UNIVERSIDAD DE OVIEDO

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INDUSTRIALES E INGENIEROS INFORMÁTICOS DE GIJÓN

ÁREA DE LENGUAJES Y SISTEMAS INFORMÁTICOS

PROYECTO FIN DE CARRERA Nº 1022036

DESARROLLO DE UN SISTEMA DE PERSISTENCIA IMPLÍCITA BASADO EN PROGRAMACIÓN ORIENTADA A ASPECTOS

DOCUMENTO Nº 2

DISEÑO

JAVIER NOVAL ARANGO
JUNIO 2003

UNIVERSIDAD DE OVIEDO

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INDUSTRIALES E INGENIEROS INFORMÁTICOS DE GIJÓN

ÁREA DE LENGUAJES Y SISTEMAS INFORMÁTICOS

PROYECTO FIN DE CARRERA Nº 1022036

DESARROLLO DE UN SISTEMA DE PERSISTENCIA IMPLÍCITA BASADO EN PROGRAMACIÓN ORIENTADA A ASPECTOS

DOCUMENTO Nº 2

DISEÑO

JAVIER NOVAL ARANGO
JUNIO 2003
TUTOR: FRANCISCO ORTÍN SOLER

Índice general

Índice de figuras

Capítulo 1

CONSIDERACIONES PARA EL DISEÑO

1.1 Aspectos generales

Las principales consideraciones al diseñar el sistema fueron sencillez, claridad y facilidad de expansión y modificación, pensando que el proyecto sería usado como un posible punto de partida para nuevos proyectos de investigación. Por tanto, se han dejado de lado aspectos secundarios como pueda ser el rendimiento.

Tampoco se ha tenido en cuenta la concurrencia de aplicaciones, esto es, no se ha tenido en cuenta el caso de dos instancias de una aplicación intentando acceder simultáneamente a los objetos almacenados en el sistema de persistencia.

Los diagramas de diseño del sistema y del API del lenguaje Java−− han sido creados empleando la notación UML (ref. [2]).

1.2 División en módulos

El sistema se ha dividido en tres módulos:

- Representación interna del lenguaje
- Intérprete y tablas de símbolos
- Sistema de persistencia

Esta división se realiza fundamentalmente para facilitar la comprensión del sistema, puesto que en realidad todas sus partes están muy ligadas, y por tanto hay dependencias mutuas entre todos los paquetes.

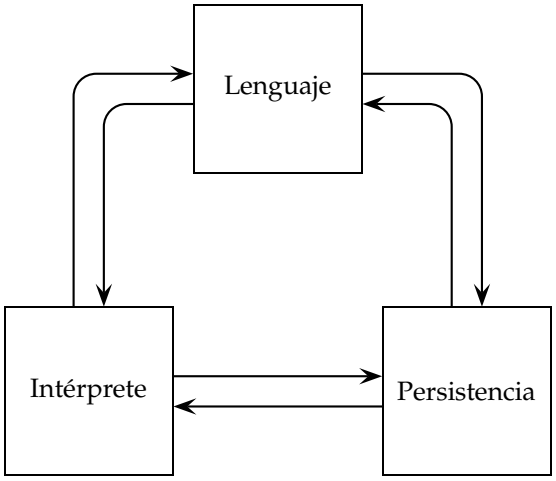


Figura 1.1: Módulos en los que se ha dividido el sistema

Capítulo 2

EL LENGUAJE JAVA−−

2.1 Diagrama de clases

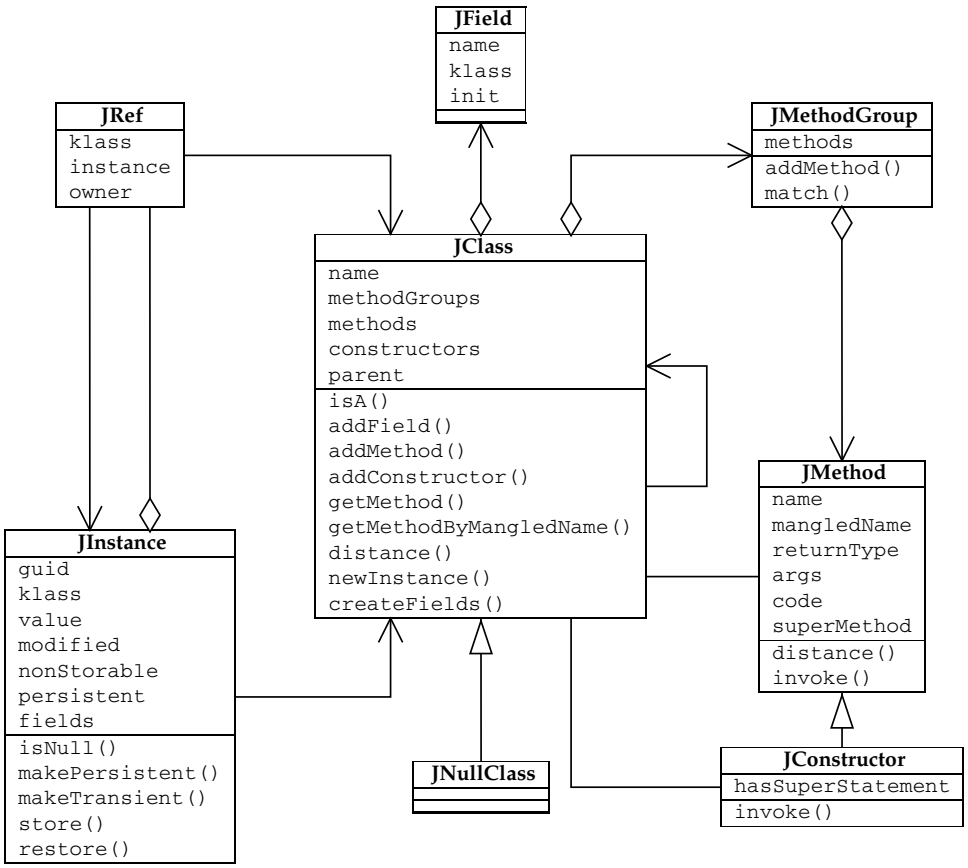


Figura 2.1: Diagrama de clases de la representación interna del lenguaje

La figura 2.1 muestra la sencilla estructura de clases de los elementos que componen el lenguaje Java−−. Las clases poseen métodos (JMethod) y constructores (JConstructor), ambos agrupados con la ayuda de instancias de JMethodGroup, pero manteniendo accesos directos para los casos en que se sabe exáctamente qué método o constructor debe invocarse y no se necesita buscar el más apto dentro del grupo. Las clases también contienen campos, que servirán como modelo para la creación de los atributos al inicializar una nueva instancia.

Las referencias son punteros a las instancias, pudiendo ser variables del usuario o bien

atributos de una instancia, en cuyo caso sabrán también de quién forman parte, para notificarle cuándo son modificadas y que de esta forma una instancia persistente pueda avisar al gestor de persistencia de que ha cambiado de estado y es necesario que se vuelva a almacenar.

Existe una clase especial, JNullClass, pensada para ser la clase de una instancia especial que represente el valor nulo (patrón *NullValue*, ref. [4]). Su único cambio es el comportamiento de su método isA(), que devolverá verdadero sea cual sea el tipo por el que se le pregunte (pues efectivamente un valor nulo puede asignarse a una referencia de cualquier tipo).

2.2 El modelo, el lenguaje Java

El lenguaje a implementar se ha denominado Java−−. Su nombre ya indica claramente que se ha creado tomando como modelo el lenguaje Java, concretamente la versión 1.0 del mismo (ref. [1]), con algunas simplificaciones y modificaciones que se detallarán en los siguientes puntos.

2.3 Se desechan los tipos primitivos

A diferencia de Java, en el lenguaje Java−− todas las variables son referencias a objetos, no existiendo tipos primitivos. Esta decisión fue motivada por el diseño del sistema de persistencia, al requerir que todo objeto almacenado tuviese un GUID asignado (ver sección 4.2). De permitir tipos primitivos se dificultaría la implementación del sistema al tener que considerar en varios puntos del mismo casos especiales, y de acuerdo a las consideraciones expuestas en la sección 1, no sería deseable—especialmente en lo que referente a la facilidad de expansión y modificación, pues es bien sabido que el tener que mantener casos especiales es una clara fuente de errores al modificar un sistema.

2.4 Los valores lógicos

Una consecuencia de la decisión de no tener tipos primitivos (sección 2.3) es que es necesario cambiar el sistema de representación de los valores lógicos en el lenguaje. En Java existe un tipo básico *bool* para representar el resultado de las operaciones lógicas. Sin embargo, Java−− carece de tipos primitivos, por lo que hubo que buscar otra forma de representar esos resultados. La elección fue emplear el mismo sistema que en Lisp: una referencia nula tiene valor lógico *falso*, mientras que una no nula tiene valor lógico *verdadero*.

El problema que aparece ahora es el siguiente: ¿qué tipo devuelven los operadores lógicos? El tipo *bool* en Java tiene una serie de propiedades apetecibles: no se puede ahormar a otro tipo, y con él sólo es posible efectuar operaciones lógicas. Finalmente se optó por crear una clase primitiva Bool (ver capítulo 5), la única del lenguaje con constructor privado (se pretendía que sólo se pudiesen obtener como resultado de una operación lógica) y que sólo contaría con una única instancia llamada *true* (nuevamente a semejanza de Lisp con su valor *T*).

El único punto que queda por aclarar se refiere a los operadores soportados por los valores lógicos, pero esto se explicará en la sección 2.5.

2.5 Operadores internos y externos

En el lenguaje Java−− existen dos tipos de operadores, que hemos llamado *internos* y *externos*. Los primeros reciben su nombre por estar implementados como métodos de las instancias (por lo que podrían ser modificables, aunque actualmente el lenguaje no lo permite—sin embargo la estructura necesaria ya está implementada). Los operadores externos, en cambio,

trabajan sobre las referencias, y por tanto son ajenos a las instancias. La tabla 2.1 muestra qué operadores pertenecen a cada categoría.

	Internos	**Externos**
Aritméticos	+, -, *, /, %, ++, –	
Lógicos	<, <=, >, >=, ==, !=	&&, ||, !, is, ?:

Cuadro 2.1: Clasificación de operadores internos y externos

Como se puede observar en la tabla, todos los operadores aritméticos son internos (como es lógico, puesto que necesitan el valor asociado a la instancia), mientras que los lógicos están repartidos entre aquellos que emplean el valor (y por tanto son internos) y los que se limitan a consultar si sus argumentos son o no nulos, o si dos referencias apuntan a la misma instancia (operador is), por lo que son externos.

En la sección

2.6 Elección del método a invocar

Dado que se permite la sobrecarga de métodos, al encontrarnos con la invocación de un método debemos elegir uno entre todos aquellos que la clase tenga definidos con ese nombre. El escogido será aquel en el que los tipos de los parámetros que deseamos pasarle sean lo más parecidos que sea posible a los que se le indicaron en su definición.

Antes de explicar el algoritmo, detallaremos el concepto de *distancia entre dos clases*: dadas las clases X e Y, la función $\text{distancia}_C\,(X, Y)$ se define de la siguiente manera:

- Si $X \equiv Y$, esto es, son la misma clase, entonces $\text{distancia}_C\,(X, Y) = 0$
- Si X no es descendiente directa o indirectamente de Y, entonces $\text{distancia}_C\,(X, Y) = -1$ (matemáticamente sería más correcto que la función no estuviese definida, pero el darle un valor que no se puede alcanzar de forma normal hará que se pueda operar con ella de una forma más sencilla, sin preocuparse de posibles casos de indefinición).
- En otro caso, $\text{distancia}_C\,(X, Y) = 1 + \text{distancia}_C\,(\text{padre}\,(X), Y)$

Apoyándonos en la función distancia_Cdefiniremos ahora $\text{distancia}_M\,(def, act)$, que nos dará la *distancia entre los parámetros de un método y sus argumentos actuales*. Dado un método con n_{def} parámetros de tipos $tipodef_i$, y una llamada con n_{act} argumentos de tipos $tipoact_i$, y refiriéndonos como def y act a los argumentos y parámetros en su conjunto respectivamente, distancia_Mse define de la siguiente forma:

- Si $n_{def} \neq n_{act}$ (el número de parámetros y el de argumentos no coinciden), entonces $\text{distancia}_M\,(def, act) = -1$ (nuevamente sería un caso de indefinición, pero optamos por el valor -1 para simplificar el tratamiento).
- Si $\text{distancia}_C\,(tipoact_i, tipodef_i) = -1$ para algún valor de $i \in \{1, n_{def}\}$ (algún argumento no es de un tipo adaptable a los parámetros), entonces $\text{distancia}_M\,(def, act) = -1$
- En otro caso, $\text{distancia}_M\,(def, act) = \text{máx}\left\{\text{distancia}_C\,(tipoact_i, tipodef_i)\;\forall i \in \{1, n_{def}\}\right\}$

Una vez definidas estas funciones, podemos explicar qué método se elige al encontrarnos con una invocación. Si tenemos una llamada a un método de nombre M, seguiremos los siguientes pasos para elegir aquel que se ejecutará:

1. Desechamos los métodos cuyo nombre base no sea M.

2. Calculamos $d_m = \text{distancia}_M\,(def, act)$ para todos los métodos que nos queden.
3. Desechamos aquellos métodos en que $d_m = -1$
4. Los métodos aptos serán aquellos en que $d_m = \text{mín}\,\{d_m\}$

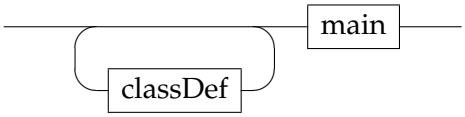
Si no nos quedase ningún método apto no habría candidatos válidos, y si hubiera más de uno que fuese apto estaríamos ante un caso de ambigüedad que el usuario debería resolver con un *cast* (o varios). Si por el contrario sólo nos queda un método apto, ése sera el que se ejecute.

2.7 Diagramas sintácticos

Los siguientes diagramas son una representación de la gramática que se implementará para el lenguaje Java−−. Por supuesto se admiten variaciones en la gramática siempre que el resultado sea equivalente al aquí presentado.

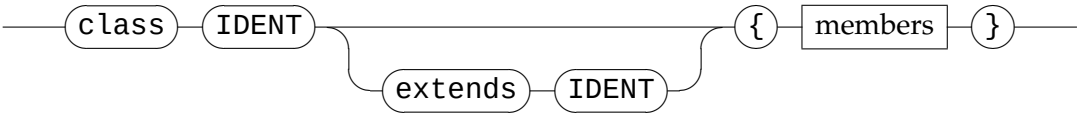
S

bodyStatements

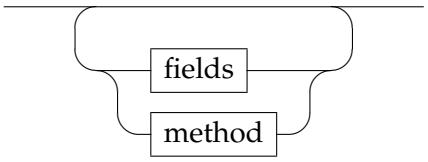


main

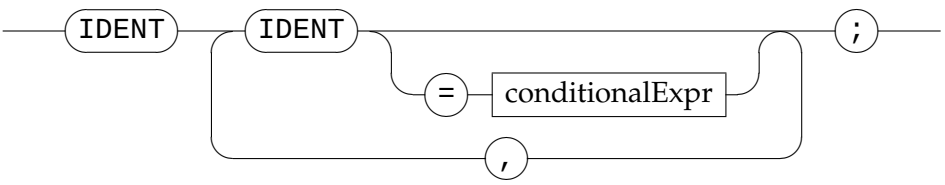
classDef



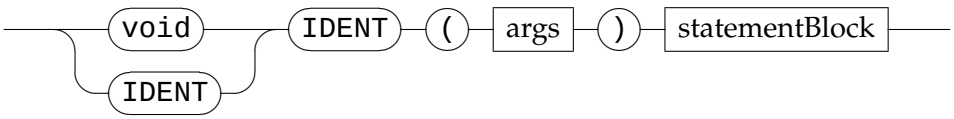
members



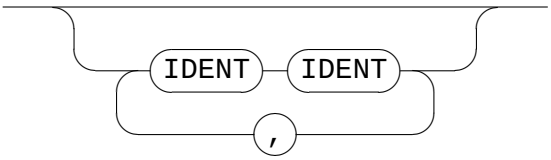
fields



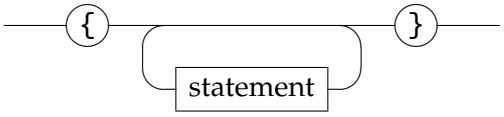
method



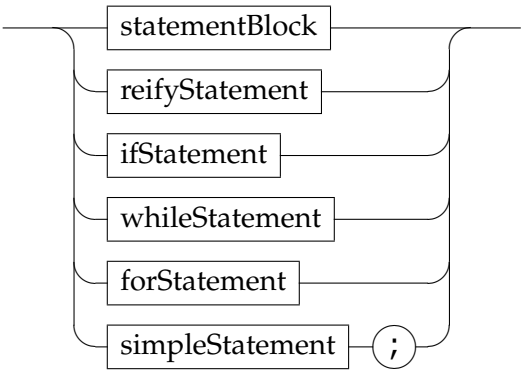
args



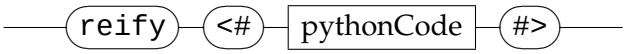
statementBlock



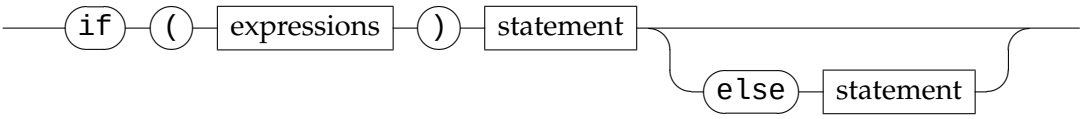
statement



reifyStatement

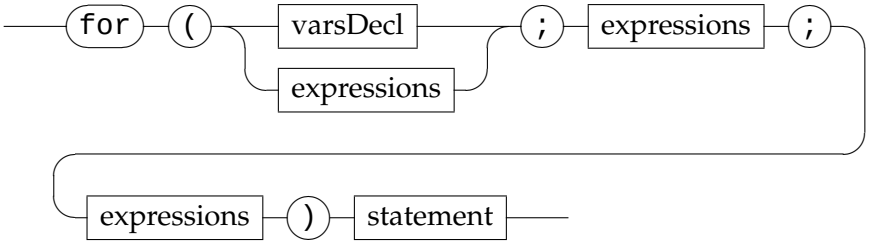


ifStatement

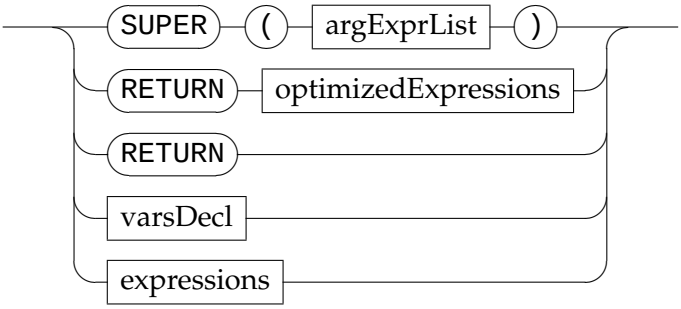


whileStatement

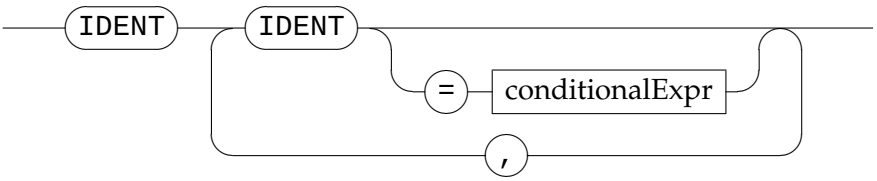
forStatement



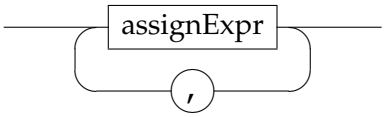
simpleStatement



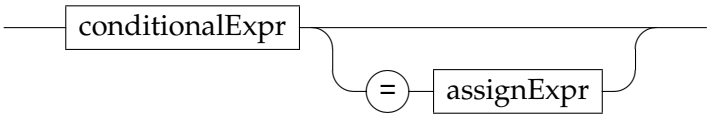
varsDecl



expressions

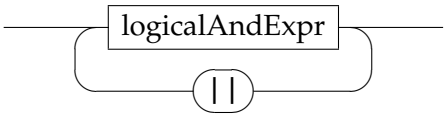


assignExpr

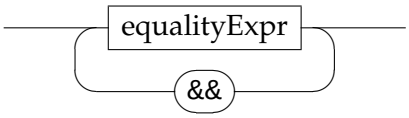


conditionalExpr

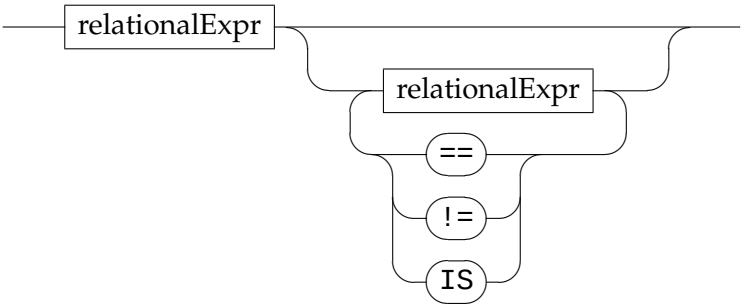
logicalOrExpr



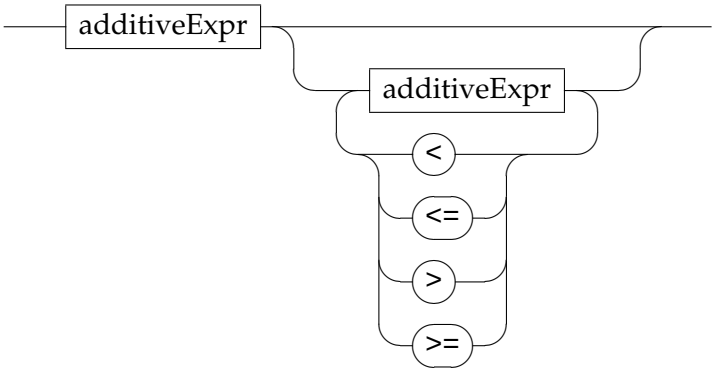
logicalAndExpr



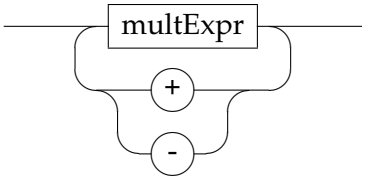
equalityExpr



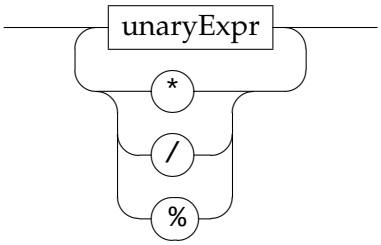
relationalExpr



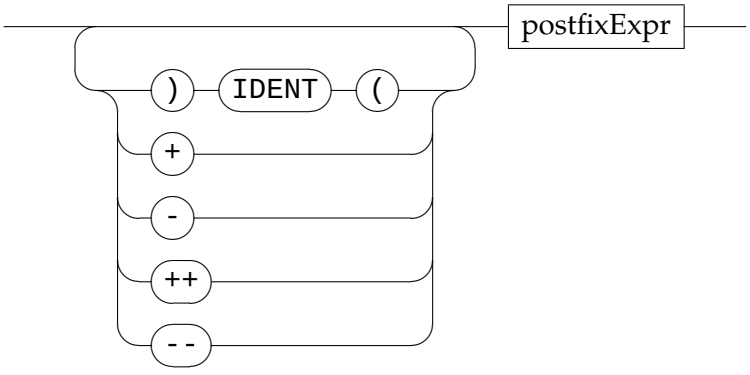
additiveExpr



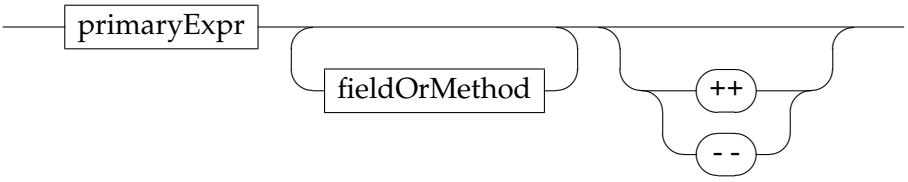
multExpr



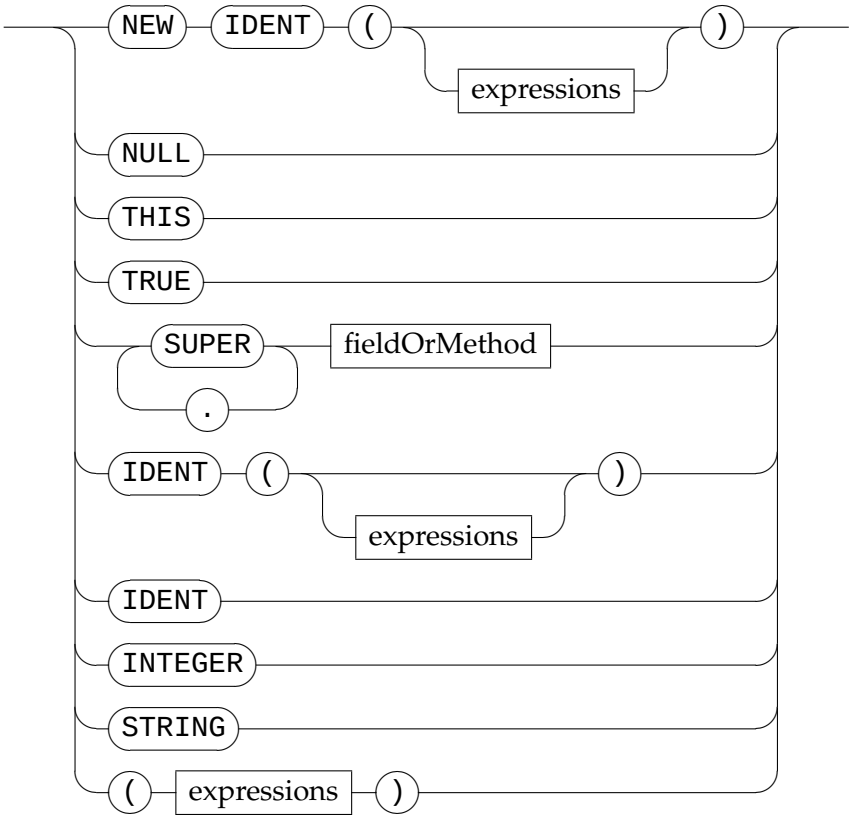
unaryExpr



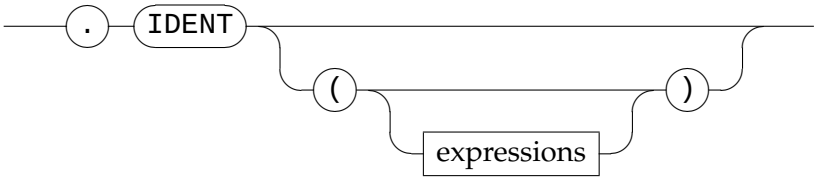
postfixExpr



primaryExpr



fieldOrMethod



Capítulo 3

EL INTÉRPRETE

3.1 Diagrama de clases

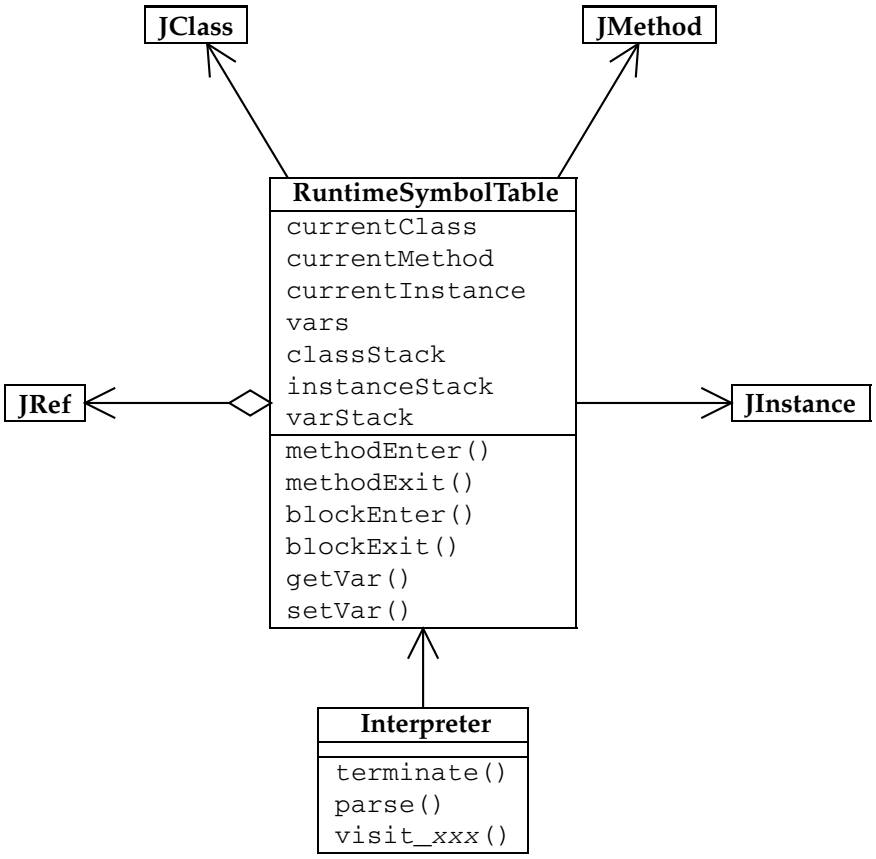


Figura 3.1: Diagrama de clases del intérprete y la tabla de símbolos

El sencillo intérprete desarrollado se muestra en la figura 4.1. La clase Interpreter será un *Singleton* (ref. [3]). Su funcionalidad la dan los métodos visit_xxx(), siendo *xxx* el nombre de la instrucción correspondiente (no se detallan todos por considerar que es un detalle de implementación la decisión de cuál es el mejor conjunto de instrucciones). Estos métodos se apoyan en el método parse(), que decide a cuál de ellos hay que llamar en función del primer elemento de la lista a evaluar que se le pase como parámetro.

La tabla de símbolos es otro *Singleton* que contiene un diccionario de referencias, vars, con las variables accesibles desde el punto de la ejecución en el que se encuentre el intérprete en cada momento. También conoce el método que se está ejecutando, así como la instancia sobre la que se está ejecutando (el argumento this), y emplea un par de pilas para almacenar ambos datos. Además, existe una tercera pila en la que se guarda el diccionario de variables cada vez que se entra a un nuevo bloque de código.

3.2 Modelo para las instrucciones

Si bien no se pretende especificar el juego de instrucciones que aceptará el intérprete, se supondrá que se toma como referencia [5] para su implementación.

3.3 Representación del código

A pesar de que nitrO genera un AST sobre el que podríamos realizar directamente la interpretación del código, este árbol no presenta una estructura adecuada para trabajar cómodamente con él, ya que se genera directamente a partir de la gramática. Por tanto se decidió que el intérprete trabajase sobre otro AST creado a partir del generado por nitrO pero que fuese más simple de manejar. Además, también se optó por el empleo del patrón *Visitor* (ref. [3]) frente al *Command* que nitrO empleaba originalmente, para simplificar los problemas producidos por mantener el estado del intérprete (ver sección 3.4 para más información al respecto).

La elección final fue una representación del código como un AST creado con listas anidadas, indicándose en su primer elemento qué función del intérprete debería llamarse para tratar ese código. De esta forma evitamos el tener que crear una clase por cada tipo de nodo, a la vez que simplificamos su tratamiento, ya que la lista es un concepto que Python maneja de forma nativa. Como ventaja añadida, esta estructura resulta bastante similar a la empleada por Lisp, así como a la que emplean muchos compiladores como representación intermedia del código, facilitando así la aproximación al proyecto a aquella gente que tenga experiencia con alguno de ellos.

3.4 Por qué se empleó el patrón Visitor

En los AST que genera nitrO se emplea el patrón *Command* (ref. [3]) para recorrerlos, sin embargo con este patrón se presentan dificultades para mantener el estado del intérprete. Existen dos alternativas para tal empeño, el empleo de variables globales y la copia de las variables entre los nodos, cada una presentando sus propios problemas.

- El empleo de variables globales, además de todos los problemas que siempre se asocian con su uso (debido al alto grado de acoplamiento que se provoca), obligan al programador a recordar que tiene que usar la sentencia global para marcar todas aquellas variables globales *que se modifiquen*, no siendo necesario con las que sólo se consulten. Por tanto, de emplearlas sería fácil sufrir errores de dificil localización (si se modifica una variable y no se empleó global), y a quien revisara el código posteriormente podría serle difícil comprender por qué en unos casos se emplea global y en otros no.
- Si las variables de estado se copian entre los nodos, además de una pérdida de eficiencia, debido a la naturaleza dinámica de Python se podrían producir errores por olvidarse de copiar cierta variable que luego fuese necesaria más adelante, como consecuencia de cargar con este trabajo al programador.

Para evitar ambos problemas, se optó por emplear el patrón *Visitor* (ref. [3]) en nuestro AST transformado. No necesitamos emplear variables globales (consiguiendo así reducir el acoplamiento y evitando el problema del `global`), y no hace falta duplicar el estado (manteniendo una adecuada eficiencia y librando al programador de tareas adicionales).

Capítulo 4

EL SISTEMA DE PERSISTENCIA

4.1 Diagrama de clases

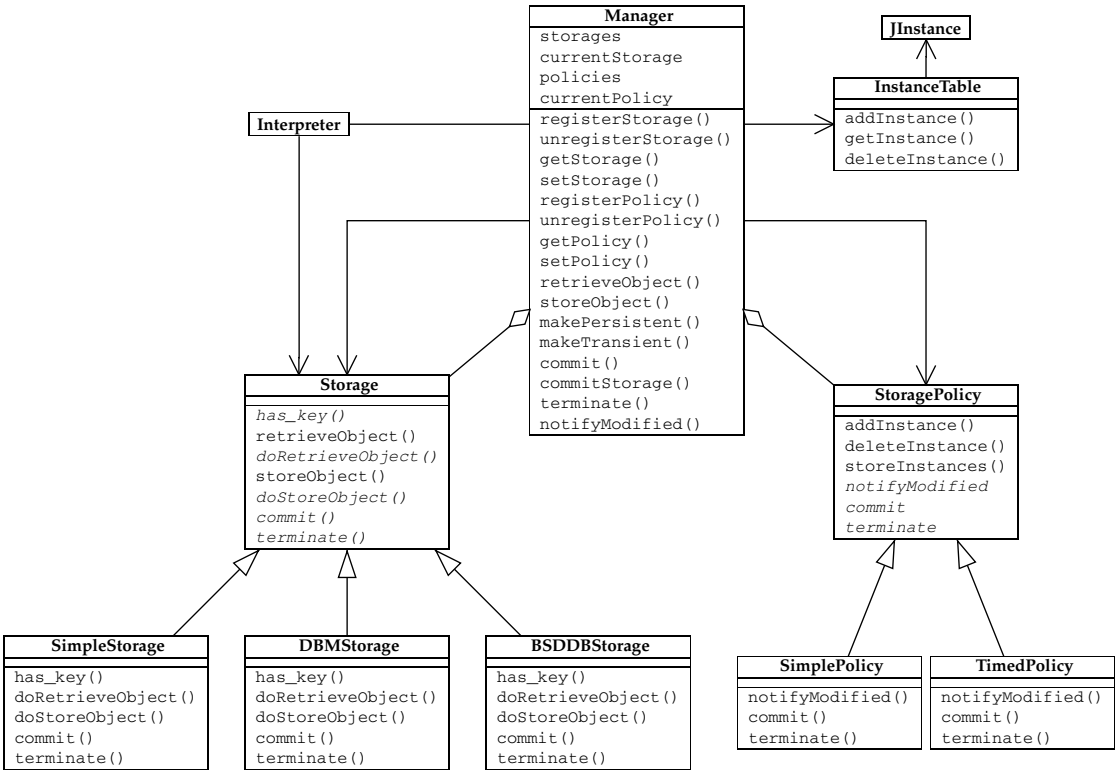


Figura 4.1: Diagrama de clases del sistema de persistencia

En la figura 4.1 se muestra el sistema de persistencia. La clase Manager será la fachada del sistema de persistencia con el resto del lenguaje (patrón *Façade*, ref. [3]), además de ser un *Singleton* al igual que el intérprete. La funcionalidad de persistencia vendrá dada por las instancias de Storage y StoragePolicy que estén registradas.

Inicialmente se crearán tres tipos de almacenamiento:

- **SimpleStorage**: consistirá en un simple diccionario que se cargará y volcará a un fichero de una sola vez, y será la que se use por defecto.

- **DBMStorage**: será un interfaz con el módulo AnyDBM de Python. Este módulo nos permite emplear diversas variantes de las bases de datos DBM, consistentes en un único fichero en disco y que se comporta como un diccionario que mapea cadenas a cadenas. El módulo elegirá la implementación DBM más adecuada según el sistema en que se ejecute.
- **BSDDBStorage**: otro interfaz, en esta ocasión con el módulo BSDDB de Python. Éste es uno de los módulos que pueden empleados por AnyDBM, sin embargo en ese caso sólo se utilizaría uno de los tipos de fichero que soporta—concretamente el tipo *Hash*, sin posibilidad de usar los tipos *B-Tree* o *Record*, que sí son accesibles empleando este almacenamiento. Además de permitir elegir el tipo de fichero, también daremos facilidades para enviar parámetros extra a las funciones de inicialización del fichero, de forma que se pueda ajustar más finamente para el uso previsto.

En cuanto a las políticas, se crearán dos tipos:

- **SimplePolicy**: hará que se ejecute un *commit* cada n cambios, dándosele el valor de n al crearse, y siendo $n = 1$ en la política por defecto.
- **TimedPolicy**: al modificar una instancia se arrancará un temporizador. Desde ese momento se dispondrá de un tiempo t (indicado al crear la política) para efectuar más cambios antes de que se ejecute un *commit*.

Hay que recordar que los almacenamientos y políticas descritos son básicamente ejemplos, no se intenta crear un sistema "perfecto".

4.2 Identificación de las instancias

Toda instancia tendrá un identificador único o GUID. Dicho identificador será una cadena de texto que constará de las siguientes partes, separadas por el carácter '.':

- La dirección IP de la máquina
- El PID del proceso, o el carácter '1' si no está disponible.
- El UID del usuario, o el carácter '1' si no está disponible.
- El TID del hilo activo, o el carácter '1' si no está disponible.
- Los segundos transcurridos desde la medianoche del 1 de Enero de 1970 (si el reloj del sistema lo soporta se devuelve un resultado en coma flotante, en cuyo caso se mostrarán 3 dígitos decimales).
- Un número aleatorio en el rango $[0, 32767]$.

Se ha elegido un formato tan complejo para el identificador con el fin de evitar en la medida de lo posible la colisión, teniendo en cuenta que actualmente se soportan múltiples aplicaciones y almacenamientos. Además, en un futuro podría extenderse el sistema para soportar concurrencia o incluso distribución en múltiples máquinas de una forma transparente, y en ese caso éste sistema para generar el GUID podría seguir utilizándose sin cambios.

4.3 Almacenamiento de instancias

Las instancias que se almacenan se transforman previamente en una cadena de texto ayudándose del módulo *pickle* de Python. Las referencias a otras instancias miembros se deshacen "al vuelo" y se convierten en cadenas que contienen el GUID de la instancia miembro (que también se almacenará)[1]

La carga de instancias se efectúa en dos pasos, debido a limitaciones en el módulo *pickle*. En primer lugar se cargan las cadenas desde el disco y se convierten de nuevo en las instancias correspondientes. A continuación, se rehacen los enlaces entre las instancias.

4.4 Memoria de instancias almacenadas

Se crea una tabla de referencias débiles con todas las instancia que se almacenen o carguen, para emplearse al hacer *unfizzling*. No se almacenan todas las instancias puesto que para el resto no es necesario que figuren en esa tabla. Asimismo, una vez que una instancia se almacene en la tabla se deja ahí mientras la instancia siga en memoria, puesto que se podría cargar desde un almacenamiento otra instancia que la referenciase, y de no "recordarla" no tendríamos forma de acceder a ella más que creando otra copia (que no sería lo deseado).

4.5 Modificación de una instancia persistente

Cuando una instancia marcada como *persistente* se modifica (se llama a su método setValue() o se llama al método setInstance() de alguna de las referencias que componen sus miembros), la instancia avisa al gestor de persistencia llamando a su método notifyModified(), quien a su vez llama al método homónimo de la política de persistencia activa, tal y como se ve en la figura 4.2.

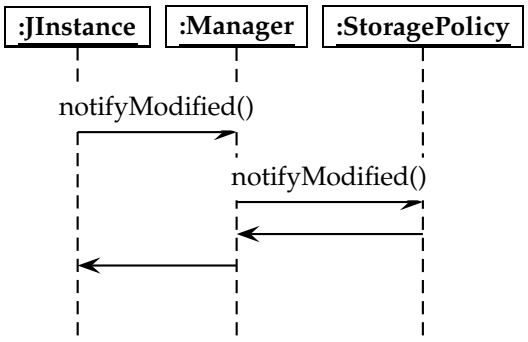


Figura 4.2: Diagrama de secuencia de la modificación de una instancia persistente

Tan pronto como a la política le llegue la notificación marcará la instancia como modificada, y la próxima vez que se actualicen los cambios en el almacenamiento ordenará que esa instancia se guarde.

La figura 4.3 muestra el proceso que se sigue una vez que se entra en el método storeInstances() de la política (pensado para ejecutarse cuando la política decida hacer un *commit* o el usuario ordene hacerlo explícitamente). La política va indicando al gestor todas las instancias que tiene que almacenar. El gestor, por su parte, le indica a la instancia que debe guardarse en el almacenamiento que se tenga activo en esos momentos.

[1] Este proceso se conoce como *fizzling*, y su opuesto, efectuado al cargar una instancia, como *unfizzling*.

El proceso de almacenar una instancia es un recorrido en profundidad del grafo formado por cada instancia y sus campos. La instancia le indica primero a cada uno de sus campos que debe almacenarse, y cuando todos lo han hecho ella misma se guarda en el almacenamiento. Cuando se ha terminado con todas las instancias, el último paso es que la política le pida al gestor que ordene al almacenamiento hacer un *commit*, de forma que los cambios se hagan permanentes.

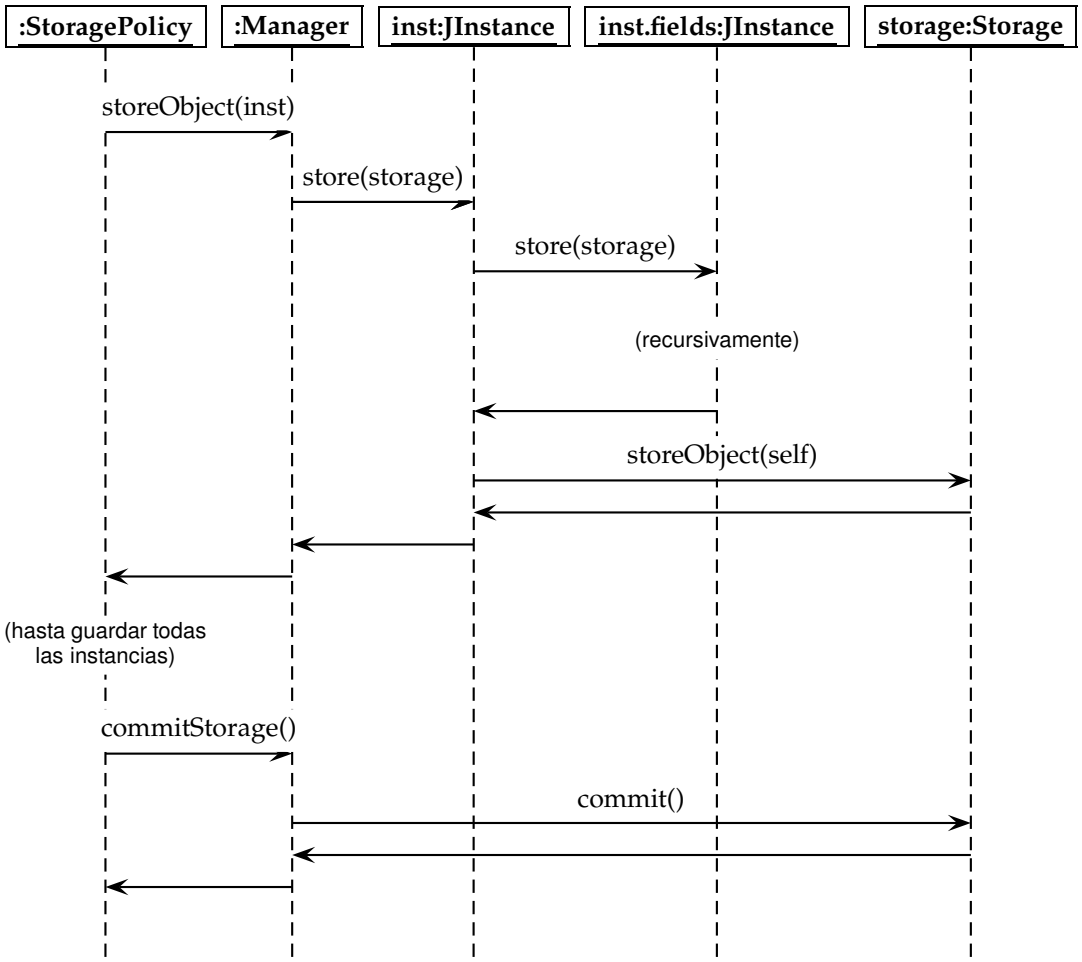


Figura 4.3: Diagrama de secuencia de la ejecución de un *commit* por la política

Capítulo 5

EL API DE JAVA−−

5.1 Diagrama de clases

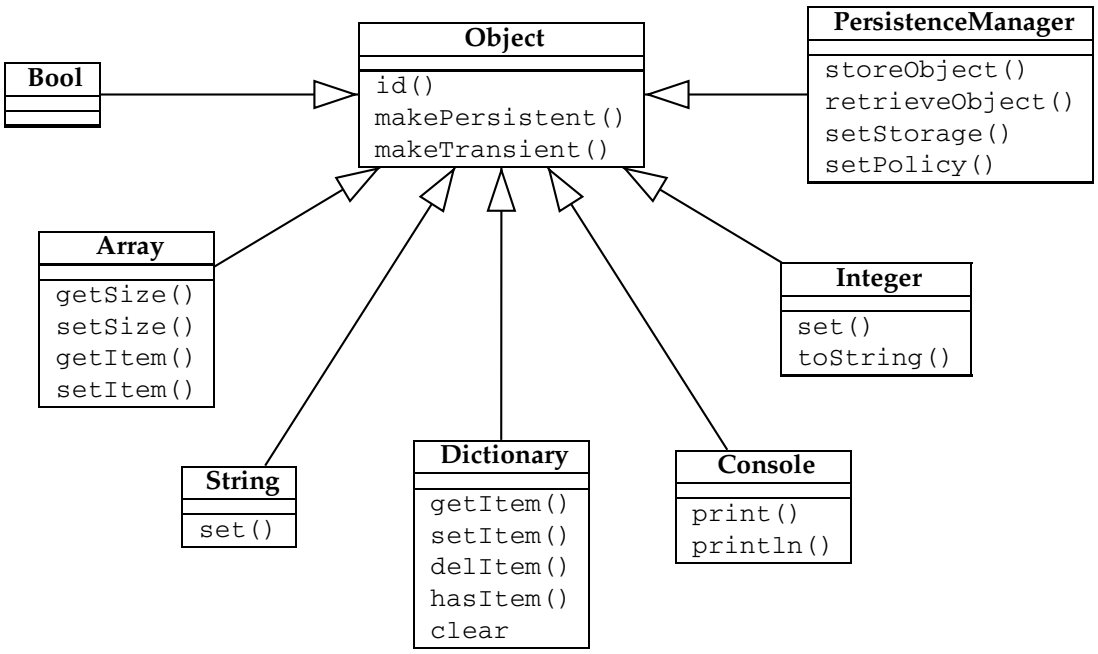


Figura 5.1: Diagrama de clases de la API del lenguaje

Para el lenguaje Java−− se ha decidido que se implemente el API mostrado en la figura 5.1. Existirá una clase base universal, Object, que poseerá los métodos que permitirán a cualquier instancia convertirse en persistente o no persistente. Directamente de Object se derivarán una serie de clases que proporcionarán una funcionalidad básica para los programas.

Las clases Bool, Integer y String se corresponden con los tipos básicos de Java−−. Console permite mostrar texto por pantalla, mientras que Array y Dictionary proporcionan dos contenedores básicos. PersistenceManager, por su parte, es el reflejo del Manager del sistema de persistencia en el lenguaje, y proporciona algunas operaciones para controlar el almacenamiento y recuperación de objetos y para configurar el sistema de persistencia.

BIBLIOGRAFÍA

[1] Ken Arnold y James Gosling. *El lenguaje de programación Java*. Addison-Wesley Iberoamericana y Domo Editores, 1997.

[2] Grady Booch, James Rumbaugh, y Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998. ISBN 0-201-57168-4.

[3] Erich Gamma, Richard Helm, Ralph Johnson, y John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.

[4] Mark Grand. *Patterns In Java: a Catalog Of Reusable Design Patterns*, tomo 1. John Wiley & Sons, septiembre 1998.

[5] Tim Lindholm y Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2ª edición, 1999.