> [creo].
32 v_conjugado(estar) -> [esta].
33 v_conjugado(actualizar) -> [actualizo].
34 %
35 prep -> [a];[como];[con];[de];[desde];
36 [durante];[en];[entre];[hacia];[mediante];
37 [para];[por];[sin];[sobre].
38 atributo(T,T,-).

```

Figura 1.10: El Léxico Español de Kally

### El Cálculo de Eventos y El disparo de Yale

El **sentido común** se ha constituido en uno de los mayores desafíos en la IA, al punto que algunos problemas referenciales al tema se han vuelto muy populares.

En la **historia del Disparo de Yale**, una persona es asesinada de un disparo. Para una posible formalización uno considera tres acciones: *cargar (el arma)*, *esperar* y *disparar*; y dos propiedades: *vive* y *cargada*. La acción cargar coloca una bala en el arma. La víctima muere después del disparo, siempre que el arma esté cargada en ese instante. Se asume que la víctima vive al principio y también que el arma está descargada entonces.

Considere el axioma central del Cálculo de Eventos:

0) Se cumple un Hecho en un Momento si un Evento ocurrió antes y ese Evento inició el Hecho y no hay Otro evento que ocurra luego del Evento iniciador, antes del Momento y que termine el Hecho.

Suponga que nos piden completar este axioma con las reglas y hecho necesarios para describir el problema del Disparo de Yale. Con todas esas reglas, debemos probar (con un argumento formal) que: no se cumple que (la víctima) vive en en último momento del cuento.

La historia se puede describir, con algo de formalización, así:

- 1) La (presunta) víctima nace en el momento 0.
- 2) No es cierto que el arma está cargada en el momento 0.
- 3) El (presunto) asesino carga el arma en el momento 1.
- 4) El asesino espera entre el momento 1 y el momento 2.
- 5) El asesino dispara en el momento 2.

Con la misma terminología podemos escribir las reglas complementarias específicas a esta situación:

- 6) El evento **Un agente (le) dispara** en el momento T termina el hecho **la víctima vive** en T si se cumple que el arma está cargada antes de T.
- 7) El evento **Un agente (le) dispara** en el momento T termina el hecho **el arma esta cargada** en T si se cumple que el arma está cargada antes de T.
- 8) El evento **Un agente carga el arma** en el momento T inicia el hecho **el arma está cargada** en el momento T.
- 9) El evento **Un agente nace** en el momento T inicia el hecho **el agente vive** en el momento T.

Con esas reglas razonamos hacia atrás a partir del hecho de que no se cumple que la víctima vive en el último momento del cuento. Digamos que ese momento es momento 3.

- 10) no se cumple que la víctima vive en el momento 3.

de 10) y 0) se obtiene:

- 11) no (es cierto que) un evento ocurrió antes del momento 3 y ese evento inició (el hecho de) víctima vive y no hay otro evento que ocurra luego de ese evento iniciador y antes del momento 3 y que termine el hecho de que la víctima vive.

```

1  si inspiracion_sistemica entonces
2      crear_obra_tipo(holistica , Obra).
3
4  para crear_obra_tipo(holistica , Obra) haga
5      seleccionar_elementos(holistica , Elementos),
6      plasmar(Elementos , Obra).
7
8  para seleccionar_elementos(Tipo , [Elemento]) haga
9      tomar_elemento(Tipo , Elemento).
10
11 para seleccionar_elementos(Tipo , [R|Resto]) haga
12     tomar_elemento(Tipo , R),
13     seleccionar_elementos(Tipo , Resto).
14
15 para tomar_elemento(holistica , nota(do)) haga true.
16 para tomar_elemento(holistica , color(azul)) haga true.
17 para tomar_elemento(holistica , textura(suave)) haga true.
18 para plasmar([nota(do)], sonido_bajo) haga true.
19 para plasmar([nota(do), color(azul)], azul_profundo)
20     haga true.
21 para plasmar([nota(do), color(azul), textura(suave)],
22     extasis ) haga true.

```

Figura 1.11: El Artista

Pero 11) se puede reescribir (por equivalencia lógica  $\text{no } (a \text{ y } b) = (\text{no } a) \text{ o } (\text{no } b)$ ), así:

**11')** no es cierto que un evento ocurrió antes del momento 3, ó no inició ese evento víctima vive, ó hay otro evento que ocurra luego de ese evento iniciador y antes del momento 3 y que termine el hecho.

11') se transforma en 12) al considerar a 1),

**12)** hay otro evento que ocurra luego del evento iniciador *víctima nace* en momento 0 y antes del momento 3 y que termine el hecho de que *la víctima vive*.

Pero, por 5), sabemos que hay un candidato a posible terminador, con lo que 12) se reduce a 14), luego de considerar a 13)

**13)** el *asesino dispara* en momento 2 termina el hecho de que *la víctima vive*.

Así, por 6), pasamos a preguntarnos si 14)

**14)** el arma está cargada en el momento 2..

y gracias a 0), 3) y 2) (y que la acción de esperar no cambia nada), podemos probar a 14 con un par de pasos similares a los anteriores. Fin de la prueba.

Visto así parece increíble que un computador lo pueda resolver. Pero lo resuelve.

### El Agente Artista

Considere las reglas en la figura artista que son parte de un agente artista holístico.

Suponiendo que este agente observa *inspiracion\_sistemica*, vemos como, paso a paso, este agente artista produciría un plan para la creación de una obra con la mayor cantidad de elementos posibles. Asegúrese de explicar en que consiste la *Obra* planeada.

Razonando hacia adelante con la primer regla, el agente produce su primer meta de logro:

1) `crear_obra_tipo(holistica, Obra)`

noten que la escribimos en notación Prolog para hacer más fácil la representación subsiguiente. Razonando hacia atrás a partir de 1) usando segunda regla obtenemos:

2) `seleccionar_elementos(holistica, Elementos)` y  
`plasmarm(Elementos, Obra).`

Para resolver la primer submeta de 2) tenemos dos reglas (la tercera y cuarta arriba), por lo que 2) se convierte en 3)

3) `tomar_elemento(holistica, [Elemento])` y  
`plasmarm([Elementos], Obra), o`  
`tomar_elemento(holística, R)` y  
`seleccionar_elementos(holistica, Resto)` y  
`plasmarm([R|Resto], Obra).`

Uno podría, si no es cuidadoso con lo que se nos pide, terminar la prueba con la primera opción (antes del ,o). Pero eso produciría una obra con un solo elemento. Eso es justamente lo que NO se nos pide. Por esta razón, un agente inteligente optaría por la segunda alternativa y, luego de sucesivos refinamientos obtendría algo como 4)

4) `tomar_elemento(holistica, [Elemento])` y  
`plasmarm([Elementos], Obra), o`  
`tomar_elemento(holistica, nota(do))` y  
`tomar_elemento(holistica, color(azul))` y  
`tomar_elemento(holistica, textura(suave))` y  
`plasmarm([nota(do), color(azul), textura(suave)], Obra)`

el cual, al ser ejecutado, implicaría *Obra = extasis* que consiste en esta lista de elementos: `[nota(do), color(azul), textura(suave)]`. El truco acá era observar que una reducción hacia atrás, izquierda a derecha (tal como lo hace Prolog) no nos conduce a la obra pretendida (que debe tener el máximo numero de elementos posibles). Fin de la prueba.

### Como lidiar con la inflación

Presentamos un último ejemplo en este capítulo para ilustrar un caso mucho más complejo (y humano) del modelado lógico de un agente. El Prof. Carlos Domingo se plantea el siguiente conjunto de reglas para enfrentar el problema de la inflación que afectará sus ingresos como académico.

Meta: Evitar las pérdidas en mis ahorros causadas por la inflación de este año.

Creencias:

1. Durante el 2008, los precios se espera que aumenten en un 30 por ciento en artículos como libros, del hogar, medicinas y ropas. En los artículos básicos (comida y transporte nacional), el incremento puede ser menor al 10 por ciento debido al control de precios y, además, es probable que sea compensado con un aumento salarial. Pero mis ahorros anteriores podrían ser afectados por la inflación.
2. Las tasas de interés estimadas para este año en un 13 por ciento no compensarán la inflación. Por lo tanto, depositar el dinero el banco no es una buena estrategia para proteger mis ahorros. Otras alternativas deben ser consideradas.

Restricciones:

1. No debería tomarme mucho tiempo para encontrar una solución. Tengo trabajo pendiente.
2. No debo comprometerme a trabajos posteriores a este año.
3. Cualquiera que sea la decisión no deben disminuir mi ingreso total anual.

Tomando eso en consideración, Carlos Domingo decide que sus metas inmediatas son:

1. Conseguir un ingreso adicional que no dependa de sus ahorros para compensar por la inflación. Para ello puedo:
  - a) Participar en proyectos universitarios extra que le den un ingreso extra (obvenciones). Esto podría chocar con la restricción 1 porque es difícil conseguir proyectos en mi área de trabajo.
  - b) Ocuparme en negocios privados que siempre están disponibles. Esto choca con todas las restricciones.
2. Invertir mis ahorros (total o parcialmente) para obtener beneficios equivalentes a las pérdidas esperadas. Para ello puedo:
  - a) Comprar acciones (pero esto requiere tiempo y puede contrariar a 1 y probablemente a 2).
  - b) Comprar bienes “durables” para vender en el futuro. Tendría que tener cuidado con los precios futuros y la obsolescencia.
  - c) Mudar mis ahorros a moneda extranjera. Difícil debido al control cambiario y muy riesgoso debido a una posible devaluación de esa moneda extranjera.

El Prof. Carlos ha concluido que una combinación de 1.a y 2.b parece ser la mejor decisión posible (una combinación minimizaría el riesgo de fallar completamente). Suponga que Carlos posee una cierta cantidad en AHORROS al INICIO del año y monto total por su SALARIO en todo el año. Ayudemos a Carlos a proponer una justificación formal de su decisión.

Un análisis de utilidad completo supondría estimar las utilidades esperadas de cada opción. Esto puede consumir mucho tiempo y, en algunos casos, aún así no se mejoraría la evaluación. Así que, entendiendo las conclusiones de Carlos, uno se puede concentrar en las dos alternativas de acción que le parecen las mejores decisiones.

Carlos fallaría si sus ahorros son consumidos por la inflación. Es decir, si al final del año sus AHORROS no satisfacen:

$$AHORROSAHORROS\_INICIALES * 1,3 \tag{1.1}$$

Es decir, ese es el mínimo valor razonable para la utilidad esperada global de las acciones que se plantea Carlos.

Las acciones que se plantea Carlos son:

1. ganar obvenciones por proyectos extras y
2. invertir en durables.

La probabilidad de ganar obvenciones por proyectos extras,  $ProbObv$ , es menor al 0.5. Así que la utilidad estimada de la acción 1 es:

$$utilidad_1 = ProbObv * OBVENCIONES \quad (1.2)$$

Invertir en durables significa: 1) Adquirirlos a buen precio y, desde luego, con un costo no superior a los ahorros en ese momento y 2) venderlos al precio justo. La probabilidad de adquirirlos a buen precio es  $ProbAdDur$ . La probabilidad de venderlos es  $ProbVenta$ . Así, la utilidad de acción 2 puede ser aproximada por:

$$utilidad_2 = ProbAdDur * ProbVenta * (PRECIOJUSTO - AHORROS_ESPERADOS) \quad (1.3)$$

donde  $AHORROS_ESPERADOS$  es una variable que depende del  $SALARIO$ , un probable aumento, la inflación y los gastos de cada tipo en que incurra Carlos. Por ejemplo:

$$AHORROS_ESPERADOS = SALARIO + ProbAumento * AUMENTO - GASTOS_BASICOS * 1,10 - OTROS_GASTOS * 1,3 \quad (1.4)$$

Así que los escenarios que satisfacen la siguiente fórmula son los que justificarían la decisión de Carlos:

$$ProbObv * OBVENCIONES + ProbAdDur * ProbVenta * (SALARIO + ProbAumento * AUMENTO - GASTOS_BASICOS * 1,10 - OTROS_GASTOS * 1,3) > AHORROS_INICIALES * 1,3 \quad (1.5)$$

Hay todavía varios otros ejemplos que discutir. En particular, hemos reservado espacio separado para los motores de minería de datos y los agentes aprendices (en el capítulo ??).



# Simulación con Agentes y Lógica

---

## Introducción

En capítulo anterior presentamos una práctica de lógica. En este presentaremos una práctica de simulación. Como quiera que una práctica requiere conocimientos mínimos de un lenguaje, referimos al lector al apéndice A para una breve introducción al lenguaje de programación que empleamos en el texto.

### 2.1. Directo al grano

Este segundo capítulo tiene como propósito dirigir a un simulista en la creación de un modelo de simulación Galatea <sup>1</sup>. Nos proponemos hacer eso en dos fases. En la primera, mostramos paso a paso cómo codificar en Java un modelo básico para Galatea. En la segunda fase, repetimos el ejercicio, pero esta vez sobre un modelo de simulación multi-agente de tiempo discreto.

La intención trascendente del capítulo es motivar ejercicios de “hágalo Ud mismo”, luego de darle al simulista las herramientas lingüísticas básicas. Explicar la infraestructura de simulación que las soporta nos tomará varios capítulos en el resto del libro. El lector no debería preocuparse, entretanto, por entender los conceptos subyacentes, más allá de la semántica operacional de los lenguajes de simulación.

En otro lugar explicamos que Galatea es una *familia de lenguajes de simulación*. En este capítulo, por simplicidad, sólo usamos Java.

### 2.2. El primer modelo computacional

Galatea heredó la semántica de Glider[6]. Entre varias otras cosas, esto significa que Galatea es, como Glider, *orientado a la red*. Esto, a su vez, refleja una postura particular de parte del modelista que codifica el modelo computacional en Galatea. En breve, significa que el modelista identificará componentes del sistema modelado y los ordenará en una red, cuyos enlaces corresponden a las vías por las que esos componentes se comunican. La comunicación se realiza, normalmente (aunque no exclusivamente como veremos en los primeros ejemplos) por medio de pase de mensajes.

---

<sup>1</sup><http://galatea.sourceforge.net>

La simulación se convierte en la reproducción del desenvolvimiento de esos componentes y de la interacción entre ellos. Esta es, pura y simple, la idea original de la orientación por objetos que comentábamos al principio del capítulo anterior.

En Glider, y en Galatea, esos componentes se denominan **nodos**. Los hay de 7 tipos: —Gate—, —Line—, —Input—, —Decision—, —Exit—, **Resource**, **Continuous** y —Autonomous—<sup>2</sup>

Estos 7 tipos son estereotipos de componentes de sistemas que el lenguaje ofrece “ya pensando” al modelista. El modelista, por ejemplo, no tiene que pensar en cómo caracterizar un componente de tipo recurso. Sólo debe preocuparse de los detalles específicos de los recursos en el sistema a modelar, para luego incluir los correspondientes nodos tipo —Resource— en su modelo computacional. Esta es la manifestación de otra idea central de la orientación por objetos: la reutilización de código.

Una de las líneas de desarrollo de Galatea apunta a la creación de un programa traductor para la plataforma, como explicaremos en un capítulo posterior. Ese interpretador nos permitirá tomar las especificaciones de nodos en un formato de texto muy similar al que usa Glider (lo llamaremos **nivel Glider**) y vertérlas en código Java, primero y, por esta vía, en programas ejecutables.

Aún con ese traductor, sin embargo, siempre es posible escribir los programas Galatea, respetando la semántica Glider, directamente en Java (lo llamaremos **nivel Java**). Hacerlo así tiene una ventaja adicional: los conceptos de la orientación por objetos translucen por doquier.

Por ejemplo, para definir un tipo de nodo, el modelista que crea una clase Java que es una subclase de la clase —Node— del paquete `galatea.glider`. Con algunos ajuste de atributos y métodos, esa clase Java se convierte en un `Node` de alguno de los tipos Glider. Para *armar* el simulador a nivel Java, el modelista usa las clases así definidas para generar los objetos que representan a los nodos del sistema.

De esta forma, lo que llamamos un modelo computacional Galatea, a nivel Java, es un conjunto de subclases de `Node` y, por lo menos una clase en donde se definen las variables globales del sistema y se coloca el programa principal del simulador.

Todo esto se ilustra en las siguientes subsecciones, comenzando por las subclases `Node`.

### 2.2.1. Nodos: componentes del sistema a simular

Tomemos la caracterización más primitiva de un sistema dinámico: Una máquina abstracta definida por un conjunto de variables de estado y una función de transición, normalmente etiquetada con la letra griega  $\delta$ , que actualiza ese estado a medida que pasa el tiempo. Una caracterización tal, es *monolítica*: no hay sino un solo componente y sus régimen de cambio es el que dicta  $\delta$ .

La especificación de ese único componente del sistema sería, en el Nivel Java, algo como:

```

1  /*
2  *  Delta.java
3  *
4  *  Created on April 22, 2004, 10:18 PM
5  */
6
7  package demos.grano;
8
9  //import galatea.*;
10 import galatea.glider.*;
11
12 /**
13  *
14  *  @author  Jacinto Dávila
15  *  @version beta 1.0

```

<sup>2</sup>Los primeros 5 dan cuenta del nombre del lenguaje GLIDER. Note el lector que Galatea también es un acrónimo, sólo que seleccionado para rendir tributo al género femenino como nos enseñaron varios maestros de la lengua española.

```

16  */
17  public class Delta extends Node {
18
19      /** Creates new CambioEdo */
20      public Delta() {
21          super("Delta", 'A');
22          Glider.nodesl.add(this);
23      }
24
25      /** una funcion cualquiera para describir un fenomeno */
26      double f(double x) {
27          return (Math.cos(x/5) + Math.sin(x/7) + 2) * 50 / 4;
28      }
29
30      /** funcion de activacion del nodo */
31      public boolean fact(){
32          Modelo.variableDep = f((double)Modelo.variableInd);
33          Modelo.variableInd++;
34          it(1);
35          return true; }
36
37 }

```

Como el lector reconocerá (seguramente luego de leer el capítulo ??), ese código contiene la especificación de una clase Java, `Delta`, del paquete `demos.Grano`, que acompaña la distribución Galatea (como código libre y abierto).

Noten que esta clase “importa” (usa) servicios de los paquetes `galatea` y `galatea.glider`. El primero de estos paquetes, como dijimos en el capítulo ??, contiene servicios genéricos de toda la plataforma (como la clase `List`). Ese paquete no se usa en este ejemplo tan simple. Pero seguramente será importante en ejemplos más complejos.

El segundo paquete almacena toda la colección de objetos, clases necesarios para que la plataforma Galatea ofrezca los mismos servicios que el tradicional compilador Glider, incluyendo el núcleo del simulador (en la clase `galatea.glider.Glider`).

Los tres métodos que se incluyen en la clase `Delta` son paradigmáticos (es decir, un buen ejemplo de lo que suelen contener esas clases).

El primero es el constructor. Noten la invocación al constructor de la superclase `super("Delta", 'A')`. Los parámetros corresponden a un nombre para la “familia” de nodos y, el segundo, a un carácter que identifica el tipo de nodo en la semántica Glider<sup>3</sup>. Este es, por tanto, un nodo autónomo lo que, por el momento, significa que es un nodo que no envía, ni recibe mensajes de otros nodos.

La segunda instrucción del constructor agrega el nuevo nodo a la lista de todos los nodos del sistema.

El método con el nombre `f` es simplemente una implementación auxiliar que podría responder a la especificación matemática de la función de transición del sistema en cuestión.

Finalmente en esta clase `Delta`, el método `fact()` contiene el código que será ejecutado cada vez que el nodo se active. Es decir, en el caso de los nodos autónomos, cada vez que se quiera reflejar un cambio en el componente que ese nodo representa.

Este código de `fact` merece atención cuidadosa. La instrucción:

```
Modelo.variableDep = f((double)Modelo.variableInd);
```

<sup>3</sup>Decimos “familia” cuando estrictamente deberíamos decir tipo. Esto para evitar una confusión con el uso de la palabra tipo que sigue. Lo que está ocurriendo es que tenemos tipos de nodos en Glider, tipos de objetos, clases en Java y, por tanto, tipos de tipos de nodos: las clases Java que definen un tipo de nodo Glider para un modelo particular. Aquí se abre la posibilidad de compartir tipos de tipos de nodos entre varios modelos computacionales. El principio que ha inspirado la **modeloteca** y que se explicará más adelante

invoca a la nuestra función `f` para asignar lo que produzca a una variable de un objeto que no hemos definido aún. Se trata de un objeto anónimo asociado a la clase `Modelo` que es aquella clase principal de nuestro modelo computacional. Las variables globales de nuestro modelos son declaradas como atributos de esta clase que, desde luego, puede llamarse como el modelista prefiera. En la siguiente sección ampliaremos los detalles sobre `Modelo`. Noten que el argumento de entrada de `f`, es otra variable atributo de la misma clase.

La siguiente instrucción incrementa el valor de aquella variable de entrada, en 1 (Ver el manual de Java para entender esta sintaxis extraña. `x++` es equivalente a `x = x + 1`).

Para efectos de la simulación, sin embargo, la instrucción más importante es:

```
it(1);
```

Se trata de una invocación al método `it` de la clase `Node` (y por tanto de su subclase `Delta`) cuyo efecto es el de reprogramar la ejecución de este código (el de la `fact`) para que ocurra nuevamente dentro de 1 unidad de tiempo. En simulación decimos que el tiempo entre llegadas (*interarrival time*, `it`) de estos *eventos de activación* del nodo `Delta` es de 1. Veremos en capítulos posteriores que estos eventos y su manejo son la clave del funcionamiento del simulador.

### 2.2.2. El esqueleto de un modelo Galatea en Java

Como el lector sabrá deducir, necesitamos por el momento, por lo menos dos clases para nuestro modelo computacional: `Delta`, con la función de transición del único nodo y `Modelo`, con las variables y el programa principal. Vamos a agregar la clase —`Analisis`—, que explicamos a continuación, con lo cual nuestro modelo queda compuesto por tres clases:

—**Delta**— Como ya hemos explicado, representa a todo el sistema e implementa su única función de transición.

—**Analisis**— Es un nodo auxiliar para realizar análisis sobre el sistema simulado mientras se le simula. Se explica a continuación.

—**Modelo**— Es la clase principal del modelo. Contiene la variables globales y el método —`main()`— desde el que se inicia la simulación. También se muestra a continuación.

Esta es la clase `Analisis`:

```

1  /*
2  *  Analisis.java
3  *
4  *  Created on April 22, 2004, 10:22 PM
5  */
6
7  package demos.grano;
8
9  //
10 import galatea.glider.*;
11
12 /**
13  *
14  *  @author Jacinto Davila
15  *  @version beta 1.0
16  */
17 public class Analisis extends Node {
18
19     /** Creates new Analisis */

```

```

20     public Analisis() {
21         super(" Analisis", 'A');
22         Glider.nodesl.add(this);
23     }
24
25     public void paint() {
26         for ( int x = 0; x < (int) Modelo.variableDep ; x++ ) {
27             System.out.print(' ');
28         }
29         System.out.println('x');
30     }
31
32     /** funcion de activacion del nodo */
33     public boolean fact(){
34         paint();
35         it(1);
36         return true; }
37 }

```

Como habrán notado, su estructura es muy parecida a la de la clase —Demo—. No hay atributos de clase (aunque podrían incluirse (No hay ninguna restricción al respecto) y se cuentan 3 métodos. Aparecen el constructor y —fact()—, como en —Demo—. La diferencia está en —paint()—. Pero este no es más que un mecanismo para visualizar, dibujando una gráfica en la pantalla del computador (con puntos y x's), el valor de una variable.

—Analisis— es también un nodo autónomo que quizás no corresponda a ningún componente del sistema simulador, pero es parte de los dispositivos para hacer seguimiento a la simulación. Por eso se le incluye en este ejemplo. El lector debe notar, como detalle crucial, que la ejecución de los códigos —fact()— de —Demo— y —Analisis— se programan concurrentemente. Concurrencia es otro concepto clave en simulación, como veremos luego.

### 2.2.3. El método principal del simulador

Estamos listos ahora para conocer el código de la clase Modelo:

```

1  /*
2   * Modelo.java
3   *
4   * Created on April 22, 2004, 10:13 PM
5   */
6
7  package demos.grano;
8
9  //import galatea.*;
10 import galatea.glider.*;
11
12 /**
13  * Este es un primer ejemplo del como implementar un modelo computacional
14  * para simulacion en Galatea.
15  *
16  * @author Jacinto Davila
17  * @version beta 1.0
18  */
19
20 public class Modelo {
21
22     /** Variables globales a todos el sistema se declaran aqui */
23     public static double variableDep = 0d ; // esta es una variable dependiente.
24     public static int variableInd = 0 ; // variable independiente.

```

```

25
26  /** No hace falta , pues usaremos una sola instancia "estatica"*/
27  public Modelo() {
28  }
29
30  /**
31   * Este es el programa principal o locus de control de la simulacion
32   * @param args the command line arguments
33  */
34  public static void main(String args[]) {
35      /** La red de nodos se inicializa aqui */
36      Delta delta = new Delta();
37      Analisis analisis = new Analisis();
38
39      Glider.setTitle("Un_modelo_computacional_muy_simple");
40      Glider.setTsim(50);
41      //Glider.setOutf(5);
42      // Traza de la simulacion en archivo
43      // Glider.trace("Modelo.trc");
44      //Glider.trace();
45      // programa el primer evento
46      Glider.act(delta,0);
47      Glider.act(analisis,1);
48      // Procesamiento de la red
49      Glider.process();
50      // Estadísticas de los nodos en archivo
51      //Glider.stat("Modelo.sta");
52      Glider.stat();
53  }
54
55 }
```

Quizás el detalle más importante a destacar en esa clase es el uso del modificador Java **static**, en la declaración (la firma decimos en la comunidad OO) de los atributos y métodos de la clase. El efecto inmediato de la palabra **static** es, en el caso de los atributos, es que los convierte en **variables de clase**. Esto significa que “existen” (Se reserva espacio para ellos y se les puede direccionar, pues su valor permanece, es estático) existan o no instancias de esa clase. No hay que crear un objeto para usarlos.

Esto es lo que nos permite en el simulador usarlos como variables de globales de la simulación. Lo único que tiene que hacer el modelista es recordar el nombre de su clase principal y podrá acceder a sus variables. Esto hemos hecho con `—Modelo.variableDep—` y `—Modelo.variableInd—`.

El lector habrá notado que el `—static—` también aparece al frente del `—main()—`. Es obligatorio usarlo en el caso del `—main()—`, pero se puede usar con cualquier método para convertirlo en un método de clase. Como con las variables de clase, un método de clase puede ser invocado sin crear un objeto de esa clase. Simplemente se le invoca usando el nombre de la clase. En Galatea usamos mucho ese recurso. De hecho, todas las instrucciones en el código de `—main()—` que comienzan con `—Glider.—` son invocaciones a métodos de clase (declarados con `—static—` en la clase `—galatea.glider.Glider—`). Veamos cada uno de esos:

`—Glider.setTitle(“Un modelo computacional muy simple”);—` Le asigna un título al modelo que será incluido en los reportes de salida.

`—Glider.setTsim(50);—` Fija el tiempo de simulación, en este caso en 50.

`—Glider.setOutf(5)—` Fija el número de decimales en las salidas numéricas.

- Glider.trace("Modelo.trc");**— Designa el archivo `Modelo.trc` para que se coloque en él la traza que se genera mientras el simulador corre. Es una fuente muy importante de información a la que dedicaremos más espacio posteriormente. El nombre del archivo es, desde luego, elección del modelista. El archivo será colocado en el directorio desde donde se ejecute el simulador<sup>4</sup>
- Glider.act(delta,0);**— Programa la primera activación del nodo `delta` (el objeto que representa el nodo `delta` del sistema simulado. Aclaremos esto un poco más adelante) para que ocurra en el tiempo `0` de simulación. Las activaciones posteriores, como vimos, son programadas desde el propio nodo `delta`, con la invocación `it(1)`.
- Glider.act(analysis,1);**— Programa la primera activación del nodo `analysis`. Luego del tiempo `1`, y puesto que `analysis` también invoca a `it(1)`, ambos nodos se activarán en cada instante de simulación. Esta es la primera aproximación a la concurrencia en simulación.
- Glider.process();**— Comienza la simulación.
- Glider.stat("Modelo.sta");**— Designa al archivo

Solamente nos resta comentar este fragmento de código:

```
/** La red de nodos se inicializa aqui */
Delta delta = new Delta();
Analisis analisis = new Analisis();
```

En estas líneas se crean los objetos que representan, en la simulación, los componentes del sistema simulado. Ya explicamos que el único componente “real” es `delta`, pero `analysis` también debe ser creado aquí aunque solo se trate de un objeto de visualización.

Note el lector que podríamos usar clases con elementos `static` para crear esos componentes. Pero quizás lo más trascendental de poder modelar al Nivel Java es precisamente la posibilidad de crear y destruir componentes del sistema simulado durante la simulación, usando el código del mismo modelo. Esto no es posible en el viejo Glider. Será siempre posible en el Nivel Glider de Galatea, pero sólo porque mantendremos el acceso directo al entorno OO de bajo nivel (Java seguirá siendo el lenguaje matriz).

Esa posibilidad de crear y destruir nodos es importante porque puede ser usada para reflejar el **cambio estructural** que suele ocurrir en los sistemas reales. Este concepto se ha convertido en todo un proyecto de investigación para la comunidad de simulación, pues resulta difícil de acomodar en las plataformas tradicionales.

## 2.2.4. Cómo simular con Galatea, primera aproximación

Hemos concluido la presentación del código de nuestro primero modelo de simulación Galatea. Para simular debemos 1) compilar los códigos Java a `.class` (con el `javac`, como se explica en el capítulo ??) y ejecutar con la máquina virtual (por ejemplo con `java`, pero también podría ser con `appletviewer` o con un Navegador y una página web, si convertimos el modelo en un applet).

La figura 2.1 muestra la salida que produce nuestro modelo en un terminal de texto (un *shell* GNU, Unix o Windows).

No es un gráfico muy sofisticado, pero muestra el cambio de estado del sistema a lo largo del tiempo, con un mínimo de esfuerzo. Mucha más información útil se encontrará en los archivos de

<sup>4</sup>Cuidado con esto. Si usa un ambiente de desarrollo como el Sun One Studio, este colocará esa salida en uno de sus directorios.



traza (—Modelos.trc—) y de estadísticas (—Modelo.sta—). Hablamos sobre ese tipo de salida más adelante en el libro.

Dejamos hasta aquí, sin haber conocido demasiado, la simulación tradicional. En la siguiente parte del capítulo conoceremos un modelo de simulación multi-agente.

## 2.3. El primer modelo computacional multi-agente

En esta parte del tutorial usamos elementos fundamentales de lo que se conoce como la tecnología de agentes inteligentes. En beneficio de una primera lección sucinta, no entraremos en detalles conceptuales. Sin embargo, un poco de contexto es esencial.

La ingeniería de sistemas basados en agentes (*Agent-Based Modelling*, ABM, como se le suele llamar) es un desarrollo reciente en las ciencias computacionales que promete una revolución similar a la causada por la orientación por objetos. Los agentes a los que se refiere son productos de la Inteligencia Artificial, IA, [7], una disciplina tecnológica que ha sufrido un cambio de enfoque en los últimos años: los investigadores de la antigua (pero buena) Inteligencia Artificial comenzaron a reconocer que sus esfuerzos por modelar un dispositivo inteligente carecían de un compromiso sistemático (como ocurre siempre en ingeniería) con la posibilidad de que ese dispositivo “hiciera algo”. Ese “hace algo inteligentemente” es la característica esencial de los agentes (inteligentes) tras la que se lanza el proyecto de IA aproximadamente desde la década de los noventa del siglo pasado.

El poder modelar un dispositivo que puede hacer algo (inteligente) y el enfoque modular que supone concebir un sistema como constituido por agentes (algo similar a concebirlo constituido por objetos) hacen de esta tecnología una herramienta muy efectiva para atacar problemas complejos [8], como la gestión del conocimiento y, en particular, el modelado y simulación de sistemas [9].

La noción de agente tiene, no obstante, profundas raíces en otras disciplinas. Filósofos, psicólogos y economistas ha propuesto modelos y teorías para explicar qué es un agente, pretendiendo explicar al mismo tiempo a todos los seres humanos. Muchas de esas explicaciones refieren como es que funcionamos en una suerte de ciclo en el que observamos nuestro entorno, razonamos al respecto y actuamos, procurando atender a lo que hemos percibido, pero también a nuestras propias creencias e intenciones. Ese ciclo en cada individuo, lo convierte en causante de cambios en el entorno o ambiente compartido normalmente con otros agentes, pero que puede tener también su propia disciplina de cambio. Algunos de los cambios del ambiente son el efecto sinérgico de la acción combinada de dos o más agentes.

Esas explicaciones generales han inspirado descripciones lógicas de lo que es un agente y una sociedad de agentes. En el resto del libro revisamos algunas de ellos, las mismas que usamos como especificaciones para el diseño de Galatea. No hay, sin embargo, ningún compromiso de exclusividad con ninguna de ella y es muy posible que Galatea siga creciendo con la inclusión de nuevos elementos de la tecnología de agentes.

### 2.3.1. Un problema de sistemas multi-agentes

**Biocomplejidad** es el término seleccionado por la *National Science Foundation* de los EEUU para referirse a “el fenómeno resultante de la interacción entre los componentes biológicos, físicos y sociales de los diversos sistemas ambientales de la tierra”. Modelar sistemas con esas características es una tarea difícil que puede simplificarse con el marco conceptual y las plataformas de sistemas multi-agentes.

El ejemplo que mostramos en esta sección es una versión simplificada de un “modelo juguete” que ha sido construido como parte del proyecto “*Biocomplexity: Integrating Models of Natural and Human Dynamics in Forest Landscapes Across Scales and Cultures*”, NFS GRANT CNH BCS-0216722.

Galatea está siendo usada para construir modelos cada vez más complejos de la dinámica apreciable en la Reserva Forestal de Caparo<sup>5</sup>, ubicada en los llanos de Barinas, al Sur-Occidente de Venezuela.

Hemos simplificado el más elemental de esos modelos juguetes<sup>6</sup> y lo hemos convertido en un **juego de simulación**, en el que el **simulista** debe tomar el lugar del ambiente. Los agentes están implementados en una versión especial del paquete —galatea.glorias—<sup>7</sup>.

El resultado es un modelo en el que el simulista es invitado a llevar un registro manual de la evolución de una Reserva Forestal (inventando sus propias reglas, con algunas sugerencias claro), mientras sobre ese ambiente actúa un conjunto de agentes artificiales que representan a los actores humanos de la Reserva.

Nuestra intención con el juego es familiarizar a los lectores con el desarrollo de un modelo multi-agente, sin que tengan que sufrir sino una pequeña indicación de la complejidad intrínseca de estos sistemas.

### 2.3.2. Conceptos básicos de modelado multi-agente

Siguiendo la práctica habitual en el modelado de sistemas, para crear el *modelo socio-económico-biológico* de la Reserva Forestal de CAPARO, construimos las descripciones a partir de un conjunto de conceptos básicos. Esos conceptos son: estado del sistema, dinámica, proceso y restricciones contextuales.

**El estado del sistema** es la lista de variables o atributos que describen al sistema en un instante de tiempo dado. Es exactamente equivalente a lo que en simulación se suelen llamar variables de estado. Pero también puede ser visto como una lista de atributos del sistema que cambian a lo largo del tiempo y que no necesariamente corresponde a magnitudes medibles en números reales. Los juicios de valor (cierto o falso) también se pueden incluir entre los atributos. En virtud de su naturaleza cambiante, nos referiremos a esos atributos como **fluentes** o **propiedades** del sistema que cambian al transcurrir el tiempo. Ya sabemos que, en Galatea, corresponderán a atributos de algún objeto.

**Una dinámica** es una disciplina, normalmente modelada como una función matemática del tiempo, describiendo como cambia una o varias de esas variables de estados (atributos o fluentes). Noten, sin embargo, que esas funciones del tiempo no necesariamente tienen que ser sujetas a otros tratamientos matemáticos. Para describir agentes, por ejemplo, puede ser preciso recurrir a funciones discretas (no derivables) parciales sobre el tiempo.

**Un proceso** es la conflagración de una o más de esas dinámicas en un arreglo complejo donde se afectan unas a otras. Es decir, el progreso de una dinámica al pasar el tiempo está restringido por el progreso de otra dinámica, en cuanto a que sus trayectorias (en el sentido de [10] están limitadas a ciertos conjuntos particulares (de trayectorias posibles).

**Una restricción contextual** es una limitación sobre los valores de las variables de estado o sobre las dinámicas que pueden estar involucradas en los procesos. Es decir, es una especificación de cuáles valores o dinámicas NO pueden ser parte del sistema modelado.

Le pedimos al lector que relea estas definiciones. Las usaremos a continuación en la medida en que conocemos el modelo juguete el cual está compuesto por:

<sup>5</sup>Detalles en —<http://cesimo.ing.ula.ve/INVESTIGACION/PROYECTOS/BIOCOMPLEXITY/>—

<sup>6</sup>Programado originalmente por Niandry Moreno y Raquel Quintero.

<sup>7</sup>adaptado al caso por Niandry Moreno, Raquel Quintero y Jacinto Dávila.

**La clase —Colono—** que define a los agentes de tipo colono que ocupan una Reserva. Varias instancias de esta clase hacen de este un modelo multi-agente.

**La clase —Delta—** que implementa la función de transición del ambiente natural. En este caso, es solo un mecanismo de consulta e interacción con el simulista, quien estará “simulando”, por su cuenta, la evolución del ambiente natural.

**La clase —Modelo—** que, como antes, contiene el programa principal y las variables globales del simulador.

**La clase —Interfaz—** que es uno de los elementos más importantes del simulador multi-agente: el conjunto de servicios que median entre los agentes y el ambiente cuando aquellos actúan y observan sobre este.

Con este modelo computacional queremos caracterizar el proceso en la Reserva Forestal que está siendo ocupada por Colonos con distintos planes de ocupación. La pregunta principal del proyecto es si existe una combinación de planes de ocupación que pudiese considerarse como intervención sustentable sobre la reserva, dado que no compromete la supervivencia de sus biodiversidad.

En nuestro ejemplo, estaremos muy lejos de poder abordar esa pregunta, pero confiamos que el lector encuentre el juego interesante e iluminador en esa dirección.

### 2.3.3. El modelo del ambiente

Llamamos —Delta— a la clase que implementa el modelo del ambiente. En este modelo juego, —Delta— es solo una colección de rutinas para que el simulador interactúe con el simulista y sea este quien registre el estado del sistema.

El propósito de esto es meramente instruccional. El simulista tendrá la oportunidad de identificar los detalles operativos de la simulación y no tendrá problemas de automatizar ese registro en otra ocasión.

Sin embargo, para no dejar al simulista desválido frente a la complejidad de un ambiente natural, incluimos las siguientes recomendaciones y explicaciones:

Una reserva forestal es un ambiente natural sumamente complejo. Cualquier descripción dejará fuera elementos importantes. Con eso en mente, nos atrevemos a decir que son 3 los procesos macro en la reserva forestal: 1) clima, 2) hidrología y relieve, con la dinámica de suelos y 3) vegetación que siempre será un resumen de la extraordinaria colección de dinámicas de crecimiento, reproducción y muerte de la capa vegetal.

Noten que estamos dejando de lado, expresamente, el proceso de la población animal, salvo, claro, por aquello que modelaremos acerca de los humanos, usando agentes. Sin embargo, el efecto del proceso animal debe ser tomando en cuenta en el proceso vegetación, particularmente el consumo de vegetación por parte del ganado que puede ser devastador para la reserva.

Para ayudar a nuestro simulista a llevar un registro de esos procesos y sus dinámicas, le sugerimos llevar anotaciones en papel del estado global de la reserva a lo largo del tiempo. Quizás sobre un mapa de la reserva, con la escala adecuada (al tiempo disponible para jugar), se pueda llevar ese registro mejor usando un código de símbolos como el que mostramos en la figura 2.2<sup>8</sup>

---

<sup>8</sup>El simulista bien podría fotocopiar esa imagen y recortar los símbolos para marcar el mapa, como solían hacer los cartógrafos (ahora lo hacen las máquinas, se admite, pero no es igual de divertido).

Los símbolos de clima son suficientemente claros. Los de hidrología y relieve son un poco más difíciles de interpretar, pero como cambian con poca frecuencia no debería haber mucho problema. La dinámica de cambio más compleja será la de la vegetación.

Lo que mostramos en la lámina es una idea de los tipos de cobertura vegetal. El más apreciado en una reserva de biodiversidad forestal es el que llamamos bosque primigenio, no tocado por la acción humana. Una vez que ha sido intervenido se convierte en bosque intervenido (y nunca más será primigenio). Pero los espacios naturales también pueden ser usados para plantaciones de especies explotables por su madera (como las coníferas que se ilustran con la figura). Los matorrales son, normalmente, bosques que han sido intervenidos y luego de ser abandonados (por los humanos y sus animales), han prosperado por su cuenta. El otro uso del espacio que es crucial, sobretodo para la ganadería, es la sabana. Finalmente, cuando el espacio es devastado sin posibilidad de recuperación se crea un desierto irrecuperable. Nuestro simulista querrá aprovechar estas indicaciones para modelar sus propias reglas de **cambio de uso de la tierra**.

El único detalle adicional a tener presente es la escala de tiempo en el que ocurren los cambios (por lo menos para la observación superficial). En el caso del clima, todos sabemos, es de minutos, horas, días y meses. En el caso de vegetación es de meses y años. En el caso de hidrología y suelos, va desde años (con las crecidas e inundaciones estacionales) hasta milenios.

Ahora podemos discutir el código de —Delta—:

```

1 package demos.biocaparo.toy;
2
3 import galatea.glider.*;
4 import galatea.hla.*;
5 import galatea.glorias.*;
6
7 public class Delta extends Node {
8     /**this variable indicates the quantity of settler agent at the forest reserve.*/
9     */
10    public int NUMCOLONOS=3;
11    /**This is an array of references to settler agents.*/
12    */
13    public Colono[] agente= new Colono[ NUMCOLONOS];
14
15    /** Creates new Delta */
16    public Delta() {
17        super("Delta", 'A');
18        Glider.nodesl.add(this);
19
20        /* Las instrucciones System.out.println( han sido editadas
21        "Este es un juego de simulacion multiagente de una Reserva Forestal");
22        "Tiene Ud "+NUMCOLONOS+" agentes actuando sobre una reserva forestal.");
23        "Su mision es decidir como evolucionará ese ambiente a medida
24        que pasa el tiempo y");
25        "que esos agentes actuan. Ud deberá indicar, tambien, como ven los
26        agentes su entorno");
27        "en la medida en que pasa el tiempo y cual será el efecto de sus acciones");
28        "Las consideraciones sobre el ambiente incluyen decisiones sobre:";
29        "clima, relieve, hidrologia y vegetacion, esta ultima, al parecer,
30        el factor mas importante");
31        "_____");
32        "Le aconsejamos que tome nota de todos los detalles");
33        "";
34        "Comienza el juego!");
35    */

```

```

36
37 //Se crean las instancias de cada uno de los agentes colono
38 //Esto deberia ir junto con la declaracion de la red en el programa
39 //principal del modelo
40 for (int i=0;i<NUMCOLONOS; i++){
41     agente[i]=new Colono();
42     agente[i].agentId= i+1;
43     GInterface.agentList.add(agente[i]);
44 }
45 }
46
47 /** funcion de activacion del nodo */
48 public boolean fact(){
49     it(1);
50
51     try{
52         /*
53         "_____ Es el tiempo "+Glider.getTime());
54         "Ambiente: Responda a cada una de estas preguntas con cuidado:";
55         "Ambiente: 1.- Donde esta cada uno de los agentes?";
56         "Ambiente: 2.- Como es el clima en cada lugar?";
57         "Ambiente: 3.- Como se estan comportando los torrentes
58         y el nivel freatico";
59         "Ambiente: 4.- Que cambios tienen lugar en la vegetacion";
60         "Ambiente: 5.- Que puede observar cada uno de los agentes";
61         "Ambiente: Continuar?(S/N)";
62         InputStreamReader ir = new InputStreamReader(System.in);
63         BufferedReader in = new BufferedReader(ir);
64         if (!si(in)) {
65             System.out.println("Ambiente: Fin del juego!");
66             System.exit(0);
67         }
68     } catch( Exception e) { };
69
70     for (int l=0;l<NUMCOLONOS;l++){
71         // Actualiza el reloj de cada agente
72         agente[l].clock=Glider.getTime();
73         // les transmite lo que deben ver
74         actualizarsensores(agente[l]);
75         // activa el razonado de cada agente
76         agente[l].cycle();
77     }
78     System.out.println("Simulador: Comienzo a procesar las influencias");
79     GInterface.gatherInfluences_test();
80     GInterface.process_test();
81     return true ;
82 }
83
84 public void actualizarsensores(Colono agente){
85
86     try {
87         InputStreamReader ir = new InputStreamReader(System.in);
88         BufferedReader in = new BufferedReader(ir);
89         System.out.println("Ambiente: Hablemos del agente: "+agente);
90         // La entrevista
91         System.out.println("Ambiente: Se establecio ya? (S/N)");
92         if (!si(in)) {
93             agente.inputs.add("No establecido");
94         }
95         // La entrevista 2
96         System.out.println("Ambiente: Ve una celda desocupada? (S/N)");

```

```

97         if (si(in)) {
98             agente.inputs.add("Celda desocupada");
99         }
100         // La entrevista 3
101         System.out.println("Ambiente: Ve una celda apta para establecerse? (S/N)");
102         if (si(in)) {
103             agente.inputs.add("Celda apta para establecerse");
104         }
105         // La entrevista 4
106         System.out.println("Ambiente: Ve una celda apta para sembrar? (S/N)");
107         if (si(in)) {
108             agente.inputs.add("Celda apta para sembrar");
109         }
110         // La entrevista 5
111         System.out.println("...: Se agotaron los recursos en su parcela? (S/N)");
112         if (si(in)) {
113             agente.inputs.add("Se agotaron los recursos");
114         }
115     } catch (Exception e) { };
116     System.out.println("Ambiente: Este agente observara "+agente.inputs);
117 }
118
119 public boolean si(BufferedReader in) {
120     int c, i;
121     int resp [] = new int[10];
122     i = 0;
123     try {
124         while ( (c=in.read()) != -1 && c!=10) { resp[i]=c ;}
125     } catch (Exception e) {};
126     return (char) resp[0]== 'S' ;
127 }
128
129 }

```

Comentaremos sobre tres fragmentos de ese código en detalle:

```

for (int i=0;i<NUM_COLONOS;i++){
    agente[i]=new Colono();
    agente[i].agentId= i+1;
    GInterface.agentList.add(agente[i]);
}

```

Esta pieza de código crea los agentes de esta simulación. Suele colocarse en el `—main()` de `—Modelo—`, pero puede aparecer, como en este caso, en otro lugar, siempre que se invoque al comienzo de la simulación (en este caso, al crear el ambiente).

Note la invocación a `—GInterface.agentList.add(agente[i])—` con la que se registra a cada agente en una base de datos central para el simulador, gestionada por la clase `GInterface` del paquete `galatea.hla`. Así es como el simulador conoce a todos los agentes de la simulación.

El siguiente fragmento es puro código Java. Es la manera de leer desde el teclado y poder identificar las respuestas del usuario. El método `—si—`, incluido en el código simplemente lee el teclado y si el texto leído comienza con el carácter `—S—`, devuelve `true`. Si la respuesta no es afirmativa, el simulador termina su ejecución con la invocación a `—System.exit(0)—`.

```

InputStreamReader ir = new InputStreamReader(System.in);
BufferedReader in = new BufferedReader(ir);
if (!si(in)) {
    System.out.println("Ambiente: Fin del juego!");
    System.exit(0);
}

```

El último fragmento es sumamente importante para el simulador multi-agente:

```

for (int l=0;l<NUM_COLONOS;l++){
    // Actualiza el reloj de cada agente
    agente[l].clock=Glider.getTime();
    // les transmite lo que deben ver
    actualizarsensores(agente[l]);
    // activa el razonado de cada agente
    agente[l].cycle();
}

```

Allí se le dice a cada agente que hora es, que cosas está observando a esta hora y se le pide que piense y decida que hacer (`—cycle()`—).

Los detalles del cómo funciona todo esto aparecen en el resto del libro. Por lo pronto, le pedimos al lector que recuerde que este es un modelo de tiempo discreto. Cada vez que el ambiente cambia, el mismo ambiente indica los cambios a los agentes y espera su respuesta.

Implementaciones en las que el ambiente y los ambientes corren “simultáneamente” también son posibles con Galatea. Pero requieren más esfuerzo de implementación, como mostraremos más adelante.

### 2.3.4. El modelo de cada agente

En Galatea, un agente es un objeto con una sofisticada estructura interna. Considere este código<sup>9</sup>:

```

1 package demos.biocomplejidad.toy;
2
3 import galatea.glider.*;
4 import galatea.glorias.*;
5 import galatea.hla.*;
6
7 public class Colono extends Ag{
8     //Atributos que definen el estado interno del agente.
9     int x=-1;
10    int y=-1;
11    int recursos_economicos;
12    int num_celdas=0;//numero de celdas que posee
13    int MAX_NUM_CELDAS=9;
14    int [][] propiedad=new int [MAX_NUM_CELDAS][2];
15    int tr; //tiempo en la region invadida.
16
17    public Colono() {
18        super(6,"colono");
19        this.population++;
20        this.agentId=population;
21        init();
22    }
23    //-----
24    public void init(){
25        outputs = new LOutputs();
26        //finding a place(0)
27        addPermanentGoal("buscarsitio",0,new String []{"No_establecido"});
28        //settling down(1)
29        addPermanentGoal("establecerse",1,
30            new String []{"Celda_desocupada","Celdaapta_para_establecerse"});
31        //cleaning the land and seeding agriculture of subsistence(2)
32        addPermanentGoal("limpiarsembrar",2,
33            new String []{"Celda_desocupada","Celdaapta_para_sembrar"});
34        //deforesting and selling wood to illegal traders(3)
35        addPermanentGoal("talarvender",3,
36            new String []{"Celda_desocupada","Celda_con_madera_comercial"});

```

<sup>9</sup>Por claridad, hemos retirado todos los comentarios en extenso. Estarán en el software que acompaña el libro.

```

37     //moving(4)
38     addPermanentGoal("mudarse",4,
39         new String [] {"Se_agotaron_los_recursos"});
40     //expanding settler's farms(5)
41     addPermanentGoal("expandirse",5,
42         new String [] {"Celda_desocupada", "Celdaapta_para_expandir"});
43 }
44 //-----
45 public void buscarsitio(){
46     Object [] args=new Object [1];
47     args[0]=this;
48     //Cargar los otros atributos necesarios en el arreglo args.
49     //args.add(x);
50     //args.add(y);
51     Output o= new Output(" buscarsitio", args);
52     outputs.add(o);
53     System.out.println(" Agent:_" +this.agentId+"_tratara_de_buscarsitio()");
54 }
55 public void establecerse(){
56     Object [] args=new Object [] { this };
57     Output o= new Output(" establecerse", args);
58     outputs.add(o);
59     System.out.println(" Agent:_" +this.agentId+"_tratara_de_establecerse()");
60 }
61 public void limpiarsembrar(){
62     Object [] args=new Object [] { this };
63     Output o= new Output(" limpiarsembrar", args);
64     outputs.add(o);
65     System.out.println(" Agent:_" +this.agentId+"_tratara_de_limpiarsembrar()");
66 }
67 public void talarvender(){
68     Object [] args=new Object [] { this };
69     Output o= new Output(" talarvender", args);
70     outputs.add(o);
71     System.out.println(" Agent:_" +this.agentId+"_tratara_de_talarvender()");
72 }
73 public void mudarse(){
74     Object [] args=new Object [1];
75     args[0]=this;
76     //Cargar los otros atributos necesarios en el arreglo args.
77     //args.add(x);
78     //args.add(y);
79     Output o= new Output(" mudarse", args);
80     outputs.add(o);
81     System.out.println(" Agent:_" +this.agentId+"_tratara_de_mudarse()");
82 }
83 public void expandirse(){
84     Object [] args=new Object [] { this };
85     Output o= new Output(" expandirse", args);
86     outputs.add(o);
87     System.out.println(" Agent:_" +this.agentId+"_tratara_de_expandirse()");
88 }
89 }

```

De nuevo concentraremos la atención, esta vez en 2 piezas del código:

El método `—init()` es una muestra de lo que tenemos que hacer cuando queremos fijar las metas del agente desde el principio. Un agente tiene metas, es decir, las reglas que determinan lo que querrá hacer.

```

1 public void init(){
2     outputs = new LOutputs();

```

```

3 | //finding a place(0)
4 | addPermanentGoal("buscarsitio",0,new String []{"No_establecido"});
5 | //settling down(1)
6 | addPermanentGoal("establecerse",1,new String []{"Celda_desocupada",
7 | "Celda_apta_para_establecerse"});
8 | //cleaning the land and seeding agriculture of subsistence(2)
9 | addPermanentGoal("limpiarsembrar",2,new String []{"Celda_desocupada",
10 | "Celda_apta_para_sembrar"});
11 | //deforesting and selling wood to illegal traders(3)
12 | addPermanentGoal("talarvender",3,new String []{"Celda_desocupada",
13 | "Celda_con_madera_comercial"});
14 | //moving(4)
15 | addPermanentGoal("mudarse",4,new String []{"Se_agotaron_los_recursos"});
16 | //expanding settler's farms(5)
17 | addPermanentGoal("expandirse",5,new String []{"Celda_desocupada",
18 | "Celda_apta_para_expandir"});
19 | }

```

En este método, luego de crear la lista que guardará las salidas que este agente envía a su ambiente, se le asignan (al agente) sus metas permanentes (es decir, sus metas durante la simulación).

El formato de invocación de `addPermanentGoal` es, básicamente, una regla de **si** condiciones **entonces** acción. Por ejemplo, para decirle al agente que **si no se ha establecido**, **entonces** debe *buscar un sitio* para hacerlo, escribimos:

```

1 | addPermanentGoal("buscarsitio",0,
2 | new String []{"No_establecido"});

```

La cadena “No establecido” será una entrada desde el ambiente para el agente (como se puede verificar en la sección anterior). “buscarsitio” es el nombre de uno de sus métodos, definido en el código que sigue y que prepara las **salidas** del agente para que las **ejecute** el ambiente como **las influencias de ese agente**.

```

1 | public void buscarsitio(){
2 |     Object[] args=new Object [1];
3 |     args[0]=this;
4 |     //Cargar los otros atributos necesarios en el arreglo args.
5 |     //args.add(x);
6 |     //args.add(y);
7 |     Output o= new Output("buscarsitio",args);
8 |     outputs.add(o);
9 |     System.out.println("Agent:_"+this.agentId+"_tratará_de
10 | _buscarsitio()"); }

```

Hay razones precisas para esta forma del objeto `Output` que tendrán un poco más de sentido en la próxima sección.

### 2.3.5. La interfaz Galatea: primera aproximación

La clase Interfaz es sumamente difícil de explicar en este punto, sin el soporte conceptual de las influencias. Baste decir que se trata de los métodos que ejecuta el ambiente como respuesta a las salidas de los agentes. Los agentes le dicen al ambiente que quieren hacer, con sus salidas, y el ambiente responde ejecutando estos métodos.

Es por ello que, en el código que sigue, el lector podrá ver métodos con los mismos nombres que usamos para crear los objetos `Output` de cada agente. La diferencia, decimos en Galatea, es que aquellos métodos de la sección anterior son ejecutados por el propio agente. Los que se muestran a continuación son cargados (en tiempo de simulación) y ejecutados por el simulador desde el objeto `GInterface` del paquete `galatea.hla`.

```

1 | package demos.biocaparo.toy;
2 |
3 | import galatea.glider.*;

```

```

4 import galatea.hla.*;
5 import galatea.glorias.*;
6
7 public class Interfaz {
8     public void buscarsitio(double t, Agent a, Colono agente) {
9         System.out.println(" Interface: _El_agent_"
10             +a.agentId+" _busco_un_sitio_y_lo_encontro.");
11     }
12     public void establecerse(double t, Agent a, Colono agente){
13         System.out.println(" Interface: _El_agent_"
14             +a.agentId+" _se_ha_establecido.");
15     }
16     public void expandirse(double t, Agent a, Colono agente) {
17         System.out.println(" Interface: _El_agent_"
18             +a.agentId+" _expandio_su_parcela.");
19     }
20     public void limpiarsembrar(double t, Agent a, Colono agente) {
21         System.out.println(" Interface: _El_agent_"
22             +a.agentId+" _limpio_el_terreno_y_sembro.");
23     }
24     public void mudarse(double t, Agent a, Colono agente) {
25         System.out.println(" Interface: _El_agent_"
26             +a.agentId+" _ha_abandonado_su_terreno.");
27     }
28     public void talarvender(double t, Agent a, Colono agente) {
29         System.out.println(" Interface: _El_agent_"
30             +a.agentId+" _talo_y_vendio_la_madera.");
31     }
32     public void haceAlgo(double m, Agent a) {
33         System.out.println(" Interface: _El_agent_" +a.agentId+" _hizo_algo.");
34     }
35 }

```

Los métodos en este ejemplos son sólo muestras. No hacen salvo indicar al simulista que la acción se ha realizado (con lo que el simulista deberá anotar algún cambio en su registro del ambiente). Pero, desde luego, el modelista puede colocar aquí, cualquier código Java que implemente los cambios automáticos apropiados para el sistema que pretende simular.

Noten, por favor, la lista de argumentos de estos métodos. Siempre aparecen un `double` y un `Agent` como primeros argumentos. Esta es una convención Galatea y la usamos para transferir el tiempo actual (en el `double`) y una referencia al objeto agente que ejecuta la acción (en `Agent`). Los siguiente argumentos serán aquellos que el modelista desee agregar desde sus `Outputs`. Noten como, a modo ilustrativo, en este caso enviamos el objeto `Colono` como tercer argumento de algunos de los métodos.

Esta explicación, desde luego, tendría que ser suplementada con ejercicios de prueba. Pueden usar el código que se anexa y consultar con sus autores.

### 2.3.6. El programa principal del simulador multi-agente

El código de la clase principal Modelo tiene pocos cambios respecto al ejemplo anterior, pero hay uno muy importante:

```

1 package demos.biocaparo.toy;
2
3 import galatea.*;
4 import galatea.glider.*;
5 import galatea.hla.*;
6 import galatea.gloriosa.*;
7
8 /**

```

```

9 |  *@author Niandry Moreno.
10 |  *@version Juguete #1.
11 |  */
12 |  public class Modelo {
13 |      public static Delta ambiente = new Delta();
14 |      /** Simulator. Main process. */
15 |      public static void main(String args []) {
16 |          GInterface.init("demos.Bioc.toy.Interfaz");
17 |          Glider.setTsim(30);
18 |          System.out.println("Inicio de simulacion");
19 |          // Traza de la simulaci{\'o}n en archivo
20 |          // Glider.trace("DemoCaparo.trc");
21 |          Glider.act(ambiente,0);
22 |          //Procesa la lista de eventos futuros.
23 |          Glider.process();
24 |          // Estadisticas de los nodos en archivo
25 |          //Glider.stat("DemoCaparo.sta");
26 |          System.out.println("La simulacion termino");
27 |      }
28 |  }

```

El cambio importante es la identificación de la clase donde se almacenan los métodos de interfaz:

```
GInterface.init("demos.Bioc.toy.Interfaz");
```

Con este cambio, el sistema está listo para simular.

### 2.3.7. Cómo simular con Galatea. Un juego multi-agente

Como antes, simplemente compilar todos los `—.java—` y, luego, invocar al simulador:

```
java demos.biocaparo.toy.Modelo
```

Disfrute su juego!

## 2.4. Asuntos pendientes

No hemos hablado todavía de varios temas muy importantes para la simulación y realizables con Galatea. Galatea es una familia de lenguajes. Es más fácil describir reglas para los agentes en un lenguaje más cercano al humano, tales como:

```
si esta_lloviendo_a_las(T) entonces saque_paraguas_a_las(T).
```

Ese código puede ser incorporado a Galatea de inmediato via Prolog y Java, como se mostrará a continuación.

El otro tema especial es concurrencia. Java es multihebrado. Galatea también, pero no solamente gracias a la herencia Java. Tenemos provisiones para que una multiplicidad de agentes hebras se ejecuten al mismo tiempo que el simulador principal (y cualquier otra colección de hebras que disponga el modelista) en una simulación coherente.

Modelar para aprovechar Galatea de esa manera requiere un dominio un poco más profundo los conceptos de sistemas multi-agentes a los que se le dedican los siguientes capítulos.

Confiamos, no obstante, que el lector estará motivado, con ejemplos anteriores, para continuar la exploración de las posibilidades que ofrece la simulación de sistemas multi-agente.



Figura 2.2: Símbolos de estado en un ambiente natural

## Referencias

---

- [1] I. Copi and C. Cohen, *Introducción a la Lógica*. . LIMUSA, 1998.
- [2] D. C. Dennett, *Consciousness Explained*. Penguin Books, 1991.
- [3] N. Gilbert and K. Troitzsch, *Simulación para las Ciencias Sociales*, 2005.
- [4] J. Martinez Coll, “A bioeconomic model of hobbes’ state of natura,” *Social Science Information*, no. 25, pp. 493–505, 1986.
- [5] “Dublin core metadata initiative,” 2006, dublin, USA. [Online]. Available: <http://www.dublincore.org/>
- [6] C. Domingo, “GLIDER, a network oriented simulation language for continuous and discrete event simulation,” in *International Conference on Mathematical Models*, Madras, India, August, 11-14 1988.
- [7] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Prentice Hall, Inc, 1995.
- [8] N. R. Jennings, “An agent-based approach for building complex software systems,” *Communications of the ACM*, vol. 44, no. 4, pp. 35–41, April 2001.
- [9] J. A. Dávila and K. A. Tucci, “Towards a logic-based, multi-agent simulation theory,” in *International Conference on Modelling, Simulation and Neural Networks [MSNN-2000]*. Mérida, Venezuela: AMSE & ULA, October, 22-24 2000, pp. 199–215. [Online]. Available: <http://citeseer.nj.nec.com/451592.html>
- [10] B. P. Zeigler, H. Praehofer, and T. G. Kim, *Theory of Modelling and Simulation*, 2nd ed. Academic Press, 2000.
- [11] G. Birtwistle, *Discrete Event Modelling on Simula*. Macmillan Press, 1979.
- [12] M. Cabral, “Prototipo del módulo GUI para la plataforma de simulación galatea,” Universidad de Los Andes. Mérida. Venezuela, 2001, tutor: Uzcátegui, Mayerlin.
- [13] O. Garcia and R. Gutierrez, “An update approach to complexity from and agent-centered artificial intelligence perspective,” *Encyclopedia of Library and Information Science*, vol. 68, supplement 31.



## De la mano con Java

---

Este apéndice comienza con una revisión, estilo tutorial, de los conceptos básicos subyacentes al desarrollo de software orientado a los objetos (OO) en el lenguaje de programación Java.

No se trata, simplemente, de pagarle servicio al lenguaje Java que estamos usando en casi todo este documento para modelar sistemas que simularemos en la plataforma Galatea. Lo que queremos hacer va un poco más allá. Los primeros esfuerzos en simulación con el computador estuvieron íntimamente ligados con esfuerzos en computación dirigidos a desarrollar lenguajes de programación más expresivos. La programación orientada a objetos surgió de esos esfuerzos y como respuesta a la urgencia de los simulistas por contar con herramientas lingüístico-computacionales para describir los sistemas que pretendían modelar y simular. Java es la culminación de una evolución que comenzó con el primer lenguaje de simulación, SIMULA [11].

Nuestra intención con Galatea es repatriar la expresividad de la orientación por objetos, junto con la enorme cantidad de mejoras de Java, para su uso, ya no en simulación tradicional, sino en simulación orientada a los AGENTES.

Este tutorial Java ha sido dispuesto en dos partes: 1) Un recorrido por lo más elemental del lenguaje, incluyendo la instalación más económica que hemos podido imaginar, de manera que el simulista pueda comenzar a trabajar con los pocos recursos que pueda reunir; y 2) Un ejercicio guiado de desarrollo que termina con una aplicación Java, con el simulador incluido y con una interfaz gráfica. Las interfaces gráficas (a pesar de que ya contamos con un IDE [12]) es uno de los aspectos más débiles de Galatea. Así que queremos enfatizar, desde el principio, que toda ayuda será bien recibida.

### A.1. Java desde Cero

Sin suponer mayor conocimiento previo, procedemos con los detalles de instalación.

#### A.1.1. Cómo instalar java

El software básico de Java<sup>1</sup> está disponible en:

<http://java.sun.com/javase/downloads/> (Abril 2009)

Al instalar el J2SE, un par de variables de ambiente serán creadas o actualizadas. En “path” se agregará el directorio donde fue montado el software y en “classpath” directorio donde están los

---

<sup>1</sup>En minúscula cuando nos refiramos al programa. En Mayúscula al lenguaje o a toda la plataforma.

ejecutables .class originales del sistema Java. Ambas variables deben se actualizadas para Galatea como se indica a continuación.

### A.1.2. Cómo instalar galatea

Galatea cuenta con una collección de clases de propósito general (no exclusivo para simulación), algunas de las cuáles aprovecharemos en este tutorial. Por ellos explicamos de inmediato como instalarla. Todo el software de Galatea viene empaquetado en el archivo galatea.jar. Para instalarlo, haga lo siguiente:

Configure el ambiente de ejecución así:

Unix:

```
setenv $GALATEA_HOME\ $ \${HOME}/galatea
setenv CLASSPATH \${CLASSPATH}:\${GALATEA_HOME}:
```

Windows:

```
set CLASSPATH = \${CLASSPATH};Galatea;Galatea\galatea.jar
```

donde */galatea*, en Unix y *Galatea*, en Windows, son nombres para el directorio donde el usuario alojara sus programas. No tiene que llamarse así, desde luego.

Coloque el archivo galatea.jar en este directorio (noten que aparece listado en el CLASSPATH).

### A.1.3. Hola mundo empaquetado

Para verificar la instalación, puede usar el siguiente código, ubicándolo en el archivo Ghola.java en el directorio donde ha instalado Galatea que llamamos directorio de trabajo en lo sucesivo:

```
1  /*
2  *  Ghola.java
3  *
4  *  Created on April 17, 2004, 12:57 PM
5  */
6
7  package demos.Ghola;
8
9  import galatea.*;
10
11 /**
12 *
13 * @author El simulista desconocido
14 * @version 1.0
15 */
16 public class Ghola {
17
18     /* @param argumentos, los argumentos de linea de comandos
19     */
20     public static void main(String args []) {
21         List unalista = new List();
22         unalista.add("Simulista");
23         System.out.println("Hola_Mundo!,_esta_lista_" + unalista);
24         System.out.println("contiene_la_palabra:_" + unalista.getDat());
25     }
26
27 }
```

Este es un código simple de una clase, GHola, con un solo método, main, que contiene una lista de instrucciones que la máquina virtual ejecutará cuando se le invoque. El programa debe antes ser “compilado con la instrucción siguiente (suponiendo que estamos trabajando en ventana con el shell del sistema operativo, desde el directorio de trabajo):

```
javac Ghola.java
```

Si todo va bien, el sistema compilará en silencio creando un archivo Ghola.class en el subdirectorio *demos* del subdirectorio *Ghola* del directorio de trabajo (verifiquen!).

Para correr<sup>2</sup> el programa use la instrucción:

```
java demos.Ghola.Ghola
```

Noten ese nombre compuesto por palabras y puntos. Ese es el nombre completo de su programa que incluye el paquete al que pertenece<sup>3</sup>. Si el directorio de Galatea está incluido en los caminos de ejecución (la variable PATH que mencionamos antes), podrá invocar su programa de la misma manera, no importa el directorio desde el que lo haga.

La salida del programa al correr será algo así:

```
Hola, esta lista List[1;1;/] -> Simulista
```

```
contiene la palabra: Simulista
```

Ud. ha corrido su primer programa en Java y en Galatea.

#### A.1.4. Inducción rápida a la orientación por objetos

Considere los siguientes dos problemas:

##### EJEMPLO 1.0

El problema de manejar la nómina de **empleados** de una institución. Todos los empleados son personas con *nombres* e *identificación*, quienes tienen asignado un *salario*. Algunos empleados son **gerentes** quienes, además de los ATRIBUTOS anteriores, tienen la responsabilidad de dirigir un departamento y reciben un *bono* específico por ello.

El problema es mantener un registro de empleados (los últimos, digamos, 12 pagos a cada uno) y ser capaces de **actualizar** salarios y bonos y de **generar** cheques de pago y reportes globales.

Para resolver el problema con la orientación por objetos, decimos, por ejemplo, que cada tipo de empleado es una **CLASE**. Cada empleado será representado por un objeto de esa clase. La descripción de la CLASE incluye también, los **MÉTODOS** para realizar las operaciones que transforman el ESTADO (los *atributos*) de cada objeto, de acuerdo a las circunstancias de uso. Observen que necesitaremos otros “objetos” asociados a los empleados, tales como cheques y reportes.

##### EJEMPLO 2.0

Queremos controlar la **dedicación** de ciertos empleados a ciertas tareas. Todo **empleado** tiene una *identificación* asociada. Además, cada empleado puede ser responsable de un número no determinado de *tareas*, cada una de las cuales consume parte de su

<sup>2</sup>correr es nuestra forma de decir: ejecutar o activar el programa.

<sup>3</sup>Esa notación corresponde también al directorio donde está el ejecutable. Revise `./demos/Ghola/Ghola` si en Unix y `.\demos\Ghola\Ghola` en Windows

*horario* semanal y de su *dedicación* anual al trabajo. Hay algunas **restricciones** sobre la dedicación anual al trabajo y sobre el número de horas asignadas a los diversos tipos de tareas.

En este caso, el objeto principal es el **reporte** de carga laboral de cada **empleado**. El empleado puede ser modelado como un objeto sencillo, uno de cuyos atributos es la *carga* en cuestión. Necesitamos también, algunos objetos que controlen la comunicación entre el usuario del sistema y los objetos anteriores y que permitan elaborar el informe de carga progresivamente.

Lo que acabamos de hacer con esos dos ejemplos es un primer ejercicio de diseño orientado a los objetos, OO. En sendos problemas, hemos identificado los tipos de objetos involucrados (a los que identificamos con **negritas** en el texto) y los atributos (*marcados así*) y sus métodos (*marcados así*). También hemos asomado la posibilidad de incluir otro elemento importante en OO, las **restricciones**.

En la secciones siguiente trataremos de aclarar cada uno de esos términos<sup>4</sup> y el cómo se reflejan en un código Java.

### A.1.5. Qué es un objeto (de software)

En OO, un objeto es una cosa virtual. En algunos casos (como en los problemas anteriores) es la representación de un objeto físico o “social”. Los objetos de software son MODELOS de objetos reales, en ese contexto.

Desde el punto de vista de las estructuras de datos que manipula el computador, un objeto es una cápsula que envía y recibe mensajes. Es una estructura de datos junto con las operaciones que la consultan y transforman. Estos son los métodos, los cuáles corresponden a las rutinas o procedimientos en los lenguajes de programación imperativa.

En lo sucesivo y siempre en OO, un objeto es simplemente una *instancia de una clase*. La clase es la especificación del objeto. Una especie de descripción general de cómo son todos los objetos de **este tipo**.

### A.1.6. Qué es una clase

Ya lo dijimos la final de la sección anterior. Una clase es una especificación para un tipo de objetos. Por ejemplo, para nuestro primer problema podríamos usar:

```

1 class Empleado {
2     String nombre;
3     String ID;
4     float salario; }

```

**Empleado** es una clase de objetos, cada uno de los cuáles tiene un **nombre** (cadena de caracteres, **String**), una identificación (otro **String**) y un **salario** (un número real, **float** en Java).

### A.1.7. Qué es una subclase

Una de las ventajas más interesantes de la OO es la posibilidad de **reusar** el software ajeno extendiéndolo y adaptándolo a nuestro problema inmediato. Por ejemplo, si alguien más hubiere creado la clase **Empleado** como en la sección anterior, para crear la clase **Gerente** (que entendemos como un tipo especial de empleado) podríamos usar:

<sup>4</sup>Salvo el de restricciones que queda lejos del alcance de un tutorial elemental

```

1  class Gerente extends Empleado {
2      String departamento;
3      float bono;
4
5      Gerente(String n, String i, float f) {
6          Empleado(n,i,f);
7          departamento = "Gerencia";
8          bono = 1.20f*f; // salario + 20% = bono
9      }
10 }

```

### A.1.8. Qué es una superclase

La posibilidad de crear subclases implica la existencia de una jerarquía de tipos de objetos. Hacia abajo en la jerarquía, los objetos son más especializados. Hacia arriba son más generales. Así la superclase (clase inmediata superior) de Empleado podría ser la siguiente si consideramos que, en general, un Empleado es una Persona:

```

1  class Persona {
2      String nombre;
3      String id;
4
5      public void Persona( String n, String id ) {
6          this.nombre = n;
7          this.id = id;
8      }
9
10     public void identifiqese( ) {
11         System.out.println( Mi nombre es  + this.id );
12     }
13 }

```

Con lo cual, tendríamos que modificar la declaración de la clase Empleado, así"

```

1  class Empleado extends Persona {
2      float salario ;
3
4      public void Empleado(String n,String id,float primerSueldo) {
5          Persona( n, id );
6          this.salario = primerSueldo ;
7      }
8
9      public void identifiqese() {
10         super.identifiqese() ;
11         if (this.salario < 1000000 ) {
12             System.out.println( ... y me pagan como a un profesor ) ;
13         }
14     }
15 }

```

En estas dos últimas especificaciones de las clase **Persona** y **Empleado** ha aparecido el otro elemento fundamental de un objeto de software: los métodos. Un método es una rutina (como los procedimientos en Pascal o las funciones en C) que implementan el algoritmo sobre las estructuras de datos y atributos del objeto (y de otros objetos asociados).

En OO, los métodos se heredan hacia abajo en la jerarquía de clases. Por eso, los métodos de nuestra clase **Persona**, son también métodos de la clase **Empleado**, con la posibilidad adicional, en

Java, de poder invocar desde una sub-clase, los métodos de su superclase, como se observa en nuestro ejemplo (`super.identifiquese()`<sup>5</sup>).

Un último detalle muy importante es la presentación de un tipo especial de método: **el constructor**, que es invocado antes de que el objeto exista, justamente al momento de crearlo (con algo como `new Constructor(Parametros);`) y cuya objetivo suele ser el de iniciar (*inicializar* decimos en computación siguiendo el anglicismo) las estructuras de datos del objeto. Noten el uso del constructor `Empleado( String n, String id, Float primerSueldo )` en el último ejemplo. Los métodos constructores no se heredan. Las previsiones para crear los objetos deben ser hechas en cada clase. Si no se declara un constructor, el compilador Java introduce uno por omisión que no tiene parámetros.

### A.1.9. Qué es poliformismo

Con lo poco que hemos explicado, es posible introducir otro de los conceptos revolucionarios de la OO: el polimorfismo. En corto, esto significa que un objeto puede tener varias formas (cambiar sus atributos) o comportarse de varias maneras (cambiar sus métodos).

Por ejemplo, suponga que Ud declara la variable `fulano` de tipo `Persona` y crea una instancia de `Persona` en `fulano`:

```
Persona fulano = new Persona();
```

Y luego, ejecuta la instrucción

```
fulano.identifiquese() ;
```

sobre ese `fulano`. La conducta que se exhibe corresponde a la del método `identifiquese` en la clase `Persona`.

Sin embargo, si Ud declara:

```
Empleado fulano = new Empleado();
```

Y luego, ejecuta la instrucción

```
fulano.identifiquese() ;
```

sobre ese `fulano`. La conducta que se exhibe corresponde a la del método `identifiquese` en la clase `Empleado`, pues en esta clase se **sobreescribió el método**. Es decir, se inscribió un método con el mismo nombre y lista de argumentos que el de la super-clase que, queda entonces, oculto.

Noten que este comportamiento diverso ocurre, a pesar que un `Empleado` es también una `Persona`. Esta es una de las dimensiones del polimorfismo.

El polimorfismo, no obstante, tiene un carácter mucho más dinámico. Por ejemplo, esta instrucción es perfectamente válida en Java, con los códigos anteriores:

```
Empleado fulano = new Gerente(Fulano de Tal,"F", 1000000f) ;
```

La variable `fulano` de tipo `Empleado`, se refiere realmente a un `Gerente` (que es una subclase de `Empleado`).

En este caso, `fulano` es un `Empleado`. Si Ud trata de utilizar algún atributo específico de la subclase, no podrá compilar. Trate con:

<sup>5</sup>Algunas versiones de la máquina virtual permiten usar `super` para invocar al constructor de la superclase, siempre que aparezca como la primera instrucción del constructor de la clase hijo. Esto, sin embargo, no siempre funciona.

```
Empleado fulano = new Gerente(Fulano de Tal,"F", 1000000f) ;
System.out.println("mi bono es "+fulano.bono);
```

Aquí bono, a pesar de ser un atributo de la clase `Gerente`, es desconocido si el objeto es de tipo `Empleado`. El compilador lo denuncia.

Al revés, en principio la instanciación también es posible:

```
Gerente fulano = (Gerente) new Empleado(Fulano de Tal, "F",
1000000f) ;
```

El compilador lo admite como válido al compilar. Sin embargo, la máquina virtual<sup>6</sup> lanzará una excepción (el error es `ClassCastException`) en tiempo de ejecución. Todo esto es parte del sofisticado sistema de seguridad de tipos de Java. No hay espacio en el tutorial para explicar todas sus implicaciones.

#### A.1.10. Qué es un agente

En la jerga de la Inteligencia Artificial, un agente es un dispositivo (hardware y software) que percibe su entorno, razona sobre una representación de ese entorno y de su conducta y actúa para cambiar ese mismo entorno (o para impedir que cambie). El término, desde luego, está asociado a la etimología tradicional (del latín *agere*: hacer) y se le usa para indicar a aquel (o aquello) que **hace algo** (posiblemente en beneficio de alguien más). La adopción del término en la literatura ingenieril de la Inteligencia Artificial tiene que ver con la aparición de una corriente o escuela de trabajo que enfatiza que los dispositivos inteligentes deben demostrar que lo son “haciendo algo”. En el extremo opuesto (los que no hacen nada) se pretendió dejar a los primeros razonadores simbólicos que concentraban todo el esfuerzo en emular el “pensar” de los humanos, con un mínimo compromiso por la acción.

Como ilustramos en este libro, en Galatea nos reencontramos con esa noción del razonador simbólico que procesa una representación de cierto dominio de conocimiento (asociado a un sistema) y usa el producto de ese procesamiento para guiar la acción de un agente. El objetivo de proyecto Galatea, en cuanto a agentes se refiere, es proveer a modelistas y simulistas de un lenguaje con suficiente expresividad para describir a los humanos que participan de un sistema que se quiere simular. Por esta razón, nuestros agentes son inteligentes.

Un agente inteligente tiene su propio “proyecto” de vida, con metas, creencias y capacidades que le permiten prestar, con efectividad, servicios para los que se requiere inteligencia

#### A.1.11. Qué relación hay entre un objeto y un agente

Algunos autores reduce la caracterización de agente a una entidad que percibe y actúa [7]. Usando la matemática como lenguaje de especificación, dicen ellos que un agente es una función con dominio en una “historia perceptual” y rango en las acciones:

$$f : \mathcal{P}^* \rightarrow \mathcal{A}$$

Esta caracterización matemática es ciertamente muy útil conceptualmente (con algunos complementos, como mostraremos en los capítulos centrales de este libro), pero lo es poco para el desarrollador de software.

Es quizás más provechoso decir que un agente es un objeto, en el sentido que hemos venido discutiendo, con una sofisticada estructura interna (estructuras de datos, atributos y métodos) y con

<sup>6</sup>Algunas máquinas virtuales) no reportan error de ejecución.

una relación intensa de intercambio con su entorno. En Galatea, esta visión de agente se refleja en la implementación de la clase `galatea.hla.Agent` y en el paquete `galatea.gloria`, como explicaremos más adelante.

### A.1.12. Por qué objetos

La programación orientada a los objetos se ha impuesto tan arrolladoramente que pocos harán ahora la pregunta que intitula esta sección. Sin embargo, queremos hacer énfasis en lo apropiado de la OO, y ahora la orientación a los agentes, OA, como herramientas para abordar la complejidad. Los profesores Oscar García y Ricardo Gutierrez, de Wright University [13], explican que una medida del esfuerzo de desarrollo en programación imperativa tal como:

$$\text{Esfuerzo de programación} = \text{constante} * (\text{líneas de código en total}) * 1,5 \quad (\text{A.1})$$

Se convierte en la OO:

$$E. \text{ de pr. en OO} = \text{constante} * \Sigma \text{objetos}(l. \text{ de c. de cada objeto}) * 1,5 \quad (\text{A.2})$$

Debemos admitir, sin embargo, el esfuerzo considerable que debe realizar un programador para aprender los lenguajes OO, buena parte del cual debe emplearlo en aprender lo que otros programadores han hecho. De su habilidad para re-utilizar, dependerá su éxito casi definitivamente.

En particular, un lenguaje como Java que conserva parte de la sintaxis de nivel intermedio del lenguaje C puede resultar muy desafiante. Por esta razón, este texto incluye el siguiente recorrido Tutorial sobre Java.

## A.2. Java en 24 horas

### A.2.1. Java de bolsillo

El material incluido en esta sección está basado en un curso dictado por Andrew Coupe, Sun Microsystems Ltd, en 1994. Ha sido actualizado con los detalles relevantes para Java 2, la nueva generación de Java y hemos cambiado su ejemplo para adecuarlo a un problema más cercano a los simulistas.

Coupe resumió las características de este lenguaje OO, así: “Java es un lenguaje pequeño, simple, seguro, orientado a objetos, interpretado o optimizado dinámicamente, que genera byte-codes, neutral a las arquitecturas, con recolección de basura, multihebrado, con un riguroso mecanismo para controlar tipos de datos, diseñado para escribir programas dinámicos y distribuidos”.

Más allá del ejercicio promocional, tradicional en un entorno industrial tan competido como este, esa última definición debe servir para ilustrar la cantidad de tecnologías que se encuentran en Java. Esa combinación junto con una política (más) solidaria de mercadeo y distribución, han convertido a Java, a nuestro juicio, en una plataforma de desarrollo muy efectiva y muy robusta.

Para el usuario final, el programador de aplicaciones, quizás la virtud que más le puede interesar de Java es la facilidad para desarrollos de software distribuidos: “Escríbalo aquí, córrelo dondequiera.”<sup>es</sup> uno de los panfletos publicitarios de Java. Significa que uno puede producir el software acabado en una máquina con arquitectura de hardware completamente diferente de aquella

en la que se ejecutará el software. Una máquina virtual se encarga de unificar las “interfases” entre programas Java y las máquinas reales.

Estas fueron las metas del diseño de Java. Las listamos aquí con una invitación al lector para que verifique su cumplimiento:

- Java tiene que ser simple, OO y familiar, para lograr que los programadores produzcan en poco tiempo. JAVA es muy parecido a C++
- Java debe ser seguro y robusto. Debe reducir la ocurrencia de errores en el software. No se permite el goto, labels, break y continue, si se permiten. Pero sin las armas blancas de doble filo: No a la aritmética de punteros (apuntadores)
- Java debe ser portátil, independiente de la plataforma de hardware.
- JAVA tiene que ser un lenguaje útil (usable).
- Java implanta un mecanismo para recolección de basura. No a las filtraciones de memoria.
- Java solo permite herencia simple entre clase (una clase solo es subclase de una más). Es más fácil de entender y, aún, permite simular la herencia múltiple con `interface`.

Java trata de eliminar las fuentes más comunes de error del programador C. No permite la aritmética de apuntadores. Esto no significa que no hay apuntadores, como algunas veces se ha dicho. En OO, una variable con la referencia a un objeto se puede considerar un apuntador al objeto. En ese sentido, todas las variables Java (de objetos no primitivos, como veremos) son apuntadores. Lo que no se permite es la manipulación algebraica de esos apuntadores que sí se permite en C o C++.

El programador no tiene que preocuparse por gestionar la memoria. Esto significa que no hay errores de liberación de memoria. El trabajo de reservar memoria para los objetos se reduce a la invocación `new` que crea las instancias, pero sin que el programador tenga que calcular espacio de memoria (como se puede hacer con algunas invocaciones `malloc` en C). Para liberar la memoria, tampoco se requiere la intervención del programador. La plataforma incorpora un “recolector de basura” (garbage collector) que se encarga de recuperar a memoria de aquellos objetos que han dejado de ser utilizados (ninguna variable apunta hacia ellos). El mecanismo puede ser ineficiente sin la intervención del usuario, pero el mismo recolector puede ser invocado a voluntad por el programador (Ver objeto `gc`).

Java es seguro por tipo. Esto quiere decir que se establecen tipos de datos muy rígidos para controlar que la memoria no pueda ser empleada para propósitos no anticipados por el programador. El mapa de memoria se define al ejecutar.

Hay un cheque estricto de tipos en tiempo de compilación que incluye chequeo de cotas, una forma de verificación antivirus inteligente, cuando las clases se cargan a través de la red. El cargador de clases les asigna a los applets de diferentes máquinas, diferentes nombres de espacios.

Los applets, estas aplicaciones Java que se descargan desde la red, operan en lo que se conoce como una “caja de arena” (como las que usamos para las mascotas) que impide que el código desconocido pueda afectar la operación de la máquina que lo recibe. Esto es protección contra caballos de troya. Los applets son revisados al ejecutarlos. El cargador chequea los nombres locales primero. No hay acceso por la red. Los applets sólo pueden acceder a su anfitrión. Los applets más allá de una firewall, sólo pueden acceder al espacio exterior.

Java pretende ser portátil (neutral al sistema operativo). Nada en JAVA es dependiente de la implementación en la máquina donde se le ejecute. El formato de todos los tipos está predefinido. La definición de cada uno incluye un extenso conjunto de bibliotecas de clases, como veremos en un momento.

Los fuente JAVA son compilados a un formato de bytecode independiente del sistema. Esto es lo que se mueve en la red. Los bytecodes son convertidos localmente en código de máquina. El esfuerzo de portabilidad es asunto de quienes implementan la máquina virtual en cada plataforma.

Java tiene tipos de datos llamados primitivos. Son enteros complemento dos con signo. byte (8 bits), short (16 bits), int (32 bits), long (64 bits); Punto flotantes IEEE754, sencillo (float) y doble (double); y Char 16 bit Unicode.

Java es interpretado o compilado justo a tiempo para la ejecución. Esto significa que código que entiende la máquina real es producido justo antes de su ejecución y cada vez que se le ejecuta. La implicación más importante de esto es que el código Java tarda más (que un código compilado a lenguaje máquina) en comenzar a ejecutarse. Esto ha sido la fuente de numerosas críticas a la plataforma por su supuesta ineficiencia. Un análisis detallado de este problema debe incorporar, además de esas consideraciones, el hecho de que los tiempos de descarga a través de la red siguen siendo un orden de magnitud superiores a la compilación just-in-time y, además, que la plataforma Java permite enlazar bibliotecas (clases y paquetes) en tiempo de ejecución, incluso a través de la red.

La otra gran idea incorporada a Java desde su diseño es la plataforma para multiprogramación. Los programadores Java decimos que Java es multihebrado. El programador puede programar la ejecución simultánea<sup>7</sup> de “hilos de ejecución”, hebras, en un mismo programa. La sincronización (basada en monitores) para exclusión mutua, un mecanismo para garantizar la integridad de las estructuras de datos en los programas con hebras, está interconstruida en el lenguaje (Ver *Threads*).

### A.2.2. Los paquetes Java

Un paquete es un módulo funcional de software y la vía de crear colecciones o bibliotecas de objetos en Java. Se le referencia con la instrucción `import` para incorporar sus clases a un programa nuevo. Lo que llamamos la Application Programming Interface, API, de Java es un conjunto de paquetes con la documentación apropiada para que sean empleados en desarrollos. La especificación JAVA incluye un conjunto de bibliotecas de clases que acompañan cualquier distribución de la plataforma: `java.applet`, `java.awt`, `java.io`, `java.lang`, `java.net`, `java.util`.

El paquete `java.applet` proporciona una clase para programas Java imbuidos en páginas web: Manipulación de applets. Métodos para manipular audioclips. Acceso al entorno del applet.

El paquete `java.awt` es la caja de herramientas para armar GUIs, recientemente mejorada con la introducción de Swing. Tanto Awt como Swing implementan los componentes GUI usuales: `java.awt.Graphics` proporciona algunas primitivas de dibujo. Se apoya en el sistema de ventanas del sistema operativo local (`java.awt.peer`). `java.awt.image` proporciona manipuladores de imágenes.

El paquete `java.io` es la biblioteca de entrada y salida: Input/output **streams**. Descriptores de archivos. **Tokenizers**. Interfaz a las convenciones locales del sistema de archivos.

El paquete `java.lang` es la biblioteca del lenguaje JAVA: Clases de los tipos (`Class`, `Object`, `Float`, `String`, etc). La Clase `Thread` provee una interfaz independiente del sistema operativo para las funciones del sistema de multiprogramación.

El paquete `java.net` contiene las clases que representan a objetos de red: Direcciones internet. URLs y manipuladores MIME Objetos `socket` para clientes y servidores.

El paquete `java.util` Un paquete para los “utilitarios”: `Date`, `Vector`, `StringTokenizer`, `Hashtable`, `Stack`.

---

<sup>7</sup>Por lo menos pseudosimultánea. El paralelismo real depende, desde luego, de contar con más de un procesador real y una máquina virtual que paralelice.

JAVA es una plataforma completa de desarrollo de software muy económica (la mayor parte es gratis). Muchos proveedores de tecnología teleinformática son propietarios de licencias de desarrollo JAVA (IBM por ejemplo) aunque Sun Microsystems sigue siendo la casa matriz.

Los Java Developers Kits son los paquetes de distribución de la plataforma y ahora se denominan Software Developer Kits. Se distribuyen gratuitamente para todos los sistemas operativos comerciales. Incluye herramientas de documentación y SDKs. Hay muchos IDEs (NetBeans, por ejemplo). Browsers Explorer y Netscape lo incorporaron desde el principio como uno de sus lenguajes script (pero, cuidado!, que Java NO ES JAVASCRIPT).

Es muy difícil producir un resumen que satisfaga todas las necesidades. Por fortuna, la información sobre Java es muy abundante en Internet. Nuestra intención aquí es proveer información mínima y concentrar la atención del lector en algunos detalles útiles para simulación en las próximas secciones.

### A.2.3. Java Libre

En el 2006, y luego de considerable presión por parte de miles de desarrolladores alrededor del mundo<sup>8</sup>, la empresa Sun, propietaria de los derechos de autor de Java (y empleadora de su creador), decidió liberar el código de la plataforma.

En la comunidad de software libre se dice que una pieza de software ha sido liberada cuando se permite distribuirle de acuerdo a ciertos términos explicados en un texto de licencia. Sun, dueños de Java, decidieron adoptar una muy conocida licencia de software libre, la GPL, con una excepción también muy popular: la excepción *classpath*. En términos simples, eso significa que el código de Java es libre para ser usado, leído, copiado y mejorado, siempre que cualquier mejora a la plataforma sea también compartida en los mismos términos. Es decir, el código Java de Sun tiene ahora “izquierdo de copia”.

Sin embargo, para no forzar a que cualquier aplicación hecha con esa plataforma fuese también libre, Sun incorporó la excepción *classpath*, inventada por la Free Software Foundation para su proyecto homónimo, con la cuál sólo las modificaciones de la plataforma obligan al izquierdo de copia. Es decir, si uno usa la plataforma como una librería para construir sus programas, no está obligado al izquierdo de copia (tal como ocurre con otras implementaciones de Java, e.g. Harmony, del proyecto Apache). Ese derecho del autor<sup>9</sup> sólo se aplicará para los cambios y mejoras sobre la plataforma de Sun.

En el proyecto Galatea hemos decidido liberar el código exactamente en los mismos términos: **GPL con la excepción *Classpath***. Esto significa que no obligaremos a nadie a liberar sus códigos si construyen invocando a Galatea. Sólo esperamos que las mejoras a la propia plataforma sean también libres y podamos así seguir construyendo una herramienta de simulación libre, eficiente y efectiva.

### A.2.4. Un applet

Un applet<sup>10</sup> es una forma inmediata de agregar dinamismo a los documentos Web. Es una especie de extensión indolora del browser, para las necesidades de cada página Web. El código se descarga automáticamente de la red y se ejecuta cuando la página web se expone al usuario. NO se requiere soporte en los servidores Internet, salvo para alojar el programa y la página Web.

<sup>8</sup>por ejemplo, aquellos que alimentan el repositorio de software libre sf.net

<sup>9</sup>sí, el izquierdo de copia es un derecho del autor

<sup>10</sup>El nombre es una variación de la palabra inglesa para aplicación “enchufable”. Estas variaciones se han vuelto comunes para la comunidad Java con la aparición de los *Servlets*, los *Midlets* y los *Aglets*, todos componentes **enchufables**.

Sin más preámbulo, esto es un applet:

```

1 import java.awt.*;
2 import java.applet.*;
3
4 public class EjemploApplet extends Applet {
5
6     Dimension app_size;
7
8     public void init() {
9         app_size = this.size();
10    }
11
12    double f(double x) {
13        return (Math.cos(x/5) + Math.sin(x/7) + 2) * app_size.height / 4;
14    }
15
16    public void paint(Graphics g) {
17        for ( int x = 0; x < app_size.width ; x++ ) {
18            g.drawLine(x, (int) f(x), x+1, (int) f(x+1));
19        }
20    }
21 }

```

Este último es un programa muy sencillo que dibuja una gráfica de la función  $f$  al momento de cargar el applet (que es cuando el sistema invoca el método `init()`. Noten que los applets no usan la función `main()` como punto de arranque los programas. Esta es reservada para las aplicaciones normales).

Para ejecutar el applet puede hacerse (luego de generar el `.class`):

```
appletviewer EjemploApplet
```

o invocarlo desde una página web como se mostrará más adelante.

### A.2.5. Anatomía de un applet

Una applet es una aplicación Java construida sobre la colección de clases en el paquete `java.applet` o `java.swing`.

Así se declara la clase que lo contiene:

```

import java.applet.*;

public class AppletMonteCarlo extends Applet {

}

```

Donde `AppletMonteCarlo` es el nombre que hemos escogido para nuestro ejemplo (y que el programador puede cambiar a voluntad, desde luego).

A diferencia de las aplicaciones “normales”, que solo requieren codificar un método `main()` como punto de arranque, los applets deben configurar, por lo menos 2, puntos de arranque. Con `init()` se indica al sistema el código que debe ejecutarse cada vez que se descargue la página web que contiene el applet. Con `start()` se indica el código que habrá de ejecutarse cada vez que el applet se exponga a la vista del usuario del navegador (cada vez que la página web se exhiba o se “maximice” la ventana del navegador).

```
import java.applet.*;

public class AppletMonteCarlo extends Applet {

    public void init() {
    }

    public void start() {
    }
}
```

### A.2.6. Parametrizando un applet

Los applets, imbuidos en páginas Web como están, tienen acceso a información en esas páginas. Por ejemplo, nuestro applet podría obtener un nombre que necesita, de una de las etiquetas de la página que lo aloja, con este código:

```
import java.applet.*;

public class AppletMonteCarlo extends Applet {
    String param;

    public void init() {
        param = getParameter("imagen");
    }
}
```

y con este texto incluido en la página, justo en el punto donde se inserta el applet. En los ejemplos que acompañan este texto, se muestra la página web (`AppletMonteCarlo.html`) que invoca a este applet, sin ese detalle del parámetro que se sugiere como ejercicio al lector.

```
<param name=imagen value=imagen.gif\>
```

### A.2.7. Objetos URL

El applet puede, además, usar la clase de apoyo para construir direcciones Web bien formadas y usarlas en la recuperación de información:

```
1  import java.applet.*;
2  import java.net.*;
3
4  public class AppletMonteCarlo extends Applet {
5      String param;
6
7      public void init() {
8          param = getParameter("imagen");
9          try {
10             imageURL = new URL(getDocumentBase(), param);
11         } catch (MalformedURLException e) {
12             System.out.println("URL mal escrito");
13             return;
14         }
15     }
16 }
```

### A.2.8. Gráficos: colores

El applet es una “aplicación gráfica”. Es decir, toda la salida se realiza a través de un panel que es una ventana para dibujar. No es de sorprender que exista una relación muy estrecha entre applets y herramientas de dibujo.

El siguiente código crea un arreglo para almacenar una paleta de 3 colores que se usará más adelante.

```

1 import java.applet.*;
2
3 int paints [] ;
4
5 /* prepara la caja de colores */
6 paints = new int [3];
7
8 paints [0]= Color . red . getRGB ();
9
10 paints [1]= Color . green . getRGB ();
11
12 paints [3]= Color . blue . getRGB ();

```

Lo más interesante de esta relación entre applets y dibujos no es sólo lo que uno puede programar, sino lo que ya ha sido programado. Este código, por ejemplo, contiene una serie de objetos que nos permiten cargar una imagen a través de la red. La imagen puede estar en alguno de los formatos profesionales de internet (.gif, jpeg, entre otros). El sistema dispone de un objeto para almacenar la imagen (**Image**) y, además de todo un mecanismo para garantizar que la imagen es cargada íntegramente (**MediaTracker**).

```

1 Image picture ;
2
3 MediaTracker tracker ;
4
5 tracker = new MediaTracker ( this ) ;
6
7 picture = this . getImage ( imageURL ) ;
8
9 tracker . addImage ( picture , 0 ) ;

```

Java provee también de objetos para que el programador puede manipular gráficos (**Graphics**) que se dibujan en su panel.

```
Graphics gc;
```

```
gc = getGraphics;
```

### A.2.9. Ejecutando el applet

El código a continuación, es una implementación del método `start()` que muestra como cargamos la imagen (con el **tracker**) y luego la dibujamos en el panel del applet (**gc.drawImage**), suponiendo, por supuesto que los objetos han sido declarados e inicializados en otros lugares del código.

```

1 import java.applet.*;
2
3 public class AppletMonteCarlo extends Applet {
4     String param;
5     ...
6     public void start () {

```

```
7     try {
8         tracker.waitForID(0);
9     } catch (InterruptedException e) {
10        System.out.print( No pude! );
11    }
12
13    image_width = picture.getWidth(this);
14    image_height = picture.getHeight(this);
15
16    gc.drawImage(picture, 0, 0, this);
17 }
18 }
```

### A.2.10. Capturando una imagen

Con muchas herramientas de visualización científica se hace un esfuerzo, quizás exagerado, por aislar al usuario de los detalles de manipulación de su data. En muchos casos, por el contrario, el usuario-programador necesita todas las facilidades para manipular esa data para sus propósitos particulares.

El código que se muestra a continuación es un ejemplo de cómo cargar la imagen que hemos obtenido a través de la red (en el objeto `picture`) en un arreglo de pixels. Un pixel corresponde a un punto en la pantalla del computador. Para el software, el pixel es un número que designa el color e intensidad que se “pinta” en cierta posición de la pantalla.

```
1 PixelGrabber pg;
2 int pixels[] ;
3
4 pixels = new int[image_width*image_height] ;
5
6 pg = new PixelGrabber(picture, 0, 0, image_width, image_height,
7 pixels, 0, image_width);
8
9 try {
10    pg.grabPixels();
11 } catch (InterruptedException e) {
12    System.err.println( No pude transformar la imagen );
13 }
```

Lo que hemos hecho ese código es capturar (agarrar, *Grabber*. Ver `PixelGrabber`) los píxeles de nuestra imagen en un arreglo desde donde los podemos manipular a voluntad (como se muestra en el ejemplo completo).

### A.2.11. Eventos

Las aplicaciones de software moderna rara vez interactúan con el usuario a través de un comando escrito con el teclado. Las formas más populares de interacción tienen que ver con el uso del ratón, de menús, de botones y de otras opciones gráficas.

Un problema con esos diversos modos de interacción es que requieren ser atendidos casi simultáneamente y en forma coherente con la aplicación y con su interfaz al usuario.

Para resolver ese problema de raíz, la plataforma Java incorporó, desde el principio, un sistema de manejo de eventos en su interfaz gráfica. Los eventos son, para variar, representados como objetos (de la clase `Events`) y dan cuenta de cualquier cosa que pase en cualquier elemento de interfaz (o en cualquier otro, de hecho). Un evento es un objeto **producido** por aquel objeto sobre el que ocurre algo (por ejemplo un botón que es presionado, *click*). El objeto que produce el evento deben tener

asociado un manejador del evento. Este es otro objeto que **escucha** los eventos (un **Listener**), los **reconoce** (de acuerdo a características predefinidas por el programador) y los **maneja**, ejecutando ciertos códigos que han sido definidos por el programador.

En un dramático cambio de dirección, Java 2 transformó el mecanismo de manejo de eventos y es por ello que algunos viejos programas Java que manipulaban eventos, tienen dificultades para funcionar.

Lamentablemente, los detalles del manejo de eventos son difíciles de resumir. Es mucho más productivo ver el código de algunos ejemplos. Este, por ejemplo, es el que usamos en el applet que hemos venido construyendo. Observen como cambia la declaración de la clase para “implementar” los objetos que escuchan eventos en cada dispositivo:

```

1 public class AppletMonteCarlo extends Applet implements
2 KeyListener , MouseListener {
3     ...
4     public void keyPressed(KeyEvent e) {
5         System.out.println("keyPressed");
6     }
7
8     public void keyReleased(KeyEvent e) {
9         System.out.println("keyReleased");
10
11         /* pasa al nuevo color */
12         paint_col++;
13         paint_col = paint_col % 3;
14
15         System.out.println("Nuevo_color:_ " + paint_col );
16     }
17
18     public void keyTyped(KeyEvent e) {
19         System.out.println("keyTyped");
20     }
21
22     public void mouseClicked(MouseEvent e) {
23         System.out.println("mouseClicked");
24
25         /* Captura la posicion del ratón */
26         int x = e.getX();
27         int y = e.getY();
28
29         /* Un click dispara la reconstrucción de la imagen */
30         if (x < image_width & y < image_height) {
31             /* llamar a la rutina de llenado: floodFill */
32             floodFill(x,y, pixels[y*image_width+x]);
33         }
34
35         /* .. y luego la redibuja */
36         repaint(); // desde el browser llama a paint()
37     }
38
39     public void mouseEntered(MouseEvent e) {
40         System.out.println("mouseEntered");
41     }
42
43     public void mouseExited(MouseEvent e) {
44         System.out.println("mouseExited");
45     }
46
47     public void mousePressed(MouseEvent e) {
48         System.out.println("mousePressed");

```

```

49     }
50
51     public void mouseReleased(MouseEvent e) {
52         System.out.println("mouseReleased");
53     }
54     ....
55 }

```

En este caso, los interesantes son los métodos `keyRelease()` y `mouseClicked()`. Los demás no hacen nada (salvo imprimir un mensaje en salida standard), pero **deben** ser implementados (Ver **interfaces** en Java).

Sugerimos al lector que, sobre el código completo de este ejemplo (en el anexo ?? ubique a los productores de eventos y a los escuchas.

### A.2.12. `paint():` Pintando el applet

Un detalle importante para cerrar la discusión sobre visualización (sobretudo la animada) es quien pinta el dibujo del applet. Lo pinta el usuario cada vez que quiere, pero también lo pinta el propio navegador (browser) se exhibe es la página, en respuesta a su propia situación (abierta, cerrada, bajo otra ventanas, etc.).

Para proveer una interfaz uniforme al sistema y al programador, se ha dispuesto del método `paint()`. El programador escribe el código de `paint`, para definir como desea que “se pinte” sus applet. Pero el método es invocado por el navegador. Si el usuario quieren forzar una llamada a `paint()`, usa `repaint()`, como se muestra en el ejemplo.

Este es el código que usamos para `paint()`. Noten el uso de otro método `update()`.

```

1 public void paint(Graphics gc) {
2     update(g);
3
4     public void update(Graphics g) {
5         Image newpic;
6
7         newpic = createImage( new
8             MemoryImageSource(image_width,
9                 image_height, pixels, 0, image_width );
10
11         g.drawImage(newpic, 0, 0, this );
12     }

```

### A.2.13. El AppletMonteCarlo completo

Como dijimos al principio de esta parte, hemos querido ofrecer un ejemplo completo de una aplicación Java funcional, destacando el manejo gráfico en el que Galatea no es tan fuerte (por falta de tiempo de desarrollo). Eso es el AppletMonteCarlo. Pero su nombre también sugiere una herramienta conceptual sumamente popular en simulación.

El método de MonteCarlo es una aplicación de la generación de números aleatorios. En nombre rememora el principado Europeo, célebre por sus casinos y sus juegos de azar (entre otras cosas).

No vamos a diluirnos en los detalles, que se oscurecen fácilmente, de la estocástica. Lo que tenemos en el ejemplo es una aplicación simple de la rutina de MonteCarlo de generación de números aleatorios para estimar áreas encerradas por un perímetro que dibuja una función.

Lo interesante del ejemplo es que la función no tiene que ser alimentada la ejemplo con su fórmula matemática o con una tabla de puntos. Una imagen (.gif o .jpeg) es el forma que el programa espera

para conocer la función sobre la que calculará el área, al mismo tiempo que cambia los colores de puntos para ilustrar el funcionamiento del algoritmo. No se ofrece como un método efectivo para el cálculo de áreas, sino como un ejemplo sencillo de la clase de manipulaciones que se pueden realizar con Java.

Noten, por favor, que en este ejemplo, el simulador de números aleatorios empleado no es el originario de Java, sino la clase **GRnd** de **Galatea**<sup>11</sup>.

Ahora pueden ir hasta el repositorio a descargar la última versión.

---

<sup>11</sup>Es muy importante notar como inicializamos la “semilla” del generador Galatea en el método `init()` del `AppletMonteCarlo`.