Universidad Carlos III de Madrid

Departamento de Ingeniería Telemática

Proyecto Fin de Carrera

NWTJava v2.0: Ampliación y mejora de un entorno de programación orientado a la docencia.

Autor: José Francisco González Ruiz.
I.T.T.: Sistemas de Telecomunicación.
Tutor: Dr. José Jesús García Rueda.

Leganés, Septiembre de 2009.

PROYECTO FIN DE CARRERA

Departamento de Ingeniería Telemática

Universidad Carlos III de Madrid

Título: NWTJava v2.0: Ampliación y mejora de un entorno de programación orientado a la docencia.

Autor: José Francisco González Ruiz.

Tutor: Dr. José Jesús García Rueda.

EL TRIBUNAL

Presidente: Jesús Arias Fisteus

Secretario: Jaime José García Reinoso

Vocal: Silvia De Castro García

Realizado el acto de defensa y lectura del Proyecto Fin de Carrera el día 18 de Septiembre de 2009 en Leganés, en la Escuela Politécnica Superior de la Universidad Carlos III de Madrid, acuerda otorgarle la CALIFICACIÓN de:

Presidente Secretario Vocal

Resumen:

NWTJava v2.0 es una aplicación informática, interactiva y multimedia creada mediante la ampliación y mejora de su anterior versión. Se trata de una herramienta desarrollada para servir de apoyo al aprendizaje y enseñanza de la programación, concretamente de la programación estructurada. Surge como respuesta a la problemática detectada en los alumnos de primeros cursos de ingenierías e informática, que encuentran considerables dificultades en las primeras etapas del aprendizaje de esta área. A parte del desarrollo software, se han realizado en este proyecto: un experimento de campo (en el marco de una asignatura de programación y cuyo formato fue el de un seminario), un estudio de las nuevas tecnologías aplicadas a la docencia de esta materia (programación estructurada) y una evaluación técnica de la misma.

Agradecimientos

Desde esta página de mi Proyecto Final de Carrera quiero agradecer, de todo corazón, el esfuerzo, apoyo y compañía que me han dado las personas a la que más quiero, y sin las cuales, me hubiese sido imposible llegar hasta aquí.

Estas personas a las que me refiero son, sin duda alguna, mi familia y amigos.

Gracias a mis tíos José y Cecilia, que han convertido su casa, en mi segundo hogar.

A mis tíos Bernardo y Loli, por haberme acogido siempre a mi y a mis amigos.

Y a todos mis otros tíos porque todos los momentos compartidos con ellos han sido siempre felices.

A mi primo Alejandro, que con tan solo 15 años me demuestra cada día ser el más inteligente y mejor informático que he conocido. Y por supuesto, a mi primo Artu, por aguantar juntos esas batallas de ordenador hasta el amanecer.

También quiero darle las gracias a mis amigos Alex y Vari, sin los que no hubiese podido sacar esta carrera adelante y por darme los mejores momentos que he vivido en esta universidad. Y como no, a todos esos amigos que han hecho de cada salida un fiestón: Javi Serrano, Chico, David Baz, la Ali, Fran, Oleguer, Juanele, P. Granero, Cobretti, Iván y Toni, la Familia V2, Xavi, mi mejor amigo del colegio Miguel, mis nuevos compañeros de Joyfe, Estela, Ramón, Mario y Jackie…

Y por supuesto, a mi primo Bernardo, una de las personas que más ha influido en mi vida. Junto a él he vivido las mejores aventuras desde que tuvimos el chalet del Rincón, y los mejores veranos que cualquier persona haya podido pasar.

Con locura quiero también a todos mis otros primos: los Migueles, Ana y Eva, a Varinia y Jorge, y como no, a Rocío y Juanjo, que hacen de todas las bodas, bautizos y comuniones las mejores de la historia.

Mención especial merecen mis abuelas, Leo y Maruja. No puedo describir con palabras lo que siento por ellas.

Y agradecer también, a mi hermano Miguel, y por supuesto y por encima de todo, a mis padres, el haber sido mi más fuerte apoyo en la vida. Sin vosotros no hubiese podido llegar hasta aquí. Os quiero.

Índice General

PLIEGO I: MEMORIA 9
CAPÍTULO 1: INTRODUCCIÓN 11
1.1 INTRODUCCIÓN A NWTJava v2.0 12
1.2 CONTEXTO 13
1.3 MOTIVACIÓN 15
1.4 OBJETIVOS 17
1.5 FASES Y METODOLOGÍA DE TRABAJO 21
1.6 GUÍA DEL DOCUMENTO 23
CAPÍTULO 2: ESTADO DEL ARTE 25
2.1 ORIGEN 26
2.2 BASE PSICOPEDAGÓGICA 27
2.3 EXPERIENCIAS EN LA ENSEÑANZA DE LA PROGRAMACIÓN ESTRUCTURADA 30
2.4 DISEÑO E IMPLEMENTACIÓN DE NWTJava v1.0 34
2.4.1 Diseño 34
2.4.1.1 Interfaz de usuario 35
2.4.1.2 Núcleo de la aplicación 38
4.2.2 Implementación 40
2.4.2.1 Interfaz de usuario 40
2.4.2.2 Interrelación entre bloques 46
2.4.2.3 Núcleo de la aplicación 48
CAPÍTULO 3: EXPERIMENTACIÓN 57
3.1 OBJETIVOS 58
3.2 DESCRIPCIÓN 59
3.3 DESARROLLO 61
3.4 CONCLUSIONES 75
CAPÍTULO 4: DISEÑO E IMPLEMENTACIÓN DE NWTJava v2.0 81
4.1 DISEÑO 82
4.1.1 Requisitos 82
4.1.2 Planteamiento General 83

4.2 IMPLEMENTACIÓN 86
4.2.1 Bloque Otros 87
4.2.2 Bloque Tipos de Variables 98
4.2.3 Bloque Arrays 117
4.2.4 Bloque Subrutinas 132
4.2.5 Bloque Ámbitos de Variables 150
CAPÍTULO 5: CONCLUSIONES Y TRABAJOS FUTUROS 159
5.1 CONCLUSIONES 160
5.2 TRABAJOS FUTUROS 162
APÉNDICE A: DTDs 165
APÉNDICE B: TEST PREVIO 171
APÉNDICE C: TEST FINAL 177
APÉNDICE D: EJERCICIOS DEL MÓDULO 183

PLIEGO II: PLANOS 189

PLIEGO III: MANUAL DE USUARIO 191

PLIEGO IV: PRESUPUESTO 219

REFERENCIAS 223

Pliego I
Memoria

Capítulo 1:

Introducción

1.1 INTRODUCCIÓN A NWTJava v2.0 12
1.2 CONTEXTO 13
1.3 MOTIVACIÓN 15
1.4 OBJETIVOS 17
1.5 FASES Y METODOLOGÍA DE TRABAJO 21
1.6 GUÍA DEL DOCUMENTO 23

1.1 Introducción a NWTJava v2.0

Antes de entrar a abordar ningún otro tema, vamos a presentar, de forma muy sencilla y concreta, la aplicación resultante de todo el proceso. De este modo, siempre dispondremos de una reseña o referencia al respecto.

NWTJava es una herramienta desarrollada para servir de apoyo al aprendizaje y enseñanza de los principios básicos de la algoritmia y la programación. Surge como respuesta a la problemática detectada en los alumnos de primeros cursos de ingenierías e informática, que encuentran múltiples dificultades en las primeras etapas del aprendizaje en estas áreas.

Si nos preguntamos, desde el punto de vista técnico, ¿qué es NWTJava? la contestación es: se trata de un entorno de programación en miniatura, que utiliza un lenguaje visual basado en diagramas de flujo. La construcción de los diagramas es totalmente gráfica y para ello dispone de un conjunto de componentes reducido y universal.

Sin embargo, esta definición, enfocada desde dos puntos de vista, podría abarcar (y de hecho lo hace) tanto a la versión primera de NWTJava, como a la actual. Pues bien, cabe ahora preguntarse, ¿cuál sería la diferencia entre ambas versiones?

Como era de esperar, los cambios no son conceptuales, sino técnicos, y en este caso, además, de profundidad en lo que se refiere a la enseñanza y aprendizaje. Para ello, y a nivel de introducción, diremos que hemos pasado de tener un único ámbito de enseñanza (principios de algoritmia y programación) a profundizar aún más y tener un segundo: programación estructurada.

De esta forma, la primera idea que surge es dividir la enseñanza en fases o niveles de aprendizaje. Ésto supone la primera y una de las más fundamentales diferencias entre NWTJava v1.0 y v2.0.

Y decimos fundamental, pues en esta segunda versión, lo que hacemos es adoptar NWTJava v1.0 como nivel 1 de aprendizaje, crear y desarrollar el nivel 2 y hacer las modificaciones pertinentes para que ambos se integren y complementen en pos de una enseñanza continua y progresiva, y ésto, en esencia, es lo que da lugar a la nueva y mejorada herramienta NWTJava v2.0.

Por lo tanto, y siendo conscientes de todas las limitaciones que se nos presentan al tratarse éste de un apartado introductorio, podemos decir que NWTJava v2.0 responde al mismo concepto de enseñanza que v1.0 pero se estructura de la siguiente forma:

- NIVEL 1:
  Enseñanza de principios básicos de algoritmia y programación (engloba a NWTJava v1.0).

- NIVEL 2:
  Enseñanza de los principios de la programación estructurada.

Este gráfico lo muestra de forma muy intuitiva:

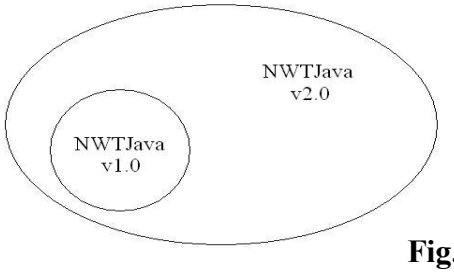


Fig. 1.1

1.2 Contexto

El trabajo cuyo estudio propició el origen de nuestra aplicación que nos ocupa ahora, y que por tanto, estableció unas bases, fue el realizado por el Doctor Raúl Ramírez Velarde[1] y su herramienta NEWT.

NEWT era una herramienta de programación visual desarrollada en el Instituto Tecnológico y de Estudios Superiores de Monterrey (Méjico) como un programa educativo para los alumnos de primeros cursos de ingenierías y cuya primera versión se remonta al año 1991.

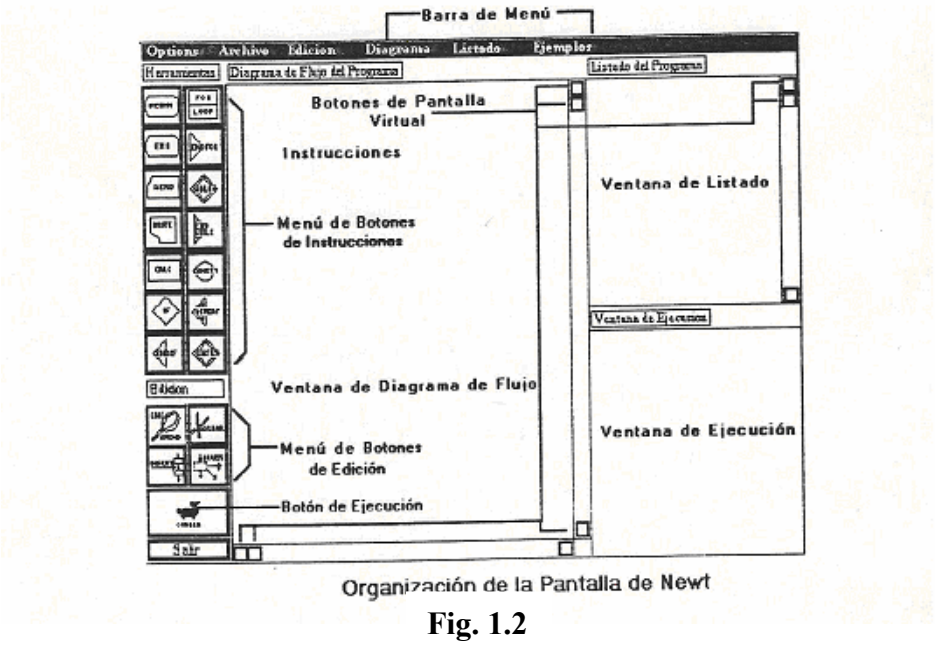


Fig. 1.2

Permitía crear, ejecutar y traducir diagramas de flujo en un entorno gráfico, aunque el usuario no podía operar directamente sobre el diagrama, sino mediante distintos botones de modo. El dibujo en ésta pantalla se hacía de modo automático. Para construir un diagrama, se pulsaba una tras otra las estructuras que se deseaban, y la aplicación iba insertándolas en sus posiciones correspondientes y conectándolas.

---

[1] Doctor Raúl Ramírez Velarde, doctorado en Informática y Sistemas Distribuidos Multimedia, desarrolla actualmente su actividad docente en los centros de Investigación en Informática, y Electrónica y Telecomunicaciones, con sede en Monterrey (Méjico). Entre sus especialidades destacan las de diseño de Interfaces Humano-Computador en 3D, y E-Aprendizaje. Para más información, consultar referencia [1].

El diagrama creado era traducido en tiempo real, a alguno de los lenguajes disponibles (PICO C, PICO PASCAL, y PICO LEEP) y enviado como texto al panel de la derecha. Si se ejecutaba el diagrama, tanto las entradas del usuario como las salidas del código creado se realizaban por medio del panel inferior derecho.

En opinión de quienes[2] pusieron en marcha el proyecto NWTJava (y de los que ahora pondremos nuestro granito de arena) la idea es valiosa incluso en el entorno actual. Es por ello que se apostó por diseñar esta nueva aplicación, basada en la de 1991 pero totalmente actualizada.

Partiendo del manual de usuario de la antigua NEWT, se creó NWTJava (el que a posteriori sería NWTJava v1.0) mediante rediseño tanto la interfaz gráfica, como de toda la arquitectura software.

Dicho rediseño, no atendió únicamente a cuestiones técnicas. También se tuvieron en cuenta otros factores, como la psicopedagogía o el estudio de otras tecnologías en alza y que, al menos a priori, pudieran servir de apoyo y resultar útiles a la hora de conseguir los objetivos marcados.

En cuanto a sus características, desarrollo y funcionamiento, realizaremos un estudio más exhaustivo y profundo en el siguiente capitulo, *Capítulo 2: ESTADO DEL ARTE*, por lo que para concluir este pequeño análisis del contexto, mostramos una imagen de la aplicación en funcionamiento:

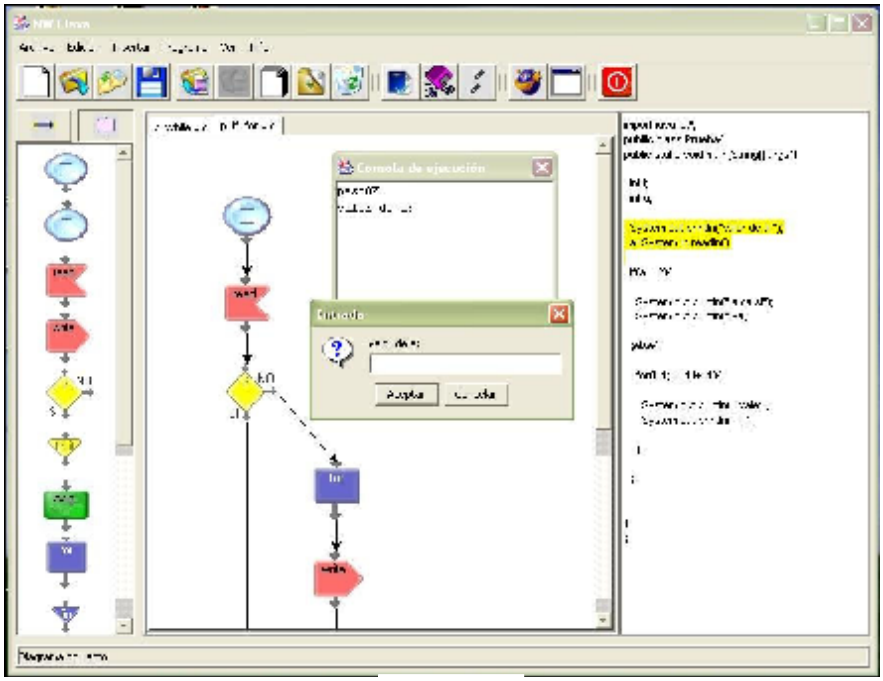


Fig. 1.3

Y es aquí donde recogemos el testigo dejado por el Doctor Velarde y la Srta. Fernández para continuar en la misma línea, y ahondar aún más en la enseñanza práctica de la programación.

[2] Patricia Fernández Garrido como autora, y Dr. José Jesús García Rueda como tutor.

El resultado, como ya sabemos, ha sido NWTJava 2.0, que no sólo absorbe los conocimientos y desarrollos alcanzados en sus proyectos predecesores, sino que profundiza en ellos e innova en otros muchos sentidos, alcanzando así una mayor complejidad, sí, pero desde el punto de vista del software, y que redunda siempre en el beneficio de la consecución de los nuevos objetivos planteados, del cumplimiento de la motivación que nos mueve, y sobretodo y más importante, el de convertir esta aplicación en una útil y accesible herramienta de referencia para nuestros potenciales usuarios.

1.3 Motivación

Existe un momento en la vida de todo estudiante de ingeniería o informática que marca un hito para el resto de su carrera. Nos referimos al instante en el que toma contacto por primera vez con alguna de las asignaturas de programación.

Este primer contacto supone un choque más o menos violento contra un conjunto de conceptos muy novedoso y una forma de pensar que no han cultivado hasta el momento. Por ello, es de suma importancia la primera impresión que se de en dichos alumnos. Su predisposición a la materia, dependerá de las experiencias personales que se den en estos primeros pasos.

Si además hacemos un análisis basándonos en sus resultados académicos, o simplemente reflexionamos acerca de nuestras propias experiencias, comprobamos como, los alumnos en sus primeros cursos, presentan serias deficiencias en el planteamiento de algoritmos, incluidos los más sencillos (concepción a nivel abstracto del problema, planteamiento de la solución, su traducción a código). También en el uso de las estructuras básicas de control de flujo. Y ésto incluso después de haber cursado toda una asignatura cuatrimestral de programación. Sorprendente.

Si nos detenemos un instante y analizamos la actualidad, observamos como la programación orientada a objetos establece un marco en el cual los programadores dedican una gran parte de su tiempo, a componer sus aplicaciones como si de puzzles se tratasen. Integrando, encajando piezas ya prefabricadas.

Una consecuencia directa de este proceso es que los entornos de desarrollo pasan a ser cada vez más complejos. Para cualquier persona experimentada en la programación supondrá en general más ventajas que inconvenientes, al liberarles de una tediosa carga de trabajo. Conmutamos la complejidad de implementación por la de gestión del sistema que la implementa. Y el cambio resulta rentable.

Sin embargo, para alguien no familiarizado con estos modelos, esquemas y paradigmas, puede resultar arto difícil el intuir las fronteras que delimitan la algoritmia, el lenguaje de programación y el entorno en el que está trabajando. De hecho, el entorno en sí mismo es para él algo que tiene que aprender al mismo nivel que todo lo demás, y no sólo un elemento auxiliar. Si tenemos ésto en cuenta, ¿resulta inverosímil el hecho de que el aprendiz no pueda discernir entre qué elementos merece la pena centrar su atención?, o ¿acaso es difícil de entender que la fije en los detalles que tiene más inmediatamente accesibles, y deje de lado los más ocultos y abstractos?

Por si todo lo contemplado hasta el momento fuera poco, se añade el hecho de que el sistema actual de docencia para la programación, se basa en la utilización de lenguajes "reales" para la enseñanza de la misma. El profesor Juan Segovia[3], al igual que el equipo de personas que idearon NWTJava, declara con las siguientes palabras las consecuencias negativas que pueden desentrañar la utilización de estos lenguajes en las etapas iniciales de aprendizaje:

"La utilización de un lenguaje de computación específico como medio de expresión de algoritmos no está exenta de riesgos. La elección de un lenguaje inapropiado puede terminar dificultando el proceso tanto para el alumno como para el profesor: El alumno no solo debe estudiar el contenido mismo de la asignatura, sino que además debe estudiar las peculiaridades del lenguaje en cuestión; y el profesor debe dedicar tiempo para explicar aspectos sintácticos y semánticos del lenguaje, características específicas de una implantación del compilador y hasta detalles del entorno de programación. En resumen: un lenguaje inapropiado se convierte en un distractor y ataca los objetivos educacionales antes que apoyarlos".

Desde luego, el estudiante aprenderá a desenvolverse y a realizar ciertas tareas en el entorno que se le ha propuesto. Sin embargo no resulta complicado comprobar como, en prácticamente la totalidad de su iniciación, la codificación de sus primeros algoritmos se ha basado en el simple hecho de memorizar un conjunto de acciones-resultado. Lejos queda el entendimiento de los procesos llevados a cabo.

Debido a esta serie de cuestiones, cabe preguntarse: ¿en qué merece la pena incidir cuando se enseña a alguien que pretende iniciarse a día de hoy en la programación? Pues bien, la tesis, por la cual surge, y en la que se fundamenta NWTJava, es que en las primeras etapas del aprendizaje, es conveniente hacer hincapié activamente en los conceptos básicos, más que dejar que el alumno los infiera de entre un conjunto amplio de información más o menos compleja.

Para conseguir un aprendizaje de calidad, los alumnos deben empezar por comprender el dinamismo natural de un ordenador. Con esta premisa, deben por tanto comenzar por aprender las estructuras básicas de programación, comunes a todos los lenguajes, y en su forma abstracta (bucles, condicionales, etc.).

En este sentido, la aplicación de las técnicas clásicas de enseñanza, basadas en pseudocódigos y diagramas de flujo, resulta de gran utilidad, ya que se centran precisamente en estos mismos conceptos básicos.

Es por ésto que se consideró la idea de crear un mecanismo, algo que abarcase y pretendiese solucionar todo lo contemplado hasta el momento. Que resolviese en gran medida los problemas que hemos comentado, y las dificultades por las que atraviesa un estudiante neófito en este campo. Surgió así una idea, un concepto. Y tras unos estudios preliminares, entre el que se engloba también todo lo comentado hasta este preciso instante, nació NWTJava, una herramienta que sería muy útil y beneficiosa para el aprendizaje de la programación.

[3] Juan Segovia Silvero, profesor del Centro Nacional de Computación de la Universidad Nacional de Asunción (Paraguay).

Ahora bien, si una vez desarrollada dicha herramienta, si todas sus consignas y fundamentos han demostrado ser válidos para la enseñanza y aprendizaje de los principios básicos de la algoritmia y la programación, cabe preguntarnos: ¿y no sería igualmente lícito? O mejor dicho, ¿y no podría ser igualmente efectivo para transmitir conceptos de un paso más allá?, ¿de una mayor implicación y profundidad?, ¿de la programación estructurada? Es decir, y si pidiéramos ir todavía más lejos, no anclarnos en lo más básico y fundamental. Si este método ha resultado ser bueno, positivo, útil para inculcar estos principios, ¿por qué no aplicarlo para otros más avanzados?

Hasta aquí la respuesta es clara, y ambiciosa. Pero podríamos también plantearnos si sería conveniente, rentable, o incluso necesario, ampliar el número de conceptos a abarcar. La respuesta surge casi de forma inmediata: ¿es sencillo para el alumno inexperto la compresión y manejo de subrutinas ("métodos" en Java)? En teoría si, en la práctica sabemos, y prácticamente vemos a diario, que es todo lo contrario. Otro ejemplo. Éste además atañe directamente a NWTJava. En los seminarios impartidos con esta herramienta (hasta el momento han sido dos), la experiencia nos demuestra que no es necesaria prácticamente ninguna experiencia en el estudiante para que rápidamente solicite el manejo de arrays (al cabo de dos sesiones, tres máximo). Apenas concibe su uso, y mucho menos su concepto, pero lo demanda para la resolución de sus ejercicios. Pues bien, en la primera versión dicha estructura ni siquiera se contempla.

Es por éste motivo, y por todos los vistos a lo largo de este apartado, que consideramos muy útil, y necesaria, la ampliación de NWTJava. Surge así la motivación para embarcarse en el diseño y construcción de una actualización. Nace NWTJava v2.0.

1.4 Objetivos

Como ya sabemos, nuestro proyecto responde al título de:

"Ampliación y mejora de un entorno de programación orientado a la docencia".

Simplemente examinándolo mediante un escueto y superficial análisis, seríamos capaces de extraer una lista de objetivos que pretende alcanzar. Sin embargo, sólo uno responde a la finalidad última del proyecto, el objetivo general.

Objetivo General

- Instruir, a los potenciales usuarios, sobre los principios básicos de la programación mediante un mecanismo más eficiente y menos traumático que lo que demuestran ser los actuales.

Más adelante tendremos ocasión de evaluar los mecanismos actuales de enseñanza de la programación, y poder así fundamentar, nuestro razonamiento referente a que la utilización de éstos suele desentrañar serias dificultades al alumno inexperto[4].

[4] Ver *Capítulo 2*: ESTADO DEL ARTE.

No obstante, si es fundamental comentar en este instante, algunos aspectos a los que hacemos referencia.

Con la expresión "*potenciales usuarios*" hacemos referencia a que podría ser útil para cualquier persona que tuviera el deseo o la necesidad de adentrarse en el mundo de la programación. Sin embargo, la herramienta ha sido diseñada para que alcance su máximo potencial sobre alumnos de nuevo ingreso en ingenierías e informática, y a ser posible, bajo la tutela y supervisión de un profesor especializado[5].

Comentar también el hecho de, que por obvio que resulte, nos vemos en la obligación de recalcar. Para aplicar el mecanismo que pretendemos, la fórmula utilizada ha sido la de emplear una herramienta cuyo formato es el de aplicación informática.

Y por último añadir, que el objetivo de instruir sobre los principios básicos de la algoritmia y la programación corresponden principalmente a la versión anterior, NWTJava 1.0. En este sentido, lo que pretendemos en esta actualización presenta una doble vertiente: reforzar el aprendizaje de los ya mencionados principios fundamentales de la algoritmia y programación, e incidir directamente en el nuevo ámbito del que ya hemos hecho mención en anteriores apartados, los principios esenciales de la programación estructurada.

Por tanto, al hilo de la definición del objetivo, y de las correspondientes matizaciones, podemos incluso reescribir dicho objetivo desde una óptica más tangible y cercana a lo que es nuestro material de trabajo:

- Ampliación de la herramienta NWTJava, capaz de fomentar entre los alumnos, el manejo, la reflexión y la comprensión de los conceptos y estructuras básicas de: la algoritmia, la programación básica, y ahora también, de la programación estructurada. Y todo ello, mediante la recuperación y actualización de las técnicas clásicas.

Sin más dilación, pasamos a plantear los objetivos específicos.

Objetivos Específicos

Estos objetivos son las partes en que se divide al Objetivo General para resolverlo o darle cumplimiento, es decir, son las partes básicas del proyecto.

Teniendo ésto en cuenta, y volviendo a considerar el título y la lista de objetivos que derivan de su simple observación, apreciamos, que destacan por su importancia en el mismo, las siguientes palabras y expresiones: *experimentación, mejora, herramienta de programación gráfica* y *docencia*.

Pues bien, con el apoyo de estas premisas, nos disponemos a plantear dichos objetivos:

[5] Una vez concluido este inciso, emplearemos los términos usuario y alumno indistintamente.

- Análisis global de la herramienta NWTJava v1.0
  - Análisis teórico.
  - Análisis práctico o *Experimentación*.
- *Mejora* y potenciación de la misma ➔ implementación de NWTJava v2.0
  - A nivel de interfaz gráfica.
  - A nivel de núcleo.

  } *Herramienta de programación gráfica*

  - A nivel didáctico (*docencia*).

Desglosamos esta clasificación de objetivos específicos en una más detallada y explicativa:

- Análisis global de la herramienta NWTJava v1.0
  - Análisis teórico.

    Evaluación de dicha herramienta mediante el estudio de:
    memoria del proyecto correspondiente[6]; manual de usuario[7]; exploración del software (código fuente, diagramas UML); reuniones con el tutor; manejo de la aplicación como usuario.

    Todo ello con el fin de extraer conclusiones sobre aspectos a eliminar, a rediseñar o a instaurar. Este punto lo veremos desarrollado en su totalidad en *Capítulo 2: ESTADO DEL ARTE*.
  - Análisis práctico o *Experimentación*.

    Evaluación técnica basada en un estudio de campo. Experimento académico, en forma de seminario, en el que evaluamos los resultados del uso de la aplicación por parte de un grupo de estudiantes voluntarios.

    Encontraremos todo lo referente a este punto en el *Capítulo 3: EXPERIMENTACIÓN.*

Una vez obtenida la información necesaria:

- *Mejora* y potenciación de la misma ➔ implementación de NWTJava v2.0
  - A nivel de interfaz gráfica.
    - Rediseño de alguno de los componentes con el fin de mejorar su calidad (iconos).

[6] Proyecto Fin de Carrera: NWTJava, un micro-entorno de desarrollo para el aprendizaje de la programación. Autora: Patricia Fernández Garrido. Tutor: José Jesús García Rueda.

[7] Lo podemos encontrar en el Pliego III de la correspondiente memoria.

- Aportación de nuevos elementos cuya función es la de proporcionar mayor información al usuario (etiquetas, nuevos paneles).

- Redimensionado de algunos componentes para comodidad del usuario (tamaño de ventanas, paneles).

o A nivel de núcleo.

- Adaptación de las antiguas estructuras a los nuevos requerimientos.
  Ej: actualización de librerías; adaptación del tipo de ámbito de las variables visto por el usuario en sus programas (de tipo global pasamos al nuevo y requerido tipo: ámbito de variables local).

- Permitir la creación y manejo, por parte del usuario, de nuevas estructuras.
  Ej: arrays; subrutinas.

o A nivel didáctico *(docencia).*

- Mayor congruencia entre las funciones del programa y lo que pretenden enseñar.
  Ej: el alumno podía asignar valores de tipos diferentes a una variable, pero siempre se inicializaba como entera (int) [8].

- Añadir el aprendizaje de los principios básicos de la programación estructurada.
  Ej: manejo de subrutinas.

- Creación de mensajes de error y reescritura de los ya existentes, con el propósito de reforzar los principios psicopedagógicos por los que aboga este proyecto. Con esta intención, procuramos que los mensajes sean sencillos, explicativos, propicien la búsqueda de razonamientos por parte del alumno...

Todo lo relacionado con el objetivo de mejora y potenciación, es desarrollado en el *Capítulo 4: DISEÑO Y CONSTRUCCIÓN DE NWTJAVA v2.0.*

Una vez indicados los objetivos, tanto el general como los específicos, haber expuesto una breve introducción, y haber comentado el contexto y la motivación, pasamos a continuación, a describir las fases y metodología seguidas para acometer el proyecto, finalizando posteriormente el capitulo 1 con una guía del documento.

[8] Ésta y otras incongruencias similares han sido solucionadas en la versión actual.

1.5 Fases y Metodología de Trabajo

La distribución de las fases es prácticamente análoga a la clasificación de los objetivos de índole específica. Este hecho no debe sorprendernos, es más, supone un motivo de satisfacción, pues como es lógico, el desarrollo de una aplicación, así como el de una investigación no es otra cosa que la de dar respuesta a los objetivos planteados como específicos.

Por este motivo, la enumeración y descripción de las fases seguidas en el desarrollo de nuestra aplicación será breve y concisa. Como ya sabemos, el planteamiento total o definitivo de cada una de ellas se encuentra extensamente expuesto en capítulos posteriores.

El presente proyecto se ha estructurado en:

Fase 1

Consistió en una profunda evaluación de la aplicación original (NWTJava v1.0). Dicha evaluación fue acometida en diversas etapas o subfases. La primera consistió en, con el software operativo, ir resolviendo ejercicios atendiendo en todo momento a su correspondiente manual de usuario. El fin era el de adquirir un alto nivel de manejo con el entorno gráfico. Una vez superada esta etapa, el planteamiento fue el mismo, solo que en lugar de acudir al manual, recurrimos a la memoria de su autora, para así entender la serie de procesos que tenían lugar a nivel interno. De esta forma, entendimos, por ejemplo, que el programa se dividía en los grandes bloques de interfaz gráfica y núcleo de la aplicación, y que éste último, a su vez, se subdividía en módulos de gestión, ejecución y traducción. Para finalizar, esta evaluación, nos adentramos en su código fuente. Se analizó su estructura de clases (los diagramas UML fueron muy útiles) y se realizaron numerosas pruebas para, entre otras razones, determinar el orden en que se ejecutaban los proceso. Todas estas etapas fueron siempre complementadas, con una frecuente comunicación con el tutor vía e-mail o mediante reuniones personales.

Si bien inicialmente el estudio tenía que empezar y terminar con esa exploración, tanto tutor como autor[9], calificamos como muy positiva, la idea de realizar un estudio de campo. El experimento consistió en impartir un seminario sobre algoritmia en el contexto de una asignatura de enseñanza de la programación. El planteamiento inicial fue el de solicitar un número mínimo de 10 alumnos voluntarios, que cursaran el primer año de cualquiera de las tres especialidades técnicas de Ingeniería de Telecomunicaciones. Sin embargo, con la sola puesta en conocimiento del seminario en la especialidad de Imagen y Sonido, el número de solicitudes fue de 58. Un número del todo inviable, tanto por razones técnicas (espacio físico de un laboratorio) como educativas (la atención sobre el alumnado sería de mayor calidad cuanto más reducido fuese el volumen de éste). Contaremos como se resolvieron éste y otros problemas, además de su desarrollo y conclusiones en un capítulo posterior. Con su puesta en práctica, lo que quisimos fue alcanzar los siguientes objetivos:

[9] José Francisco González Ruiz.

- Comprobar que la comprensión de la disciplina de la programación mejora en calidad mediante NWTJava (objetivo secundario, pues su eficacia ha quedado patente desde las primeras experiencias de Raúl V. Ramírez Velarde).

- Que un grupo de usuarios pusiesen a prueba la herramienta, con la intención de que examinasen su operatividad, detectasen posibles errores y aportaran información sobre aquellos elementos o conceptos que echasen en falta (objetivo principal, pues nuestro fin es el de mejorar y potenciar la herramienta).

Pues bien, a nuestro entender, la inclusión de esta subfase fue un rotundo éxito, pues no sólo alcanzó los objetivos propuestos, sino que superó con creces las expectativas creadas. Como ya hemos mencionado anteriormente, a esta fase, denominada Experimentación, le dedicamos un capítulo de la memoria, el *Capítulo 3: EXPERIMENTACIÓN.*

Fase 2

Una vez recopilada y analizada toda la información aportada por la primera fase, avanzamos a la segunda. En ésta tuvo lugar toda la implementación de la nueva versión.

La primera subfase que encontramos, es una que bien podría haber constituido una completa e independiente por sí misma. Hubiese supuesto una fase intermedia entre la fase 1 y 2, y el motivo de ello es que se presta a un entorno próximo al de la primera, pero que presenta el rasgo fundamental y distintivo de la segunda. Dicho rasgo o característica principal es la implementación de nuevo código. Es por este hecho que decidimos incluirla en la Fase 2.

Esta etapa de la que hablamos, consistió en la adaptación de NWTJava v1.0 a las nuevas exigencias y requisitos de la versión 2.0. Para ello hubo que modificar el hecho, e incluso el concepto, de que v1.0 pasase de ser un "todo" a convertirse únicamente en uno de los niveles de aprendizaje de la herramienta definitiva. En esta misma etapa, también tuvieron lugar otros cambios, cuyo fin seguía siendo el de minimizar los efectos de los profundos cambios que iba a sufrir la aplicación. En definitiva, lo que pretendíamos con esta subfase, era realizar una transición entre versiones lo más gradual y progresiva posible.

De esta forma, conseguíamos que el uso de la aplicación, y por tanto, el aprendizaje, quedase dividido en niveles. Así, NWTJava v2.0 quedaba estructurado en:

- NIVEL 1:
  Aprendizaje de principios básicos de la algoritmia y la programación.

- NIVEL 2:
  Aprendizaje de los conceptos básicos de la programación estructurada.

Y es el diseño y desarrollo del nivel 2, toda su implementación, lo que era abarcado por la siguiente subfase.

Ni que decir tiene que, estas fases y subfases, ocultan (al menos por el momento) un sin fin de otras etapas, más subfases, problemas y dificultades, disquisiciones y razonamientos, que irán progresivamente viendo la luz a medida que nos adentremos en el análisis de cada una de ellas.

Por ejemplo, y de esta forma concluimos la descripción a nivel de introducción del nuevo software, la subfase diseño y desarrollo del nivel 2, decidimos subdividirla en cuatro grandes bloques,

- Tipos de variables.
- Manejo de arrays.
- Subrutinas.
- Ámbito de variables.

correspondientes cada uno de ellos con los nuevos conceptos en que NWTJava v2.0 debía hacer hincapié desde el punto de vista docente.

A estos cuatro bloques podríamos añadir un quinto, que a modo de tejido conjuntivo, se encarga de la interrelación entre bloques, además de añadir un plus en la operatividad y funcionalidad de la herramienta.

Finalizando ya el apartado que nos ocupa, Fases y Metodología de Trabajo, comentar que para la consecución de todos y cada uno de los objetivos planteados como específicos, y así, haber podido alcanzar la consecución del objetivo general propuesto, además de, para el desarrollo de las fases del proyecto, la filosofía o estrategia seguida, ha sido siempre la misma. Responde a la denominación de "Divide y vencerás" [10].

1.6 Guía del Documento

Para terminar, vamos a realizar un pequeño recorrido por los capítulos que restan en este documento:

- Capítulo 2: Estado del Arte. Se describen los trabajos y tecnologías en que se basa NWTJava v2.0 (especial relevancia tiene NWTJava v1.0), así como las diferentes teorías psicopedagógicas en que se fundamenta la función docente de NWTJava.

- Capítulo 3: Experimentación. En este capítulo se describe la motivación y objetivos con los que se parten en el experimento realizado, además de su desarrollo y las conclusiones obtenidas.

[10] Implica resolver un problema difícil, dividiéndolo en partes más simples tantas veces como sea necesario, hasta que la resolución de las partes se torna obvia. La solución del problema principal se construye con las soluciones encontradas. En programación, es uno de los más importantes paradigmas de diseño algorítmico. Para más información, ver [2].

- Capítulo 4: Diseño e Implementación de NWTJava v2.0. Describe todo el proceso de diseño y desarrollo de la aplicación desde un punto de vista fundamentalmente técnico.

- Capítulo 5: Conclusiones y Trabajos Futuros. Incluye las conclusiones obtenidas a partir del trabajo realizado y las líneas en las que se puede continuar este proyecto.

- Incluimos también una serie de elementos fundamentales en la composición de un documento de esta tipología: Referencias, Apéndices, Planos, Manual de Usuario y Presupuesto.

Capítulo 2:

Estado del Arte

2.1 ORIGEN 26
2.2 BASE PSICOPEDAGÓGICA 27
2.3 EXPERIENCIAS EN LA ENSEÑANZA DE LA PROGRAMACIÓN ESTRUCTURADA 30
2.4 DISEÑO E IMPLEMENTACIÓN DE NWTJava v1.0 34
2.4.1 Diseño 34
2.4.1.1 Interfaz de usuario 35
2.4.1.2 Núcleo de la aplicación 38
4.2.2 Implementación 40
2.4.2.1 Interfaz de usuario 40
2.4.2.2 Interrelación entre bloques 46
2.4.2.3 Núcleo de la aplicación 48

En este tema, como hemos comentado ya en más de una ocasión, realizaremos un análisis de las tecnologías y teorías psicopedagógicas, trabajos y proyectos, que constituyen la base de NWTJava, y más aún, de nuestra ampliación.

2.1 Origen

Tradicionalmente, la iniciación a la programación se realizaba mediante diagramas de flujo y pseudocódigos, haciendo especial hincapié en la algoritmia. Sólo después de varios meses de práctica con estas técnicas se introducía al alumno en los detalles de un lenguaje de programación real. La tendencia actual, sin embargo, es la de enseñar a programar haciendo uso directo de un lenguaje de programación concreto.

Como ya dijimos en el capítulo anterior, con NWTJava se apostó por la continuidad de estos métodos clásicos. Por múltiples razones. Además, se dotó de una aproximación más actual a las mismas, y se suplieron ciertas carencias con la ayuda de las nuevas tecnologías disponibles.

Si retrocedemos unos años, vemos como también otros dieron su propia visión al respecto, implementando sus aproximaciones de modos muy diversos. Sin embargo fue un trabajo el que se destacó sobre todos los demás a la hora de asentar una sólida base para NWTJava. Fue el ya mencionado en el apartado *1.2 Contexto*, NEWT, de Raúl Ramírez Velarde.

Dicho proyecto surgió con la idea de impartir un curso introductorio de programación, en el que también se incluiría el funcionamiento de las computadoras y sus componentes. Esto provocó que dicho trabajo derivara en dos proyectos diferentes, con distintos enfoques y tecnología.

El primero, orientado al hardware, en el que se diseñaron una serie de esquemas y gráficas que explicaban el funcionamiento de cada uno de los componentes (CPU, teclado, discos, monitor, etc.). El modelo seguido se basaba en el dictado por la publicación *How Computers Work*[11]. La diferencia principal con el enfoque del citado libro era que, cada esquema, cada gráfica, iba acompañada de una explicación exhaustiva del funcionamiento del dispositivo, y siempre con el uso de una terminología de nivel introductorio, pues la audiencia presente contaría con unos conocimientos prácticamente nulos de informática.

En cuanto al centrado en la programación, se diseñó un software interactivo que permitía la creación de programas por medio de la edición de diagramas de flujo. La ventaja de este enfoque, era que la utilización de una interfaz gráfica intuitiva de diseño de programas, permitía aligerar la carga cognitiva[12] al estudiante. Debido a que NEWT no diseñaba el programa del usuario, sino que sólo se encargaba de la sintaxis y de la edición del programa, no era en realidad más que un laboratorio virtual donde el estudiante podía experimentar. Dicha experimentación podía ser guiada por la propia curiosidad del estudiante o, más académicamente, por una serie de prácticas que tocaban sucesivamente diferentes conceptos.

---

[11] Para más información acerca de esta publicación, ver [3].

[12] Ésto es, que el estudiante se ve libre de entender el funcionamiento del medio en el que desarrolla la programación (sistema operativo, editor, depurador, compilador) pudiendo así concentrarse en el diseño de algoritmos.

Este software fue presentado en la X Reunión de Intercambio de Experiencias de Estudios sobre Educación en Agosto de 1992, y como se puede intuir, es de los dos proyectos en que derivó la idea inicial, el que más información y conocimientos aportó para asentar las bases de NWTJava.

Y son el estudio de esta herramienta y el concienzudo análisis sobre el ámbito de la docencia las estructuras que ejercen la función de base, los pilares fundamentales sobre los que se asientan nuestros conceptos, y por tanto la herramienta, NWTJava.

Es por ello que a continuación resumimos, de forma breve y concisa, el estudio que tuvo lugar sobre el mencionado ámbito de la docencia.

Esta exposición la dividimos en dos grandes bloques. El primero, donde plasmamos, reflexionamos y justificamos las teorías del conocimiento sobre las que se basa la estrategia docente.

Y el segundo, en el que investigamos las tecnologías cuyas características se han considerado significativas a la vez que muy útiles y beneficiosas para nuestro objetivo de inculcar los principios fundamentales de la programación estructurada.

2.2 Base psicopedagógica

Para la planificación y desarrollo de NWTJava, cuyo objetivo principal, como ya sabemos, es la docencia, fue fundamental realizar un estudio sobre las diferentes teorías sobre el conocimiento, la inteligencia y la docencia.

Para ello, los primeros autores de NWTJava efectuaron un recorrido histórico y un concienzudo análisis sobre teorías y ramas del conocimiento como puedan ser la epistemología[13], la Ciencia Cognitiva[14] y otras muchas. Como no, también sobre sus influencias en la enseñanza y el aprendizaje.

Sin embargo, una explicación histórica de la evolución de las ciencias cognitivas y sus aplicaciones no parece aportar demasiado a nuestro proyecto. Ello, sumado al hecho de que podríamos cometer el error de desviar demasiado la atención de los puntos verdaderamente importantes, nos hace referenciar al lector, al apartado de mismo nombre contenido en el anterior proyecto[15].

Teniendo en cuenta las características allí recogidas, y volviendo al ámbito de este proyecto, podemos no obstante, indicar cuales son los puntos que guían nuestra hoja de ruta en pos del mejor y más óptimo aprendizaje posible.

---

[13] Rama de la filosofía cuyo estudio se basa en el conocimiento científico. Para mayor información, acudir a la referencia [4].

[14] Entendemos "cognitivismo" como adquisición de conocimiento.

[15] Memoria de Patricia Fernández. Ver [5].

Comentamos pues las consideraciones de tipo pedagógico que hemos aplicado específicamente en nuestra implementación:

1. Focaliza y hace hincapié en los conceptos básicos más universales de la programación estructurada y en la lógica propia de esta disciplina.

   Este primer punto es decisivo. Hemos comentado ya los problemas que aparecen en la enseñanza de la programación a día de hoy: el alumno tiene que partir en su aprendizaje de un punto en el cual recibe mucha más información de la que puede procesar cómodamente, para las que no tiene base previa de conocimiento en la que apoyarse, y en la que cualquier intento de actividad, de implementación, de participación por su parte, se ve truncado por un gran número de detalles que aún no comprende.

   Desde nuestro punto de vista, era conveniente centrar la atención de los alumnos en dos asuntos fundamentales:

   - La idea subyacente a cada una de las estructuras básicas de la programación estructurada, para que los alumnos logren captar exactamente qué hace cada una de ellas, las conozcan y sepan cómo se comportan.

   - La lógica propia de su computación, es decir, el modo en el que deben estructurar las órdenes para trasladar la solución de un problema a un algoritmo.

   La visión parecía acertada, pero ciertamente podía mejorarse añadiendo el resto de consideraciones que veremos a continuación. Ello, sumado a que el ordenador, proporciona una poderosa herramienta de simulación, deja a NWTJava v2.0 en la posición de una eficiente herramienta que simula un pequeño entorno de programación basado en diagramas de flujo que integran un nuevo conjunto de características deseables desde el punto de vista pedagógico.

2. Se basa en paradigmas de actuación hasta cierto punto conocidos por el alumno.

   Adaptando NWTJava al paradigma de editor que les resulta tan familiar, se les libera de una cierta carga cognitiva, quedando sumidos en una actividad mucho más cotidiana.

   Éste aspecto de nuestra aplicación queda ya avalado desde primeros estudios de *Piaget*[16] sobre la materia. *Piaget* dedujo que el conocimiento se compone de ciertas estructuras o esquemas cognitivos que se van desarrollando unas sobre otras. El niño nace con la capacidad lógica y tal vez algún otro patrón de conocimiento muy básico, y sobre ello va construyendo todo su conocimiento posterior. De este modo el pensamiento va evolucionando sobre sí mismo, creando y adaptando nuevas estructuras cognitivas. Es decir, el conocimiento se construye sobre conocimiento.

[16] Podemos encontrar su biografía y obra en la referencia número [6].

En este sentido, hemos estudiado también, entre otros muchos, el método perteneciente a lo que algunos han llegado a denominar "*escuela francesa*", enfoque propuesto por *Scholl* y *Peyrin*[17] y que, con algunas mejoras y modificaciones, ha demostrado su vigencia y eficacia a lo largo de los años y hasta nuestros días.

De hecho, la idea de estructuras mentales, y de que el conocimiento se construya en base a esquemas previamente adquiridos han constituido la base para toda una gama de nuevas aproximaciones a la materia, y para una nueva concepción de la pedagogía de la que NWTJava ya es partícipe. Más adelante, en este mismo capítulo, mencionaremos herramientas y experiencias pasadas que siguen esta misma filosofía.

3. Permite un alto grado de interactividad y de operatividad.

4. Proporciona realimentación al alumno, permitiéndole observar los resultados de sus acciones.

Con las consideraciones de los puntos 3 y 4, los chicos podrán ejecutar sus programas y observar su comportamiento, recibiendo de este modo una realimentación esencial para el proceso de acomodación de ideas y auto-corrección.

5. Mantiene un cierto enfoque lúdico.

Resulta evidente la necesidad de potenciar la motivación de los alumnos en cualquier proceso de este tipo. Es necesario facilitarles la adopción del proyecto para que el aprendiz consiga desplegar toda su capacidad.

Bajo esta línea de actuación se desarrollaron experiencias como las realizadas por *Seymour Papert*[18] con *Logo*, en las que la sensación de logro y de juego enganchaban directamente con la determinación del alumno, con su plan y con su motivación, hecho que atraía irremediablemente su atención tal y como comenta en [8].

Papert, considerado como uno de los padres de la inteligencia artificial, destacó enormemente por sus trabajos en la enseñanza basada en ordenadores. Promotor de, probablemente, la teoría pedagógica constructivista más conocida, desarrolló un lenguaje de programación de ordenadores llamado *Logo*. Dicho lenguaje funciona como un instrumento didáctico que permite a los alumnos, sobre todo a

---

[17] Estos dos autores describen y justifican en su obra, referencia [7], una amplia variedad de estrategias didácticas que reducen la sobrecarga cognitiva que supone el afrontar los primeros problemas de programación, posibilitando así una compresión más sencilla.

[18] Seymour Papert (1928-): destacado científico computacional, matemático y educador, es un pionero de la inteligencia artificial, inventor del lenguaje de programación LOGO en 1968. Trabajó con el psicólogo educativo Jean Piaget y basándose en los trabajos de éste sobre el Constructivismo, desarrolla una visión del aprendizaje llamado Construccionismo [9], cuyo apogeo se alcanzó a partir de los 80 con manifestaciones muy diversas. Fundó el Instituto de Inteligencia Artificial y creó el "*Epistemology & Learning Research Group*" ("Grupo de Investigación sobre el Aprendizaje y la Epistemología").

los más pequeños, construir sus conocimientos. Se trata de una potente herramienta para el desarrollo de los procesos de pensamiento lógico-matemáticos. En ella, se puede programar a un pequeño robot llamado la "*tortuga de Logo*" (*LogoTurtle*) que permite a los alumnos resolver problemas. Este robot, en forma de tortuga, realiza movimientos simples (izquierda, derecha, adelante, atrás) y dibuja líneas en su trayectoria. Para los niños, mover el robot y conseguir que dibuje formas (geométricas por ejemplo), es a la vez un juego y una actividad de construcción, lo que hace que mantengan la concentración durante elevados periodos de tiempo.

Pues bien, hemos llegado ya al final del análisis pedagógico que deseábamos hacer. Ya hemos argumentado y justificado cuales fueron las ideas con las que se enfocó el desarrollo de NWTJava v2.0. En el siguiente apartado haremos alusión a las recientes tecnologías, aplicaciones y experiencias cuyas características se consideraron positivas para la enseñanza de la programación.

2.3 Experiencias en la enseñanza de la programación estructurada

Como ya sabemos, la programación estructurada permite la escritura de programas relativamente fáciles de leer y modificar utilizando únicamente estructuras secuenciales, iterativas y condicionales (no de transferencia incondicional como pueda ser GOTO).

Pues bien, como ya comentamos anteriormente, desde el punto de vista de su docencia, han existido a lo largo del tiempo una serie de paradigmas muy heterogéneos, llegando incluso a coexistir diversos enfoques y tendencias. Aún hoy se puede verificar que no hay un consenso en los métodos a utilizar. Éste análisis así lo justifican, por ejemplo, las ponencias [10] y [11] que tuvieron lugar en las "*Primeras Jornadas de Educación en Informática y TICS en Argentina (2005)*".

Una de las razones por la que nos encontramos inmersos en esta vorágine metodológica es que no existe un único método para la resolución de algoritmos, así como tampoco un enfoque didáctico para materias introductorias que se haya impuesto sobre otros o demostrado una indiscutible efectividad.

No obstante, la metodología seguida por NWTJava (emplear un lenguaje algorítmico lo bastante general cómo para facilitar el salto posterior a cualquier lenguaje de programación frente al de enseñar a programar en un lenguaje de programación particular, utilizando su sintaxis y su semántica) se ha seguido por multitud de experiencias cuyo objetivo, no sólo era el de la docencia, también el de medir su eficacia en comparación a otros enfoques.

Es este el caso de la asignatura *Introducción a la Algorítmica y Programación*, recogida en el marco de los "*Proyectos de Innovación e Investigación para el Mejoramiento de la Enseñanza de Grad*o", durante los años 2004 y 2005 en la Universidad Nacional de Río Cuarto, Argentina.

Su programa, guiado por el nuevo enfoque del citado plan de investigación, responde con exactitud a nuestra filosofía docente.

Su metodología consistió en la ejecución de distintas fases que implican la elaboración de un algoritmo con el que se expresaba la solución del problema planteado en un lenguaje algorítmico. Una vez obtenido el mismo, se traducía a un lenguaje de alto nivel (en este caso PASCAL) y se implementaba en un ordenador. De esta forma, y según sus precursores, se conseguía independencia respecto a los lenguajes reales de programación, evitando así encasillar al alumno en un lenguaje particular.

Las estructuras de programación se iban integrando paulatinamente en el proceso de aprendizaje. Inicialmente se trabajaban las estructuras secuenciales con las que se elaboran algoritmos cuya solución era exclusivamente una secuencia de acciones. En segundo lugar se atendía al concepto de estructuras condicionales e iterativas, para finalmente acabar con los de abstracción y modulación. Esta fase del aprendizaje progresivo se basaba en acciones con parámetros (entrada, salida y referencia), primero en acciones simples y luego en funciones. Se hacía aquí hincapié en la necesidad de parametrizar para conseguir independencia entre los módulos y también entre las funciones.

La última fase de esta asignatura es la que contiene los principios en que nosotros basamos la introducción de funciones y subrutinas en NWTJava v2.0, como ya veremos en el *Capítulo 4: DISEÑO Y CONSTRUCCIÓN DE NWTJAVA v2.0*.

En cuanto a los resultados arrojados por dicha asignatura con temario experimental, sus precursores, Guillermo Rojo y Ariel Ferreira comentan los siguientes hechos[19]:

"El enfoque presentado anteriormente se aplicando desde 2004 y obedece la necesidad que se planteó en ese momento de realizar cambios pedagógicos, metodológicos, didácticos y organizacionales producto de los magros resultados obtenidos por los alumnos en cuanto a la calidad de sus aprendizajes, la alta deserción y el bajo rendimiento académico…"

"…En líneas generales podemos decir que se evidencian mejoras significativas en los últimos dos años respecto al porcentaje de alumnos que regularizan la materia, 31% en 2004 y 30% en 2005, contra el 20% en 2001, 21% en 2002 y 21% en 2003. No obstante estos números indican que todavía hay mucho trabajo por realizar. Por otra parte se realizaron encuestas a los alumnos ingresantes y recursantes a través de las cuales se pudo verificar una apreciación positiva de los mismos con respecto de los cambios de enfoque en la materia…"

Desde un punto de vista más técnico, podemos hacer referencia a una aplicación muy relacionada con nuestra metodología y nuestro diseño: **RAPTOR**.

RAPTOR (*Rapid Algorithmic Prototyping Tool for Ordered Reasoning*) nace como respuesta a los ya citados sistemas de enseñanza basados en un lenguaje de programación particular, y es, como lo definen sus creadores, un programa visual para la enseñanza de soluciones en problemas de algoritmia.

[19] Extractos recogidos de [11].

En la siguiente figura podemos observar una imagen de su interfaz gráfica:

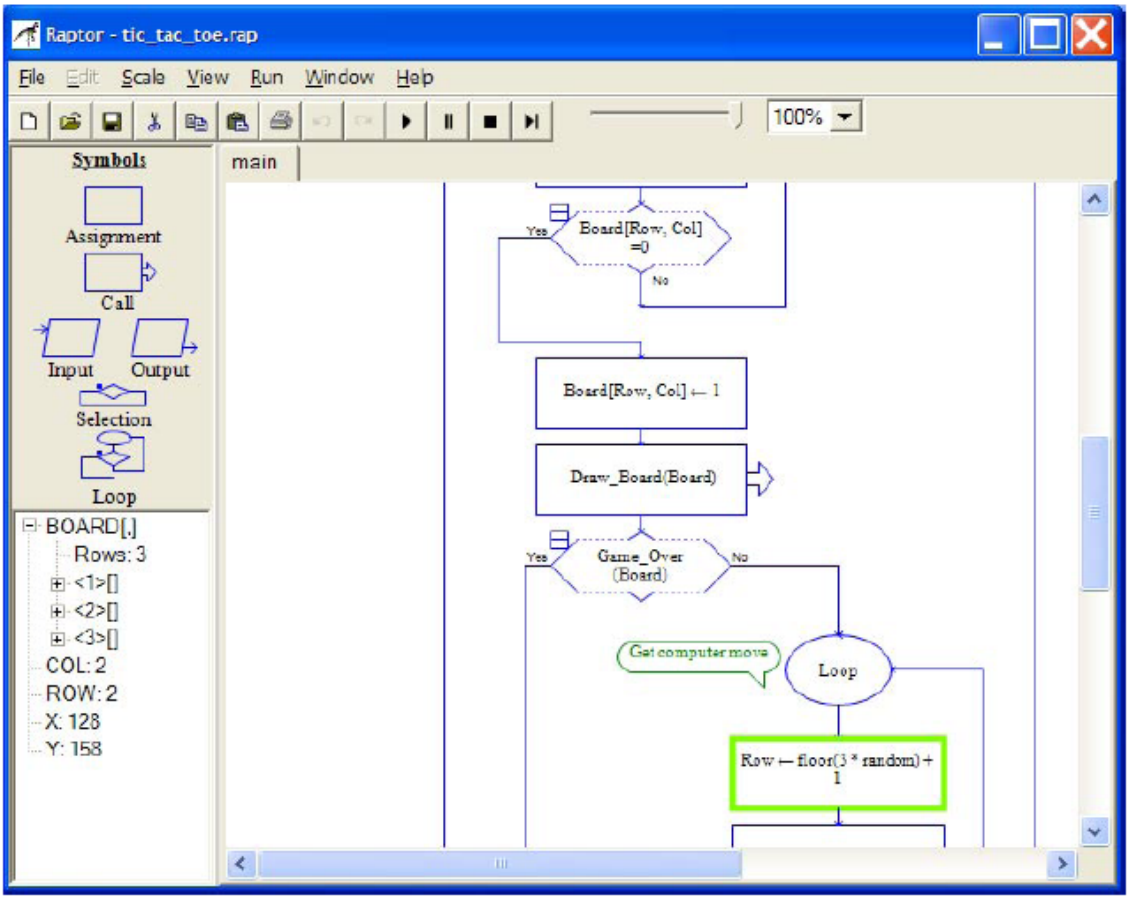


Fig. 2.1

NWTJava presenta una gran cantidad de similitudes con RAPTOR:

- Desde una paleta colocada en la esquina superior izquierda, el usuario de RAPTOR añade a la zona central de construcción de algoritmos una serie de símbolos, asignaciones, entradas y salidas, etc.

- Distingue también, al igual que esta nueva versión de NWTJava, entre diferentes tipos de variable, concepto que, desde el punto de vista de sus diseñadores, resulta fundamental.

- Ambos presentan una sintaxis muy flexible.

RAPTOR presenta sin embargo, una biblioteca de funciones o métodos de la que no goza NWTJava v2.0. De hecho, éste es uno de los trabajos futuros propuestos en el *Capítulo 5: Conclusiones y Trabajos Futuros*. Con estas funciones, RAPTOR es capaz de generar números aleatorios, realizar cómputos trigonométricos, dibujar la gráfica, etc.

No obstante, NWTJava v2.0 es capaz de utilizar y manejar una estructura que los creadores de RAPTOR echaron en falta como así citan en sus conclusiones:

"*Of particular attention will be the sigue of teaching and using arrays in RAPTOR as he results indicated this to be an area for improvement*."

Como podemos observar, dicha estructura de datos es el array. Comprobamos de esta forma que en aplicaciones de similares características ya construidas y utilizadas, los arrays son un elemento imprescindible. Ésto, unido a la insistente petición de los alumnos inscritos en nuestro experimento de campo (ver *Capítulo 3: Experimentación*), fue lo que nos motivó para incluir la utilización y manejo de este tipo de estructura de datos como un objetivo primordial.

De esta forma, pretendemos así abarcar con NWTJava v2.0 un gran compendio de elementos y conceptos grandes en importancia. En este apartado hemos justificado, o al menos planteado, la necesidad de incluir aspectos decisivos en la programación estructurada: tipos de variable, manejo de arrays, la posibilidad de codificar y utilizar funciones. Faltarían por incluir otros conceptos, como es, por ejemplo, el de ámbitos de variable, pero todos y cada uno de los restantes por analizar en este Cap2 son incluidos, y debidamente tratados, en las tecnologías en que se basó el NWTJava original, o en su defecto, en posteriores capítulos de esta memoria.

Citando rápidamente algunas de estas tecnologías: **lenguaje SL**[20], **puzzles en la enseñanza de la algoritmia** o **programación visual**[21], entramos ya en el último, y quizás más importante apartado de este segundo capítulo. Como sabemos, la función de éste es el de realizar un recorrido y analizar los trabajos, investigaciones, tendencias y experiencias que constituyen la base de la que parte nuestro proyecto.

Sin duda alguna, la mayor influencia en nuestro trabajo la ejerce NWTJava, su primera versión. Es lógico, pues es nuestro punto de salida, nuestro origen, de donde partimos para construir una segunda versión más desarrollada, potente y con un mayor alcance y profundidad en sus objetivos.

Una vez visto de donde parte la idea, la necesidad de ampliar NWTJava, y los puntos de apoyo sobres los que se cimenta esta segunda versión (la docencia, la pedagogía, las tecnologías asociadas…), nos queda por tanto analizar algo que consideramos fundamental, básico, de vital importancia. Nos referimos al diseño e implementación de NWTJava v1.0. Creemos que es imprescindible su estudio por la sencilla razón de que facilitará en gran medida el entendimiento de la dinámica seguida para la construcción de NWTJava v2.0: las ideas surgidas, las disquisiones, las conclusiones alcanzadas, las soluciones tomadas…, es decir, todo tipo de planteamiento asociado a este proyecto.

Por lo tanto, y sin más dilación, pasamos al citado apartado.

---

[20] Este lenguaje queda totalmente descrito en las referencias [12] y [13].

[21] Encontrará un detallado análisis de todas estas tecnologías en el proyecto antecesor a éste, referencia [5].

2.4 Diseño e Implementación de NWTJava v1.0

A partir de toda la información expuesta hasta el momento, vamos a ordenar y resumir muy brevemente los requisitos con los que partía esta antigua versión:

- Enseñar los conceptos básicos de la programación y la algoritmia.
- Con total independencia de un lenguaje de programación.
- Hacerlo de un modo sencillo, práctico e interactivo, que busque el razonamiento del alumno a la hora de afrontar la solución de un problema y su posterior codificación.
- Todo ello abordado desde un enfoque atrayente y divertido.

2.4.1 Diseño

Para hacer frente a estos requisitos y cumplimentarlos, se decidió dividir el diseño de la aplicación en dos bloques bien diferenciados.

Por un lado, la *interfaz gráfica*, que englobaría todo el conjunto de funcionalidades gráficas y el tratamiento formal[22] de los diagramas generados por el usuario. Con ello nos referimos al tratamiento de los elementos gráficos básicos (como puedan ser menús, botones, barras de scroll…) y a las funciones de construcción gráfica de los diagramas o programas[23] generados por el usuario (un ejemplo de una de estas funciones puede ser el proceso de validación al que se someten dichos diagramas).

Por otro lado, y como segundo bloque, tenemos el que se denominó como *núcleo de la aplicación*. En él se incluirían todos los procesos y funcionalidades internas de la aplicación, entre los que podemos citar el tratamiento gramatical, la ejecución o la traducción a lenguaje real.

Una vez introducido el diseño, podemos sintetizarlo con el siguiente esquema:

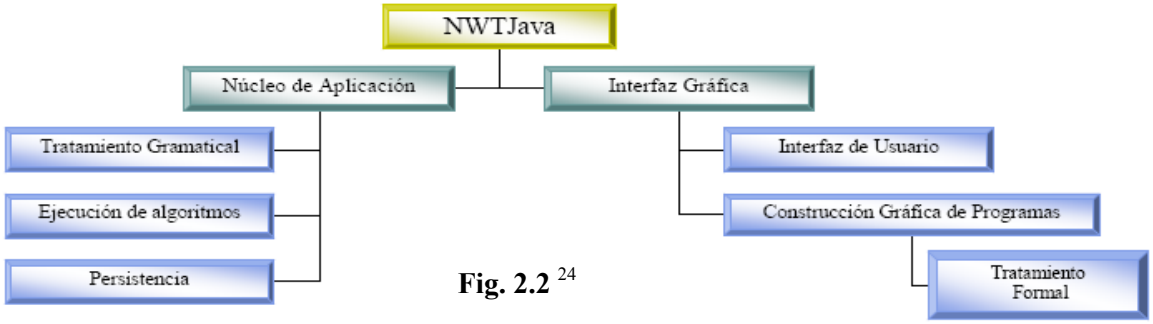


Fig. 2.2 [24]

[22] Con tratamiento formal nos referimos a la escritura, o en este caso dibujo, de los diagramas o programas que el usuario construye a partir de los símbolos de los que dispone.

[23] A partir de ahora utilizaremos los términos diagrama y programa indistintamente.

[24] Figura extraída de la memoria de Patricia Fernández. Ver [5].

Pasamos a continuación a describir el diseño[25] de cada uno de los bloques con mayor profundidad.

2.4.1.1 Interfaz de usuario

De acuerdo con los objetivos asumidos en el desarrollo de esta aplicación, la interfaz de usuario resultaba ser una pieza clave del proyecto. Debía también cumplir con una serie de objetivos adicionales:

a. Proporcionar una representación adecuada de los conceptos abstractos, es decir, de las estructuras de programación. Para ello se decidió seguir el siguiente convenio:

1. Para la representación gráfica de cada sentencia se utilizaría el dibujo tradicional empleado en los diagramas de flujo de código.

2. Se asignaría un color distinto a cada pieza[26].

3. Cada pieza de fin de sentencia tendría el mismo color que su pieza de inicio.

Como resultado de este convenio surgieron las siguientes piezas:

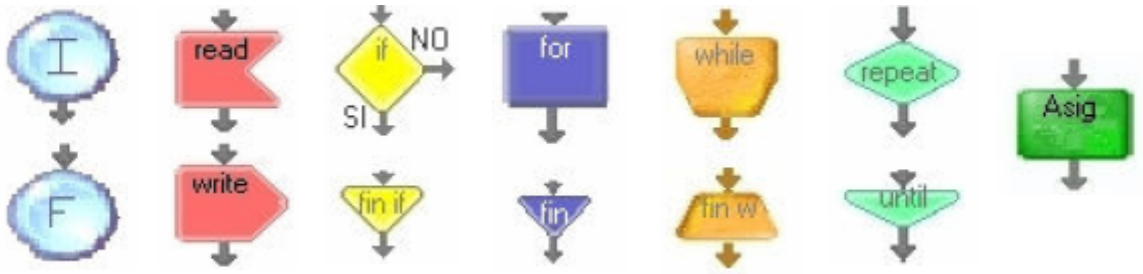


Fig. 2.3

b. El acceso a estas piezas debía ser sencillo e inmediato, de modo que el proceso de su utilización supusiese la menor carga cognitiva posible al usuario.

Para este fin, se decidió mantener el diseño de la interfaz del antiguo NEWT. Las piezas se situarían en un área auxiliar situada a la izquierda, dejando libre la zona central para la construcción de los diagramas. Además el mecanismo de inserción de piezas sería tan sencillo como pinchar y soltar en el punto deseado.

---

[25] La justificación de este diseño queda fuera del alcance y propósito de este proyecto. Es posible contemplarla en [5].

[26] Con el término *pieza* se hace referencia al componente gráfico que representa a una estructura de programación.

c. A pesar de ser una herramienta muy específica, debía asemejarse lo más posible a cualquier aplicación convencional que los alumnos hubiesen manejado con anterioridad (procesadores de texto, programas de dibujo…).

De esta forma la curva de aprendizaje se reduciría drásticamente y se crearía un marco más favorable, permitiendo a los alumnos concentrarse con mayor facilidad en los conceptos adecuados.

Se decidió por tanto, implementar utilidades como:

- Menú con las funciones estándar (más las específicas de NWTJava).
- Barra de herramientas con redundancia de las operaciones más frecuentes del menú.
- Activación de teclas rápidas.
- Etc…

d. La construcción de los algoritmos debía ser un proceso absolutamente gráfico, pues éste era uno de los atractivos que pretendía aportar la aplicación.

Para ello los programas se diseñarían dibujando un diagrama de flujo en el panel central de la aplicación. Para la creación de este dibujo, las piezas contenidas en su correspondiente cuadro de herramientas (recordamos que estaba situado a la izquierda) serían arrastradas al *Área de Trabajo*, e interconectadas por flechas que se trazarían con el ratón del mismo modo que se trazan líneas en cualquier programa de dibujo.

Al efecto, el diagrama que se generase, constituiría por sí mismo un objeto gráfico. Si hacemos un pequeño esfuerzo de visión espacial, rápidamente caeremos en la cuenta de que dicho objeto gráfico se puede asociar perfectamente con una estructura de grafo.

Por tanto, se tendría un objeto general, el diagrama (grafo), compuesto a su vez por otros objetos: cada nodo (que representaría una instrucción) y cada enlace (una dirección de flujo).

Sin embargo, no cualquier combinación de nodos y conexiones tendría validez. Se debería por tanto seguir un conjunto de reglas en la construcción de estos diagramas. Ante esta situación, y analizando el conjunto de directrices que se emplean para la construcción de diagramas de flujo clásicos, se definió un modelo de grafo[27]. Algunas de las normas impuestas para este modelo de grafo fueron:

- Los enlaces debían ser direccionales y un único sentido.
- Sólo se permitiría la entrada o salida de enlaces en determinados puntos (o puertos) de un nodo. Estos puertos serían: superior, inferior y lateral.

[27] Un modelo de grafo corresponde a un conjunto de reglas que definen una gramática. Éste debe cumplirse para todos los grafos.

- Cada nodo (o estructura de programación, como pueda ser un *while*) estaría definido por una serie de propiedades, y no todos tendrían todos los puertos disponibles. Ejemplo: un nodo correspondiente a la estructura *for* sólo tendría un puerto de entrada (o superior) y otro de salida (o inferior), mientras que el correspondiente a un *if* tendría un puerto de entrada y dos de salida (uno inferior y otro lateral) correspondientes a cada una de las ramas de decisión posible (*true* o *false*).

- Deberían respetarse las siguientes restricciones en cuanto a número, posición y sentido de los enlaces (o ejes) entre nodos:

  - Un eje únicamente podría tener su origen en un puerto inferior o lateral.
  - Un eje sólo podría tener su destino en un puerto superior.
  - Un eje tendría un único origen y un único destino.
  - De un puerto solamente podría salir un eje.
  - A un puerto podrían llegar uno o más ejes.

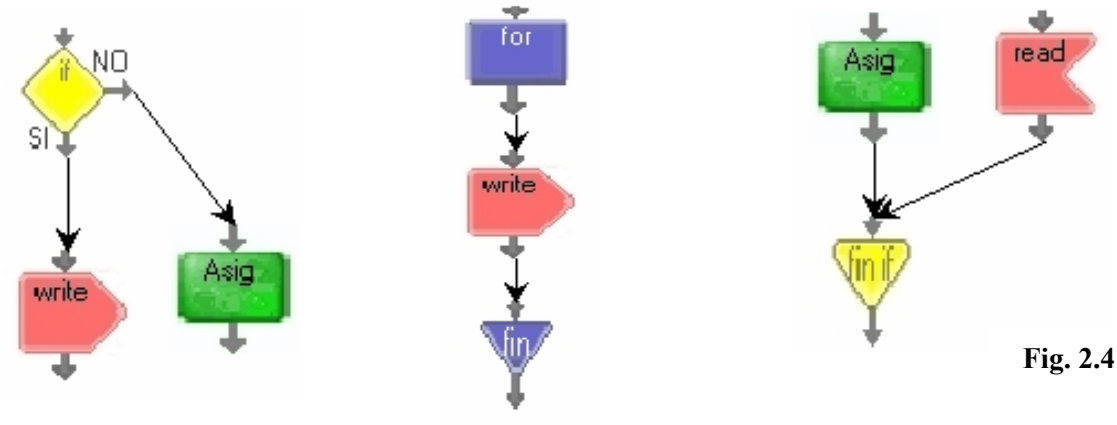


Fig. 2.4

La interfaz gráfica debe por tanto controlar y asegurar el cumplimiento de estas restricciones en los grafos que se dibujen en pantalla.

e. Finalmente la interfaz debía implantar mecanismos para que los algoritmos en ejecución dispusiesen de una consola de entrada/salida con la que el usuario pudiese interactuar, así como una consola de información sobre las acciones de dicho algoritmo durante la ejecución.

También hubo que tener en cuenta que el módulo del núcleo de la aplicación debía ofrecer una serie de prestaciones, que aunque a nivel interno, debían tener medios para acceder al usuario en determinados momentos. Para cumplir con este cometido, la interfaz también debía implementar otras funciones como: control para la validación[28] gramatical del grafo; control para iniciar la ejecución del programa; control para finalizar la ejecución en curso; ventana de ejecución donde se mostrarían las salidas del programa ejecutado; etc.

[28] Podemos entender la validación como un proceso análogo al de la compilación.

Podemos dar por acabada la descripción del diseño de la interfaz gráfica. Más adelante estudiaremos su implementación, ahora vamos a analizar el diseño del segundo bloque de NWTJava, el núcleo de la aplicación.

2.4.1.2 Núcleo de la aplicación

Este bloque se compone del conjunto de funcionalidades que se realizan a nivel interno. En él se pretendía implementar el código necesario para satisfacer los siguientes objetivos:

a. Tratamiento gramatical de los programas o algoritmos.

- Validación gramatical.
- Traducción a lenguajes reales.

b. Ejecución de los mismos.

c. Tareas auxiliares de la aplicación como:

- Persistencia de datos.
- Gestión (a nivel interno) del conjunto de programas que se encuentran en edición simultáneamente.

Se partió de la base de que cualquier tratamiento gramatical se compone de dos fases:

- Las reglas. Establecimiento del conjunto de normas que definen la gramática.
- Como aplicarlas. Diseño de los mecanismos que permiten evaluar una estructura mediante ese conjunto de reglas.

Y para ello se hizo uso de un mecanismo prácticamente estándar para la definición de gramáticas en computación: XML. Gracias a XML, un parser del mismo y DOM[29], se resolvieron automáticamente los dos puntos anteriores. Además, DOM proporcionó un soporte en forma de árbol muy útil para el almacenamiento de programas en memoria y un amplio surtido de clases y funciones muy eficaces y optimizadas para este trabajo.

---

[29] Para extraer la información que contiene un documento XML, se podría escribir código para analizar el contenido del archivo XML, pues no deja de ser un archivo de texto, tal y como lo podríamos hacer con HTML. Sin embargo, esta solución no es muy aconsejable y desaprovecharía una de las ventajas de XML: el ser una forma estructurada de representar datos.

La mejor forma de recuperar información de archivos XML es utilizar un parser de XML, que sea compatible con el modelo de objeto de documento (DOM) de XML. DOM define un conjunto estándar de comandos que los parsers devuelven para facilitar el acceso al contenido de los documentos XML desde sus programas. Un analizador de XML compatible con DOM toma los datos de un documento XML y los expone mediante un conjunto de objetos que se pueden programar.

Mediante la codificación del citado parser, se pudo también implementar la traducción de los programas. Éste generaría cadenas representativas de cada uno de los nodos y sus propiedades. Por ejemplo, un nodo *for* con los parámetros de, una variable llamada *var*, inicio igual a 0, límite de iteración hasta 10 e incremento igual a 2, se representaría por la cadena:

for (int i=0 ; i<=10 ; i+=2) {

Del mismo modo, al encontrarse los programas creados por el usuario en forma de árbol (en memoria), la ejecución se simplificó sobremanera, pues bastó con implementar un recorrido preorder de dicho árbol (con una pequeña peculiaridad[30]).

Por último, la utilización de DOM facilitó la implementación de la persistencia de los programas. El almacenamiento en memoria era inmediato salvando el programa, representado como árbol DOM, en un archivo de texto XML.

Los objetivos que se plantearon a priori se encontraban perfectamente cubiertos con la utilización de DOM. Sin embargo, éste proporcionaba todavía más ventajas en cuanto a posibles compatibilidades y separación de código.

No obstante surgió una dificultad. Todo lo expuesto hasta el momento referente a este bloque, dependía de si era o no posible construir una DTD[31] que definiese la gramática deseada. Lo cierto es que fue posible.

A continuación podríamos exponer todas y cada una de las reglas que definen esta gramática, sin embargo, escapa del objetivo marcado para este apartado. Únicamente haremos constar una pequeña lista de puntos que se tuvieron en cuenta[32]:

- El conjunto de elementos (estructuras) a incluir en la gramática y sus propiedades (atributos).
- El conjunto de elementos (hijos) que contendría cada elemento, además de su tipo, número y carácter de obligatoriedad si fuese necesario.
- El conjunto de valores de los atributos de cada elemento.

Pues bien, con esto damos por concluida la descripción de lo que fue el diseño de NWTJava v1.0.

---

[30] Comentada por supuesto en su correspondiente proyecto. [5].

[31] Document Type Definition. Podemos dividir los parsers XML en dos grupos principales:
- sin validación: el parser sólo chequea que el documento esté bien formado de acuerdo a las reglas de sintaxis de XML (sólo hay una etiqueta raíz, las etiquetas están cerradas, etc).
- con validación: además de comprobar que el documento está bien formado según las reglas anteriores, comprueba el documento utilizando una DTD (ya sea interna o externa).

[32] Volveremos a hablar de la DTD en el *Capítulo 4: DISEÑO E IMPLEMENTACIÓN,* cuando tengamos que mencionar nuestro trabajo realizado al respecto de las DTDs.

2.4.2 Implementación

Pasamos ahora a analizar la implementación de cada uno de los bloques descritos en el diseño. En opinión del autor, lo que veremos a continuación es fundamental para el posterior entendimiento de NWTJava v2.0.

2.4.2.1 Interfaz de usuario

Para saber en todo momento de lo que estamos tratando, y no perder así la referencia, lo primero que vamos a hacer es ver literalmente, cual es la interfaz de usuario. Es por tanto la *Fig. 2.5* el resultado de la implementación que vamos a describir a continuación:

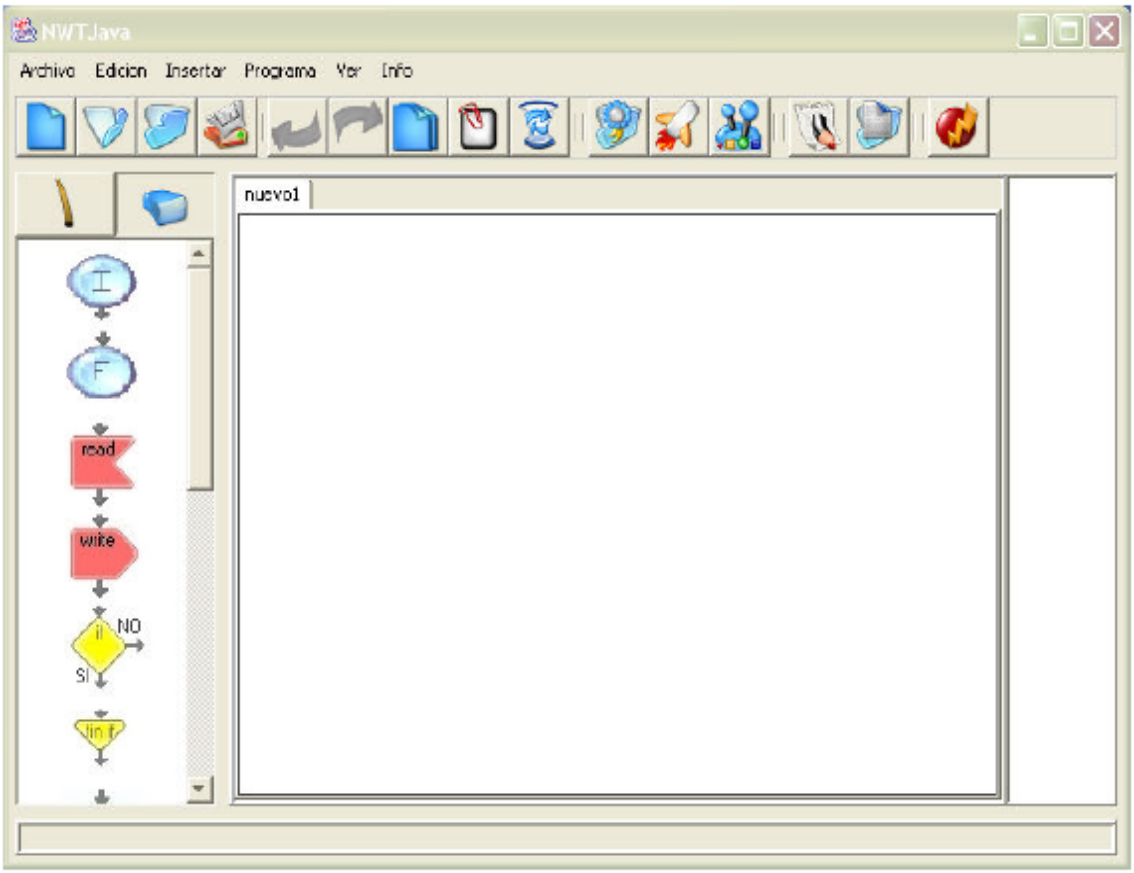


Fig. 2.5

En el panel de la izquierda podemos observar el conjunto de piezas con los que el usuario puede construir su diagrama. El área de la derecha queda reservada para la traducción de los diagramas a código real, y el panel central es el dedicado a dibujar los diagramas. Para la creación de estos diagramas, y como ya explicamos con anterioridad, las piezas contenidas en su correspondiente cuadro de herramientas, serían arrastradas al *Área de Trabajo* mediante un mecanismo tan sencillo como pinchar y soltar en el punto deseado. Después se unen por flechas trazadas con el ratón del mismo modo que se trazan líneas en cualquier programa de dibujo.

La clase central de éste bloque (y también de la aplicación completa) es *NWTJava*. Esta clase es la que contiene el método *main* de la aplicación, y se encarga fundamentalmente de gestionar los flujos de interacción entre programa y usuario. Para el desempeño de esta función incluye:

- Los controles que el usuario puede accionar (menús, botones, teclas rápidas,…).
- Los mecanismos de realimentación con el usuario (ventanas de información, de ejecución,…).
- El soporte para otros objetos obligados a residir en un contenedor gráfico (como pueden ser los *NWTGraph* o los *Traductores*[33]).

Sin embargo, como clase principal de la aplicación, *NWTJava* también se encarga de la dirección de flujos de trabajo del resto de clases.

Para revelar cuales son el resto de clases cardinales que constituyen la interfaz, como quedan estructuradas, y cual es la relación entre ellas, hacemos uso del siguiente diagrama UML:

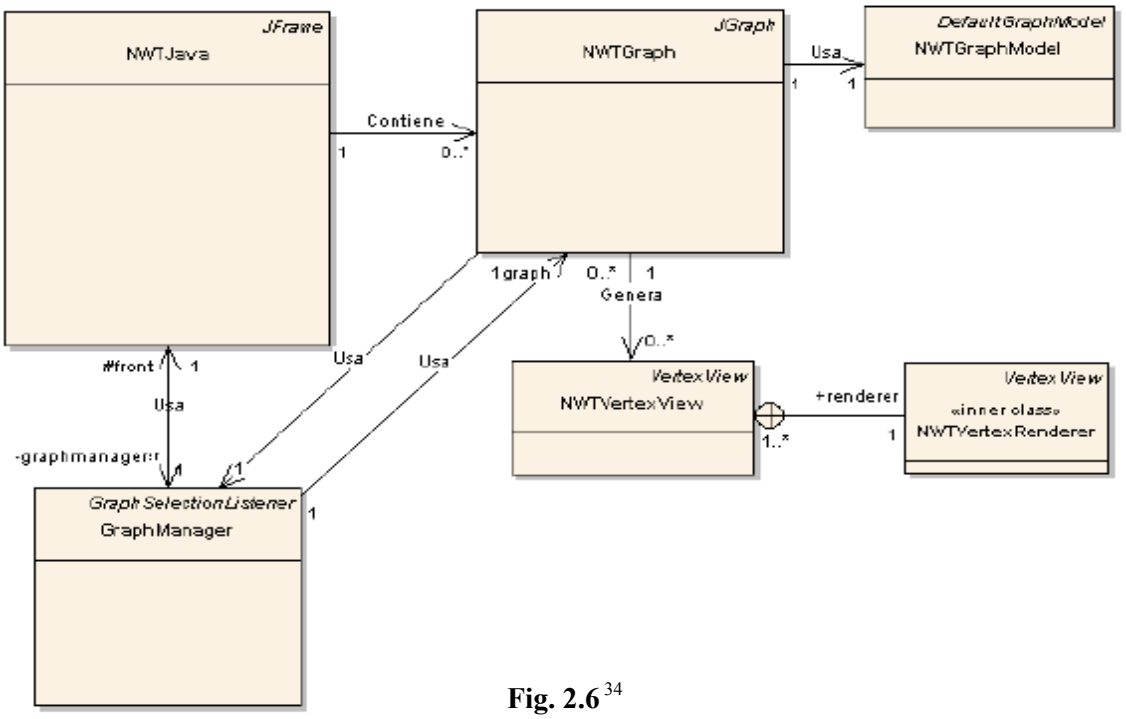


Fig. 2.6[34]

¿Qué implementan pues el resto de clases? Para responder a esta pregunta vamos a seguir uno de los posibles hilos que desentrañan el funcionamiento de éste bloque.

---

[33] Hablaremos de ellos más adelante.

[34] Para ver el diagrama UML al completo, del cuál hemos extraído este esquema, se puede consultar el *Pliego II: Planos* de la memoria: "NWTJava, un micro-entorno de desarrollo para el aprendizaje de la programación". Referencia [5].

Vamos a establecer el origen de este recorrido por la ya conocida clase *NWTJava*.

Oculta entre sus funciones ya mencionadas, se encuentra la de que, dicha clase, permite manejar en el área central varios diagramas simultáneamente mediante un clasificador de tipo *JTabbedPane*[35]. Pues bien, cada uno de estos diagramas es un objeto de tipo *NWTGraph*.

Por tanto, tenemos que *NWTGraph* es una clase que implementa grafos. Pero no sólo eso, también contiene las limitaciones de formato que exige NWTJava (son las limitaciones que describimos en el *punto d* del diseño de la interfaz gráfica, y que permiten que un grafo sea completamente editable).

Como se puede observar en el diagrama UML anterior, *NWTGraph* hereda de la clase *JGraph*, la cual pertenece a la librería *jgraph*[36].

Detengámonos unos instantes en este punto.

Jgraph es una librería externa, de código abierto, y que fue utilizada por su adecuación a las necesidades del proyecto. Aporta toda una estructura de clases capaz de proporcionar objetos gráficos idóneos para la representación de grafos. Además son objetos totalmente compatibles con el entorno gráfico que proporciona *Swing*, con todas las facilidades que ello conlleva.

De hecho, la gran utilidad que supone su uso es la capacidad que tiene su clase *JGraph* de encapsular toda la complejidad de visualización de un grafo, convirtiéndolo en un componente más de *Swing* (de esta forma, este complejo objeto se puede situar en cualquier contenedor como si de un simple botón se tratase).

Ejemplos de funciones que encapsula son la gestión del dibujado y refresco de sus nodos y ejes o el permitir dinamismo en la posición y tamaño de los nodos, pudiendo así redimensionarlos y moverlos sin desbaratar las conexiones existentes entre ellos.

En resumen, proporciona todo el dinamismo e interactividad que se requería en el proceso de dibujado de los diagramas.

Sin embargo, y para terminar ya con *JGraph*, como ocurre con todo componente *Swing*, su diseño responde a la arquitectura "*separable model architecture*", lo que implica que los datos de control que necesita para funcionar se separan de la lógica de pintado y de cómo se comporta ante acciones del usuario. Estos tres puntos son los que corresponden respectivamente al *Model*, *View* y *Controller* del componente. *View* y *Controller* se fusionan y esta división se reduce a *Model* e *UI*.

Para NWTJava, el objeto *UI* de *JGraph* servía inalterado, pero no así el *Model*, que determinaba la definición de un nodo NWT. Es aquí donde entran en juego las clases *NWTGraph, NWTGraphModel, NWTVertexView* y *NWTVertexRenderex.*

Retomamos la clase *NWTGraph*. Como ya hemos comentado, es una clase hija de *JGraph*, y fue creada por diversos motivos:

---

[35] Documentación de las clases de *Swing* en referencia [14].

[36] Documentación de las clases de j*graph* en referencia [15].

- Incluir ciertas propiedades de instancia que un grafo NWT debía tener para su tratamiento dentro de la aplicación: un identificador único y una referencia al objeto *GraphManager*[37].

- Añadir métodos que envolviesen el manejo del *Model* del grafo, facilitando así ciertas operaciones comunes sobre grafos dentro de NWTJava. Como ejemplo podemos citar el método que transforma el grafo en un diagrama de flujo. Realmente esta transformación es ficticia, pues lo que realmente hace es proporcionar un determinado orden de recorrido al grafo. Sin embargo es totalmente análogo a lo que sería recorrer un diagrama de flujo.

- Proporcionar al contenedor de *Swing* que contiene al grafo un objeto *View* (de tipo *NWTVertexView*) adecuado para dibujar sus nodos, en lugar del objeto genérico de *jgraph* (*VertexView*).

Por lo tanto, podemos decir que *NWTGraph* utiliza un objeto *Model* para gestionar los datos que contiene, *NWTGraphModel*, y genera un objeto *View*, *NWTVertexView*, cada vez que sea necesario dibujar un nodo.

Vamos a examinar ahora la clase *NWTGraphModel*. Se creó para proporcionar las limitaciones ya mencionadas en la construcción de grafos[38], y posibilitar así que éstos puedan representar diagramas de flujo válidos. Debido a ésto, el cuerpo de algunos de sus métodos almacena cierto conocimiento sobre el comportamiento de cada grafo *NWTGraph*.

Para no perder la perspectiva de lo alcanzado hasta el momento, y utilizando un vocabulario muy sencillo y alejado de los tecnicismos propios de la programación, podemos decir que:

> Hasta el momento, lo que tenemos son un conjunto de estructuras (*if, for, while,…*), con sus propiedades asociadas. Estas estructuras, con sus correspondientes propiedades, se almacenan en los nodos. También existen los enlaces (también llamados vértices o ejes) que la única información que contienen es el de la dirección del flujo. Una combinación válida de estos elementos es lo que constituye un diagrama o grafo.
>
> A modo de recordatorio, decir que estos diagramas son construidos y representados en el panel central de la aplicación.

Pues bien, como se podía llegar a pensar, la información contenida en los nodos es fundamental, pues de ella dependen cosas como, por ejemplo, el icono usado para dibujar el correspondiente nodo.

Y es precisamente el almacenamiento de esta información para lo que se creó la clase *UserObject*.

---

[37] Comentaremos esta clase más adelante.

[38] Relacionadas con la edición de los nodos, las reglas impuestas a los enlaces que los unen y la combinación de ambos. Puntos *a* y *d* del diseño de la interfaz gráfica.

Por tanto, son los objetos de esta clase los que definen a un nodo, lo particularizan. Llevan también el peso de la gestión de sus correspondientes tipos. En concreto:

- Mantiene los distintos tipos de sentencias disponibles en la aplicación mediante constantes de clase públicas (ej: `public static final int IF=1; public static final int WHILE=6`).

- Cada instancia de esta clase representa una sentencia concreta. Es decir, una sentencia particular con sus atributos particulares, un "*if(a>4)*" en vez de un "*if*".

Además, los objetos *UserObject* se emplean como objetos de intercambio con las clases pertenecientes al núcleo de la aplicación[39]. Como consecuencia de ésto, *UserObject* cuenta con atributos relacionados con cada uno de los dos grandes bloques.

- *nodoGemelo* para el bloque del núcleo, del que veremos cual es su utilidad,

- e *icon* para el bloque de la interfaz. Es la imagen con la que se representa la pieza en pantalla (concretamente hace referencia al nombre del archivo de imagen que debe cargar).

Cualquiera de los dos bloques puede construir un objeto de esta clase, e incluso transferirlo al otro, pero de esta forma, siempre se estará operando con la misma estructura concreta.

Sin embargo, el hecho de que el objeto *UserObject* almacene el icono con el que dibujar la pieza, no significa que la dibuje, y es aquí donde abandonamos la exposición de *UserObject* y retomamos la implementación de la interfaz gráfica. Aparecen las clases *NWTVertexView* y *NWTVertexRenderex.*

Consultando de nuevo el diagrama de la *Fig. 2.6*, podemos observar como ambas heredan de clases de *jgraph*. Cuando se implementó esta parte, los autores del proyecto necesitaban que los nodos del objeto *NWTGraph* (el objeto grafo) se dibujasen de una forma muy específica (no como simples polígonos que era lo que hacían los *UI* de *JGraph*). Para ello implementaron la clase *NWTVertexView*, para ser utilizada por los *NWTGraph* en lugar de los objetos *View* por defecto de *JGraph* (*VertexView*).

Por otra parte, de toda la funcionalidad de dibujado se encargaría *NWTVertexRenderex*, el cual sobrescribe el método *paint* de su clase contenedora.

Para ver realmente como funcionan estas dos clases, transcribimos literalmente la explicación de Patricia Fernández al respecto[40]:

---

[39] Éste es el motivo por el que la clase *UserObject* no está representada en el diagrama 2.9 que hace referencia exclusivamente a la interfaz de usuario.

[40] Páginas 68-69 de la memoria: "NWTJava, un micro-entorno de desarrollo para el aprendizaje de la programación". Referencia [5].

"Cuando un nodo tenga que ser dibujado en pantalla, se invocará el método createVertexView del objeto NWTGraph que lo contiene. Éste devuelve un objeto NWTVertexView para el nodo solicitado, al que a su vez se le pedirá un dibujador que Jgraph utilizará para pintar el nodo.

Puesto que el proceso de pintado para cualquier nodo (independientemente de su tipo y de todo lo demás) consistía siempre en dibujar una imagen, la clase NWTVertexView, simplemente define una propiedad estática renderer que contiene un NWTVertexRenderer que dibujará todos los objetos NWTVertexView. De este modo se evita que cada NWTVertexView tenga que almacenar su propio objeto NWTVertexRenderer."

Ya sólo queda por describir una de las seis clases que aparecen en el diagrama UML. Ésta es *GraphManager.* Tiene tres funciones:

- Implementa el arduo tratamiento de los objetos *NWTGraph*, ocultando su complejidad.
- Mantiene información de control sobre los diferentes grafos abiertos, además de cuál es en cada momento el grafo activo.
- Centraliza las comunicaciones con el núcleo de la aplicación, trabajando conjuntamente con la clase *Datos*[41] para realizar las acciones ordenadas por *NWTJava*.

Estas tres acciones esconden una engorrosa y compleja estructura de atributos y métodos que no vamos a describir en esta memoria. Por el contrario, si que vamos a dedicar un instante a explicar la edición de los nodos que componen cada objeto *NWTGraph*. Es importante.

Puesto que *GraphManager* implementa el acceso a los grafos, la localización de los objetos *UserObject* correspondientes (los nodos), es su responsabilidad. Por este motivo, se añadió a la clase el método *editar*. Este método obtiene el objeto *UserObject* del nodo seleccionado en el grafo activo, y llama al método *editAtt* de *NWTJava*, el cual muestra por pantalla una tabla con las propiedades de dicho *UserObject*. Estas propiedades, que deben ser rellenadas por el usuario (pudiendo ser modificadas cuando precise), se almacenan en el nodo. De esta forma queda ya fijado el nodo o estructura a utilizar con sus correspondientes parámetros.

Un ejemplo: para nuestro programa, o diagrama de flujo, queremos emplear una estructura *for*. Pues bien, seleccionamos su nodo, lo pasamos al *Área de Dibujo*, y una vez allí, lo editamos. Como ya sabemos, editarlo consiste en particularizarlo, darle unos valores determinados, es decir, rellenar sus parámetros con los valores que creemos necesarios. Como bien sabemos, un *for* requiere de un valor inicial, uno final y un incremento. Por tanto son precisamente esos parámetros los que muestra la tabla de propiedades (denominada *Tabla de Edición*). Una vez completada dicha tabla, su información pasa a formar parte del nodo, que junto con el tipo de estructura (*for*), constituyen el objeto *UserObject*.

[41] Comentaremos esta clase más adelante.

Vemos otro ejemplo. En este caso de forma gráfica y con una estructura *if*:

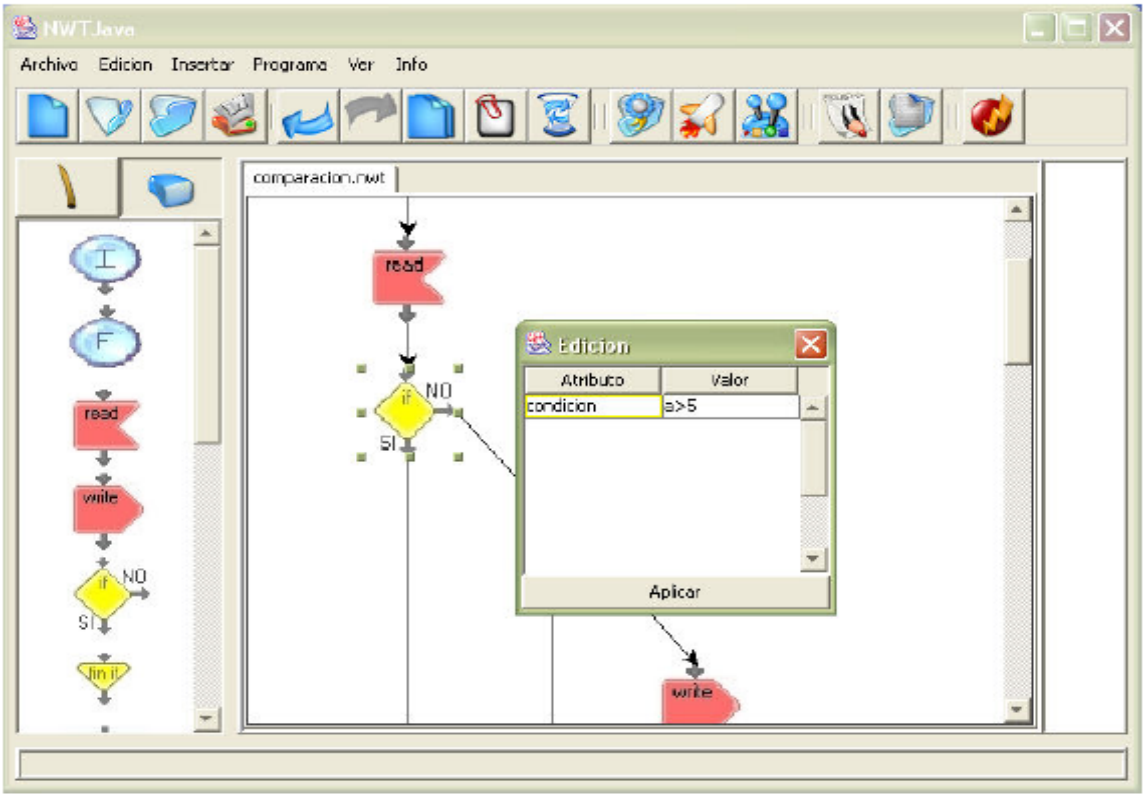


Fig. 2.7

Y hasta aquí la descripción de la implementación del primer bloque. Ya comentamos que se trataba de una versión reducida, y es por ello que existen todavía más clases en este bloque aún sin comentar. Sin embargo, encomiendan tareas auxiliares, y no merece la pena desglosarlas por dos motivos. El primero es que ya están todas explicadas en la memoria de su correspondiente proyecto (referencia [5]). El segundo es que su papel en NWTJava v2.0 no resulta relevante.

Una vez dicho ésto, y antes de pasar a exponer el segundo bloque, vamos a comentar el modo en que coexisten, se conectan y relacionan interfaz y núcleo, la interrelación entre bloques. Ésto además servirá para que el cambio entre bloques no sea brusco y forzado, sino más bien una transición en la que se puede ver la relación existente entre ambos.

2.4.2.2 Interrelación entre bloques

Volvemos aquí a mencionar la clase *GraphManager*. Si recordamos, esta clase cumplía una triple función. Pues bien, retomamos la tercera:

- Centraliza las comunicaciones con el núcleo de la aplicación, trabajando conjuntamente con la clase *Datos* para realizar las acciones ordenadas por *NWTJava*.

Como vemos, se refiere a la comunicación entre bloques, y para entender en su totalidad el papel que juega *GraphManager* en este canal de comunicación, utilizamos el siguiente diagrama de clases:

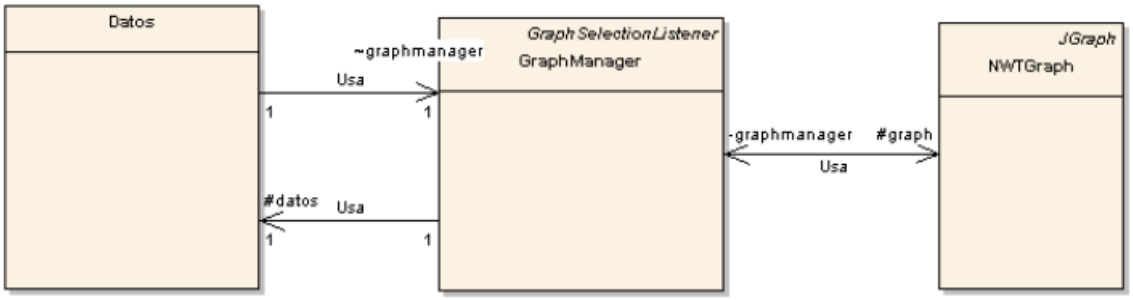


Fig. 2.8

Se encarga de ejecutar las acciones necesarias en cada momento sobre *Datos*, que como veremos, es la clase central del Módulo de Gestión[42]. Otra forma de entenderlo es que *GraphManager* hace transparente a ambos bloques entre sí.

Pero lo verdaderamente destacable en este apartado es el mecanismo de transferencia de programas de usuario[43] entre bloques. Este asunto supuso uno de los puntos más comprometidos en el desarrollo de la aplicación. El principal motivo de este hecho es que los programas de usuario eran representados mediante árboles en el bloque núcleo, mientras que en la interfaz eran grafos. Dos estructuras diferentes y que sin embargo debían poder trasladar información de un bloque a otro para que todo encajase.

Dicha traslación de programas de usuario quedó resuelta e implementada con la construcción de dos métodos de *GraphManager*:

- *resumeGrafo*: que genera un array de objetos *UserObject* correctamente ordenados que representa al grafo.

- *construyeGrafo*: que genera un objeto *NWTGraph* a partir de un array de objetos *UserObject* correctamente ordenados.

De esta forma, un grafo, constituido por nodos (objetos *UserObject* que podían intercambiarse entre bloques) y con un sentido único de flujo, podía transformarse en un array que sirviese a ambos bloques. Sin embargo, no todo este diseño fue trivial. Hubo una importante excepción a la hora de generar dichos arrays, la sentencia *if*. Al tener dos caminos de flujo, se necesitó establecer un convenio especial para el almacenamiento de cada rama en un orden concreto. De no hacerse así, hubiese sido imposible distinguir los nodos de cada una de las ramas, pues todos tenían el mismo nodo inicio y el mismo nodo final.

---

[42] El módulo de gestión es una de las partes en que se dividió el núcleo de la aplicación.

[43] Término con el que se hizo referencia a los programas o diagramas desarrollados por los usuarios en NWTJava v1.0.

2.4.2.3 Núcleo de la aplicación

Finalmente llegamos al segundo y último de los bloque en que se dividió NWTJava, el núcleo de la aplicación. Para introducirlo vamos a hacer uso del siguiente diagrama de clases, que contiene una representación resumida de la implementación final del bloque:

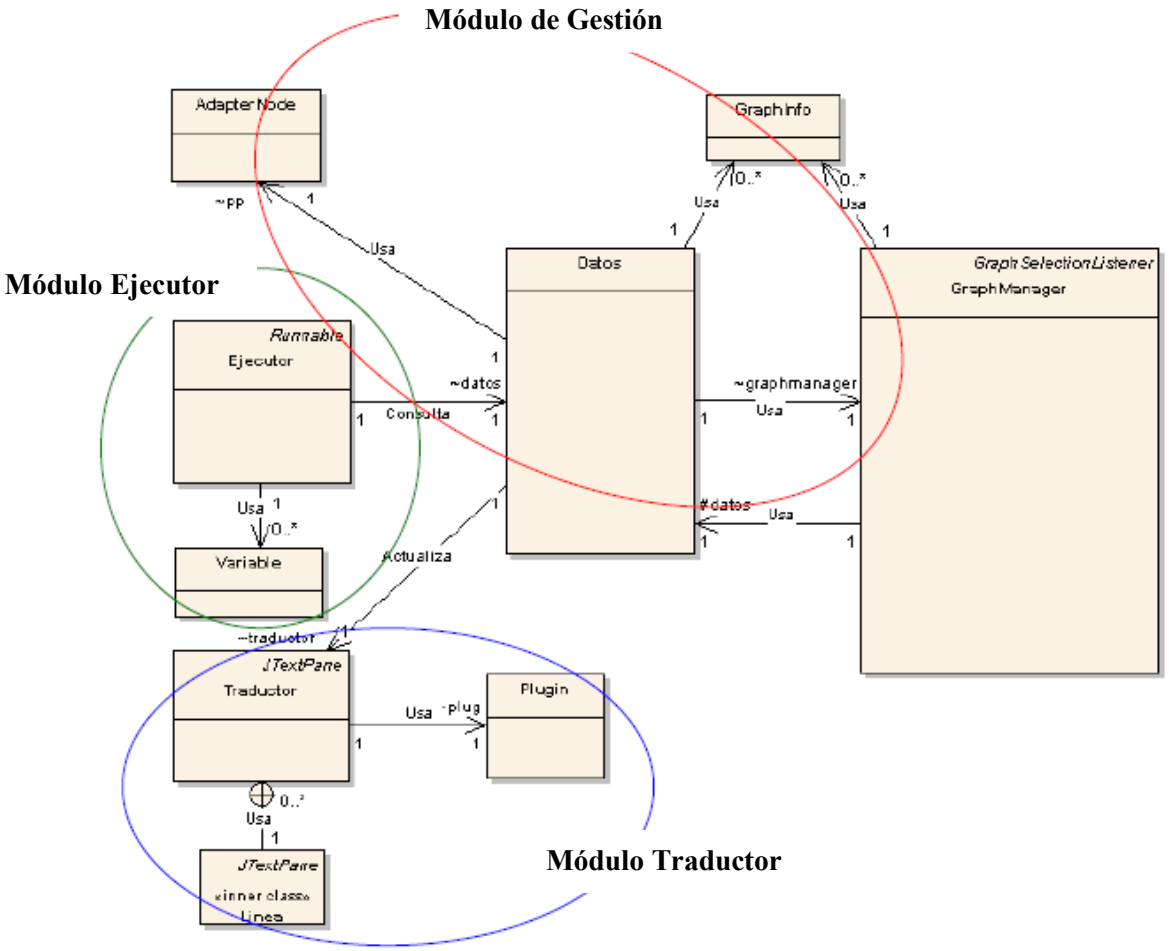


Fig. 2.9

Como podemos observar en el gráfico, este bloque se compone de tres módulos

- Módulo de Gestión: encargado de almacenar y organizar los distintos árboles[44] que están en uso, realizar las modificaciones sobre los mismos durante la ejecución del programa y responsable de las tareas de persistencia.
- Módulo Ejecutor: procesa la ejecución de un árbol.
- Módulo Traductor: traduce el árbol a un lenguaje de programación real (en este caso Java).

[44] Recordamos que en el ámbito de este bloque, con el término "árbol" nos estamos refiriendo a un programa editado por el usuario dentro de la aplicación. Se escogió este término porque precisamente es en estructura de árbol el formato con el que se implementa aquí. Si hacemos memoria, en el bloque interfaz lo denominábamos "grafo", y para hacer una mención general decíamos "programa" o "diagrama".

Como en el apartado anterior, el objetivo de la descripción que vamos a exponer a continuación no es listar de un modo exhaustivo las clases, atributos y métodos, sino realizar una presentación aclaratoria de los mismos, de sus puntos esenciales y de aquello que influyó, o de alguna manera repercutió en la toma de decisiones referentes a NWTJava v2.0[45].

Comenzamos, pero antes, dos apuntes significativos. El primero es que la clase *GraphManager* se incluye en el diagrama de la *Fig. 2.9* para tener una referencia de cual es el límite de este bloque. El segundo es que, si recordamos, este bloque implementa los programas de usuario mediante árboles *jdom*[46].

Módulo de Gestión.

Este módulo se compone fundamentalmente de dos clases. Una principal, *Datos*, y otra auxiliar llamada *AdapterNode*.

Pues bien, la clase *Datos* almacena los árboles correspondientes a los distintos programas de usuario abiertos en la aplicación. Enumeramos sus funciones:

- Persistencia de los programas de usuario (almacenamiento y recuperación de ficheros).
- Validación gramatical de los programas o árboles.
- Mantenimiento del puntero de programa utilizado en la ejecución de los árboles.

Para el almacenaje de los programas en uso, Datos mantiene en memoria una tabla que contiene objetos *Document*, con los que implementa cada programa. Dicha tabla esta indexada por los objetos *GraphInfo* que mantiene *GraphManager*, por lo que estos objetos sirven de referencia e identifican a un único árbol. Si analizamos las propiedades de estos objetos,

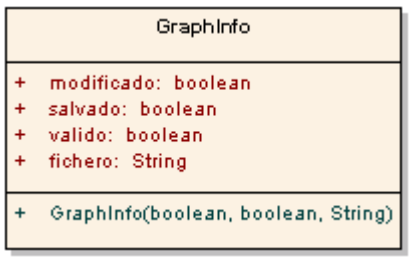


Fig. 2.10

vemos como particularizan cada diagrama (con el nombre del mismo, almacenado en el atributo *fichero*).

---

[45] Como bien sabe ya el lector, para una referencia completa puede consultar el *Pliego II: Planos* de la referencia [5].

[46] En las pgs. 33-34 expusimos los motivos por los que para implementar NWTJava se adoptó DOM.

La persistencia de los árboles se implementa de forma automática mediante las clases que proporciona el mismo *jdom*. El proceso seguido para guardar un árbol de la tabla en disco es el siguiente: *Datos*, con su método *save*, construye un objeto *Transformer* que se encarga de transformar el objeto *Document* en un fichero de texto. El nombre asignado al fichero es el contenido en el atributo *fichero* del objeto *GraphInfo* asociado al árbol.

El mencionado objeto *Transformer* se configura especificando el tipo de documento xml al que pertenece. Ésto es importante para que el xml generado haga referencia en su cláusula DOCTYPE a la DTD que se desarrolló para NWTJava v1.0.

Esta DTD define el lenguaje xml en que se representan los archivos de NWTJava, o lo que es lo mismo, contiene las reglas gramaticales a partir de las cuales, se deben construir los citados archivos de NWTJava.

Para la recuperación de un árbol almacenado en disco, *Datos*, en su método *iniciar*, hace uso de un objeto *DocumentBuilder*, el cual parsea el archivo de texto indicado, comprobando además, que el archivo contiene un documento xml válido según la DTD. Este objeto *DocumentBuilder* devolverá entonces un objeto *Document* que será almacenado en la tabla y que estará disponible para la aplicación.

La función de validación hace referencia al proceso mediante el cual se comprueba que un árbol existente en el Módulo de Gestión es gramaticalmente correcto. Como se puede deducir, la validación se reduce a comprobar que la estructura del árbol *jdom* concuerda con lo especificado en la DTD.

La clase *Datos*, volvemos a ella, se implementó como un repositorio de utilidades, pues además de las tres funciones ya mencionadas, proporciona un conjunto de métodos sueltos. Estos métodos son principalmente requeridos por *GraphManager* en calidad de clase mediadora con el bloque interfaz gráfica.

También es interesante contar el modo en que los árboles se crean en la clase *Datos*. Cuando un usuario crea un diagrama en la interfaz gráfica, éste sólo se almacenará en forma de árbol en la clase *Datos* cuando el usuario solicite la operación de validación. Esta operación, que pertenece al núcleo de la aplicación, necesita sin embargo ser accesible desde la interfaz. Para ello se dispone en la misma de un botón específico (cuya etiqueta es "*Validar programa*"). Ni que decir tiene que la función de la interfaz es tan solo ordenar la acción al objeto *GraphManager* que actúa como enlace y que a su vez invoca a *Datos*.

Una vez validado, para añadir el árbol a la tabla de programas en uso de *Datos*, el objeto *GraphManager* debe construir un array de *UserObject*, hecho que ya comentamos que se hacía mediante el método *resumeGrafo*.

Bien, hasta aquí hemos comentado conjuntamente las funciones primera y segunda de *Datos*, y por consiguiente, algunas de las funciones más importantes del Módulo de Gestión. Comentamos ahora que *Datos* tiene también la responsabilidad de gestionar el puntero de programa durante la ejecución de los árboles que se lleva a cabo en el Módulo de Ejecución. La relación "*Consulta*" que podemos observar en el diagrama de

la *Fig. 2.9* alude precisamente a ésto: la clase *Ejecutor* debe consultar a *Datos* para conocer la siguiente instrucción a ejecutar.

Nos detenemos aquí un instante para hacer una aclaración. Los términos "*puntero de programa*" e "*instrucción*" responden a los mismos conceptos usados por la computación tradicional, sólo que, en vez de apuntar a una dirección de memoria, apunta a un nodo del árbol que se está procesando. Ídem para "*instrucción*", pues no se refiere a ejecutar una línea de código sino a un nodo del diagrama.

Una vez finalizado el inciso, y retomado el anterior párrafo, decimos que es en este punto cuando surge la funcionalidad de la clase auxiliar *AdapterNode*. Se trata de una clase que envuelve a un nodo DOM y que proporciona métodos para recorrer el árbol. Es decir, los objetos *AdapterNode* proporcionan los movimientos que necesita la clase *Datos* para gestionar el puntero de programa mientras se ejecuta un árbol almacenado en su tabla.

Y con ésto podemos dar por concluido el análisis de este módulo. Pasamos al de ejecución.

Módulo de Ejecución.

Al igual que en el módulo anterior, destacan aquí también una clase principal y una auxiliar. La primera es *Ejecutor* y *Variable* la segunda. Para poder entender el funcionamiento de los procesos en que se involucra la clase Ejecutor, vamos a explicar primero en que consiste la clase Variable.

Los objetos de esta clase son creados y almacenados por un objeto *Ejecutor*, el cual los emplea durante la ejecución de un programa de usuario. Representan variables creadas por los usuarios, y son definidas únicamente por dos atributos públicos (esta clase no contiene ningún método a excepción del constructor):

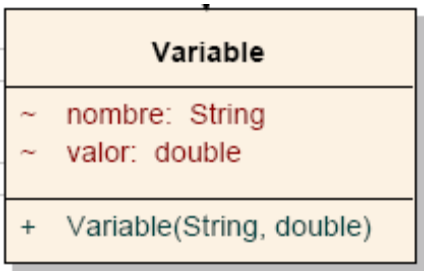


Fig. 2.11

Debido a lo descriptivo de sus nombres, no es necesaria mayor explicación.

Sin embargo, NWTJava v1.0 sólo puede operar con variables numéricas, y es aquí donde nos topamos con dos cuestiones referentes a la segunda versión. La primera es que los tipos de variables se reducen a uno (numérico). En NWTJava v2.0 dividiremos este tipo en dos (enteros y reales), y además añadiremos otros no numéricos (booleanos y cadenas de caracteres). De esta forma, se añade al aprendizaje de los principios de la programación la distinción entre tipos de variables. La segunda consiste en subsanar un error. Si nos fijamos en la *Fig. 2.11*, vemos que el atributo *valor* es de tipo *double*. No

obstante, a la hora de traducir el diagrama del usuario al lenguaje Java (recordemos: en el panel de la derecha, el diagrama de usuario se traduce a lenguaje real), este hecho no se contempla, y las variables, tengan el valor que tengan, siempre se inicializan a tipo *int*. Si por ejemplo, el usuario a definido la variable "*c = 6.8*", en el panel de traducción aparecerán las líneas de código:

```
int c;
c = 6.8;
```

Como es lógico, ésto no es correcto. Comentaremos en el *Capítulo 4: DISEÑO E IMPLEMENTACIÓN DE NWTJAVA v2.0* como acometimos estas cuestiones.

En cuanto a la clase *Ejecutor*, podemos decir que se encarga de, como no podía ser de otra forma, la ejecución de los programas de usuario. Puesto que el puntero de programa ya lo gestiona la clase *Datos* (como vimos en el Módulo de Gestión), la responsabilidad de esta clase se limita a:

a. Reconocer la instrucción que le llega a través de dicho puntero.

b. Realizar las acciones necesarias.

c. Pedir la siguiente instrucción en función del resultado de las mismas.

Exponemos estos puntos de modo resumido:

a. Las entradas que reciben los métodos de *Ejecutar* procedentes de la clase *Datos* son arrays de *Strings*, los cuales representan al nodo del árbol que corresponde ejecutar. El formato de estos arrays depende del tipo de nodo al que representa, por lo que el reconocimiento es unívoco e inmediato.

b. Las acciones que puede realizar *Ejecutor* son:

- Lectura de datos introducidos por el usuario y asignación de éstos a variables.
- Evaluación de expresiones aritméticas o lógicas.
- Escritura en pantalla.

Para la evaluación de las expresiones, se utilizan las clases pertenecientes a la librería externa llamada JEP ("*Java Expresion Parser*")[47].

[47] Su descripción y documentación se pueden encontrar en [16].

La decisión de utilizar esta librería se tomó a raíz de que el objeto *JEP* que proporciona, es un parser-evaluador muy práctico, que permite evaluar las expresiones aritméticas y lógicas que se emplean en esta aplicación.

Pero, ¿cómo se realiza una evaluación? Para ello *Ejecutor* requiere únicamente un objeto *String* con la expresión a evaluar. Como bien sabemos, toda expresión matemática se compone de operadores (+ sumas, - restas, * multiplicaciones…) y operandos.

Pues bien, no es necesario notificar al objeto *JEP* el tipo de operación, pues lo obtiene automáticamente mediante el parseo de la expresión y el análisis de los símbolos (+ - *…). Si la expresión es aritmética, el resultado de la misma es un *double*. Si por el contrario, es una expresión lógica, devuelve un *double* "*1.0*" si la solución es *true* o "*0.0*" si es *false*.

En cuanto a los operandos, en NWTJava pueden ser de dos formas: mediante variables (pi, edad, …) o mediante términos independientes (-8; 9.5; true…). El objeto *JEP* mantiene un registro interno de los pares variable-valor que emplea en la evaluación de expresiones que contengan variables. Los términos independientes también son reconocidos automáticamente por el parseador.

Para las otras dos funciones (la lectura y escritura en pantalla), *Ejecutor* contiene una referencia al objeto *NWTJava* de la aplicación sobre el que se puede ejecutar los siguientes métodos:

- *imprime*: que muestra en la ventana de ejecución el *String* que el usuario le pasa como argumento a la instrucción *WRITE* (análoga a la línea de código " *System.out.println("texto"); "* de Java).

- *read*: mediante un cuadro de texto, establece un dialogo entre la ejecución de un programa y el usuario que lo ejecuta. En este dialogo el usuario introduce un *String*.

Aparte, *Ejecutor* mantiene un repositorio con las variables que se están utilizando en el programa de usuario actual.

Por simplicidad, se decidió que no fuese necesaria la declaración de una variable antes de su uso. De esta forma, al encontrarse una referencia a una variable durante el transcurso de una ejecución, lo que hace *Ejecutor* es buscarla en su repositorio, y dependiendo del caso actuar, actuar en consecuencia:

- Si ya estaba declarada, obtiene su valor del repositorio.

- En caso contrario, la añade al repositorio con el valor por defecto de "0.0".

Ésto presenta un serio inconveniente, y es que el nombre de una variable debe ser único en todo el ámbito del programa. En el *Capítulo 4: DISEÑO E IMPLEMENTACIÓN DE NWTJAVA v2.0* solucionamos este problema, pudiendo así nombrar distintas variables con el mismo identificador.

Evidentemente ésto supone un cambio de filosofía: pasamos de un ámbito de variables global a uno local.

Realizamos así uno de los trabajos de mejora que se proponen en el *Capítulo 6: CONCLUSIONES Y TRABAJOS FUTUROS* del anterior proyecto y cumplimentamos también uno de los principales requisitos que se exigía para el desarrollo de NWTJava v2.0.

c. La información de qué instrucción debe ejecutarse a continuación se la proporciona *Datos* a *Ejecutor* a través del método *cp*. Aclaramos como funciona este proceso:

Ejecutor realiza las acciones necesarias para ejecutar el nodo indicado. Entonces, en función del resultado de esas acciones, *Ejecutor* decide el movimiento a realizar y le pasa al método *cp* una palabra clave que indica el movimiento que debe realizar el puntero de programa. . Ejemplos de estas palabras clave son: "*hijo*", según las flechas que indican el flujo, "*hijo*" corresponde al nodo que va a continuación del que se esta ejecutando; "*tio*", la siguiente instrucción corresponde al nodo que está actuando como hermano de su padre (hermano del nodo que precede al que se está ejecutando); etc. Una vez el puntero de programa se ha actualizado, *cp* se lo envía a *Ejecutor* obteniéndose así la siguiente instrucción.

Lo vemos con un ejemplo:

Ejecutamos el diagrama correspondiente a esta figura:

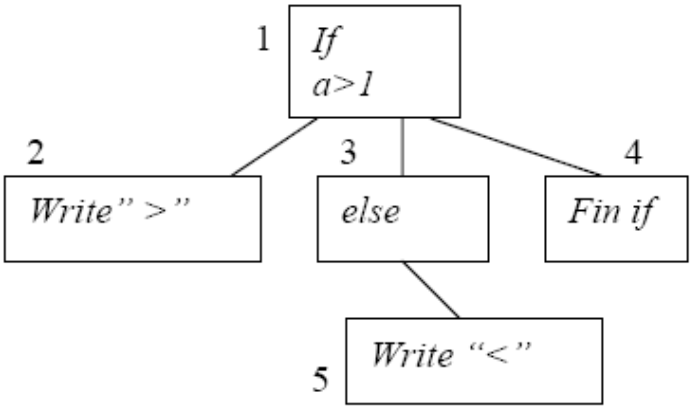


Fig. 2.12

Si la variable "*a*" tuviese un valor de 10, *Ejecutor* invocaría al método *cp* con la palabra clave "*hijo*", obteniendo así la estructura contenida en el nodo 2 como la siguiente a ejecutar. En el caso opuesto, con "*a*" valiendo 0, *Ejecutor* llamaría a *cp* con la palabra clave "*hijoderecho*" obteniendo directamente el nodo 5.

Una vez comentado el funcionamiento básico del módulo, queda aclarar que éste proporciona dos modos de funcionamiento:

- Ejecución directa
- Ejecución Paso a Paso.

Para ello, la clase *Ejecutor* presenta dos métodos distintos respectivamente: *ejecuta(String[] primeraLinea)*, y *ejecutaPaso(String[] linea)*.

En el primero de ellos, *Ejecutor* lanza un hilo en el que se recorre sin interrupción todo el árbol. Para cada ejecución se crea un objeto *Ejecutor* y se reinicia el puntero de programa.

En el segundo, tan solo se ejecuta la instrucción o línea que se le pasa como parámetro y termina. Para ejecutar la siguiente, el usuario debe generar de nuevo la acción "*ejecutar paso*" mediante su botón correspondiente en la interfaz gráfica. De esta forma, el alumno puede ver como se ejecuta el diagrama creado línea a línea con lo que ello supone para su aprendizaje: analizar el orden de ejecución de sus programas, ver donde se cometen determinados errores, etc.

A continuación pasamos al tercer y último módulo en que se divide el bloque núcleo de la aplicación.

Módulo de Traducción.

Este módulo, como su propio nombre indica, realiza la traducción de los diagramas a lenguaje de programación real. Como ya sabemos, a Java (en el *Capítulo 5: CONCLUSIONES Y TRABAJOS FUTUROS* propondremos ampliar el número de lenguajes reales a los que traducir).

Sus principales clases son *Traductor* y *Plugin* (ver *Fig. 2.9*)[48]. Lo realmente interesante de este módulo es la estructura que conforman entre sí estas clases, ya que proporcionan una enorme flexibilidad. Lo analizamos.

La clase *Traductor* está implementada como un *JPanel* de modo que el resultado de la traducción pueda ser mostrado en la interfaz de usuario de forma inmediata. Su principal característica es que contiene un objeto *StyledDocument*, que presenta la particularidad de poder almacenar texto de muy diferentes formatos en un mismo documento. Ésto a su vez, permitiría que la traducción a otros lenguajes de programación fuese relativamente sencilla.

Este objeto permite también que en el modo "*ejecución paso a paso*", el texto correspondiente a la línea de código Java perteneciente a la traducción, resalte sobre un fondo amarillo, indicando el punto por el que transcurre la ejecución del programa de usuario.

[48] En dicha figura también aparece la clase *Linea*, pero nosotros no haremos alusión alguna a ella pues su papel en NWTJava v2.0 resulta irrelevante. Si aún así desea conocer sus características e implementación, puede acudir a la referencia [5].

Por lo tanto, tenemos que *Traductor* se encarga de almacenar y gestionar el texto de la traducción. Sin embargo quien implementa la traducción en sí misma es la clase *Plugin*. El proceso es el siguiente: un objeto *Traductor* utiliza el objeto *Plugin* que se le pasa para generar el texto correspondiente al diagrama ejecutado.

El objeto *Plugin* se encarga pues de generar las palabras adecuadas y la indentación de las líneas para el caso de Java. Para traducir a un lenguaje distinto, bastaría con generar otra clase que implementase la indentación de las líneas y las palabras adecuadas a ese lenguaje. Ésto es precisamente lo que proporciona gran flexibilidad a la estructura de traducción.

Y con ésto damos por concluido el análisis sobre el diseño e implementación de NWTJava v1.0. Recordemos que el objetivo de esta sección no era el de analizar las ideas de las que surgieron su diseño, ni los problemas e inconvenientes a los que los autores tuvieron que hacer frente en su implementación. Más bien era el realizar un breve recorrido por el diseño e implementación de la primera versión de NWTJava, dando, eso si, alguna pequeña pincelada sobre su lógica, para que el posterior entendimiento de NWTJava v2.0 resulte mucho más sencillo y natural.

Capítulo 3:

Experimentación

3.1 OBJETIVOS 58
3.2 DESCRIPCIÓN 59
3.3 DESARROLLO 61
3.4 CONCLUSIONES 75

3.1 Objetivos

Si recordamos, ya en el *Capítulo 1: INTRODUCCIÓN A NWTJAVA v2.0* introdujimos el hecho de la realización de un experimento de campo. Tanto a tutor como autor nos pareció una idea muy positiva. Sin embargo, el enfoque al que fue dirigido dista en cierta medida de los experimentos académicos realizados hasta la fecha. En breve argumentaremos esta afirmación.

Fue en la enumeración de objetivos específicos de NWTJava v2.0 donde fue mencionado por primera vez. Si repasamos dicha enumeración,

- Análisis global de la herramienta NWTJava v1.0
  - Análisis teórico.
  - **Análisis práctico o *Experimentación*.**
- *Mejora* y potenciación de la misma ➔ implementación de NWTJava v2.0
  - A nivel de interfaz gráfica.
  - A nivel de núcleo.

  } *Herramienta de programación gráfica*
  - A nivel didáctico (*docencia*).

vemos como, en lugar de ejercer un papel de prueba final de la aplicación (o lo que podríamos denominar una prueba β), supone una fase más del análisis global de NWTJava v1.0. Este hecho nos lleva irremediablemente a pensar que, su objetivo primordial no era el de probar el funcionamiento de la herramienta, ni tan siquiera analizar si la comprensión de la disciplina de la programación mejoraba en calidad con NWTJava.

¿Y cuál es el motivo de que éstos no fuesen el propósito general de la experimentación? La explicación es sencilla. Se debe a que, a este respecto ya se han realizado suficientes pruebas. Desde las primeras experiencias del Dr. Raúl Ramírez Velarde hace ya casi veinte años hasta el último seminario impartido con NWTJava v1.0 en esta misma universidad.

No obstante, y en la siguiente clasificación de objetivos lo veremos, éste se incluyó como secundario.

- Objetivo Principal: estudio de posibles mejoras a incluir en NWTJava v2.0.

  Descripción: mediante la asistencia a un seminario, un grupo de "*usuarios tipo*"[49] ponen a prueba la herramienta. Haciéndoles notar que se trata de una

[49] Con la expresión "usuarios tipo" hacemos referencia a lo que ya denominamos en su momento "*potenciales usuarios*": alumnos de nuevo ingreso en ingenierías e informática con los que la herramienta alcanza su mayor potencial.

herramienta en fase de experimentación, solicitamos su colaboración para examinar su operatividad y funcionalidad, detectar posibles errores y aportar información sobre aquellos elementos o conceptos que echen en falta.

La herramienta fundamental de este objetivo era la *sugerencia.*

Y es este el objetivo principal, puesto que nuestro fin era el de mejorar y potenciar la herramienta.

- Objetivo Secundario: comprobar que la comprensión de la disciplina de la programación mejora en calidad con NWTJava.

  Descripción: los autores no pudimos resistir la tentación de volver a probar una vez más la eficacia de esta herramienta. De hecho, ello no suponía ningún esfuerzo extra, pues el seguimiento de este objetivo era análogo al seguimiento de los alumnos y sus resultados. Un seguimiento, por otra parte muy necesario para el adecuado aprendizaje de dichos alumnos.

  Las herramientas básicas de este objetivo fueron un *test inicial y otro final*, la resolución de *ejercicios* y el planteamiento de *dudas*.

- Surgió también otro objetivo a raíz de cómo plantear éste y otros futuros seminarios. Se analizaron errores cometidos en el pasado (con el fin de subsanarlos. Ej: cuestiones asociadas con la asistencia del alumnado) y se idearon nuevos planteamientos para hacerlos más útiles y llamativos. De este modo se buscaba adquirir mayor experiencia para la creación y desarrollo de futuros seminarios

3.2 Descripción

La evaluación técnica basada en un estudio de campo, consistió en impartir un seminario de introducción a la algoritmia básica. El ámbito fue el de una asignatura de programación de segundo cuatrimestre (primer curso) y sus características básicas las siguientes:

- Seminario muy elemental, orientado a alumnos para los que el mundo de la programación supusiese una enorme problemática. Se realizarían ejercicios básicos de bucles y estructuras condicionales.

- Centrado en aspectos generales de la algoritmia (y no particulares de la programación orientada a objeto o lenguajes concretos de programación).

- De carácter totalmente voluntario y al margen del horario de clases.

- Formato semipresencial, con un cierto número de horas presenciales y abundante comunicación vía web.

- Duración de 4 semanas. Ocho horas presenciales repartidas en 4 clases de 2 horas cada una y el resto no presencial.

- Mecánica del trabajo de clase: realización de una presentación inicial de la información por parte del profesor; turno de preguntas y dudas; realización de ejercicios prácticos relacionados con la explicación inicial.

- Mecánica del trabajo no presencial: envío de ejercicios por correo electrónico y recepción de sus correspondientes soluciones. Por supuesto se aceptarían todo tipo de dudas y sugerencias.

Un inciso merecen estos dos últimos puntos. Ni mucho menos fue tan estricto como dan a entender estas líneas. De hecho, una de las prioridades fue dotar al seminario de una enorme flexibilidad, atendiendo sobretodo a las necesidades del alumnado. De esta forma, en algunos días el planteamiento de dudas y su resolución ocupó más de un 50% del tiempo de clase. En otros se realizaron ejercicios enfocados a un aspecto muy concreto, e incluso se tuvieron en cuenta preguntas sobre Java y la resolución de ejercicios en este lenguaje. Este fue un factor que sin duda hizo mejorar notablemente la calidad del seminario como veremos reflejado tanto en el análisis objetivo de los resultados alcanzados por los alumnos, como en sus valoraciones personales.

Hablamos ahora de su puesta en marcha. El planteamiento inicial fue el de solicitar un número mínimo de 10 alumnos voluntarios, que cursaran el primer año de cualquiera de las tres especialidades técnicas de Ingeniería de Telecomunicaciones (Sonido e Imagen, Telemática y Sistemas de Telecomunicaciones). Además, para atraer la atención de los estudiantes y promover su participación, incluimos medidas como la de, además de utilizar NWTJava, resolver dudas y ejercicios de Java. También se pensó en una batería de medidas para casos de emergencia (en caso de que no se alcanzasen los 10 inscritos). Una de ellas era la concesión de 0´5 puntos extra para la asignatura de programación que impartía el Dr. José Jesús García Rueda.

Sin embargo, tuvo lugar algo sorprendente y del todo inesperado. Con la sola puesta en conocimiento del seminario en la especialidad de Sonido e Imagen, el número de solicitudes se disparó a 58 (recordemos que se propuso un plan de emergencia para el caso de que no se alcanzase el número mínimo de 10 solicitudes). Resultó ser un número del todo inviable, tanto por razones técnicas (espacio físico de un laboratorio) como educativas (la atención sobre el alumnado sería de menor calidad en comparación con un reducido grupo de estudiantes).

Como es lógico, la estrategia de promoción dio un giro de 180 grados. No sólo se dejó de fomentar su participación, sino que además nos vimos obligados a exigir condiciones de permanencia. Por una parte, se decidió no incluir a las otras dos titulaciones en la puesta en marcha del experimento. Nos centraríamos así en la de Sonido e Imagen. Por otra parte, se advirtió de que los alumnos asistentes deberían identificarse tanto en la entrega de ejercicios como en la resolución de los tests propuestos. Con esta medida (no planteada en un principió) matábamos dos pájaros de un tiro: no solo establecíamos unas condiciones de participación sino que además nos permitiría ofrecer una ayuda personalizada al alumno. E incluso, de modo análogo al premio de 0´5 puntos en la asignatura, se pensó en castigar con -0`5 puntos la no entrega del test final.

Finalmente no fue necesario llegar a tales extremos, pues entre la reducción del número de solicitudes y el firme compromiso de los creadores a atender a cuantos alumnos fuese necesario, se alcanzó un equilibrio.

El número final de inscripciones fue de 30 alumnos, dos de los cuales solicitaron realizar el curso total y exclusivamente vía Internet, por la coincidencia de horarios con otra asignatura. Con el tutor y autor de este proyecto, el guión del seminario ya ideado, y estos 30 voluntarios, el experimento quedaba totalmente constituido.

Nacía así TSIOCA- Seminario de algoritmia básica[50].

3.3 Desarrollo

El seminario tuvo lugar en el laboratorio 7.2J01 y su horario de clases presenciales quedó establecido en:

Lunes de 13:00 a 15:00.

El trabajo no presencial se distribuyó de la siguiente forma:

Martes: publicación del correspondiente ejercicio.
Plazo de entrega de la solución hasta las 24:00 del miércoles.
Jueves: publicación de correspondiente ejercicio.
Plazo de entrega de la solución hasta las 24:00 del viernes.

El guión inicial a seguir en el experimento, en cuanto al trabajo de los alumnos, se puede sintetizar en el siguiente esquema:

- TEST INICIAL
  - Ejemplo explicativo
  - Problemas
  - Cuestionario
  - Corrección
- EJERCICIOS
  - Publicación de enunciados
  - Envío de soluciones, dudas y sugerencias
  - Análisis de soluciones, dudas y sugerencias
- TEST FINAL
  - Ejemplo explicativo
  - Problemas
  - Cuestionario
  - Corrección

[50] TSIOCA corresponde a las siglas de la titulación y asignatura que sirvieron como marco del seminario: Ingeniería Técnica en Sonido e Imagen, asignatura Organización de Contenidos Audiovisuales.

Cada punto citado en la anterior esquematización será explicado según lo alcancemos en la siguiente explicación.

Para hacer al lector partícipe de la dinámica seguida durante el desarrollo de TSIOCA, nos parece muy adecuado el seguir los registros y datos tomados in situ, los cuales utilizábamos a modo de "cuaderno de bitácora". De esta forma, se podrá apreciar muy claramente la evolución tanto del seminario como de los alumnos.

Y sin más dilación, comenzamos con lo que es el objetivo fundamental de este apartado, el desarrollo del experimento:

Día 0

La asistencia a clase fue de 28 alumnos (recordemos que 2 más seguirían el seminario vía web).

En este primer día tuvo lugar la presentación del seminario, y por supuesto, de los creadores y directores del mismo. En dicha presentación, de aproximadamente una hora de duración, se explicaron aspectos tales como las condiciones de permanencia, la mecánica de funcionamiento y el rol de cada uno de los integrantes, así como las expectativas generadas y los objetivos marcados. Como no, también se explicó el proceso de instalación y primeras directrices sobre NWTJava.

Y fue en la segunda hora donde hacemos referencia al primer punto del anterior esquema: el *test inicial*.

El test, de unos 50 minutos de duración, y reutilizado del proyecto anterior, constaba en su primera parte de una serie de instrucciones indicativas acompañadas de un ejemplo ilustrativo de resolución de ejercicios. Por supuesto, la resolución de dicho ejemplo no correspondía con ningún lenguaje concreto de programación, ni tan siquiera a la usanza de NWTJava. La solución estaba expresada en forma de pseudocódigo, dejando libertad prácticamente absoluta al alumno para plantear su solución.

Después presentaba una batería de ejercicios o pequeños problemas cuyo objetivo era el de evaluar la capacidad de los alumnos para abordar problemas simples desde el punto de vista de la algoritmia[51].

Para finalizar, se les propuso rellenar un cuestionario. En él podrían reflejar, de forma subjetiva e individual, las circunstancias en que se encontraban respecto al ámbito de la programación. Aunque no estaba explícito, las preguntas estaban divididas en bloques cuya función era la de evaluar los siguientes aspectos:

- Nivel de programación objetivo: donde se les preguntó por su experiencia y convocatorias relacionadas con la programación.

[51] Los criterios de evaluación de los ejercicios no son competencia de este proyecto, pues ya quedaron fijados en el anterior experimento llevado a cabo por Patricia Fernández Garrido. No obstante, pueden ser consultados en la referencia [5].

- Nivel de programación subjetivo: para analizar lo que ellos pensaban respecto a su nivel de programación.

- Paradigma del seminario: su objetivo era el de comprobar si eran capaces de distinguir entre los conceptos de "*programar*" y "*utilizar un lenguaje de programación*". Considerábamos que era primordial y lo primero que debían entender para avanzar en el seminario.

- Por último, un bloque para que comentasen sus principales problemas con la programación y cómo lo solucionarían desde el punto de vista de la enseñanza (mediante una mayor cantidad de ejercicios resueltos en clase, análisis de un mayor número de ejemplos, una explicación más detallada de la teoría…).

Hay que comentar también el hecho de que, la entrega del test suponía el último requisito para cumplimentar la matrícula de inscripción. Es decir, era una de las condiciones de permanencia anteriormente citadas. La entrega del test, que era totalmente voluntaria, suponía el firme compromiso, por parte del alumno, a asistir al máximo porcentaje posible de horas del seminario.

Muy destacable es el hecho de que, a pesar de que una vez realizado el test la entrega era voluntaria, el porcentaje de éstas fue del 100%.

Una vez explicado el test y su funcionamiento, sería muy útil recurrir al *APÉNDICE B* para ver las partes y preguntas de las que constaba, y poder así entender en toda su dimensión los siguientes comentarios.

Pues bien, realizando la suma total de las tenemos 55´5 puntos. Volvemos a repetir que el sistema de puntuaciones y evaluación se escapa de los objetivos de este análisis (consultarlos en [5]).

Lo que sí es importante destacar es que el planteamiento de los ejercicios no es ni mucho menos aleatorio. De hecho están construidos de forma que permitan dirimir las lagunas que presenta cada uno de los alumnos (utilización de bucles, uso de condiciones,…).

En el siguiente cuadro vemos los criterios de evaluación:

Nivel	Puntos	Conclusión Aproximada
1	Hasta 7	Reconoce y sabe construir correctamente sentencias de decisión y sus condiciones asociadas. También conoce y sabe aplicar iteraciones simples.
2	Hasta 13	Además: Utiliza la iteración de un modo más complejo: es capaz de utilizar el índice iterativo no sólo para contar iteraciones, sino para implicarlo en el resultado, y además sabe que puede incrementarlo de cualquier modo, y no sólo mediante la suma. Esto indica que percibe patrones más complejos en el problema.
3	Hasta 23	Además: Conoce el fundamento de las estructuras repetitivas: estructuras con un número indeterminado de repeticiones. Distingue entre sí las estructuras repetitivas "while" y "repeat/until".
4	Hasta 33.5	Además: Secuencializa procedimientos sencillos en problemas que requieren que los datos se analicen más de una vez. Identifica las variables de acumulación y adicionales que debe utilizar, lo que indica que ha comprendido las distintas fases en las que debe dividir el algoritmo.
5	Hasta 55.5	Además: Implementa algoritmos más complejos que requieren la solución de más de un problema al mismo tiempo. Es capaz de entender el problema secundario como independiente del principal, definirlo y resolverlo aparte, y utilizarlo como herramienta auxiliar para resolver el principal. Además es capaz de secuencializar algoritmos que requieren el anidamiento de varios bucles y de controlar y utilizar correctamente los índices anidados.

Fig. 3.1

Los resultados obtenidos fueron los siguientes:

Nivel	Nº de alumnos
1	1
2	16
3	11
4	2
5	0

Fig. 3.2

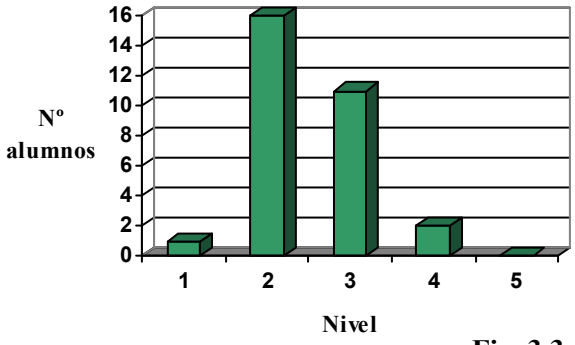


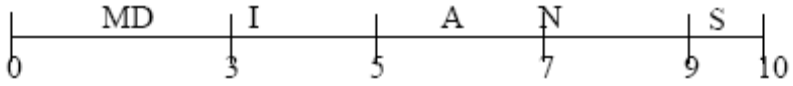
Fig. 3.3

Con un rápido vistazo a los datos comprobamos que más del 50% del alumnado no alcanza el *Nivel 3*. Eso implica que presentan importantes deficiencias a la hora de identificar las estructuras iterativas que resuelven el problema planteado. Este hecho, a priori resulta impactante, pues hay que tener en cuenta que eran alumnos que ya habían cursado una asignatura cuatrimestral de programación entera, más las empezadas en el segundo cuatrimestre. Deberían, sin una gran dificultad, haber resuelto la totalidad de los ejercicios pertenecientes al *Nivel 4* y en gran medida los del *Nivel 5*. Como curiosidad, cabe destacar que sólo un alumno fue capaz de contestar correctamente el problema 9, y ninguno los 10, 11 y 12.

Entre los errores más destacables encontramos:

- Incorrecta utilización de las estructuras condicionales *IF/ELSE*. No cometen fallos cuando la resolución de un problema exige únicamente una instrucción *IF*, pero si cuando deben combinarla con *ELSE*. Ésto lo vemos muy claramente en el ejercicio 8, donde al enunciado

  Devolver por pantalla la nota "alfabética" correspondiente a una nota numérica que proporcione el usuario, con las siguientes categorías:
  (4´5 puntos).



  muchos respondieron de la siguiente forma:

```
Introducir (nota)

Si (nota<3)
        Mostrar(MD)
Si (3≤nota<5)
        Mostrar(I)
        Si (5≤nota<7)
        Mostrar(A)
        Si (7≤nota<9)
        Mostrar(N)
Si (9≤nota≤10)
        Mostrar(S)

FIN
```

- No identifican correctamente la estructura de bucle adecuada. Es decir, presentan enormes dificultades a la hora de distinguir entre *FOR, WHILE* y *REPEAT/UNTIL*. Además, típicamente la estructura que utilizan para la gran mayoría de los casos es el *FOR*, mientras que la distinción entre *WHILE* y *REPEAT/UNTIL* es totalmente inexistente.

  En cuanto a la masiva utilización de la instrucción FOR, resulta muy curiosa la solución que muchos de los alumnos adoptaron para resolver el ejercicio 5. La vemos:

  Imprimir por pantalla la frase "hola" para siempre (infinitas veces).

```
Para (x=0 , x>-1 , x++)
        Mostrar(hola)
```

  Los problemas existentes con los bucles *WHILE* y *REPEAT/UNTIL* se hacen muy evidentes en los problemas 6 y 7, que no reproduciremos por no incidir más en esta problemática.

- Como último gran error destacable podemos resaltar la dificultad que presentan en la enunciación de las condiciones. Detectamos aquí dos tipos de problemas. El primero es la incorrecta utilización de los símbolos necesarios: *&&; ||; ≤; >; etc.* El segundo es la mala interpretación del enunciado y de esos mismo símbolos, ejemplo:

  Escribir por pantalla la secuencia de caracteres: 7 3 5 7 3 5 7 3 5 7 3 5 7 3 5.

  Si analizamos la secuencia, vemos como la serie 735 se repite en 6 ocasiones. Pues bien, errores como el ahora mostrado se repitieron en multitud de ocasiones:

  Para (i=1 , i<6 , i++)
  Mostrar(735)

  Como podemos observar, la condición no es correcta pues el bucle únicamente se ejecuta en 5 ocasiones y no 6 como es requerido.

A raíz de estos hechos y de los resultados obtenidos, podemos extraer las siguientes conclusiones:

- Los alumnos presentan enormes dificultades en la resolución de los problemas mediante algoritmos.
- Tienden a utilizar, de forma mecánica, una única estructura de control. La que más sencilla e intuitiva les resulta (típicamente el *FOR*).
- Tardan bastante tiempo en interiorizar y aprender a utilizar los distintos tipos de condiciones existentes. Este hecho se agrava aún más cuando dicha condición debe ser de tipo lógico (*AND*, *OR*, *NOR*, …)

Como conclusión general, extraemos la que hemos venido comentando a lo largo de todo este proyecto: es fundamental y estrictamente necesario un refuerzo en la enseñanza de la algoritmia.

Pasamos ahora a analizar las respuestas del cuestionario, las cuales resultaron ser mucho más sorprendentes y esclarecedoras de lo previsto en un primer momento. Podemos decir que los resultados fueron muy próximos a lo esperado, sin embargo resultaron ser mucho más contundentes. Un hecho que es de agradecer es la sinceridad con la que se expresaron los alumnos. Podemos así destacar las siguientes observaciones:

- Si recordamos, en páginas anteriores comentamos que las preguntas del cuestionario respondían a una estructura de bloques. En el primero de ellos hacíamos alusión a la posible experiencia individual de cada alumno (asignaturas cursadas, número de exámenes presentados, etc.). Pues bien, es digno de mención el siguiente dato:

De entre todos los alumnos asistentes, si sumamos el número de exámenes realizados por todos y cada uno de ellos, relacionados con el ámbito de la programación, la cifra total asciende a 103. De todos ellos, los aprobados son 15. Es decir, el porcentaje de exámenes superados alcanza el 14´56%.

Y lo que es aún más sorprendente, los alumnos reconocen en un 86% de los casos que con su actual nivel de conocimientos, es del todo imposible aprobar los que les restan.

- En cuanto al segundo bloque de preguntas, correspondientes al nivel de programación desde un punto de vista subjetivo y personal, vemos que ante la pregunta número 4 "**¿Qué te ha parecido el *Test inicial*?"**, la respuesta más repetida ha sido: complicado o muy complicado. Y ante la número 5 "**Si un -1- corresponde a -*pienso que estoy totalmente perdido*- y un -5- a -*todo lo que se ha dado en clase lo manejo sin dificultades*-, ¿qué nota te pondrías?**", la nota media de toda la clase, impuesta por ellos mismos, es de 2. Muy significativo es que los alumnos con más confianza en sus conocimientos no se autocalificaron con una nota superior a la de 3.

- Otra sorpresa más nos llevamos con la cuestión 6, donde preguntamos si tienen clara la diferencia entre los conceptos de "*programar*" y "*utilizar un lenguaje de programación*". Solamente uno de cada dos alumnos contestó correctamente. Lo que nos conduce de nuevo a la opinión de que la utilización de un lenguaje de computación como instrumento de docencia en la algoritmia no está exenta de riesgos. Las respuestas en esta pregunta fueron tan dispares como las siguientes:

  "Programar es resolver un problema y utilizar un lenguaje es la forma que toma esa resolución para que la entienda la máquina". (Respuesta correcta).

  "No lo tengo muy claro. Supongo que programar es realizar programas".

- Para acabar, el último bloque tenía como objetivo el que comentasen sus principales problemas con la programación y cómo los solucionarían desde el punto de vista de la enseñanza. En sus correspondientes contestaciones resaltan la necesidad de establecer unas sólidas bases, aunque como estudiantes que son, hacen hincapié que la requieren para aprobar sus exámenes. En cuanto a sus sugerencias, repiten constantemente la necesidad de que el correspondiente profesor resuelva una mayor cantidad de ejemplos y ejercicios, tanto en pizarra como en pc.

  También destaca por su abundante presencia comentarios del tipo:

  ¿Qué piensas sobre los métodos pedagógicos (o forma de enseñar) con los que se imparten las clases de programación? ¿Cómo los mejorarías?

  "Intentando que desde el principio comprendamos las cosas, sin partir de la suposición de que ya sabemos cosas."

Y con el análisis de *test inicial* y su correspondiente *cuestionario* acabamos con lo que fue el *Día 0* del seminario, su primer día.

Semana 0

Con *Semana 0* nos referimos a la parte del seminario que aconteció desde la primera clase presencial hasta la siguiente, *Día 1*. El trabajo aquí realizado fue exclusivamente vía mail por parte de los profesores, y tanto vía mail como desde casa por parte de los alumnos. Describimos a continuación lo acaecido.

Tal y como contamos al principio de este apartado, el primer punto del trabajo no presencial consistía en la publicación del enunciado de un ejercicio. Pues bien, así fue. Tras la clase del lunes, el martes a primera hora de la mañana se envió a las correspondientes cuentas de correo el enunciado del primer ejercicio previsto en el experimento. Por ser la primera vez en que tomarían contacto con la herramienta, y además por sí solos, el ejercicio fue acompañado de un pequeño guión o "manual de instrucciones" con los primeros pasos a seguir, además de un decálogo con los puntos más importantes vistos el día anterior, en el *Día 0*.

El enunciado de este primer ejercicio puede verse en el *APÉNDICE D* como *Ejercicio 1*.

Si lo observamos, vemos como en el enunciado se añadió una parte final en la que se ofrecen una serie de pistas o ayudas. El objetivo era orientar los primeros pasos y el modo en que debían ir pensando los alumnos a la hora de resolver algoritmos. Tras una serie de ejercicios, y a la vista de los resultados obtenidos, todos ellos muy positivos, se decidió no volver a incluir más pistas.

A parte de solicitar sus resoluciones, también se les motivó y alentó para que realizaran el máximo número de preguntas y sugerencias posibles, con el fin de resolverles la mayor cantidad de dudas y crear así una realimentación entre alumnos y profesorado, considerada muy necesaria para el óptimo desarrollo del seminario (y como no, del experimento). Esta medida tuvo por parte del alumnado una gran acogida, fue muy bien aceptada, y sobretodo y más importante, muy aprovechada.

Pues bien, los primeros resultados fueron espectaculares. Tan solo dos horas después de publicar el enunciado se recibió la primera solución, y además, correcta. Y al final del plazo de entrega (miércoles a las 24:00), el porcentaje de entregas fue del 96% (o lo que es lo mismo, sólo un alumno no hizo entrega de su correspondiente solución).

Los datos relacionados con este ejercicio son los siguientes:

	Soluc. Correctas*	**Soluc. incorrectas**	**% Soluc. correctas**
Ejercicio 1	14	15	46%

* Con "*Soluciones Correctas*" nos referimos a aquellos diagramas capaces de realizar lo solicitado por el enunciado.

De estos resultados podemos extraer numerosas conclusiones. La negativa es que ratifica la conclusión alcanzada con el *test inicial*: el nivel de programación es mucho más bajo del deseado. Si nos concentramos por un momento en el enunciado, podemos comprobar que éste es harto sencillo. Su resolución se alcanza con un único y simple *FOR* que debe ser ejecutado en cuatro ocasiones. No obstante, el resto de conclusiones propiciaban una visión optimista. La participación fue enorme, mucho mayor de la esperada, la realimentación muy positiva, pues se recibieron 14 correos cuyo contenido eran dudas y sugerencias (todos ellos tuvieron su correspondiente contestación), y un dato muy importante a destacar: "*No me adapto a NWTJava*", solamente éste y otros 2 correos más hacían alusión al manejo de la aplicación, lo cual hace pensar que el objetivo de proporcionar a NWTJava una alta y sencilla manejabilidad esta prácticamente conseguido.

La duda más común al respecto de este ejercicio fue cómo diferenciar las estructuras de control (*FOR, WHILE* y *REPEAT/UNTIL*) y cuando usar cada una de ellas. Ante la insistencia de esta duda remitimos un correo general donde incluimos una breve explicación al respecto e indicamos que la duda sería totalmente resuelta en la siguiente clase presencial (*Día 1*).

Como no, el jueves de esta misma semana se publicó el enunciado del segundo ejercicio (*APÉNDICE D, Ejercicio 2*).

La mecánica para el tratamiento de este ejercicio fue la misma que para el primero. Analizamos los datos:

	Soluc. correctas	**Soluc. incorrectas**	**% Soluc. correctas**
Ejercicio 2	15	12	55´5%

Vemos que aunque el número de entregas bajó ligeramente, el porcentaje de soluciones correctas aumentó. Ésto es una constante que se dará a lo largo de todos los ejercicios.

Curioso fue el hecho de que muchos de los alumnos, tras resolver el correspondiente ejercicio en NWTJava, lo intentaron con Java. Surgieron así una gran cantidad de emails con duda. Ésto supuso un motivo de satisfacción para los creadores del seminario, pues no hacía más que demostrar la buena disposición y la alta motivación con la que los alumnos estaban asumiendo el reto.

Y con ello llegamos a la segunda clase presencial.

Día 1

La asistencia fue de 23 alumnos (recordemos que 2 más seguían el seminario vía Internet).

De las dos horas que constaba la clase, la primera se dedicó a la explicación y resolución de las dudas surgidas en los ejercicios 1 y 2. Ésto derivó en la explicación de las estructuras de control. Se hizo hincapié en su definición, en los factores que

diferencian su uso, y también en la declaración de sus condiciones. Si recordamos, éste último factor fue una de las mayores fuentes de errores en el *test inicial*.

La segunda hora se empleó en la resolución práctica de los ejercicios de la *Semana 0*. Con ello se buscó reforzar y asentar los conocimientos adquiridos hasta el momento, mejorar en el manejo de NWTJava y explicar toda su funcionalidad (cada botón, cada opción…), con el fin de que los alumnos, conociendo toda su capacidad, fuesen capaces de detectar errores y sugerir mejoras.

Semana 1

La dinámica de la *Semana 1* fue análoga a la de la *Semana 0*. Únicamente mencionar que estuvo compuesta por una semana lectiva y otra no lectiva, correspondiente a Semana Santa. Por este motivo, fueron cuatro los ejercicios enviados. Correspondieron a los números 3, 4, 5 y 6, cuyos enunciados podemos encontrar en el *APÉNDICE D* de este mismo proyecto.

Reseñar también el hecho de que la tendencia (pequeña reducción del número de entregas y aumento del porcentaje de soluciones correctas) se mantiene. Los datos correspondientes a cada uno de los ejercicios los encontraremos en una tabla al final de este apartado. Su función será la de ver el desarrollo seguido por el alumnado en cuestión de resultados y aprendizaje.

Por último destacar el salto cualitativo que presentaron las soluciones del ejercicio 3 respecto a las del número 2. Si bien el porcentaje de soluciones correctas en el problema 2 era del 55´5%, para el número 3 fueron del 81´8%. Un buen indicador que nos aportó la información de que el sistema utilizado y NWTJava estaban dando sus frutos.

Día 2

Asistieron 18 personas, y bajo demanda general se resolvieron en la primera hora un ejercicio de examen perteneciente a la asignatura del primer cuatrimestre Representación de Datos y Aplicaciones (RDA) y las dudas surgidas a raíz de los ejercicios 5 y 6 del seminario.

En la segunda se trabajó con los ejercicios *embalse* y *proceso metalúrgico*[52].

Semana 2

Por parte de los estudiantes que seguían todavía el seminario, la actividad era aún incesante. La abundancia de correos sobre aclaraciones y dudas, y por consiguiente la realimentación alumnos-profesores, no decaía. Uno de los motivos de ésto fue el ejercicio 7, que merece especial atención por la enorme dificultad que supuso su resolución. Nos remitimos al *APÉNDICE D* como *Ejercicio 7* para leer su enunciado.

[52] Fueron creados para este seminario una serie de ejercicios auxiliares cuya denominación ya no siguió una numeración, sino que correspondía con algún elemento citado en el enunciado. Estos ejercicios son "desfibrilador", "embalse", "primitiva" y "proceso metalúrgico". También los podemos encontrar, como el resto de ejercicios, en el *APÉNDICE D* de este proyecto.

A pesar de lo aparente que nos pueda resultar la solución, aún sin el uso de arrays, únicamente 3 personas de 16 entregaron una correcta resolución. El motivo por el que argumentaban no alcanzar la solución adecuada era precisamente la falta de arrays.

Incluso llegaron a asegurar que sin ellos, este problema no tenía solución. Resultó muy importante llegar a este punto, pues quedó patente la dependencia que muestran los alumnos principiantes ante estructuras conocidas, pero de las que, sin embargo, no entienden su fundamento. Este es el caso de los arrays, pero se puede también extrapolar al de los bucles *FOR, WHILE* y *REPEAT/UNTIL* en los primeros momentos del aprendizaje (o en nuestro caso, en las primeras clases de nuestro seminario).

Ni que decir tiene, que el uso de arrays por parte de NWTJava, fue la sugerencia más repetida a lo largo del seminario.

Aparte del ejercicio 7, también fue publicado el denominado *desfibrilador*. El comentario más generalizado acerca de este nuevo ejercicio fue el de que era muy fácil en comparación al problema 7. Una buena noticia por tres motivos: la primera que todas las soluciones entregadas fueron correctas, la segunda que hacía escasamente tres semanas no fueron capaces de resolver un ejercicio muy similar en el test inicial, y la tercera, que a pesar de ser un ejercicio que implicaba una cierta dificultad, ninguno mostró la más mínima duda en su resolución.

Día 3

Y finalmente llegamos al cuarto y último día de clase presencial. Como ya sabían los asistentes, a segunda hora serían sometidos al *test final*. La asistencia al mismo fue de 12 alumnos, un muy buen dato si tenemos en cuenta que en el anterior seminario de NWTJava, la asistencia al mismo fue de tan solo 2 personas.

En la primera hora se resolvió otro problema de examen de la asignatura RDA y por supuesto el ejercicio 7, cuya solución resultó tan problemática. Fue en la segunda hora cuando tuvo lugar el *test final*, acompañado de su correspondiente *cuestionario final* (*APÉNDICE C*).

La puntuación total del test, como se puede comprobar, es de 38,5 puntos[53], y la correlación entre puntos y niveles la siguiente:

Nivel	Puntos
3	Hasta 10
4	Hasta 24´5
5	Hasta 38´5

Fig. 3.4

Si recordamos lo que implicaba cada nivel:

[53] Como ya dijimos, el sistema de puntuaciones y evaluación se escapa de nuestros objetivos (consultarlos en [5]).

3	Además: Conoce el fundamento de las estructuras repetitivas: estructuras con un número indeterminado de repeticiones. Distingue entre sí las estructuras repetitivas "while" y "repeat/until".
4	Además: Secuencializa procedimientos sencillos en problemas que requieren que los datos se analicen más de una vez. Identifica las variables de acumulación y adicionales que debe utilizar, lo que indica que ha comprendido las distintas fases en las que debe dividir el algoritmo.
5	Además: Implementa algoritmos más complejos que requieren la solución de más de un problema al mismo tiempo. Es capaz de entender el problema secundario como independiente del principal, definirlo y resolverlo aparte, y utilizarlo como herramienta auxiliar para resolver el principal. Además es capaz de secuencializar algoritmos que requieren el anidamiento de varios bucles y de controlar y utilizar correctamente los índices anidados.

Fig. 3.5

Los resultados obtenidos fueron:

Nivel	Nº de alumnos
3	3
4	7
5	2

Fig. 3.6

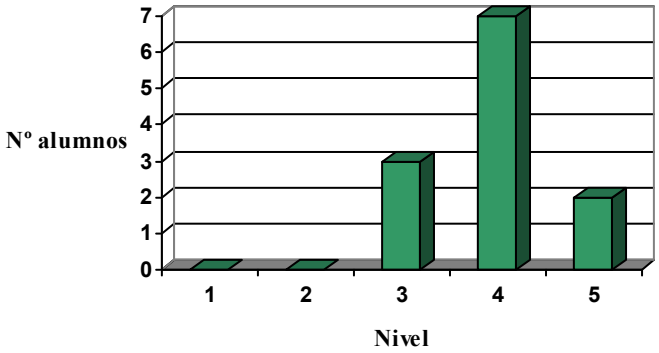


Fig. 3.7

Resulta sorprendente, e incluso gratificante, ver como a pesar de que no todas sus soluciones fueron correctas, si lo eran en gran medida las estructuras escogidas para resolverlos. Es decir, ya no utilizaban masivamente el bucle *FOR*, sino que eran capaces de distinguir los casos en los que era adecuado otro tipo de estructura iterativa. La misma línea de aprendizaje tuvo lugar en el caso de las estructuras condicionales *IF/ELSE*. En este sentido todos asimilaron los conceptos necesarios, pues desde el uso que les dieron en el *test inicial* a este *test final*, existe un enorme salto cualitativo.

Desde el punto de vista negativo, también es fundamental reseñar un cierto número de factores o hechos. Comentar que ningún alumno fue capaz de realizar correctamente la totalidad de los ejercicios. De hecho, la puntuación máxima fue de 30´5, puntuación que dista en gran medida de la máxima posible.

Pasamos a continuación a analizar los comentarios más destacables del cuestionario, donde veremos también el resto de puntos negativos.

Al igual que el cuestionario inicial, éste también divide sus preguntas en bloques muy específicos. En este caso, el primer bloque (compuesto por las preguntas 1, 2 y 3) responde a la temática de si han encontrado el seminario positivo y útil para su aprendizaje.

Prácticamente el 100% de los encuestados afirman que sí. Hacen incluso observaciones indicando que sobretodo les ha servido para asentar una sólida base. Más dubitativos se encuentran a la hora de opinar sobre si sus conocimientos han aumentado. Aunque comentan que sí, que éstos han aumentado, también reconocen que no mucho, pues tras prácticamente un año de programación (dos o más en algunos casos), lo visto les ha servido más como un repaso.

Un ejemplo muy claro y sincero, que además refleja el sentir de la mayoría de asistentes, es el de las siguientes respuestas ofrecidas por el alumno A.C. Las transcribimos literalmente:

1. **¿Crees que tras el seminario y la utilización de NWTJava, tus conocimientos sobre programación han aumentado?**

"Sí, he aprendido bastantes cosas. No quiere decir que sepa programar, pero por lo menos ya tengo la sensación de tener una base. Algo es algo."

2. **Entonces, ¿crees que ha sido positivo para ti el que lo hayas realizado? (SI, NO, ¿por qué?).**

"Sí, porque por lo menos tengo una base, algo sobre lo que aprender y empezar a entender esto de la programación."

3. **Piensa en ti antes del seminario, y ahora. ¿Crees que podrías resolver un mayor número de ejemplos, ejercicios y problemas de otras asignaturas? ¿Lo has intentado? Si la respuesta es SI, ¿qué resultados has obtenido?**

"Sí, podría resolver más ejemplos, pero es algo fácil ya que antes no era capaz de resolver ninguno. He resuelto algunos, y la mayoría soy capaz de plantearlos bien."

En cuanto al segundo bloque, el objetivo era ver si asociaban la aplicación NWTJava con el aprendizaje de los conceptos de programación. En este caso, el bloque también esta compuesto por tres preguntas (4, 5 y 6).

Pues bien, las respuestas a este bloque fueron todas afirmativas. Continuamente se hizo alusión a que con NWTJava, la visión sobre las estructuras era muy intuitiva, muy fácil de entender. Quizás sus dudas llegaban con el miedo de extrapolar estos conocimientos a Java. De hecho la mayor crítica a este seminario por parte de los alumnos deriva de este bloque de preguntas. Ante la cuestión número 6, el alumno que responde a las siglas A.M. contesta:

6. En líneas generales, ¿te ha resultado útil (NWTJava) para el aprendizaje?

"Sí, pero creo que se debería utilizar antes de empezar realmente con Java, en el primer cuatrimestre. De esta forma sabríamos lo que es programar."

Con la principal crítica nos referimos a la de que debería haberse impartido en el primer cuatrimestre. Argumentan dos razones para ello: la primera, que el aprender los conceptos básicos de la programación debería ser la primera fase del aprendizaje, antes de introducirse en el lenguaje Java. La segunda, que muchas de las cosas vistas, eran ya conocidas, aunque este segundo motivo resulta bastante discutible a raíz de lo observado en el *test inicial*.

El tercer bloque se centraba ya en las prestaciones de NWTJava. Nos interesaba, de hecho era fundamental, su opinión acerca de las características de la herramienta, como la interfaz gráfica o su manejabilidad.

Podemos decir que ambos elementos fueron muy aceptados. Como mayor virtud destacaron su sencillez (la cual, si recordamos, era uno de los objetivos esenciales de la aplicación). Mencionaron también pequeños fallos en alguna de sus funciones, que con total compresión, achacaron a ser ésta la primera versión del programa.

Como cuarto y último bloque, uno de bastas dimensiones. Hasta cinco cuestiones donde hacíamos alusión a un análisis general de NWTJava y la visión global sobre el seminario y profesores.

Respecto a la primera parte, análisis general de NWTJava, planteamos la pregunta 9:

1. ¿Puedes elaborar una pequeña lista de ventajas e inconvenientes de NWTJava?

Inconvenientes citados fueron: ante la ejecución de bucles infinitos, el botón de "terminar ejecución" (o "botón del pánico") no funciona bien, de hecho la aplicación se cuelga; la necesidad de tener que declarar variables sin ser éstas necesarias[54]; no poder elegir el tipo de las variables utilizadas; algunos mensajes de error no eran nada aclaratorios; se echa en falta el manejo de arrays…

Las ventajas más citadas fueron: una interfaz muy clara; fácil manejo; traducción de los algoritmos a lenguaje Java; aclara en gran medida como se debe estructurar el código de un algoritmo…

Si atendemos a la naturaleza de los inconvenientes, podemos observar como la gran mayoría corresponden a fallos técnicos y no de concepto. Por supuesto, todos los errores detectados a éste respecto han sido solucionados en la nueva versión de la aplicación. Por si fuera poco, los fallos mencionados por los alumnos a nivel teórico, como puedan ser la ausencia de tipos de variables o manejo de arrays, son algunos de los requisitos propuestos desde un principio para el desarrollo de NWTJava v2.0.

[54] Esta situación se daba principalmente en la edición de propiedades del icono o pieza WRITE. Más adelante, términos como los mencionados en la primera sentencia de esta referencia, resultarán muy familiares.

En cuanto a la última parte del cuestionario, opinión personal sobre el seminario, el trabajo desarrollado en el mismo, y los profesores, las respuestas son más que satisfactorias.

Comentarios como

10. ¿Crees que la forma de trabajo que hemos seguido todos en el seminario favorece el aprendizaje?

"Por supuesto, me he enterado casi más en este seminario que en todo el primer cuatrimestre."

12. Qué os ha parecido (puntos fuertes, puntos débiles, críticas, esperabais algo diferente, alguna decepción…)

"Me ha sorprendido gratamente porque no esperaba tanta dedicación a alumnos tan perdidos como yo."

13. ¿Y en cuanto a los profesores?

"Me han parecido bastante bien, atentos, se esforzaban en las explicaciones, promueven la participación del alumno,…¿qué más se puede pedir?.**"**

nos llenan de orgullo y satisfacción[55].

3.4 Conclusiones

Para analizar las conclusiones obtenidas en el experimento de campo, seguiremos el guión establecido para los objetivos del mismo.

Estos objetivos eran los siguientes:

- Objetivo Principal: estudio de posibles mejoras a incluir en NWTJava v2.0.

- Objetivo Secundario: comprobar que la comprensión de la disciplina de la programación mejora en calidad mediante NWTJava.

- Objetivo auxiliar: adquisición de experiencia para la creación y desarrollo de futuros seminarios.

[56] Quizás no resulte demasiado apropiado introducir este tipo de comentarios en un proyecto de índole técnica y científica como es éste. Sin embargo, a nivel personal si me ha parecido importante resaltarlo. También considero esencial agradecer el esfuerzo y dedicación de los alumnos asistentes al evento, pues sin su participación, diversos aspectos abordados en este proyecto no hubiesen tenido el mismo peso específico.

Siguiendo un orden inverso en cuanto a la prioridad de los objetivos, podemos decir que alcanzamos el de adquisición de experiencia en cuanto a la creación y desarrollo de futuros seminarios. O por lo menos se encontró la dirección correcta en la que seguir trabajando.

Los datos objetivos sobre los que apoyamos esta afirmación los obtenemos de la comparación con el anterior experimento de NWTJava. En él, la asistencia a la primera sesión presencial fue del 90% de los estudiantes matriculados. Sin embargo, solamente 2 asistieron al *test final*. En opinión de los autores del anterior proyecto, el seminario no respondió a lo que finalmente esperaban dichos alumnos. Además, aunque ambos coinciden en ser de carácter voluntario, en el primero de ellos no se establecieron métodos de control (entregas de ejercicios, correcciones, etc.), por lo que no se pudo precisar el grado de implicación de los alumnos durante el transcurso del mismo.

Gracias a toda esta información, y al análisis de la misma, se pudieron poner en marcha planes cuya intención era la de subsanar o remediar estos problemas. Medidas de este tipo fueron:

- Se decidió no sólo utilizar NWTJava, también se resolverían dudas y ejercicios de Java.

- Se pensó premiar con 0´5 puntos en la asignatura asociada para mantener la asistencia (ante el aluvión de solicitudes, y debido a algunas dudas de tipo ético y/o moral, se decidió no tomar esta medida).

- Los alumnos deberían identificarse tanto en la entrega de ejercicios como en la resolución de los tests propuestos. En esta medida no vimos más que ventajas: establecíamos unos métodos de control, nos permitiría evaluar el grado de implicación, y también el rendimiento a nivel individual. Ésto último, a su vez, nos ayudaría a prestar una atención mucho más personalizada.

- Finalmente, el resto de medidas tomadas, y muy al contrario de lo esperado, tuvieron que ser de naturaleza restrictiva. Como ya mencionamos anteriormente, la explosiva demanda que alcanzó el seminario rompió con todas las expectativas, por lo que en lugar de incentivar la participación, hubo que restringirla. Para ello se dejó de informar al resto de carreras previstas, se exigió un compromiso, e incluso se pensó en castigar con -0´5 puntos en la asignatura asociada si la asistencia a clase no alcanzaba un porcentaje determinado (finalmente esta medida tampoco se contempló).

Como hemos comentado, estas medidas cumplieron con su cometido, por lo que su estudio aportará una valiosa información en la creación de nuevos seminarios de índole similar. Sin embargo, no quisimos desperdiciar la oportunidad de contemplar la opinión personal de los alumnos asistentes. Por ello elaboramos las preguntas 10 y 11 del cuestionario final:

10. **¿Crees que la forma de trabajo que hemos seguido todos en el seminario favorece el aprendizaje?**

11. **¿Piensas que sería bueno para posteriores promociones el repetir este seminario?**

La respuesta en todos los casos fue afirmativa, lo que incide aún más en nuestra conclusión de objetivo logrado. Aún así, muchos alumnos puntualizaron cosas como que el curso debería ser más largo o que para ser realmente positivo debería abarcar todavía más temario. Todas estas observaciones, sin duda son muy positivas y dan por finalizado el análisis de este objetivo.

En cuanto al secundario, comprobar que la comprensión de la disciplina de la programación mejora en calidad mediante NWTJava, podemos desglosarlo, estudiarlo y profundizar en él gracias a uno de los planes recién comentados.

Para ello se elaboró un registro en el que constaban los ejercicios, las calificaciones y los errores cometidos por cada uno de los alumnos. También se apuntaron en él todas las sugerencias y dudas, teniendo así un elaborado informe con el que se decidían y adoptaban las medidas a seguir en cada clase. De ahí muchas de las actividades que se mencionan en el apartado *3.3 DESARROLLO*, como la de resolver ejercicios de examen pertenecientes a la asignatura Representación de Datos y Aplicaciones o repasar algunos de los ejercicios mandados.

Gracias a éste registro podemos construir las siguientes tablas y gráficas sobre las que podremos ver los resultados obtenidos y decidir así si NWTJava influye positivamente, y en que medida, sobre los conocimientos de programación de los alumnos usuarios.

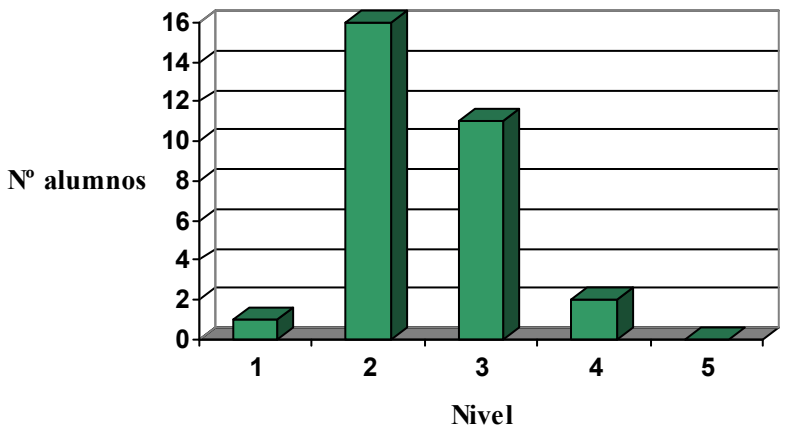


Fig. 3.8

Ejercicio número	Presentados	Correctos	Incorrectos	% de correctos
1	29	14	15	46
2	27	15	12	55´5
3	22	18	4	81´8
4	21	19	2	90´4
5	19	12	7	63
6	18	17	1	94´4
7 *	16	3	13	18´75

Fig. 3.9

Test final

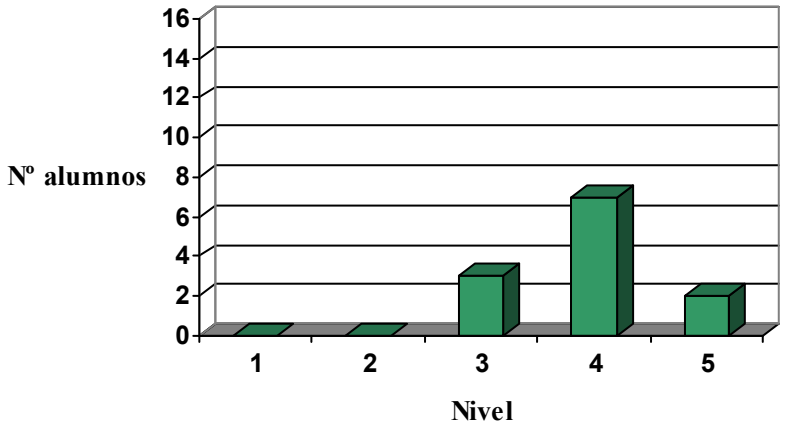


Fig. 3.10

Apenas es necesario comentar o comparar el resultado de las gráficas referentes a los tests. Basta una simple observación de las mismas para extraer una conclusión acerca de los resultados. Mucho más interesante es estudio de los ejercicios.

Examinando la tabla generada a partir de la información recopilada sobre estos ejercicios, vemos, en primer lugar como la constante de la que ya hablamos (menos ejercicios presentados pero mayor porcentaje de resoluciones correctas) se cumple inexorablemente en cada ejercicio. Claramente ésto tiene dos vertientes, una negativa (la participación decrece), y otra positiva, donde, salvo dos excepciones, la gráfica que representaría el porcentaje de soluciones correctas sería claramente ascendente. Y ello a pesar de que la dificultad de cada uno de los ejercicios es relativamente mayor a su predecesor.

Muy interesante es el caso del ejercicio 7 (remarcado con un *). El porcentaje de resoluciones correctas cayó brutalmente, instaurándose en un pobre 18´75%. Sin embargo, el motivo de lo ocurrido, argumentado por los propios alumnos, es muy claro, y enlaza perfectamente con lo que veremos a continuación, el análisis del objetivo primordial.

Todos estos alumnos declararon necesitar de una estructura en forma de array para resolver el problema. Por ello, todos sugirieron la inclusión de esta estructura en la

aplicación. Como podemos razonar sin mucha dificultad, es precisamente lo que buscábamos con este experimento, recopilar una serie de sugerencias que, según los participantes, potenciase las funciones docentes de esta herramienta.

Pasamos por tanto a examinar el objetivo principal de este experimento: el estudio de posibles mejoras a incluir en NWTJava v2.0.

Para este fin era fundamental estudiar las reacciones de los asistentes al experimento. Reacciones ante el manejo de la herramienta. Para ello, les sometimos a diferentes ejercicios y horas de "vuelo" con la misma. Todo ello, acompañado de una motivación: la de que fuesen los más critico posible con la aplicación. Ésto dio lugar a una lista de sugerencias y errores detectados.

Los errores eran, como ya dijimos en una ocasión, fundamentalmente técnicos (deficiente funcionamiento del "botón del pánico"; obligatoriedad de declarar algunas variables sin ser éstas necesarias; confusos mensajes de error; etc.). Los analizaremos en el siguiente capítulo según vayan siendo solucionados.

En cuanto a las sugerencias, principalmente se referían a la inclusión de nuevos conceptos, que dotasen a la aplicación de una mayor profundidad. Y precisamente era eso lo que pretendíamos con esta expansión de NWTJava, con su segunda versión. Destacaron las propuestas de añadir tipos en las variables, que no sólo fuesen numéricas, y sobretodo, por su insistencia, incluir el manejo de arrays.

Para finalizar el capítulo, y a modo de resumen, podemos decir que, a nuestro entender la inclusión de esta subfase fue un rotundo éxito, pues no sólo alcanzó los objetivos propuestos, sino que superó con creces las expectativas creadas.

A continuación veremos el que puede ser considerado capítulo clave de este proyecto. Todo lo relacionado con el objetivo de mejora y potenciación, es desarrollado en él. *Capítulo 4: DISEÑO Y CONSTRUCCIÓN DE NWTJAVA v2.0.*

Capítulo 4:

Diseño e Implementación de NWTJava v2.0

4.1 DISEÑO 82

4.1.1 Requisitos 82

4.1.2 Planteamiento General 83

4.2 IMPLEMENTACIÓN 86

4.2.1 Bloque Otros 87

4.2.2 Bloque Tipos de Variables 98

4.2.3 Bloque Arrays 117

4.2.4 Bloque Subrutinas 132

4.2.5 Bloque Ámbitos de Variables 150

Como ya introdujimos en los últimos párrafos del capítulo anterior, nos adentrarnos ahora en el que puede ser considerado capítulo central de esta memoria. Lo que queremos decir con ello es que, sin menospreciar la importancia de los anteriores capítulos, su función fue la de llegar hasta este punto. A partir de aquí, todos los análisis, los estudios, las discusiones y razonamientos, desembocan en lo que propiamente es la aplicación, su implementación.

4.1 Diseño

Lo primero que vamos a hacer es comentar el diseño, que a raíz de toda la información obtenida, analizada y sintetizada, se planteó para acometer este proyecto. Para describirlo, lo dividiremos en dos partes. La primera, *Requisitos*, donde haremos una breve recapitulación de lo que pretendemos lograr con la expansión de NWTJava. Y la segunda, *Planteamiento General*, donde describimos el proceso seguido para alcanzar el cumplimiento de los citados requisitos.

Sin más dilación, comenzamos.

4.1.1 Requisitos

Desde los orígenes de NWTJava, su intención fue la de profundizar lo máximo posible en el ámbito de la docencia de la programación y la algoritmia. Sin embargo, como todo sistema, debía evolucionar paulatinamente. El primer objetivo en su inicio se convirtió por tanto, en el de asentar unas bases, alcanzar unos mínimos, y examinar lo desarrollado, para así cerciorarse de que tanto la idea como el planteamiento eran los correctos.

Hoy día esa etapa inicial esta superada. Es por ello que surge la necesidad de seguir avanzando. La aplicación, sus creadores, e incluso los usuarios demandaban un mayor alcance y profundidad en diferentes aspectos: manejo, operatividad, funcionalidad, y como no, conceptual[57].

Surge así la siguiente pregunta: ¿cuál es el siguiente paso que debemos dar? Acudiendo a la evolución histórica sufrida por el mundo de la programación, observamos la siguiente sucesión:

Programación básica → ***Programación estructurada*** → ***Programación orientada a objetos***

La lógica pues, inducía a ello. Tras la concepción de los elementos básicos de la programación en la primera versión, tocaba incluir en esta segunda los referentes a la programación estructurada[58].

---

[57] Con este término nos referimos a la ampliación del número de conceptos a enseñar.

[58] Comentamos ya los principios de este tipo de programación en el *Capítulo 2: ESTADO DEL ARTE*. Analizaremos la inclusión de la programación orientada a objetos en el *Capítulo 5: CONCLUSIONES Y TRABAJOS FUTUROS.*

Ésto, unido a los requerimientos de los usuarios (sus sugerencias, recordar *Capítulo 3: EXPERIMENTACIÓN*), y al planteamiento general del proyecto, hizo de los requisitos la inclusión en el programa de la siguiente lista de elementos:

- Tipos de variables.
- Manejo de arrays.
- Subrutinas.
- Ámbito de variables.

Y todo ello, manteniendo siempre los principios esenciales de NWTJava:

- Enseñar los conceptos de la programación y la algoritmia.
- Con independencia de un lenguaje concreto de programación.
- De un modo que resulte sencillo, interactivo, que busque el razonamiento e incluso el autoaprendizaje.
- Con una interfaz gráfica familiar y sencilla.
- Con un enfoque atrayente y divertido.

Una vez conocido los requisitos, pasamos a describir el planteamiento general.

4.1.2 Planteamiento General

La primera decisión crítica que hubo de tomarse fue, cómo debíamos estructurar la aplicación para delimitar los dos grandes ámbitos de enseñanza: principios fundamentales de la programación y programación estructurada.

Pues bien, la decisión tomada, sobre la que más tarde argumentaremos, fue la de dividir la enseñanza en dos niveles de aprendizaje.

- Básico.
- Intermedio o Estructurado.

Una vez decidida esta división, podemos decir que es en el *NIVEL INTERMEDIO o ESTRUCTURADO* en el que se centra este proyecto. Y decimos "*se centra*" porque también se ha trabajado sobre el *BÁSICO* (solucionando errores detectados, adecuándolo a la nueva estructura de la aplicación, etc.), y además, se han establecido

las bases necesarias para poder implementar un posible *NIVEL AVANZADO* en la siguiente versión.

Para cada uno de los bloques ya mencionados (tipos de variables; manejo de arrays; subrutinas; ámbito de variables), tratados como entidades independientes (hasta llegar al punto en el que era necesario relacionarlos con el resto), se ha seguido para su construcción 6 grandes fases. Éstas han englobado a groso modo el desarrollo de cada uno de ellos. Son los siguientes:

- Planteamiento teórico.
- Reflejo en la Interfaz Gráfica.
- Funcionamiento en el Núcleo de la Aplicación.
    - Condiciones de validación (o Tratamiento Gramatical).
    - Ejecución.
- Módulo de Traducción.
- Interrelación entre bloques.
- Pruebas definitivas.

Desglosamos esta clasificación de fases en una más detallada y explicativa:

- Planteamiento teórico.
    - Presentación del objetivo por parte del tutor.
    - Razonamiento sobre la forma de abordar el problema y encontrar su solución.

      Este punto se realizaba mediante un estudio de pros y contras, tanto a nivel de implementación como desde el punto de vista de la docencia. La pregunta clave en cuanto a este aspecto era: ¿cómo resultaría más beneficioso para la instrucción de los usuarios? ¿Y para nosotros como programadores?
    - Enumeración de problemas y dificultades surgidas.
    - Exposición de las conclusiones al tutor.
    - Puesta en común (entre ambos).
    - Decisión final.

- Reflejo en la Interfaz Gráfica.

  Como es lógico, cada bloque debía ser representado de cara al usuario mediante la interfaz gráfica. Cada uno de ellos, a su vez, presentaba unas características muy concretas, y debía decidirse cómo serían simbolizados.

  Por ejemplo: la forma en la que el usuario asigna el tipo a una variable. Como veremos en su correspondiente apartado, en la *Tabla de Edición* de aquellos iconos donde corresponda crear o inicializar variables, aparecerá un desplegable (un objeto de la clase *JComboBox*[59]) donde al pinchar, se podrá seleccionar de entre todas las opciones existentes, el tipo de la variable.

  Otro ejemplo: ¿qué nomenclatura utilizar para definir arrays?, ¿y si queremos que la definición sea intensiva?, ¿o extensiva[60]?

- Funcionamiento en el Núcleo de la Aplicación.

  - Condiciones de validación (o Tratamiento Gramatical).

    Como ya pasaba en la versión original, cada diagrama era sometido a un tratamiento gramatical, con la intención de detectar errores, incompatibilidades e incumplimientos de normas en su construcción[61].

    Por supuesto, estos nuevos elementos también deben someterse a reglas, y como es lógico, no solo a las ya implementadas, también a las nuevas surgidas a raíz de la aparición de estas estructuras.

    Por lo tanto, en esta subfase, se analizarán e implementarán las reglas de tipo gramatical asociadas a cada bloque.

  - Ejecución.

    La aparición de nuevos elementos exige también una nueva estructura de ejecución.

    Ahora no sólo ejecutamos diagramas con variables de tipo entero, también podrán contener de tipo cadena (*Strings*) o incluso arrays de tipo booleano.

- Módulo de Traducción.

  Una vez implementadas tanto la validación como la ejecución de las nuevas estructuras, resulta también fundamental reflejarlas en el código equivalente del lenguaje al que traducimos el diagrama (recordemos que el área reservada para tal efecto está en la parte derecha de la interfaz, y que en este caso, el lenguaje real al que se traduce es Java).

[59, 60] Si el lector no asocia ahora mismo estos términos a nada conocido, no debe preocuparse, todos serán definidos y explicados en su momento.

[61] Podemos entender la validación como un proceso análogo al de la compilación.

- Interrelación entre bloques.

  Fundamental para el correcto funcionamiento de la aplicación. Si bien en sus fases iniciales, los bloques eran tratados de forma independiente (con el fin de acotar problemas y hacer uso del "divide y vencerás"), posteriormente debían enlazarse y funcionar tal y como lo haría una máquina formada por distintas piezas y bien engrasada.

  Un usuario podría querer implementar un diagrama con un array, de tipo entero, pasado como parámetro a una subrutina, y que ésta estuviera inmersa en un bloque de *FOR´s* anidados.

- Pruebas definitivas.

  A parte de la infinidad de pruebas iniciales e intermedias, cuyo fin era el de detectar problemas, fallos, incongruencias, para finalmente encontrar el camino correcto, era estrictamente necesario una masiva batería de pruebas que intentase abarcar la mayor cantidad de casos posibles. Que probase el funcionamiento del propio bloque, pero también el de la aplicación en su conjunto. Y siempre con un nivel de exigencia exponencial. De esta forma, surgió también una batería de ejemplos y ejercicios que ponemos a disposición del usuario junto con la aplicación. Podrá así, si lo desea, probar toda la operatividad de NWTJava v2.0.

Para finalizar con este apartado, no podemos dejar escapar lo que ya comentamos en el punto *1.5 FASES Y METODOLOGÍA DE TRABAJO* de esta memoria. Y es que a estos cuatro bloques añadimos un quinto, que a modo de tejido conjuntivo, se encarga de la interrelación entre ambas versiones, además de añadir un plus en la operatividad y funcionalidad de la herramienta.

4.2 Implementación

En los siguientes subapartados analizaremos la implementación de cada uno de los bloques citados en el diseño. Expondremos las ideas surgidas para su implementación y funcionamiento, además de las disquisiciones y argumentaciones que tuvieron lugar para finalmente, alcanzar y aplicar la solución final.

Aunque puede resultar chocante, vamos a comenzar la exposición con el que hemos llamado hasta ahora "*quinto bloque"*, al cual pasaremos a denominar *Bloque Otros*, por englobar todo aquello que no pertenece o no tiene cabida en ninguno de los restantes bloques principales. A continuación expondremos los motivos por los que abrimos la redacción de los subapartados con él.

4.2.1 Bloque Otros

El motivo por el que iniciamos la descripción de todos los bloques con el denominado "*Otros*" es porque, sencillamente, los primeros pasos en la implementación del código de ésta la segunda versión, pertenecen a su ámbito. Y es que, si recordamos cuando mencionamos a este bloque en el último párrafo del apartado 4.1, dijimos que éste actúa a modo de tejido conjuntivo, es decir, entre sus funciones está la de compatibilizar lo construido en la versión original con los requisitos y exigencias de esta segunda.

- La primera gran decisión correspondió a una importante reestructuración de los contenidos de la aplicación. Hicimos ya mención de este aspecto con las siguientes líneas:

"*…debíamos estructurar la aplicación para delimitar los dos grandes ámbitos de enseñanza: principios fundamentales de la programación y programación estructurada*."

Pues bien, como también comentamos, la decisión tomada fue la de dividir la enseñanza en dos niveles de aprendizaje.

La idea inicial era clara. Nada más abrir la aplicación, saldría por pantalla un *JPanel*[62], que contendría un objeto *ButtonGroup* compuesto a su vez por dos *JRadioButton*. Con este panel se brindaría al usuario la posibilidad de escoger el nivel de aprendizaje deseado. Más adelante veremos una figura donde podremos ver el resultado final.

La problemática surgió cuando pensamos en dónde incluir la implementación de todo ésto. Y no sólo de ésto, también de la gestión de niveles. El programa debía saber en todo momento en que cota de aprendizaje se encontraba, pues dependiendo de él, los diagramas estarían sujetos a unas condiciones u otras (por ejemplo: a *NIVEL BÁSICO* no debe permitirse la utilización de arrays).

Las opciones eran dos: crear una nueva clase para la gestión de los niveles (*class Nivel*), o añadir esta función a alguna ya existente. A priori, la elección parece sencilla, por aquello del encapsulamiento y la modularidad de los lenguajes orientados a objeto. Sin embargo, hubo un cierto número de matices a tener en cuenta.

La clase *NWTJava*, la principal de la aplicación, soporta la mayor parte de las funciones referentes a la interfaz gráfica de usuario, y además, gestiona la entrada de información usuario-máquina. Como en la elección de nivel influían ambas cosas, la segunda opción no era descabellada.

Finalmente se tomó la primera, pues rápidamente caímos en la cuenta de que los niveles tendrían un rol fundamental en el resto de funcionalidades.

La implementación de la solución fue la siguiente: creamos la clase *Nivel*, que simplemente contiene la información seleccionada por el usuario y los métodos necesarios para informar de ello a las entidades que lo soliciten. Al invocarse el método

[62] Documentación de las clases de *Swing* en referencia [14].

main, *NWTJava* (la clase que lo contiene) crea un objeto *Nivel*, arranca la interfaz y muestra el panel *"Escoja nivel de aprendizaje: "*. Es entonces cuando el usuario lo selecciona, información que se pasa al objeto *datos* con el método *actualizarNivel*. Es pues, la clase *Datos* (la clase central del Módulo de Gestión[63]) la que gestiona dicha información.

El resultado en pantalla es el siguiente:

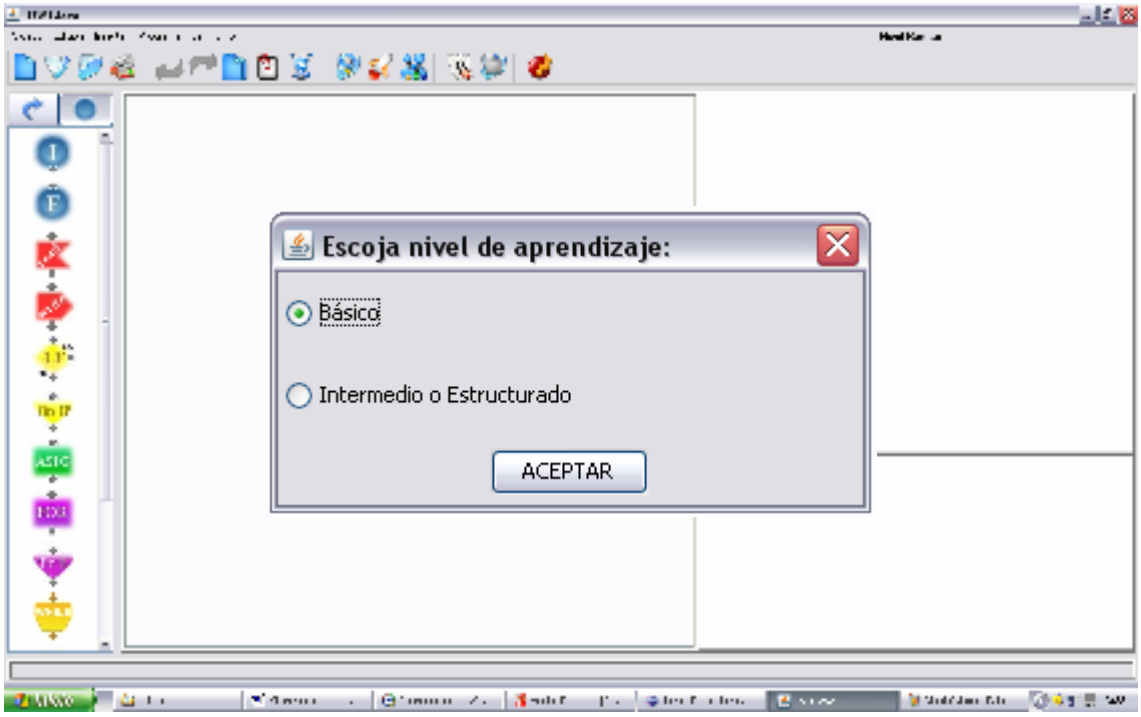


Fig. 4.1

Para acabar con los niveles de aprendizaje, al menos por el momento, ya que volveremos a mencionarlos cuando expliquemos la utilización de *DTD´s*, es necesario comentar tres asuntos.

El primero de ellos surgió de la pregunta: ¿y cómo cambiamos de nivel? Se plantearon numerosas alternativas al respecto (en la barra de menú, una opción para cambiar de nivel; cambiarlo automáticamente al abrir un diagrama perteneciente a otro nivel; etc.). Sin embargo, y dado que los gráficos de un nivel, para facilidad del usuario, serían incompatibles con los de otro, muchas de las opciones quedaron descartadas.

Ésto, sumado a la vorágine de diferentes posibilidades en que desembocaban cada una de ellas (al cambiarlo deberían guardarse y cerrar los diagramas abiertos, o avisar mediante mensajes de información sobre las posibles incompatibilidades…) nos hizo tener que revisar los principios básicos de NWTJava. Rápidamente caímos en la cuenta: SENCILLEZ, SIMPLICIDAD.

[63] La descripción e información necesaria de la clase *Datos* la encontramos en la referencia al Módulo de Gestión, apartado *2.4.2.3 Núcleo de la aplicación*.

Dos principios esenciales. Lo que se pretende en todo momento es que los usuarios vayan aumentado su nivel de programación y aprendizaje progresivamente, pero con la menor carga cognitiva posible. Como además teníamos claro desde un principio, que cada nivel suponía un modo de trabajo diferente, la solución era la opción más sencilla posible. Cada aplicación abierta funcionaría únicamente en el nivel seleccionado. Para trabajar en otro, el usuario debería abrir una nueva instancia. Ésta solución, además de ser la más simple desde el punto de vista de la implementación, también lo era para el de la docencia. El alumno pasa así a estar muy concienciado, en todo momento, del nivel de aprendizaje en el que se encuentra, además de las opciones y limitaciones que presenta dicho nivel, sin perder así ni un ápice de su concentración en si debería, mediante un simple botón u opción en el menú, cambiar de nivel.

El segundo asunto es, una evolución del primero. Una vez asimilado que el cambio de nivel no es posible en una misma instancia del programa, ¿cómo podríamos dar un paso más en el aprendizaje? Pensamos que sería muy positivo el que, estando en un nivel, se pudieran utilizar diagramas construidos por los propios usuarios en niveles inferiores. Es decir, a *NIVEL INTERMEDIO* deberían poderse abrir diagramas construidos en el *NIVEL BÁSICO*. Ésto por supuesto con ciertas limitaciones y restricciones. Por ejemplo, al abrir un diagrama de *NIVEL BÁSICO* (donde las variables no tienen tipo) en el *INTERMEDIO*, e intentar validar, saldría un mensaje de error informando de que la validación no es posible, pues en dicho nivel, las variables deben tener asignado un tipo obligatoriamente. De esta forma, una vez el alumno ascendiese a una cota superior de su aprendizaje, podría analizar cuales son los cambios y diferencias entre niveles.

No obstante, este proceso a la inversa no esta permitido. La razón es sencilla: mientras que todas las posibilidades de un nivel inferior son contempladas por uno superior (como ocurre en toda política de versiones, en sistemas operativos por ejemplo), no todas las estructuras de un nivel superior son aceptadas por uno inferior. Por ejemplo: si en un diagrama perteneciente al *NIVEL INTERMEDIO* utilizamos arrays, ¿cómo los manejaría el *NIVEL BÁSICO* sino reconoce dicha estructura?

El tercer y último asunto hace referencia a qué subyace a cada nivel de aprendizaje. Como ya sabemos, el objetivo principal de NWTJava v1.0 era el de instruir sobre los principios básicos de la programación y la algoritmia. El de NWTJava v2.0 es el de dar un paso más y añadir al aprendizaje los principios de la programación estructurada. Parece lógico pues, asignar todo el contenido didáctico de la primera versión al *Nivel de Aprendizaje Básico*, y el de esta segunda al *NIVEL INTERMEDIO*. El último nivel queda abierto a una futura versión.

Con esta reestructuración inicial comenzamos la creación de NWTJava v2.0. Los siguientes pasos dados dentro de este *Bloque Otros* se destinaron a actualizar y preparar la interfaz gráfica para la serie de cambios que iba a sufrir la aplicación. Los comentamos.

- El primer cambio al respecto fue automático, debido al traspaso del código de NWTJava a una nueva versión de Java. Pasamos de j2sdk1.4.2_12 a jdk1.6.0_11. El tratamiento de esta nueva y mejorada versión sobre la librería Swing es diferente. Proporciona a sus componentes visuales una línea más estilizada y ornamentada, dando una impresión de mayor dinamismo y modernidad.

Ésto supuso un importante cambio en el diseño de la interfaz gráfica, añadiendo además un conjunto de nuevos elementos y cambios en la estructura.

- El primero de estos nuevos elementos fue la inclusión de una etiqueta (objeto *JLabel*) en la esquina superior derecha de la ventana principal. En ella se indica en todo momento el nivel de aprendizaje seleccionado por el usuario. Es una etiqueta cuyo función es la mera información de este hecho y su fin, el de no hacer perder la perspectiva al alumno de en que cota de programación se encuentra.

La construcción de esta etiqueta se realiza inmediatamente después de presionar el botón "*ACEPTAR*" del panel "*Escoja nivel de aprendizaje:* " y está implementada en el método *ponerEtiqueta* de la clase *NWTJava* (recordemos que es esta clase la que soporta la mayor parte de las funciones referentes a la interfaz gráfica de usuario).

Veremos la inclusión de esta etiqueta y de los demás cambios descritos sobre la interfaz gráfica en breves instantes, mediante dos imágenes correspondientes a cada una de las interfaces (versión 1.0 y 2.0). De esta forma, podremos realizar una comparativa.

- El tercero de estos cambios resultó sencillo. Simplemente mediante un nuevo método hicimos que, para comodidad del usuario, la ventana principal del programa apareciese maximizada ya por defecto. Anteriormente, la forma en que estaba codificada la apertura de la aplicación hacía que su ventana principal apareciese minimizada. Es más, se abría con unas dimensiones concretas, predeterminadas (500x400 píxeles). Actualmente, al iniciar la aplicación, el método *maximizar* obtiene mediante las dos siguientes instrucciones

```
Toolkit kit =  Toolkit.getDefaultToolkit();
Dimension screen = kit.getScreenSize();
```

las dimensiones de la pantalla sobre la que se va a ejecutar NWTJava. Una vez conocida esta información, se ordena que el tamaño de la *JFrame* principal (ventana principal) sea de esas dimensiones. Ni que decir tiene que este nuevo método también pertenece a la clase *NWTJava*.

- La siguiente modificación resultó algo más tediosa. El estilismo de la interfaz gráfica había cambiado, evolucionado. Sin embargo, los iconos representativos de cada instrucción o estructura (*FOR*, *WHILE*, *IF*,…), los ficheros gráficos que los definían, quedaron obsoletos. En comparación con las líneas y componentes de la interfaz (botones, menús,…), los iconos resultaron ser toscos y vetustos. Se apreciaba cada píxel y desentonaba con la línea general de la interfaz. Por ello decidimos rediseñarlos. Seguiríamos su estructura general (forma, color) pues ésta no era aleatoria y sin razón.
De hecho, los motivos por los que se decidió seguir esta nomenclatura de símbolos son importantes y tienen una sólida argumentación[64].

Decimos que resultó tedioso no por la labor de seleccionar el programa de dibujo más adecuado a nuestras exigencias (finalmente se optó por *GIMP Portable*), o por aprender a controlar sus funciones correctamente (aspecto lógico en cualquier desarrollo de un

[64] Podemos encontrar dicha argumentación tanto en el apartado *2.4.1.1 Interfaz de usuario* de este mismo proyecto como en la referencia [5].

proyecto), sino por la tarea de ir icono por icono transformándolo hasta alcanzar el resultado deseado. Una muestra de la evolución sufrida por cada una de estas piezas la podemos mostrar mediante un ejemplo. Vemos el proceso de transformación de los iconos de "*inicio*" y "*asignación*":

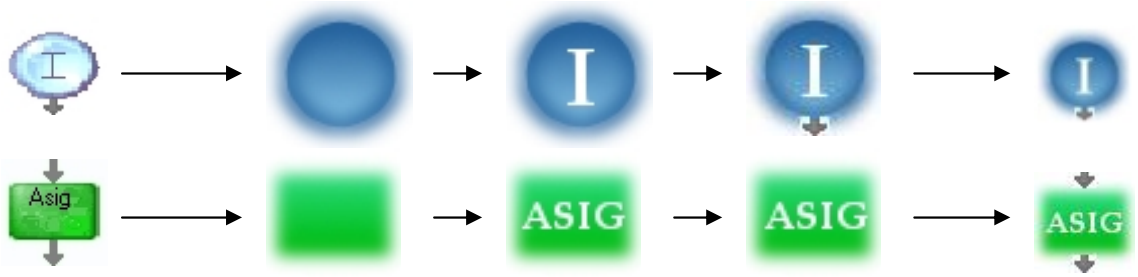


Fig. 4.2

A continuación exponemos la totalidad de los iconos tanto de la antigua versión como de la v2.0:

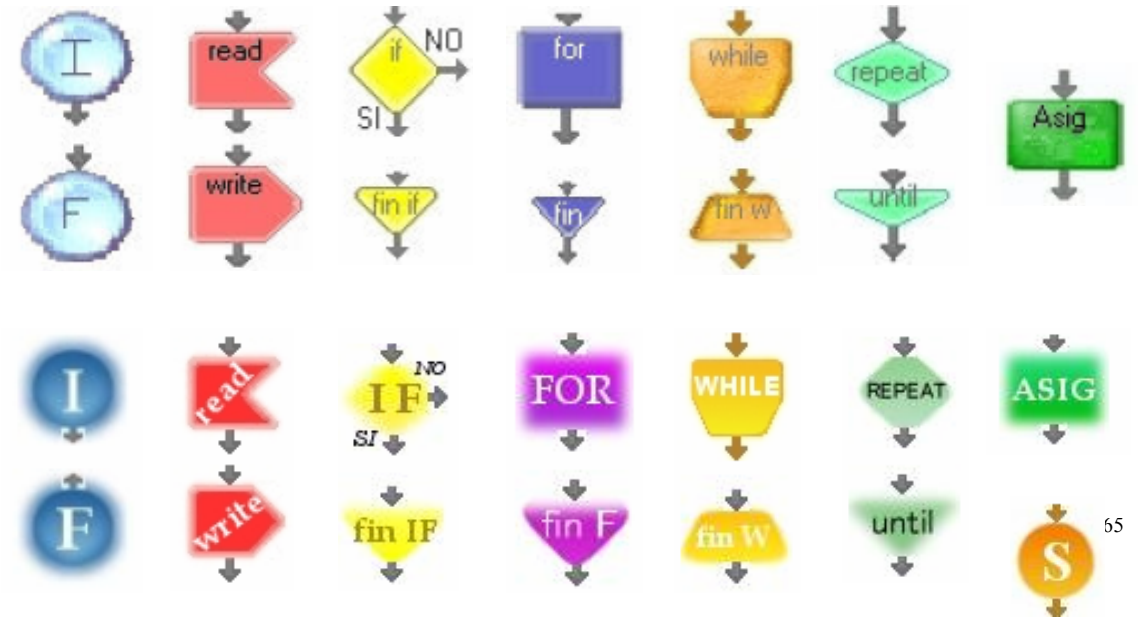


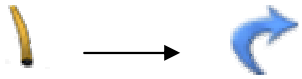
Fig. 4.3

- También nos vimos obligados a rediseñar la representación de los botones de modo. La función que desempeñan estos botones es la de activar el modo "*colocación de piezas*" (con él activado se pueden arrastrar los iconos desde su panel hasta el *Área de Dibujo*) o el modo "*colocación de ejes*" (ídem que el anterior pero para representar y direccionar los ejes que unen las distintas piezas).

[65] Esta pieza no tiene equivalencia en la antigua versión de NWTJava pues surge como respuesta al requisito de permitir el uso de subrutinas. Lo veremos en su correspondiente bloque.

El motivo de este cambio no fue tanto su diseño como el de las continuas sugerencias, por parte de los alumnos asistentes al experimento, de modificarlos. Comentaban que sus símbolos, su representación, era confusa y no daba una idea fidedigna de la que era su labor.

El cambio fue el siguiente:

Botón de modo ejes: 

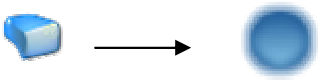
Botón de modo piezas: 

Fig. 4.4

Para visualizar los cambios comentados hasta el momento, y ver sus efectos en la nueva interfaz, mostramos las siguientes dos figuras, correspondientes a cada una de las interfaces de las dos versiones de NWTJava:

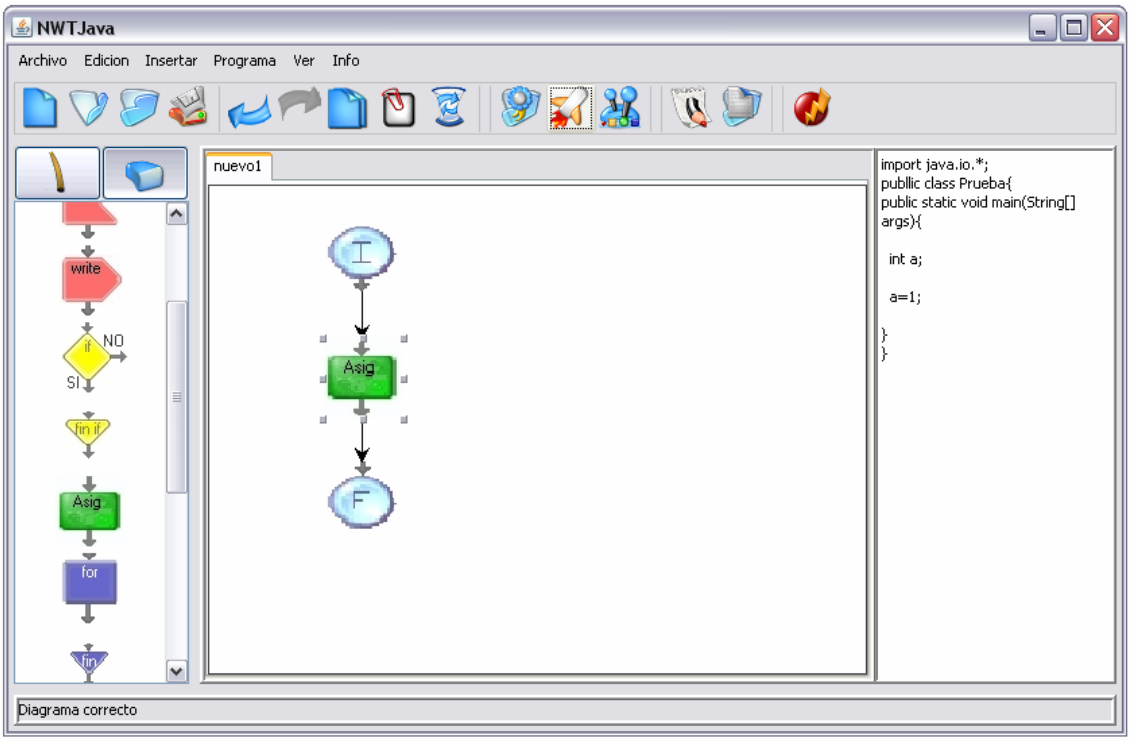


Fig. 4.5 - Interfaz gráfica de NWTJava v1.0

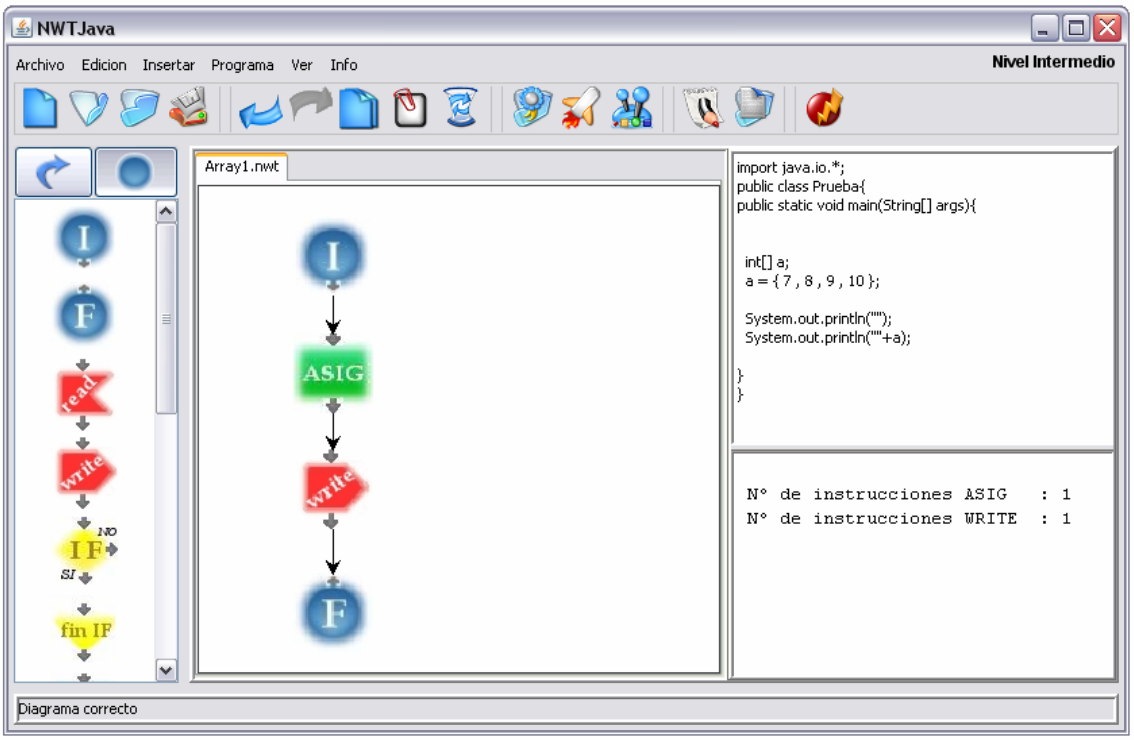


Fig. 4.6 - Interfaz gráfica de NWTJava v2.0

- Si nos fijamos detenidamente, podremos apreciar otra variación no comentada hasta el momento. Si llevamos nuestra atención a la zona derecha, vemos que en la primera versión queda reservada para la traducción de los diagramas a código real. Sin embargo, en NWTJava v2.0 este área está dividida en dos. Cada una de las divisiones pasa a tener una tarea específica. La superior sigue dedicada a la traducción del diagrama, *Área de Traducción*, pero ¿cuál es la función del segundo y nuevo panel?

Y es con esta pregunta con lo que pasamos a exponer el último cambio realizado al respecto de la interfaz gráfica.

Pensamos en añadir una nueva área de texto (*JTextArea*) que ofreciese al usuario una información muy concreta acerca de sus diagramas construidos. Dicho panel mostraría el número de apariciones de cada uno de los tipos de estructuras utilizado. Es decir, si un alumno implementase un diagrama con tres bucles *REPEAT* y dos *IF*, la información mostrada sería:

Nº de instrucciones REPEAT : 3
Nº de instrucciones IF : 2

La función que desempeña esta información presenta varias vertientes. La primera es llevar la contabilidad sobre el número de estructuras utilizadas en el diagrama. Muy útil para cuando la extensión del mismo no permite visualizarlo de un solo vistazo. Digamos que funciona a modo de un contador de líneas de código como pueda haber en cualquier editor. La segunda y más importante es, que a través de los datos mostrados, el usuario busque una cierta competitividad consigo mismo e intente abordar la solución desde la perspectiva de la optimización de código. Es decir, una vez analizado el funcionamiento del diagrama, la estructura del mismo y los datos que arroja este panel, busque alcanzar los mismos resultados pero rebajando el número de instrucciones empleadas, lo que en programación denominamos: optimización.

Además, este aspecto no sólo esperamos que lo infiera el alumno por sí mismo. Al respecto también hay ejercicios preparados para ello, donde su enunciado exige un número máximo de determinadas instrucciones[66]. Para lo citado en el anterior párrafo, y lo mencionado en éste, es para lo que resulta positivo la inclusión de este nuevo panel, al cual pasamos a denominar *Área de Detalles*.

El modo en que construimos el *Área de Detalles* fue el siguiente. Vamos a representar primero un boceto de la anterior distribución de paneles:

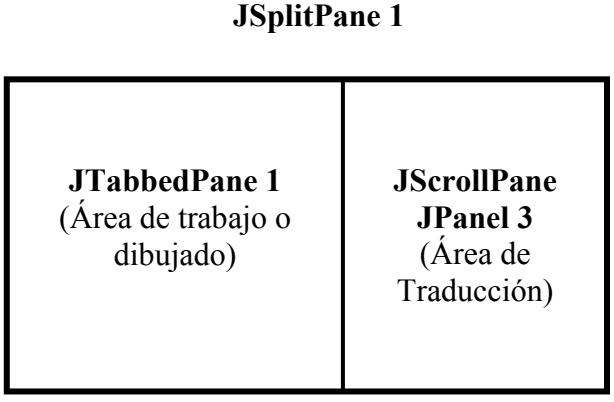


Fig. 4.7

[66] Ver enunciado del Ejercicio 1 en *APENDICE B*.

Pues bien, la nueva y actual disposición, en la que ya incluimos el nuevo *Área de Detalles*, es la siguiente:

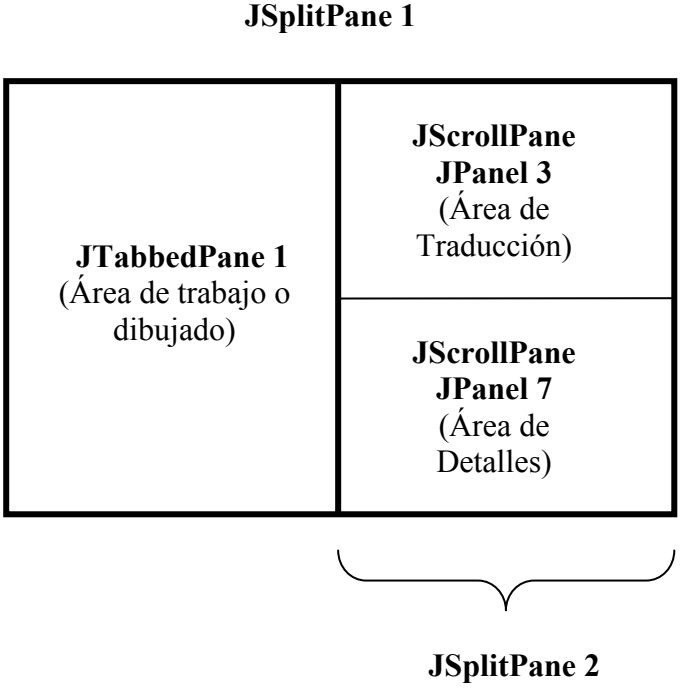


Fig. 4.8

Teniendo ya una idea general de la estructura de paneles, podremos entender más fácilmente que fue lo que hicimos a nivel de código.

Lo primero fue crear tanto el *JSplitPane2*[67] como el *JPanel7* mediante los métodos *getJSplitPane2* y *getJPanel7* respectivamente. Pero no fue sólo eso lo que hicimos con ambos métodos. En el segundo, *getJPanel7*, también creamos su correspondiente *JScrollPane* y desactivamos la posibilidad de que su *JTextArea* correspondiente fuese editable. Con ésto último impedimos que los usuarios pudiesen escribir en él, y por tanto, que su labor fuese exclusivamente informativa. En cuanto a *getJSplitPane2*, le proporcionamos unas dimensiones dentro del contexto *JSplitPane1* y ordenamos el contenido colocando *JPanel3* en la posición *TOP* y *JPanel7* en la posición *BOTTOM*.

Además les aseguramos unas dimensiones mínimas para que puedan ser redimensionados todo lo que se desee pero en ningún caso lleguen a desaparecer.

En cuanto al funcionamiento de la nueva incorporación, el *Área de Detalles*, se llegó a la conclusión de que debía funcionar de la misma forma que el *Área de Traducción*, es decir, cuando se mostrase la traducción del diagrama a código real, también debería mostrarse su información; cuando se reiniciara el primero, también lo haría el segundo, y así sucesivamente. Por ello, los métodos relacionados con el *Traductor* (no todos) tienen un análogo referenciado al *Área de Detalles*.

Por último comentar, que para conocer el número de apariciones de cada estructura, utilizamos el método *resumeGrafo* de *GraphManager*, el cual devuelve un array en el

[67] *JSplitPane* se usa para dividir dos componentes, que a su vez pueden ser alineados de izquierda a derecha o de arriba hacia abajo.

que cada posición hace referencia a un icono (un objeto *UserObject*) del diagrama actual. Lo que hacemos es contar las apariciones de los iconos que nos interesan (*IF*, *FOR*, *WHILE*, *REPEAT*, *ASSIGN*, *READ* y *WRITE*[68]) y mostrar el nº de apariciones de cada tipo en el panel de *Detalles*.

Y de esta forma, queda ya descrita toda la funcionalidad de este área, por lo que pasamos inmediatamente a la siguiente fase del *Bloque Otros*.

- Esta fase, fuera ya de las modificaciones sufridas por la interfaz gráfica, podemos incluirla en las pertenecientes a compatibilizar lo construido en la versión original con los requisitos y exigencias de esta segunda. De hecho, lo que vamos a hacer tiene dos vertientes. Una es la de subsanar un error en el funcionamiento de NWTJava v1.0, y la otra es la de hacer congruente un modo de funcionamiento, también de la primera versión, con el nuevo planteamiento de NWTJava v2.0.

Explicamos en que consistía el error. NWTJava v1.0 trataba todas las variables introducidas por los usuarios en la construcción de sus diagramas como si fuesen de tipo *double.* Y esta variable podía ser perfectamente utilizada durante todo el transcurso del programa. Sin embargo, en la traducción a código real Java, esta variable aparecía siempre como tipo *int*. Pero no sólo eso. Como ya hemos comentado, los valores que se podían asignar a las variables eran *double*, pero también de cualquier otro tipo (enteros *int*, reales *double* e incluso caracteres *char*, cadenas de caracteres *String* o booleanos *boolean*). Y todo ello sin queja alguna en los procesos de validación (compilación) y ejecución. Por supuesto, en todos los casos, al traducir a Java, seguía aplicándose únicamente el tipo *int*.

Este suceso no era lógico, por lo que era primordial su arreglo. Fueron dos las soluciones planteadas a priori, pero para entenderlas, es fundamental ser conscientes de que esta modificación actuaría únicamente a *NIVEL BÁSICO*, nivel que corresponde con lo heredado de la primera versión, y más concretamente, con los principios que pretendía inculcar.

La resolución A consistía en hacer que, a *NIVEL BÁSICO*, todas las variables fuesen reales (tipo *double*), para poder así abarcar tanto valores enteros como reales. Bastaría con cambiar en la clase *Plugin* (encargada de la traducción a lenguaje Java) la impresión de la palabra *int* por la de *double*, e impedir en la validación que los valores de estas variables pudiesen tomar otro tipo.

La B era similar, pero el cambio era más profundo, sobretodo desde el punto de vista conceptual. Radicaba en la idea de no hacer que todas las variables fuesen de tipo *double*, sino enteras, *int*.

¿Y en qué sentido difiere la solución B de la A? Pues bien, a pesar de que a primera vista parece un cambio únicamente de tipos y sin mayor relevancia, resultó ser un punto de inflexión en el caso de la docencia.

---

[68] Mostramos el número de ocurrencias de estas estructuras que son las que aportan información de interés. El lector puede estar preguntándose porqué el número de apariciones del resto de iconos no resulta relevante. La explicación es la siguiente: no aportan ninguna información. Por ejemplo, ¿debemos reflejar el número de iconos de *INICIO* y *FINAL*? No, pues sabemos que éstos sólo van a aparecer una vez en un diagrama que sea válido. ¿Y el resto, como puedan ser *FIN FOR* o *UNTIL*? Sería redundante pues tienen el mismo número de apariciones que el icono que abre el correspondiente bucle, es decir, las ocurrencias del icono *UNTIL* son las mismas que la del *REPEAT*.

Como ya sabemos, el objetivo fundamental de NWTJava v1.0 (en nuestro caso, el *NIVEL BÁSICO*) era el de instruir sobre los principios básicos de la programación y algoritmia, mientras que, uno de los requisitos de esta nueva versión, cuyo nuevos conceptos pretende enseñar a partir del *NIVEL INTERMEDIO*, es de la inclusión de tipos de variable.

Entonces, ¿por qué no dejar el tipo más sencillo para el *NIVEL BÁSICO*? Ya introduciríamos el concepto de tipos de variable y las diferentes características de cada uno de ellos en el siguiente nivel. Ésto, sumado a que los ejercicios del primer nivel no exigían más que la utilización de variables numéricas enteras, y que lo que interesaba era centrar su atención en lo más básico y fundamental (como el manejo de las diferentes estructuras condicionales e iterativas, la correcta expresión de las condiciones, etc.), hicieron de la opción B la más adecuada.

La implementación de esta solución fue la siguiente. En primer lugar accedimos a la clase *Variable* (la clase relacionada con los objetos *variable*). Cada variable viene definida únicamente por su *nombre* y su *valor* (por el momento, esta situación cambiará en el siguiente bloque). Modificamos su atributo *valor*, que como podemos deducir fácilmente, era de tipo *double*. Lo cambiamos a *int*.

El siguiente paso está relacionado con el proceso de validación. Para ello comentaremos brevemente cual es el proceso por el que creamos una variable y le aportamos un valor.

Cada estructura está representada por su correspondiente icono o pieza. Todas éstas a su vez tienen una correspondiente *Tabla de Edición*, encargada de recoger los datos introducidos por el usuario respecto a esa pieza. Estos datos son los parámetros de cada una de ellas. Por ejemplo, la *Tabla de Edición* del icono *FOR* solicita al usuario los valores correspondientes a sus parámetros *variable*, valor de *inicio*, condición de parada o *final*, e *incremento* del valor de la variable. Lo vemos mediante la siguiente figura:

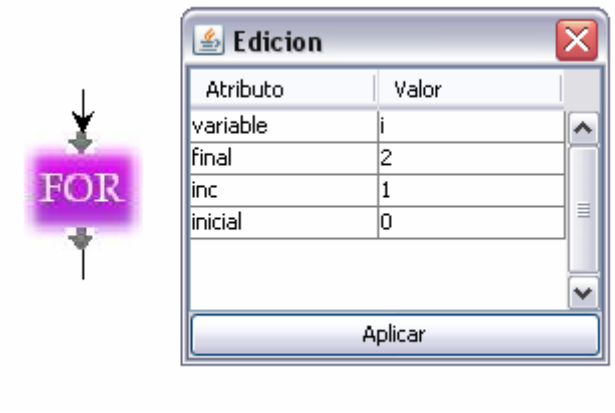


Fig. 4.9

Pues bien, de esta forma creamos la variable "*i*" y le asignamos el valor "*0*". Y siendo en el método *validar* de la clase *GraphManager* donde se realizan las oportunas comprobaciones de la validación, fue en él donde implementamos el código necesario para que, si a *NIVEL BÁSICO* el valor de una variable no correspondía con uno válido (es decir, de tipo entero), saltase un error y su correspondiente mensaje de información. Esta comprobación esta subdivida en dos. Si el valor es numérico, se le aplica el módulo (%). Si el resultado de la operación es distinto de cero, se trata de un valor real, error. Si

no es numérico, inmediatamente sabemos que el valor que se pretende introducir es un *char*, un *String* o incluso un *boolean*, error.

En cuanto a la ejecución, no es necesaria ninguna comprobación pues en el momento que se de un error de validación, no se permite ejecutar el diagrama.

Por último decir que para este tipo de comprobaciones (ésta y muchas más que veremos en los restantes bloques), el método *validar* de *GraphManager* delega en una batería de métodos muy específicos para cada comprobación. Ejemplos de éstos son *comprValor* (comprobar valor), *comprCondicion* o *comprNomSub* (que lo veremos en su correspondiente bloque).

Y es a partir de esta modificación cuando abandonamos el *Bloque Otros* y entramos de lleno en el siguiente: *Bloque Tipos de Variables*. Tenemos que decir que el aprendizaje que nos proporcionó este cambio sobre la casuística, la implementación de variables y sus tipos, nos permitió abordarlo con unas mínimas garantías, una base.

Recalcar también que volveremos a este bloque en determinadas ocasiones, en aquellas donde la implementación de una solución nos obligó a adoptar medidas independientes de cualquiera de los otros cuatro bloques principales. Es el caso, por ejemplo de, cuando tengamos que tratar la codificación de las DTD´s propias de este proyecto.

4.2.2 Bloque Tipos de Variables

Una vez sistematizado el *NIVEL BÁSICO* conforme a los criterios propios de NWTJava v2.0, pasamos al *NIVEL INTERMEDIO*.

En estos primero instantes, el funcionamiento de ambos niveles era idéntico. No habíamos hecho nada que diferenciase a ambos. Y era lo lógico, simplemente estábamos regulando el funcionamiento base de toda la versión. Es a partir de ahora cuando ya incidimos directamente en las diferencias entre los niveles *Básico* e *Intermedio o Estructurado*. Y más concretamente en este segundo, donde las exigencias y requisitos son los ya expuestos en el primer apartado de este cuarto capítulo.

Planteamiento teórico.

La primera decisión que hubo de tomarse al respecto de este bloque era, cuántos tipos y cuales, debíamos incluir en este segundo nivel. En el *NIVEL BÁSICO*, lo acabamos de comentar, la decisión fue la de tener un único tipo, y además, el más sencillo posible: *tipo int*. Pero, ¿y en este segundo nivel de aprendizaje?

Se barajaron diversas posibilidades La primera fue de la de disponer de dos tipos: *numérico* y *no-numérico*. El primer caso, como su propia denominación indica, englobaría todo tipo de valores numéricos (enteros, reales, racionales, irracionales) y el

segundo todo tipo de texto. Sin embargo, esta clasificación resultaba muy pobre para cumplir con la intención de abordar la enseñanza del concepto "*tipos de variable*".

Por tanto se decidió subdividir el tipo *numérico* en dos géneros: *int* y *double*. Lo mismo ocurrió con los *no-numéricos*: *char* y *String*. E incluso después, se resolvió incluir el tipo *boolean*.

No sería necesario incluir más tipos, porque redundarían en la misma enseñanza. Se podrían añadir otros como *long* o *short*, pero para llegar a entender el concepto, eran suficientes los arriba citados.

Es más, se eliminó uno de los propuestos: el *tipo char*, por considerarse que apenas aportaba riqueza. Es simplemente un caso muy concreto de *String*.

A continuación vamos a adelantarnos en el proceso que seguimos para la construcción de este bloque, el cual sufrió numerosas modificaciones. Sin embargo lo hacemos con el fin de: establecer definitivamente la lista y denominación de todos los tipos considerados al final del proyecto. Como hemos dicho hasta ahora, los tipos elegidos fueron: *int*, *double*, *String* y *boolean*. Sin embargo, y a pesar de estar trabajando con ellos a lo largo de gran parte del proceso, se llegó a la siguiente consideración. Estos tipos, o al menos sus nombres, hacen referencia a los existentes en Java (también en otros lenguajes). Y si recordamos los principios básicos de NWTJava (Ver *4.1.1 Requisitos*), vemos como en el segundo punto hacemos hincapié en que esta herramienta es totalmente independiente de un lenguaje de programación real y concreto. Por ello estaríamos cayendo en una incongruencia en el caso de seguir con dichas denominaciones.

Para alejar al alumno de cualquier idea o razonamiento próximo a Java o a cualquier otro lenguaje, optamos por denominar a los tipos de un modo mucho más genérico, sin referencia alguna a otros lenguajes. Los nombres pasaron por tanto a ser: *Entero*, *Real*, *Cadena de caracteres* y *Booleano*.

Por lo tanto, el esquema a seguir, y que resume todo lo planteado hasta el momento es el siguiente:

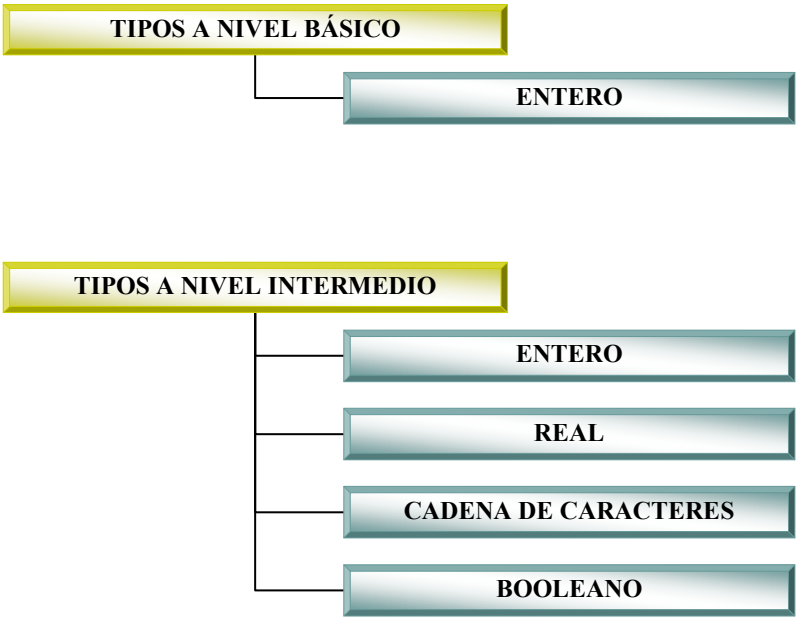


Fig. 4.10

Implementación.

Una vez hemos acordado la nomenclatura precisa, pasamos a describir el proceso de implementación seguido.

Al inicio de este bloque teníamos ya claro que los tipos a implementar eran cuatro. ¿Deberíamos intentar implementar los cuatro tipos al mismo tiempo?, ¿quizás de dos en dos? La mejor opción vino una vez más de la mano del famoso "*divide y vencerás*".

El planteamiento teórico, y el que a la postre se siguió definitivamente, fue el siguiente: ya teníamos totalmente implementado el *tipo Entero*. La descripción de este proceso tuvo lugar en el último guión del *Bloque Otros*. Pensamos pues en aplicar el mismo proceso para uno solo de los mismos, el *tipo Real*. Y a partir de lo inferido en éste, crearíamos el código necesario para los restantes tipos.

Resultó que, ni mucho menos el proceso fue tan trivial como a priori parece. De hecho comentaremos la multitud de problemas y dificultades que hubo que resolver para alcanzar el objetivo deseado. Estos conflictos aparecieron tanto en la primera fase del bloque (inclusión del segundo tipo de variable) como en la segunda (inclusión del resto de tipos).

La aparición de esta gran afluencia de obstáculos encontrados nos parece ahora evidente. La implementación del antiguo código estaba estructurada para la única existencia de un tipo de variable. De ahí que los cambios realizados fueran mucho más estructurales y profundos de lo esperado.

Indexamos a continuación el proceso seguido para la inclusión, primero del *tipo real*, y después de los restantes:

Implementación de los *tipos de variable*:

a. Tratamiento de la clase *Variable*: esta clase debe en estos momentos hacer una distinción entre dos tipos.

b. Funcionalidad de la estructura *ASSIGN*[69]: del mismo modo que, mediante la inclusión del *tipo Real* pretendíamos inferir los conocimientos necesarios para implementar el resto de tipos, aquí, e inicialmente, trabajaremos únicamente con dicha estructura.

1. Cambios a nivel de interfaz gráfica.

2. Validación.

3. Ejecución.

4. Traducción.

[69] La estructura o pieza ASSIGN es la encargada de las asignaciones, es decir, aportar un valor a la variable indicada.

c. Funcionalidad del resto de estructuras.

d. Repetición de los pasos a, b y c para los tipos C*adena de caracteres* y *Booleanos*.

Desglosamos los citados apartados:

a. Tratamiento de la clase *Variable*

Inicialmente esta clase era tan sencilla como se muestra a continuación:

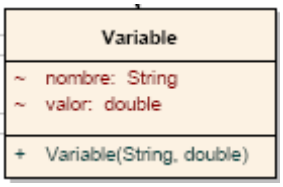


Fig. 4.11

Bastaba con un atributo *String* para almacenar el nombre de la variable y otro *double* para el valor asignado. Volvemos a ver aquí la incongruencia entre el manejo de variables *doubles* y su posterior traducción a variables *int*.

Pues bien, el primer paso fue añadir un nuevo atributo, *double valorDOUBLE*. Lógicamente modificamos el ya existente. Pasamos de *double valor* a *int valorINT*. Así teníamos ya la distinción entre variables de un tipo otro. Aún así, incluimos un cuarto y nuevo atributo: *int tipo*. Su valor indica de qué tipo es la variable. Su rango de valores es:

0: ninguno (pertenece al *NIVEL BÁSICO*) *; 1: Entero ; 2: Real*

También añadimos un nuevo constructor. Hasta la fecha era solo uno:

public Variable (String nombrecito, double valorcito[70]*)*

Se limitaba a recibir un nombre y un valor *double*. Ahora la mecánica del programa exigía otra estructura diferente. Pasamos así a tener dos constructores:

public Variable (String nombrecito, int valorcito)
public Variable (String nombrecito, double valorcito)

De esta forma, NWTJava v2.0 ya era capaz de discriminar cuando a una variable se le asignaba un valor entero o uno real.

Hasta aquí el *punto a*, pero no podemos abandonarlo sin antes comentar que, debido al cambio sufrido por el antiguo atributo *valor* (sustituido por *valorINT* y *valorDOUBLE*), la clase *Ejecutor*, entre otras, no compilaba correctamente. Así que, aunque parece un salto en cuanto a las fases enumeradas, el siguiente paso natural era ir a *b3 Ejecución* y aplicar la siguiente modificación: todas las

[70] Los nombres de atributos y otros muchos elementos han sido reutilizados en esta nueva versión (clases, constantes, etc)

invocaciones al atributo *variable.valor* eran ya erróneas, por lo que sus apariciones fueron sustituidas por las siguientes líneas de código (pseudocódigo para mayor facilidad).

```
if (nivel = = NIVEL BASICO)
        variable.valorINT

else (nos encontramos en un nivel diferente de básico){

if (valor es tipo int)
        variable.valorINT
        if (valor es tipo double)
        variable.valorDOUBLE

}
```

b. Funcionalidad de la estructura *ASSIGN*.

1. Cambios a nivel de interfaz gráfica.

Lo primero que tuvimos que hacer fue distinguir las estructuras[71] que requerían el nuevo parámetro *TIPO*. La frontera que marcaba un campo u otro era:

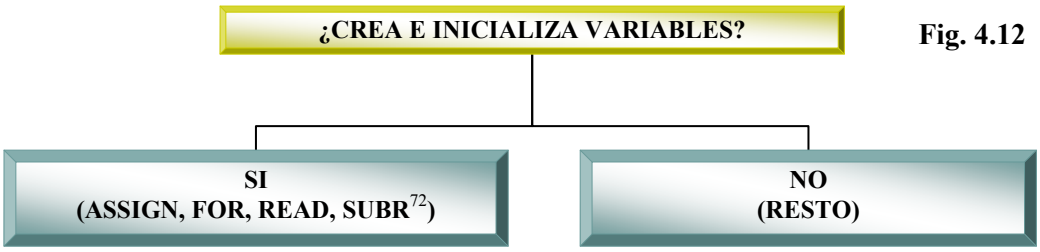


Fig. 4.12

Una vez clasificadas las estructuras que requerían este parámetro, la forma de incluirlo era la siguiente: en el método *initAtts* de la clase *UserObject*, cada estructura tiene un listado de sus correspondientes parámetros. Bastaba con incluir el "*tipo*" pero siempre con la condición de que fuera a nivel distinto de *BÁSICO*.

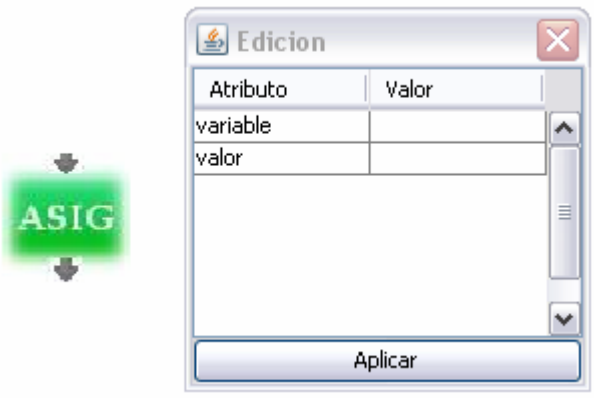
---

[71] Como ya sabemos, con el término "*estructura*" nos referimos a nuestros *FOR, IF, ASSIGN*,… y con "*parámetros*" a los requisitos que exige cada uno de ellos (*FOR*: variable, inicio, final, incremento; *IF*: condición; etc.)

[72] Veremos esta estructura *SUBR* en el *Bloque Subrutinas*.

Una vez hecho ésto, y una serie de cambios más (como el orden de invocación de algunos métodos en el *main*) que no merece la pena comentar, nos metimos de lleno en la implicación gráfica de este bloque.

Dicha implicación se ve reflejada en las *Tablas de Edición* de las correspondientes estructuras afectadas. En estas tablas se muestran los atributos que debe rellenar el usuario. Analizamos la del *ASSIGN*:



Vemos como antaño bastaba con asignarle *nombre* y *valor* a la variable. A partir de ahora también será necesario el *tipo*.

Fig. 4.13

Pero, ¿cómo añadir este parámetro a la *Tabla de Edición*? Múltiples fueron las opciones:

- Añadir una entrada más a la tabla, al modo de "*variable*" o "*valor*", donde el usuario escribiese el *tipo*. Sin embargo esta opción fue desechada por dos motivos. El primero que el usuario podría no saber que tipos son los admitidos ni en esta versión ni en este nivel. Además, aún sabiéndolo existiría una gran fuente de error en esa libre escritura (ej: ¿en mayúsculas?, ¿sólo la inicial?, ¿todo en minúsculas?, ¿qué pasaría si en lugar de "*Entero*" escribiese "*Entierro*"?...).
- Para minimizar esta posibilidad de error pensamos en añadir a la tabla una serie de *JRadioButton*[73] (ver *Fig. 4.14*) con los correspondientes nombres de los tipos. Así el usuario seleccionaría el deseado sin más.

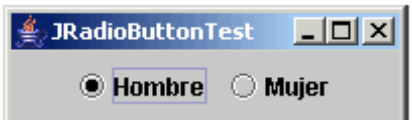


Fig. 4.14

- Pero sin duda la mejor opción para estos casos es la utilización de un *JComboBox*[74]. Consiste en un desplegable capaz de mostrar una lista de elementos deseados. Así, por defecto, aparece "*Tipo:*" al modo de "*Variable*" y "*Valor*", pero al pinchar sobre él, se despliega la lista, pudiendo el usuario elegir y seleccionar el adecuado de entre *Entero* y *Real* (más *Cadena de caracteres* y *Booleano* en un futuro).

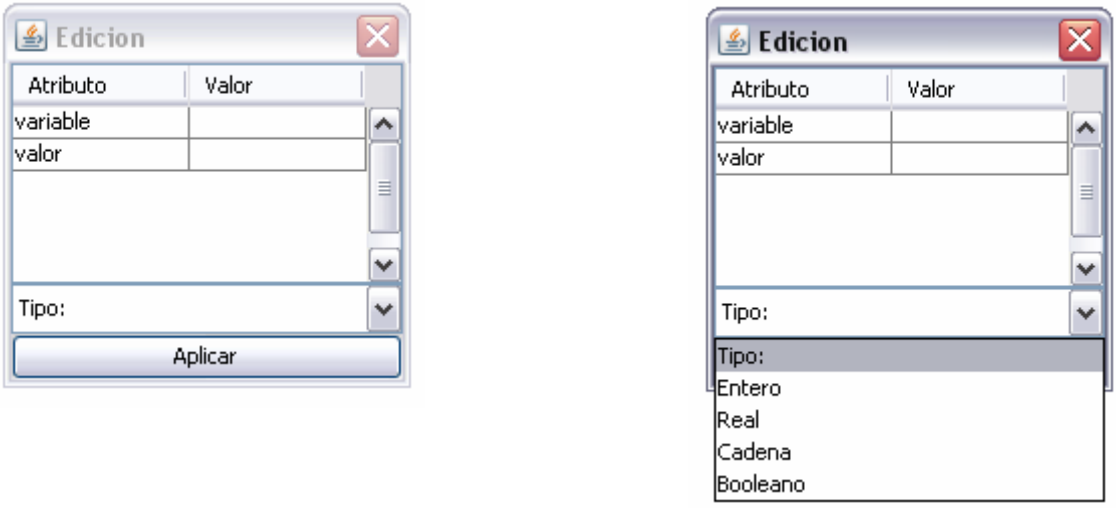
[73, 74] Documentación de las clases de *Swing* en referencia [14].

Vemos el resultado de todo lo comentado hasta el momento mediante una comparativa:

Antes:

Fig. 4.15

Ahora:



El código implementado para este apartado corresponde casi exclusivamente a *editAtt* de *NWTJava*. Como ya podemos intuir, este método se encarga de la edición de las propiedades asociadas a un nodo del grafo en uso. Para ello genera un cuadro de diálogo que contiene la tabla editable donde aparecen las propiedades contenidas en el objeto *UserObject* (la pieza). Se utiliza para permitir la modificación por parte del usuario de las propiedades de cada nodo del grafo.

Sin embargo, y a pesar de que las bibliotecas gráficas de Java (*AWT*, *Swing*...) están muy claramente estructuradas y explicadas en sus correspondientes documentaciones, en la práctica, su uso es bastante engorroso.

Resultó muy arduo y algo desesperante la inclusión de los *JComboBox* en aquellas tablas (de edición) que lo requerían, sobretodo por el hecho del posicionamiento y dimensionado de los elementos.

Es por ello que este apartado *b1 del Bloque Tipos de Variables* supuso una gran inversión de tiempo.

No obstante, una vez superados todos los obstáculos, y añadidos los escuchadores necesarios, la información recogida sobre el *tipo* era ya tangible y manejable. Debíamos por tanto pasar a tratarla desde el punto de vista de la validación, y posteriormente de la ejecución. Pasamos pues al siguiente punto.

2. Validación.

Esta fue la primera vez que nos topamos con el proceso de *validación*. En un principio, muchas de las comprobaciones realizadas en este proceso fueron incluidas en el método *retrieveAtt* de *NWTJava*, por tener una estrecha relación con el método *validar* de *GraphManager*.

No obstante, a medida que avanzaba el proyecto, el aprendizaje y los conocimientos sobre el código base aumentaron rápidamente. Ésto nos permitió, no sólo progresar sino corregir y perfeccionar el código implementado anteriormente.

Por ello, una vez citado dicho problema inicial, pasamos a lo que fue su correcta y adecuada resolución.

Es en el método *validar* de *GraphManager*, como su propio nombre indica, donde se realiza prácticamente la totalidad de las comprobaciones pertinentes. Pero no todas. Aproximadamente un 5% de ellas las realiza *retrieveAtt* de *NWTJava*. ¿Qué clase de pruebas son las que realiza este método? Si recordamos, la clase *NWTJava* es la que soporta la mayor parte de las funciones gráficas del programa, resulta casi evidente decir, que se encarga de supervisar la correcta validación de este ámbito. Lo vemos muy claramente con el siguiente ejemplo:

> Construimos un bucle *FOR*, por tanto trasladamos la pieza correspondiente al *Área de Dibujo*. Abrimos su *Tabla de Edición* para aportarles valor a sus correspondientes parámetros.
>
> *Variable*: i - *Inicial*: 0 - *Incremento*: 1 - *Final*: ____
>
> Sin embargo, sin rellenar el campo *Final*, pulsamos el botón "*Aplicar*" y validamos.

El proceso interno llevado a cabo por la validación es analizar primero si existe alguna anomalía o irregularidad desde el punto de vista gráfico. En este caso existe: uno de los campos de una *Tabla de Edición* consta como "*vacío*". Por tanto no es necesario pasar a las comprobaciones gramaticales (*validar* de *GraphManager*), se ha detectado un error de índole gráfico → *Diagrama No Válido*.

Es por tanto de este tipo de comprobaciones de las que se encarga *retrieveAtt*. Podemos así decir, que se trata de una primera etapa o filtro del proceso de validación.

Ni que decir tiene, que no todos los campos deben ser rellenados obligatoriamente. Un ejemplo de ello es el parámetro *Texto* de la pieza *WRITE* (estructura análoga al "*System.out.println( )*" de Java).

Bien, pasamos pues al proceso puro de la validación, al enfoque gramatical. Por ser esta la primera vez que hablamos de dicho proceso, describiremos como hemos estructurado el método *validar*. De esta forma, y a partir de ahora, nos será mucho más sencillo entender su funcionalidad, tanto para este bloque como para los restantes.

Dicho método divide su código en pequeños bloques, cada uno de ellos específico a cada pieza del programa (seguiremos centrándonos en la estructura *ASSIGN*). Y más concretamente se analiza cada uno de sus parámetros, los valores introducidos por el usuario. Para ello, *validar* invoca y delega en una red de métodos auxiliares que son los realmente encargados de comprobar la validez de dichos valores. Cada uno de estos métodos está muy especializado en un parámetro concreto. Mencionamos algunos:

- *comprTipo(String tip)*: Método auxiliar de validar(). Hace las comprobaciones pertinentes sobre el atributo "tipo".

- *comprValor(String val)*: Método auxiliar de validar(). Hace las comprobaciones pertinentes sobre el atributo "valor".

- *comprCondicion(String con)*: Método auxiliar de validar(). Hace las comprobaciones pertinentes sobre el atributo "condicion".

Un método por cada uno de los parámetros existentes en la aplicación.

Las pruebas que nos conciernen ahora, son las relativas al *tipo*, e indirectamente, y como veremos a continuación, las del atributo *valor*.

Gracias a que la decisión adoptada sobre cómo el usuario debía tipificar la variable, fue la del *JComboBox*, nos ahorramos todo el tratamiento del texto introducido por el usuario. En estos momentos, las opciones eran exclusivamente: "*Entero*", "*Real*" y "*Tipo:*". Lógicamente, aquí el único

error posible era el de seleccionar "*Tipo:*", pues esta opción es la de por defecto, la de no seleccionar ningún tipo.

El mensaje de error sacado por pantalla debido a este caso es:

"El TIPO de las variables debe ser Entero o Real."

El mayor trabajo, a parte de la reestructuración del método *validar*, fue el que describimos a continuación. Dependiendo del tipo seleccionado, el valor de la variable introducido, ya no podía ser cualquiera, debía responder a unas normas. Estas reglas, evidentemente, son intrínsecas al tipo elegido.

El listado de comprobaciones que citamos a continuación crecerá a medida incluyamos nuevos tipos y avancemos en los sucesivos bloques. Hasta el momento son:

- Si el tipo seleccionado es "*Entero*", el valor no puede corresponder con uno "*Real*". Para esta prueba, aplicamos el operador *módulo* (*%*) al valor introducido. Si el resultado de la operación es distinto de 0 → Error.

  "No existe concordancia entre el TIPO y el VALOR de las variables."

- Si el tipo seleccionado es "*Real*", el valor abarca tanto a los "*Entero*" como a los "*Real*".

- Si el tipo seleccionado es tanto "*Entero*" como "*Real*", y en la comprobación del valor salta excepción, es que el término introducido no es numérico → Error. Ej: tipo "*Real*", valor "*Hola*".

  "No existe concordancia entre el TIPO y el VALOR de las variables."

Por el momento es suficiente con esta explicación. Iremos adentrándonos más en la validación a medida que desarrollemos los diferentes bloques. Pasamos pues al proceso de ejecución.

3. Ejecución.

Una vez el usuario ha construido su diagrama, ha pulsado el botón de *Validar*, y el proceso ha devuelto un: *Diagrama Válido*, el siguiente paso a dar es la ejecución.

Como ya comentamos en el *Capítulo 2: Estado del Arte*, la clase *Ejecutor* es la principal del módulo encargado de este proceso.

Para el funcionamiento de lo descrito hasta el momento, no fue necesario realizar grandes cambios en esta clase, simplemente los ya comentados al final del apartado *a. Tratamiento de la clase Variable*.

Este hecho se debió a que en la anterior versión ya estaba permitida la operación entre variables de diferente tipo, aunque con muchas limitaciones. Sin embargo, como el caso que nos ocupa, tipos "*Entero*" y "*Real*" son bastante compatibles, no hubo mayores dificultades.

Estudiamos los casos posibles:

- Operación entre "*Enteros*" → Ningún problema.
- Operación entre "*Reales*" → Ningún problema.
- Operación entre "*Enteros*" y "*Reales*"
  → Ningún problema cuando la variable que almacena el resultado es "*Real*".
  → Ningún problema cuando la variable que almacena el resultado es "*Entero*" (resultado cortado).

Será en el apartado *d. Repetición de los pasos a, b y c para los tipos Cadena de caracteres y Booleanos* cuando comencemos a comentar importante y profundos cambios en el Módulo de Ejecución.

4. Traducción.

La estructura y funcionamiento del Módulo de Traducción también lo comentamos en el *Estado del Arte*. Sus principales clases son *Traductor* y *Plugin*.

Retomamos las últimas líneas referentes a dicho módulo:

"Por lo tanto, tenemos que *Traductor* se encarga de almacenar y gestionar el texto de la traducción. Sin embargo quien implementa la traducción en sí misma es la clase *Plugin*. El proceso es el siguiente: un objeto *Traductor* utiliza el objeto *Plugin* que se le pasa para generar el texto correspondiente al diagrama ejecutado.

El objeto *Plugin* se encarga pues de generar las palabras adecuadas y la indentación de las líneas para el caso de Java. Para traducir a un lenguaje distinto, bastaría con generar otra clase que implementase la indentación de las líneas y las palabras adecuadas a ese lenguaje. Ésto es precisamente lo que proporciona gran flexibilidad a la estructura de traducción."

Pues bien, no sólo para traducir a otro lenguaje distinto bastaría con crear otra clase *Plugin*, también es muy útil y práctico para cada uno de los niveles. Ésto se debe a que cada nivel tiene una entidad suficiente como para asociarse con una clase *Plugin* propia. Nos podemos a continuación preguntar: ¿y por qué cada nivel? Por la utilización de diferentes estructuras, y a que, a las comunes, se les proporcionan usos diferentes.

Lo vamos a ver más claramente con la siguiente explicación: a *NIVEL BÁSICO* las variables son sólo *Variables* (objetos de tipo *Variable*), y de tipo

"*Entero*". La traducción de esta estructura a lenguaje Java es tan simple como:

int nom_variable = valor;

En el *NIVEL INTERMEDIO* una variable puede ser tanto un objeto *Variable* como un objeto *Array*, de tipos *Entero, Real, Cadena de caracteres* o *Booleano*, y además, tratarse de una variable local en una subrutina determinada. La traducción de esta estructura ya no es tan trivial como la anterior.

Por todo ello, creamos e implementamos la nueva clase *PluginINTERM*. Se encarga pues de generar las palabras adecuadas y la indentación de las líneas para el caso de Java a *Nivel Intermedio*.

Sin embargo, la creación de esta nueva clase únicamente suponía la punta del iceberg.

El origen del proceso de traducción tiene lugar en la clase *Datos*, y más concretamente en su método *iniciaTraductor*. Y fue aquí donde topamos con la mayor dificultad de este apartado. Dicho método, lo primero que hace es, dependiendo del nivel seleccionado, cargar el plugin adecuado (para nuestro caso, la clase *PluginINTERM*). En segundo lugar, además de una serie de procesos que no merece la pena comentar, obtiene todas y cada una de las piezas del diagrama (*FOR´s, IF´s, WHILE´s*,…), pasándoselas a *Traductor* para que pueda hacer después la conversión de cada una de estas estructuras a Java. Y por último, y más importante para este bloque, crea y rellena un array de *Strings*, donde lo almacenado en cada posición son los *nombres* de cada variable.

Ésto supuso un gran problema, pues de esta forma, apenas conseguíamos información de las variables para su traducción.
La solución inicial fue crear otro array, paralelo al de los nombres, donde el contenido de sus posiciones era el *tipo*. La misma posición de ambos arrays hacía referencia a la misma variable.

Sin embargo, y con vistas a futuros bloques, esta solución no era la más óptima. Se decidió finalmente sustituir estos dos arrays por un único vector. Un vector que almacenase *Variables*, pero NO SOLO VARIABLES, también, y en un futuro próximo contendría objetos *Array*. Por ello la estructura idónea era un vector. En él, y al contrario que en un array, podríamos almacenar todo tipo de objetos, lo cual nos proporcionaba una enorme flexibilidad y dinamismo.

De esta forma, almacenando objetos *Variable*, ya podríamos acceder de forma rápida y sencilla a las características de las variables: *nombre*, *tipo*, e incluso *valor*, por lo que también modificamos la forma que tenían *Traductor* y *Plugin* de obtener dichos valores.

Llegados a este punto, ya éramos capaces de seleccionar *Nivel de aprendizaje*, escoger el *INTERMEDIO* y trabajar en una interfaz de usuario totalmente actualizada. Podíamos construir un diagrama, y utilizar ya variables de diferente tipo, *entero* o *real*, dependiendo de lo seleccionado en una lista desplegable. Estaríamos asegurados ante posibles fallos de validación y podríamos ejecutar dicho diagrama con total seguridad. Y además, ya no toda variable sería traducida e inicializada como *int*. Podríamos también analizar el número de instrucciones utilizado mediante el *Área de Detalles*.

No obstante, la única pieza con la que podríamos manejar variables de tipo *Real* era *ASSIGN*. Por ello, y una vez conocido el proceso que debíamos seguir para la inclusión de tipos en el resto de estructuras, pasamos al siguiente punto, cuya temática es precisamente esa.

c. Funcionalidad del resto de estructuras.

Como ya comentamos, y vimos en la *Fig. 4.12*, no todas las estructuras son susceptibles de seleccionar tipos de variable. Por ejemplo, no tiene ningún sentido que la pieza *WHILE* pueda seleccionar tipo pues no crea ni inicializa variables, su único cometido es el de evaluar una condición.

Por tanto, la lista de piezas que si lo requieren, acordada ya en la figura citada son: *ASSIGN, FOR, READ* y *SUBR*. Como el *ASSIGN* ha sido la estructura utilizada hasta el momento y SUBR no la veremos hasta el tercer bloque, la lista se reduce a las piezas *FOR* y *READ*.

Podemos decir que, para estas dos estructuras, el proceso fue prácticamente análogo al seguido con *ASSIGN*. Sobretodo con *READ*, cuya única diferencia conceptual con *ASSIGN* es que *READ* solicita los datos al usuario en tiempo de ejecución (es básicamente una lectura de teclado).
Cierto es que cada pieza presenta sus singularidades, y sobretodo la estructura *FOR*, muy particular con respecto a las demás. Pero las dificultades acontecidas por cada una de estos casos nos adentrarían en un bucle de un sinfín de pequeñas pegas y conflictos que nos harían, sin duda, difuminar y perder la finalidad de este texto científico-técnico.

Por este motivo pasaremos al siguiente apartado, con mucho más contenido práctico y trascendente.

d. Repetición de los pasos a, b y c para los tipos C*adena de Caracteres* y *Booleanos*.

Gracias al óptimo funcionamiento de la división entre tipos *Entero* y *Real*, el procedimiento a seguir para la inclusión de estos dos nuevos tipos era claro. Sencillamente sería el mismo. Sin embargo, este camino sería equivalente tan sólo en las primeras etapas, como veremos a continuación.

El primer paso, el denominado *a. Tratamiento de la clase Variable* se resume sencillamente en las modificaciones necesarias para incluir ambos tipos. Para ello incorporamos los atributos *String valorCADENA* y *double valorBOOLEANO* cuya función es idéntica al ya incluido anteriormente *double valorDOUBLE*: almacenar el valor de la variable dependiendo del tipo atribuido por el usuario. De esta forma, la lista de los principales atributos de la clase *Variable* queda conformada de la siguiente manera:

```
String nombre; //Nombre de la variable.

int valorINT; //Valor cuando la variable es de tipo INT.

double valorDOUBLE; //Valor para variable tipo DOUBLE.

String valorCADENA; //Valor para variable tipo STRING.

double valorBOOLEANO; //Valor para variable tipo BOOLEAN.
```

Quizás el lector se haya percatado de una anomalía en la declaración de estos atributos. Es la siguiente: al igual que los valores *Entero*, *Real* y *Cadena* son definidos por sus respectivos tipos Java (*int*, *double* y *String*), el atributo *valorBOOLEANO* es, por el contrario de tipo *double*. Comentaremos el porqué de esta alteración cuando tratemos el proceso de ejecución en este mismo apartado, precisamente cuando surgió el problema al que de esta forma dimos solución.

Para acabar con el capítulo de modificaciones de la clase *Variable*, simplemente comentar que, al igual que para el tipo *Real*, también aquí creamos un constructor para cada uno de los tipos aquí estudiados.

public Variable (String nombrecito, String valorcito)
public Variable (String nombrecito, boolean valorcito)

En cuanto a los cambios relativos a la interfaz gráfica (punto *b1*) podemos decir que fueron harto sencillos, pues debido a que la estructura del listado desplegable de tipos (*JComboBox*) y sus correspondientes *listener* estaban ya dispuestos al 100%. Bastó con añadir a dicha lista los términos "*Cadena*" y "*Booleano*".

Pasamos pues sin más dilación al punto *b2. Validación*, bastante más interesante desde el punto de vista tanto práctico como conceptual.

Para esta fase fue necesario plantearse el siguiente razonamiento: ¿cuál sería la forma de expresar las cadenas de caracteres? En primer lugar pensamos que tal cual, es decir, sin comillas "" o indicación similar. Bastaría con introducir valores tanto numéricos como no numéricos y seleccionar este tipo. Pero tras probar durante un breve período de tiempo con esta opción, nos decantamos por la alternativa. Suponía una ventaja trabajar con comillas, y las razones que avalan esta decisión fueron las siguientes:

- Facilita la búsqueda y tratamiento de errores.

- Acentúa la sensación de estar trabajando con variables de carácter no numérico.

- De forma visual, permite un reconocimiento rápido y claro del tipo utilizado.

  ¿Cómo sabríamos si el carácter, digamos 8, es de tipo *Entero* o *Cadena de caracteres*?

- *JEP* es capaz de manejar *String* siempre y cuando éstos se encuentren entre comillas.

Este último punto declinó finalmente la balanza hacia esta opción. El que *JEP*[75], del que hablaremos en breve, pudiese también manejar el tipo *String*, añadía a NWTJava v2.0 la posibilidad de operar con variables de este tipo. Este hecho no estaba contemplado en el primer planteamiento, sin embargo, no pudimos rechazar la oportunidad de añadir aún más funcionalidad a esta versión.

En estos momentos, no solo éramos capaces de definir y crear variables de tipo *String* (o *Cadena de caracteres*), sino que también podíamos realizar operaciones con ellas. Operaciones como la de concatenar sus valores.

Una vez comentado el criterio a seguir para el manejo de *Cadenas de caracteres*, pasamos a lo que realmente supuso adecuar el proceso de validación a estos nuevos tipos.

Una vez más acudimos al método *validar* de *GraphManager*. Y más concretamente a sus métodos auxiliares *comprTipo* y *comprValor*.

El procedimiento es el siguiente:

- *comprTipo* analiza si en la pieza procesada se ha seleccionado un tipo o no. Sino no es así, mensaje de error correspondiente a la ausencia de tipo en una variable.

- Si la selección ha sido válida, *comprValor* comprueba si el valor introducido por el usuario corresponde con el tipo seleccionado y su rango de valores.

La comprobación asociada al tipo *Cadena de caracteres* es que el término introducido se encuentre entre comillas. Todo aquello que cumpla este requisito es susceptible de ser una cadena de caracteres.

---

[75] JEP (Java Expresión Parser). Para la evaluación de expresiones, utilizamos las clases de esta librería externa, cuya descripción y documentación se pueden encontrar en [16]. El objeto JEP que proporciona esta librería es un parser-evaluador eficiente y flexible, que permite sobradamente las operaciones aritméticas y lógicas que maneja esta aplicación.

En cuanto a las comprobaciones pertinentes para el tipo *Booleano*, nos basamos en que sólo dos valores son posibles o válidos: *true* y *false*.

Es en el proceso de ejecución, punto *b3*, donde se dio una profunda diferenciación entre el tratamiento de las *Cadenas de caracteres* y los *Booleanos*. No en el fin, que era coincidente: poder manejar y operar con ambos tipos de variable en los diagramas NWTJava. Pero sí en el tratamiento de su ejecución.

Hasta ahora ambos tipos habían seguido vías paralelas (en los cambios de la clase *Variable*, interfaz gráfica, las comprobaciones pertinentes en validación). Sin embargo, la clase *Ejecutor* los maneja de distinta forma. Y ésto es debido al tratamiento que hace *JEP* sobre cada uno de estos tipos.

Para entender ambos procedimientos, lo primero que vamos a hacer es describir el funcionamiento de *JEP* y su función en nuestra aplicación:

El objeto *JEP* mantiene un registro interno de duplas variables-valor que emplea en la evaluación de expresiones que contengan variables. Para realizar una evaluación, tan sólo requiere que se le proporcione un objeto *String* con la expresión a evaluar (ej: "a+b/8") al método *parseExpression*, y luego, mediante el método *getValue*, se recupera el resultado. No es necesario notificar al objeto *JEP* el tipo de expresión, ya que lo obtiene automáticamente al parsearla. Si la expresión es aritmética, *getValue* devolverá un *double* con el valor resultante; si se trataba de una expresión lógica, devolverá el valor *double* "1.0" para representar el *true*. Como vemos, dada la simplicidad de su uso, su robustez y flexibilidad, resulta mucho más rentable para el proyecto la utilización de esta librería que la implementación de un evaluador propio.

La clase *Ejecutor* contiene pues un objeto *JEP* y se encarga de proporcionarle las variables necesarias para la evaluación de las expresiones que se encuentre. Para ésto y para otras funciones como la lectura o la escritura, el *Ejecutor* mantiene a su vez un repositorio con las variables que se utilizan en el programa de usuario que está en procesamiento, por supuesto en forma de objetos *Variable*.

Pues bien, es ahora cuando podemos contemplar y entender las diferencias que supone el tratamiento de estos nuevos tipos, tanto con los ya existentes (*Entero* y *Real*) como entre sí (*Cadena* y *Booleano*).

Comenzaremos por el tipo *Cadena de caracteres*, asociado como ya sabemos, a los *String* de *JEP* o Java.

La clase *Ejecutor* presenta un método para cada una de las piezas existentes.

assign(String[] línea)
read(String[] línea)
fin(String[] línea)

Dichos métodos son los invocados cuando a la hora de ejecutar un diagrama, la clase *Datos*, la cual maneja el puntero de programa, informa a *Ejecutor* de cual es la siguiente pieza a ejecutar.

Pues bien, lo primero que hicimos fue repasar estos métodos, uno por uno e identificar que partes del código hacían referencia a los tipos *Entero* y *Real* para, a continuación, añadir el bloque referente al tipo *Cadena*. Pero es aquí donde viene la peculiaridad.

Como hemos comentado en el párrafo explicativo de *JEP*, el método que devuelve el resultado de las operaciones es *getValue*. Sin embargo, este método devuelve un valor *double*, totalmente incompatible con los valores *String* necesarios para este tipo.

Pues bien, adelantamos ya el hecho de que la versión de *JEP* utilizada hasta el momento era obsoleta para los requerimientos de esta nueva evolución de NWTJava. Comentaremos el cambio realizado en el siguiente bloque donde fue del todo decisivo. Hasta entonces lo que podemos decir es que esta versión de *JEP* no contaba con los métodos necesarios para el manejo de *String*, al menos para presentar un nivel operativo tan alto como el de las variables numéricas.

Tras este encontronazo con *JEP*, añadido a algunos otros que comentaremos en su momento, fue por lo que se decidió actualizar esta librería.

Ahora sí. Teníamos ya un método equivalente a *getValue* capaz de trabajar con todo tipo de objetos al nivel de las variables numéricas: *getValueAsObject*. Tras las comprobaciones oportunas en cada uno de los métodos de *Ejecutor*, parsear la expresión y operar con ella, éramos capaces de obtener resultados tanto de tipo *Entero* y *Real* como *Cadena de caracteres*.

Para finalizar la descripción del proceso de ejecución con respecto a las variables *Cadena*, queda reseñar que queda totalmente excluida la posibilidad de operación entre este tipo y cualquiera que sea numérico. Si el alumnos intentase ésto, ya sea por inconsciencia, prueba o equivocación, por pantalla saldría el correspondiente mensaje de error.

```
"Error debido a:

      - realizar una OPERACIÓN NO PERMITIDA para el TIPO de datos
      utilizado"
```

Vemos en estas figuras ejemplos de utilización de variables:

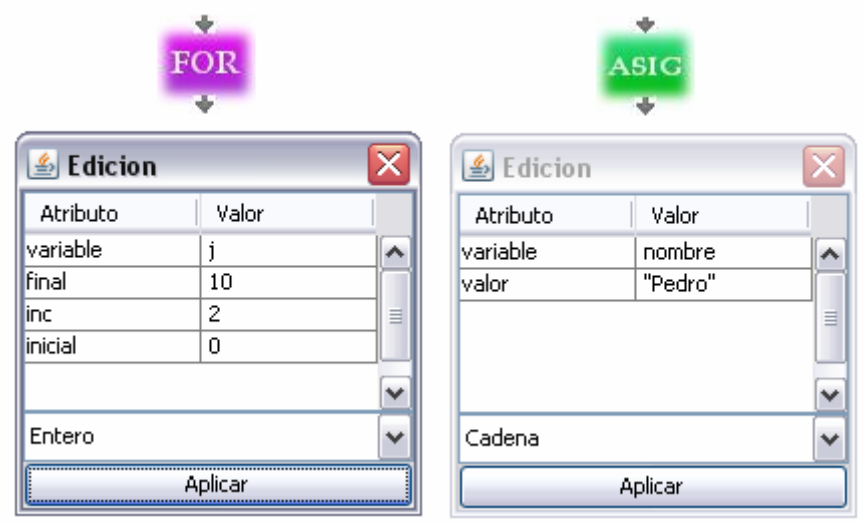


Fig. 4.16

Pasamos a describir el tratamiento de adecuación del tipo *Booleano* al proceso de ejecución. Anteriormente citamos la disonancia existente entre la definición de los atributos de tipo *int*, *double* y *String* con respecto al *boolean*. Éste último era declarado de tipo *double*, y es aquí donde contamos el porqué de esta anomalía.

La instrucción

"*myParser.addVariable(variable.nombre, variable.valorBOOLEAN)*"

no es válida, pues para este método de *JEP*, su sobrecarga únicamente permite que el segundo de sus parámetros sea un *double* o un *String* (es por ello que no hay problemas con los tipos *Entero*, *Real* y *Cadena*). Ésto conlleva a que no podamos aportar el valor *boolean* necesario a la variable de ese tipo.

Tras un largo proceso de investigación y pruebas, llegamos a la conclusión de que la mejor solución era la siguiente: adaptarnos a cómo *JEP* maneja las operaciones lógicas.

Decidimos pues asociar los términos "*true*" y "*false*" con valores *double*. ¿Cómo hicimos ésto? Pues bien, sabíamos que el tratamiento que hace esta librería con las variables es mediante asociaciones, como ya dijimos antes: almacena duplas variables-valor. Por tanto, nosotros le pasaríamos la información booleana empleando el mismo sistema.

Estábamos al corriente de que *JEP*, en operaciones lógicas, devolvía el valor *double* "1.0" en el caso de resultado afirmativo. Pero, ¿y ante un resultado negativo?, ¿sería 0, -1 quizás? Para averiguar esta incógnita, examinamos su documentación, pero nos percatamos de ciertas carencias en la misma (éste caso es un claro ejemplo), por lo que nos vimos obligados a ejecutar un diagrama rico en estructuras *IF*, tanto con condiciones de resultado negativo como de resultado positivo. Analizando los resultados de dicha ejecución a nivel interno,

obtuvimos la conclusión: Resultado afirmativo (*true*) → 1.0; Resultado negativo (*false*) → 0.0

A la vista de los resultados obtenidos actuamos en consecuencia, y la implementación en el Módulo Ejecutor a este respecto fue la siguiente:

Tras inicializar el objeto *myParser*[76], en el constructor de la clase *Ejecutor*, lo primero que hicimos fue crear las duplas con las que trabaja *JEP*. Ello mediante la asociación de los citados términos "*true*" y "*false*" (ahora como nombres de variable) con sus correspondientes valores numéricos. Meridianamente claro queda si vemos la implementación de este proceso:

```
/*
 * para el tratamiento de BOOLEANOS, JEP no acepta trabajar
 * con booleans asi que al leer NWTJava las expresiones
 * "true" y "false" las convierte en un valor double con los
 * que puede trabajar perfectamente.
 */
myParser.addVariable("true", 1);
myParser.addVariable("false", 0);
```

A partir de aquí, el manejo de los valores "*true*" y "*false*", es decir, los booleanos en general, dejó de ser un problema para convertirse en un importante avance con respecto a la versión anterior.

Sin embargo, restaba todavía el añadir esta funcionalidad a cada una de las piezas participantes en este bloque. Para ello, en cada uno de los métodos asociados fue necesario implementar una serie de comprobaciones para dilucidar si el tipo de variable tratado era efectivamente *Booleano*.

De esta forma, el cuerpo principal de estos métodos (*assign*, *read*, *write*,…) quedaba constituido por las pertinentes pruebas, capaces todas ellas de encasillar la variable tratada en su tipo correspondiente.

Una vez implementado todo ésto, más un importante número de pesquisas cuyo fin fue el de afinar y perfeccionar el proceso de ejecución, llegamos al punto de poder realizar un gran número de operaciones con los tipos propuestos. En el caso que nos ocupa, con los *booleans*, son aceptados todos los operadores lógicos habituales (&&, ||,!,...).

En cuanto a la traducción de las variables de tipo *Cadena* y *Booleano* a lenguaje Java, todo estaba ya dispuesto a partir del punto *b4. Traducción*. Gracias a la utilización del *Vector variables*, podríamos acceder de forma rápida y sencilla a las características de las variables: *nombre*, *tipo*, e incluso *valor*, y mediante el objeto *PluginINTERM* traducir sin problemas.

Y es aquí donde damos por concluido la descripción sobre el funcionamiento e implementación del *Bloque Tipos de Variables*. A continuación analizaremos el siguiente de ellos, el *Bloque Arrays*, donde explicamos también aspectos del anterior por estar ambos muy relacionados.

[76] Objeto de tipo *Parser* del evaluador de expresiones (*JEP*).

4.2.3 Bloque Arrays

A pesar de que los arrays son una estructura novedosa e independiente para NWTJava, lo cierto es que su bloque está estrechamente relacionado con el de los tipos de variable. Lo primero que hemos de entender es la visión en conjunto, la nueva incorporación unida a la estructura actual.

Hasta el momento teníamos variables (objetos de la clase *Variable*) definidas por un nombre, un valor y su tipo correspondiente. Pues bien, planteamos el array como una estructura paralela a la ya existente[77]. De esta forma tendremos variables, llamémosla pro el momento, "simples" (objetos *Variable*) y variables array (objetos de la nueva clase *Array*), con una definición análoga a las "simples" (nombre, valor y tipo) pero con unas peculiaridades y singularidades tan importantes y significativas[78] que la hacía merecedora de una clase nueva e independiente.

Por si fuera poco, el tratamiento que se realiza sobre los objetos *Variable* y *Array* a lo largo de los procesos ya conocidos es tan diferente que la necesidad de un nuevo bloque para esta estructura se hacía indispensable.

Pasamos pues, al modo de trabajo seguido para la inserción de los arrays (estructura tan requerida, si recordamos, por los alumnos del seminario TSIOCA).

A estas alturas, todos sabemos lo que es un array: "*referido a la programación, un array es un conjunto o agrupación de variables del mismo tipo cuyo acceso se realiza por índices*[79]".

También sabemos cómo y para qué utilizarlos, pero casi siempre a nivel intuitivo. Atrás quedan las definiciones, los guiones que mostraban las diferentes formas de declaración, los principios y clasificaciones en cuanto a su uso, y un largo etcétera. Sin embargo, para la implementación del código necesario para su manejo y funcionamiento, requeríamos aquellas clasificaciones. Ello nos condujo a la siguiente pregunta: ¿cómo íbamos a definirlos? Existen dos tipos de declaraciones: la que inicializa el array en su propia declaración (la denominaremos "***declaración intensiva***") y la que lo declara sin inicializarlo, aportándole únicamente el tamaño del mismo (la denominaremos "***declaración extensiva***").

Pero, ¿basta con ello?, ¿acaso trabajamos siempre con la estructura completa? No, también utilizamos sus posiciones de forma independiente, el contenido de sus casillas por así decirlo.

---

[77] Anterior a este planteamiento definitivo hubo otro en el que se pensó tratar los arrays como meros objetos *Variable*. Pero ello supuso una gran cantidad de problemas, superados mediante la resolución tomada y que comentamos a continuación.

[78] Por ejemplo, los atributos de *Variable valorINT*, *valorDOUBLE*, etc. tienen su equivalente en los *valorArrayINT*, *valorArrayDOUBLE*, etc. que podían haber sido los mismos si únicamente trabajásemos con la clase *Variable*. Sin embargo, mientras que en *Variable*, estos atributos son de tipo *int*, *double* y *String*, para trabajar con arrays de cara al usuario, es estrictamente necesario que a nivel interno los atributos de *Array* sean de tipo *Vector*. Unido a que esta nueva clase sólo debe funcionar a niveles *Intermedio* y superiores (no como *Variable* que también es obligatoria en *Básico*) y a muchos otros detalles a tener en cuenta, se tomó la decisión de crear la nueva clase.

[79] Definición extraída de la referencia [17], de donde se puede obtener una visión general sobre el concepto "*array*".

Ésto nos llevaba ya a tener que implementar tanto las declaraciones extensiva e intensiva como el uso de posiciones. Y en cuanto a dichas posiciones, ¿cómo accederíamos a ellas? Dos los mecanismos con los que hacemos referencia a una posición de array: mediante un número entero (índice) que la indica o con una variable que almacena dicho índice. De nuevo teníamos otra ramificación de casos.

Finalmente construimos un diagrama capaz de albergar todas estas posibilidades. Dicho esquema o diagrama plantea todo el ámbito de uso que la aplicación es capaz de abarcar en el manejo de arrays. Además divide las fases en que organizamos nuestro trabajo.

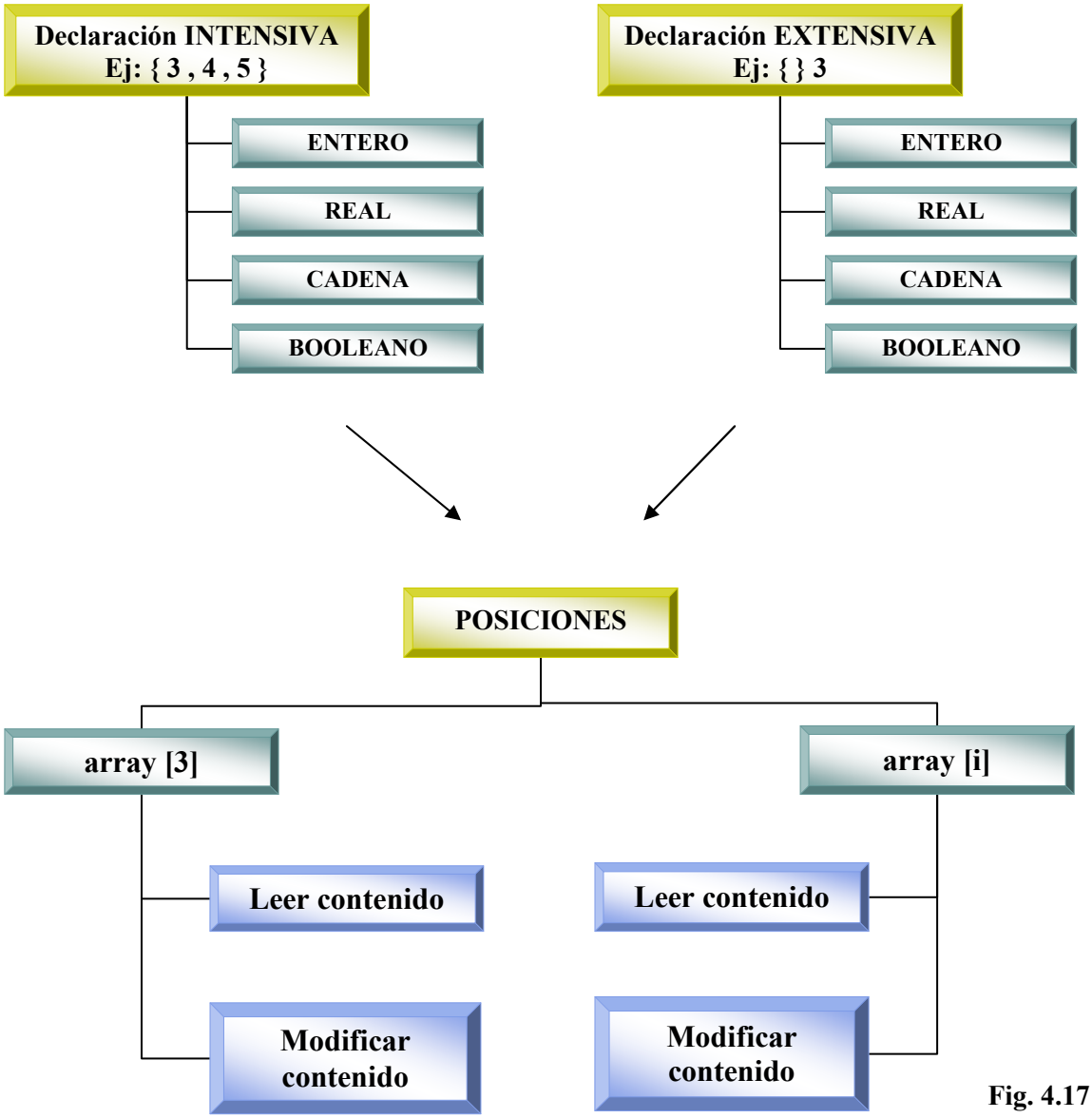


Fig. 4.17

Las grandes fases en que hemos dicho dividir el trabajo corresponden a los títulos encuadrados en las casillas doradas. Son tres: declaración intensiva, declaración extensiva y posiciones, las cuales a su vez se dividieron en subfases.

No obstante, antes de comenzar con la implementación de este bloque, lo primero que hicimos fue evaluar *JEP* a este respecto. Es decir, realizamos una gran cantidad de pruebas con dicha librería en pos de ver el tratamiento que realizaba sobre los arrays.

Pues bien, el resultado de estas pruebas fue nefasto.

En un principio, y sin ningún problema, vimos que *JEP* es con vectores con lo que trabaja. Este cambio no suponía mayor dificultad que la de elaborar un algoritmo conversor de arrays a vectores[80].

Los métodos que debían ser utilizados para estas cuestiones eran *parseExpersion* y *getValue* (métodos en los que, como su propia denominación indica, parsean la expresión y devuelven el resultado). Sin embargo, y tras muchas dificultades, discusiones y búsqueda de soluciones en webs relacionadas con *JEP*, encontramos un ejemplo de manejo de vectores. Los métodos utilizados eran *parse* y *evaluate*, dos métodos no definidos en nuestra versión de *JEP*. Fue este hecho el que nos puso en el camino de un más que probable problema de versiones.

Finalmente encontramos la solución en las siguientes líneas:

"*The package structure has changed significantly. All classes that were previously in the* ***org.nfunk.jep*** *package are now located under* ***com.singularsys.jep****. The organization of the classes has changed significantly as well*."

Efectivamente nuestra versión de *JEP*, *jep-2.23* trabajaba con el paquete *org.nfunk.jep*, el cual no había asumido todavía todo el potencial sobre el manejo de vectores que si proporcionaba *com.singularsys.jep* con sus métodos *parse* y *evaluate*.

Por tanto, y como ya adelantamos en el anterior bloque, era necesario actualizar nuestra librería. Eran dos las versiones de *JEP* que contenían el nuevo paquete: *jep-2.4.1* y *jep-3.2*. Para adquirir esta última era necesario comprar una cuantiosa licencia, por lo que, lógicamente nos decantamos por adquirir la primera, *jep-2.4.1*

Ahora sí, la funcionalidad y manejo de vectores era perfecto, por lo menos para nuestros requisitos. Todas las pruebas anteriormente realizadas daban ahora un resultado óptimo, por lo que, a partir de aquí pudimos continuar con el trabajo previamente establecido.

Retomando el trabajo genérico sobre arrays, y resumiendo que es lo que se pretendíamos de ellos, tenemos:

- Crearlos (definirlos).
- Rellenarlos.

[80] Recordemos que los índices de los arrays comienzan desde 0 mientras que en el caso de los vectores es desde 1. Para ello creamos el método *sincrArrayToVector* que sincroniza las posiciones de los arrays (con los que se trabaja de cara al usuario) y los vectores (con los que se trabaja a nivel interno). Lo que hace realmente es extraer la información almacenada desde la posición array[0] de un array, e introducirla en un vector cuya primera posición es vector[1].

- Manejas sus posiciones.
  - Leerlas (extraer contenido).
  - Modificarlas (cambiar contenido).

Comenzamos con el proceso de cómo crearlos y definirlos:

Declaración INTENSIVA.

Durante un largo período de tiempo se estuvo estudiando y decidiendo cual era la nomenclatura más adecuada para la declaración de arrays en el programa. Sirvieron de guía fundamentalmente los siguientes dos factores:

- Cómo transmitir los conocimientos necesarios para la comprensión y utilización del concepto de array. Factor de cara al usuario.

- Cómo codificarlo en la aplicación. Factor relacionado con el programador.

Una idea fue la de en la *Tabla de Edición*, mediante un *JRadioButton*[81], poder seleccionar la opción de usar un array. Marcando dicho *JRadioButton* saldría por pantalla un mensaje solicitando cada uno de los valores correspondientes al array.

Como ventajas de este planteamiento podemos citar que era una forma sencilla para el alumno. Simplemente marcaría la opción y rellenaría con valores sus posiciones. Otra es que sería fácil crear el código necesario para la salida del mensaje y la introducción de valores en la estructura. Por el contrario, muy diferente sería la dificultad para tratar los elementos gráficos de la interfaz, pues la introducción de estos elementos en la *Tabla de Edición* resulta compleja[82].

Además, como hemos dicho anteriormente, es una forma sencilla para el alumno, pero no intuitiva. Mediante este mecanismo el usuario obtendría la falsa visión de que un array es una estructura muy diferenciada de lo que es una variable, cuando en realidad ésto no es así.

Un inciso a este respecto. Aunque a nivel interno, y como hemos comentado en la justificación de este bloque, separamos los casos del tratamiento de "*variables simples*" y arrays, de cara al usuario es fundamental que esta división no se vea reflejada. Es importante, desde el punto de vista del aprendizaje, que el alumno asocie el array como una variable más.

Dicho ésto, pasamos a comentar el segundo planteamiento. El array sería declarado en la actual *Tabla de Edición* de la siguiente forma: en el parámetro *nombre*, como es

[81] Documentación de las clases de *Swing* en referencia [14].

[82] Recordemos las dificultades que encontramos en el *Bloque Tipos de Variable* para añadir el *JComboBox* en la *Tabla de Edición* (dimensiones, al posicionamiento en su correspondiente contenedor, descolocación de elementos preexistentes,…).

lógico, se escribiría el nombre asociado, y en *valor* añadiríamos su contenido mediante la siguiente nomenclatura

{ dato1 , dato2 , dato3 }

Resultó ésta ser una solución mucho más elegante, además de intuitiva para el alumno, pues se trata a los arrays exactamente igual que a las "*variables simples*". También eliminábamos así el engorroso trabajo de añadir elementos gráficos a la *Tabla de Edición*.

Sin embargo, y como es natural, esta idea no estaba exenta de dificultades. Sería necesario crear un nuevo tratamiento para los valores introducidos en el campo *valor*. En él habría primero que distinguir los casos de "*variables simples*" y arrays, y en función de la decisión tomada, someter al valor concreto a su correspondiente batería de pruebas.

Elegimos este segundo planteamiento, pero no fue la última decisión tomada. Con ella venían asociadas otras que pueden parecer livianas y sin fundamento, pero al fin y al cabo decisivas para comunicar toda la información que pretendíamos enseñar. Ejemplos de estas discusiones fueron los símbolos con los que delimitar el array (en este caso, como vemos unos párrafos más arriba, elegimos las llaves, pero otra posible opción eran los corchetes, símbolos que a la postre utilizaríamos para otro caso), los elementos con los que separaríamos los valores de cada posición (punto y coma, una coma, un guión,…).

Declaración EXTENSIVA.

El mismo proceso sufrió la nomenclatura de la definición extensiva. En Java, dicha definición se corresponde con la línea de código "*tipo[] nombre = new tipo[longitud];* " pero ¿para el caso de NWJava v2.0? La estructura base sería la misma que para la definición intensiva (*Tabla de Edición* sin modificaciones), el *nombre* en su campo correspondiente y el *tipo* se fijaría con el *JComboBox*. Nos faltaba cómo indicar en el campo *valor* que lo creado sería un array vacío con una determinada longitud.

Realizamos un rápido repaso por las principales opciones contempladas. Utilizaremos para ello un ejemplo con el que dichas opciones quedan mejor ilustradas → un array de nombre "*a*" y tamaño 3:

- a(3) : esta primera opción fue la génesis de todas las demás. Fue recogida incluso antes de tomar la definitiva decisión de no modificar la *Tabla de Edición* (recordemos que se ideó seleccionar la "*opción array*" mediante un *JRadioButton*). De hecho, vemos como en su definición incluíamos el nombre del array, cuando ya sabemos que ésto no es necesario pues para tal fin utilizamos el campo *nombre*.

- (3) : en este caso ya excluíamos el nombre del campo *valor* que es el que pretendemos rellenar. Sin embargo, los paréntesis no nos parecieron

los símbolos más adecuados para esta definición por el siguiente motivo que a continuación comentamos.

- {3} : ya que para la definición intensiva habíamos utilizado llaves, porqué no seguir en esta misma línea y conseguir una asociación entre este símbolo y el array. Al usuario le resultaría más rápido y sencillo el reconocer y manejar este tipo de estructura. De lo contrario, y al menos en sus fases iniciales con la herramienta, se vería obligado a elegir entre usar llaves o paréntesis, perdiendo así parte de la simplicidad que pretende NWTJava. Pera esta nueva opción presenta otra seria desventaja: según nuestro modelo de definición intensiva *{ dato1 , dato2 , dato3 }*, ¿cómo distinguir el caso de que {3} es una definición extensiva y no un array de tamaño 1 cuyo valor almaceno es un 3?

- { }3 : finalmente llegamos a esta nomenclatura. Contiene los símbolos que ya en la definición intensiva asociamos con los arrays. Contiene la longitud, y en ningún caso se puede confundir con otro tipo de estructura o definición. Además permitía una secuencia de pruebas de validación muy lógica e intuitiva como veremos en unos instantes. Finalmente fue la nomenclatura escogida.

Decidida ya la representación a seguir para las definiciones intensiva y extensiva, pasamos a describir lo que fue su implementación.

Implementación de ambas declaraciones.

Lo primero que hicimos fue crear la nueva clase *Array*, sobre la que pivota el funcionamiento de todo este bloque. Son los objetos de esta clase los que utiliza la clase *Ejecutor* para realizar las diferentes operaciones permitidas con arrays[83].

Como ya sabemos, el manejo de arrays que aportamos de cara al exterior, en la interfaz gráfica, es idéntico al de cualquier variable visto en el bloque anterior. Este hecho permite que las primeras fases del procesado sean las mismas: creación y manejo en los diagramas construidos por el usuario; el tratamiento de *retrieveAtt*, del que ya comentamos que es un método perteneciente a la clase *NWTJava* cuya función es la de supervisar el correcto funcionamiento del grafo a nivel de interfaz (ejemplo de función: comprobación de que los campos de las *Tablas de Edición* cuyo rellenado es obligatorio cumplan con dicho requisito. Es obligatorio rellenar los campos o atributos *nombre* y *valor* de *ASSIGN*, mientras que no lo es el de *texto* en *WRITE*).

Pero es hasta este punto donde existe el paralelismo. A partir de aquí, si bien los procesos son los mismos, validación y ejecución, el tratamiento en cada uno de ellos es totalmente diferente. De hecho, lo primero que hacen es discriminar entre "*variables simples*" o arrays.

Vamos por tanto a analizar el proceso seguido por la ramificación de los array.

[83] Array siempre desde el punto de vista externo, del usuario, pues internamente ya sabemos que trabajamos con vectores. A la hora de analizar el proceso de ejecución volveremos a tratar este asunto.

Supongamos que el usuario construye su diagrama, lo revisa y a continuación pulsa el botón que da inicio a la fase de validación. Pues bien, una vez realizada la comprobación del método *retrieveAtt* salta a la clase *GraphManager*, más específicamente a su método *validar*. Éste, lo primero que hace es ordenar el diagrama, el grafo, en una lista de piezas (nodos)[84]. Una vez ordenados mediante la instrucción "*datos.ordenaGrafo();*", los almacena en un objeto *NodeList*, y de ahí va extrayendo elemento a elemento y comprobando su validez. De hecho, es como esta estructurado el *validar*, dividido en pequeños bloques correspondientes cada uno a una instrucción de NWTJava v2.0. Pongamos por caso que la primera pieza encontrada (tras el icono *INICIO*, por supuesto), es la ya conocida *ASSIGN*. Entonces, entra en su correspondiente bloque mediante "*if(name.equals("assign"))*" y dependiendo de a que nivel (*BÁSICO* o *INTERMEDIO*) estemos trabajando, invocará unos métodos u otros de lo que ya dijimos que son auxiliares. Como estamos en *INTERMEDIO*, la primera se basa en el *tipo*. Invoca a *comprTipo* (si nos encontrásemos en el *BÁSICO* no haríamos uso del concepto *tipo de variables*, por lo que esta comprobación no tendría lugar). Una vez superadas las pruebas sobre el *tipo* seleccionado con éxito[85], pasaríamos al *comprValor* donde realmente, a la vista del valor introducido por el usuario, se decide si es "*variable simple*" o array. Para la toma de esta decisiva elección, implementamos la siguiente batería de comprobaciones:

- Requisito para ser considerado <u>array de definición intensiva</u> *{ 1 , 2 , 3 }*:

  - Que el *String* almacenado entre los símbolos "*{*" y "*}*" no sea cadena vacía.

- Requisito para ser considerado <u>array de definición extensiva</u> *{ }3* :

  - Que el *String* almacenado en el parámetro *valor* posea los símbolos "*{*" y "*}*".

  - Que lo que haya ENTRE dichos símbolos sea una cadena vacía. Es decir, no puede haber ningún carácter entre las llaves. Sin embargo, por motivos de robustez y comodidad para el usuario, si pueden existir entre 0 e infinitos espacios en blanco. Ejemplos:

    {}3 OK
    { }3 OK
    {r}3 INCORRECTO

  - Que no haya ningún carácter ANTERIOR a la primera llave "*{*".

    { }3 OK
    r{ }3 INCORRECTO

---

[84] Como ya comentamos en c2, el diagrama en determinadas partes del programa funciona como un árbol, en donde las piezas actúan como nodos de dicho árbol.

[85] En caso contrario, existencia de un error, almacenamos su mensaje de error correspondiente, pero seguimos con la validación hasta el final, con la idea de una vez acabado el proceso, imprimir por pantalla todos y cada uno de los fallos cometidos. Sería un fastidio que el proceso se interrumpiese con cada error encontrado, sobretodo para el usuario, que tras encontrarlo, solucionarlo y volver a pulsar validar, volvería a ser interrumpido con cada fallo.

- Que lo encontrado TRAS la última llave "*}*" sea un número, de tipo entero y positivo. De esta forma nos aseguramos que el término que hace referencia al tamaño del array sea válido.

{ }3	OK
{ }3.5	INCORRECTO
{ }a	INCORRECTO
{ }-2	INCORRECTO

Cada una de estas pruebas, es descalificativa, es decir, en el momento en que no cumplan una, queda descartada la definición. Si tras estas pruebas, lo almacenado en el parámetro *valor* no cumple con los requisitos de ninguna de las definiciones, pasa automáticamente al proceso de comprobaciones correspondientes a "*variables simples*".

Pero el proceso de validación no acaba aquí. Con lo realizado hasta el momento, lo único que hemos hecho ha sido conocer si el dato introducido por el usuario es una "*variable simple*" o un array. Y en este último caso, conocer el tipo de definición utilizado para crearlo.

Para la definición extensiva, sí es éste el final del proceso, por el motivo que ahora comentaremos, pero no para la definición intensiva. Aquí es necesario pasar a un nuevo nivel de comprobaciones: si los valores almacenados en sus posiciones corresponden con el tipo seleccionado. Lógicamente este nivel de comprobación no lo alcanzamos con la definición extensiva, pues no hay valores almacenados que analizar.

Para esta nueva batería de pruebas es necesario realizar antes un pre-procesado. El valor introducido mediante la *Tabla de Edición* por parte del usuario, llega a esta parte en forma de *String*. Nunca podríamos concluir si el tipo del elemento almacenado en la posición 1 del array "*{ 0.1 , 2.3 , 3.4 }*" es un *Real* o no.

Para ello, realizamos la siguiente tarea:

- Del *String* "*{ 0.1 , 2.3 , 3.4 }*" extraemos el *substring* contenido entre las llaves. Nos quedamos por tanto con "*0.1 , 2.3 , 3.4*".

- Cogemos ahora cada uno de los elementos separados por el carácter delimitador, en nuestro caso la coma "," y los vamos almacenando en un array. Este array sigue siendo *String*, pero hemos pasado de tener un único *String*, que llegados a este punto nos resulta inútil, a tener un array de *String* con los valores de cada una de las posiciones por separado.

- Vemos el *tipo* seleccionado en la *Tabla de Edición* del correspondiente icono. Suponemos que el seleccionado fue el *Real*, por lo que entramos en su sub-bloque de comprobaciones (ídem para cualquiera de los otros tipos).

- Recorremos el array, encontrado en su primera posición el valor "*0.1*". ¿Podemos transformar este *String* en un valor *double* mediante la instrucción "*Double.valueOf(array[indice].trim()).doubleValue();*"?

  SI → El valor concuerda con el tipo seleccionado.
  NO → El valor no concuerda con el tipo seleccionado. ERROR.

. Y de forma análoga con cada uno de los tipos existentes[86].

Los mensajes de error relacionados con lo comentado hasta ahora son:

```
"No existe concordancia entre el TIPO y el VALOR de las
variables."

"La DEFINICIÓN DE UN ARRAY (intensiva o extensiva) no concuerda
con la nomenclatura."
```

Y así damos por finalizado el proceso de validación. Pasamos al de ejecución.

En la clase central de este módulo, *Ejecutor*, la funcionalidad de cada estructura (*WHILE*, *UNTIL*, *READ*,…) viene encapsulada por cada uno de sus métodos asociados (*whilei*, *until*, *read*,…). Cada vez que la ejecución invoca uno de estos métodos, éste, mediante el análisis de cada uno de los atributos de la pieza (*nombre*, *condición*, *valor*) detecta con elementos va a jugar. Y es aquí cuando cobra especial relevancia la familia de métodos *buscaVariable*.

Una vez el método de la pieza descubre que va a trabajar con la variable "*a*", e interpreta que es "*variable simple*" o array, de tipo "*x*", llama al método *buscaVariable* correspondiente(*buscaVariableDouble*, *buscaVariableBoolean*, *buscaVariableArrayInt*, *buscaVariableArrayString*, etc.) los cuales se encargan de la siguiente función: si dicha variable ya existe, por haber sido ya creada con anterioridad, el *buscaVariable* la encuentra en el almacén de variables y la devuelve para que el método principal pueda trabajar con ella. Sino es así, es decir, no la encuentra en dicho almacén, significa que no ha sido creada todavía y por tanto la crea e inicializa con el valor por defecto correspondiente:

- 0 → valor *Entero* por defecto.
- 0.0 → valor *Real* por defecto.
- "Cadena por defecto" → valor *Cadena* por defecto.
- false → valor *Booleano* por defecto.

Al respecto de la creación de variables, debemos saber que aunque se pueden utilizar en prácticamente todos los iconos, son sólo en el *ASSIGN*, en el *READ* y en el *FOR* donde se pueden crear.

[86] Cada uno de los tipos tiene su propia comprobación para concluir si los valores introducidos en los arrays corresponde con el tipo seleccionado, pero creemos conveniente describir más el proceso que todas y cada una de las pruebas implementadas.

Por tanto ya tenemos que cualquier pieza, que detecta variables en sus parámetros, las busca y extrae del almacén de variables (excepto en los casos de *ASSIGN* y *READ*, que si no las encuentran, las crean). Pero, ¿qué ocurre si la variable buscada y encontrada resulta ser un array? Como ya sabemos, es en el proceso de ejecución donde se realizan las operaciones indicadas por el usuario en su diagrama. Sabemos también que estas operaciones, a nivel interno, es *JEP* el encargado de realizarlas, pero que éste no puede trabajar con arrays.

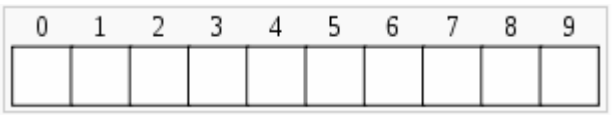
Es por tanto, como ya mencionamos con anterioridad, necesario realizar una conversión de array a vector, y para ello creamos el método *sincrArrayToVector*.

Entonces, las operaciones entre variables y términos independientes se realizan sin ningún problema, siempre y cuando los tipos de estas variables y términos permitan obtener un resultado capaz. En el caso, por ejemplo, de intentar multiplicar un número *real* con una variable *booleana*, esta claro que no es posible obtener un resultado válido, por lo que dará error, el cual será a su vez transmitido al usuario mediante un mensaje de error en tiempo de ejecución.

Encontramos a este respecto una clara e importante diferencia en la implementación entre el *ASSIGN* y el *READ*. Debido a que *READ* solicita la información en tiempo de ejecución, los datos introducidos por el usuario no son sometidos al proceso de validación, es decir, no se comprueba si dichos datos respetan la sintaxis propia del código. Lógicamente no podíamos permitir esta situación. Por tanto nos vimos obligados a implementar las mismas comprobaciones que realiza el método *validar* en el método *read*[87].

Sólo nos queda ya comentar el proceso inverso al de introducir una variable de clase array, y éste es el imprimirlo por pantalla.

El tratamiento consiste ahora en, de un vector que tenemos como resultado de una operación *JEP*, obtener un array. Pero cuidado, no un array Java, sino un *String* que salga por pantalla y simule o represente la definición de lo que sería el verdadero array Java. Es decir, si tenemos el vector



[87] El método *read* es único en el sentido de que, al no poderse comprobar la validez de sus valores en el proceso de validación, por ser sus datos introducidos en tiempo de ejecución, debe por sí solo hacer un subproceso análogo a éste. La consecuencia más directa de este hecho es que los errores detectados de ésta forma deben ser comentados al usuario durante la ejecución.

Sin embargo resultada algo tedioso el introducir un valor erróneo y que el proceso de ejecución se cortase de raíz, obligando pues a tener que validar y ejecutar el diagrama de nuevo. Decidimos por tanto que si un mensaje de error tenía su fuente en los datos introducidos vía *READ*, tras pulsar el botón ACEPTAR de dicho mensaje, volvería automáticamente a saltar la ventana de solicitud de datos *READ*. De esta forma no sería necesario interrumpir la ejecución y volver a inicializar los procesos de validación y ejecución, ahorrando de esta forma tiempo y trabajo al usuario y alumno.

tendremos que crear el *String*

{ 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 }

que es lo que el método *write* de la pieza *WRITE* puede imprimir. ¿Cómo imprimir un array vacío de tamaño 2 y tipo *Cadena de caracteres* para que el alumno sea capaz de comprender e interiorizar lo que supone este concepto? La formula elegida fue la siguiente:

{ “Cadena por defecto” , “Cadena por defecto” }

¿Y por qué la expresión *“Cadena por defecto”*? ¿Por qué no una cadena vacía? Hemos de recordar que esta aplicación esta pensada y diseñada para alumnos cuyo nivel de programación es nulo o prácticamente inexistente. ¿Recuerdan cómo, en las primeras clases de programación, una de las preguntas más repetidas por los compañeros era, qué demonios es una cadena vacía? Y aún tras dos o tres explicaciones, algunos alumnos seguían sin entenderlo del todo. Recordando esta experiencia es por lo que nos decantamos por esta fórmula.
De entender este formato a concebir una cadena vacía sólo hay un pequeño paso. Tanto a profesor como autor nos pareció una buena idea hacer ver así este concepto. Además, seguimos respetando así el paralelismo con el resto de casos, como por ejemplo, un array vacío de valores enteros y tamaño 2

{ 0 , 0 }

Y con ésto damos por concluido las fases de implementación de la declaración intensiva y extensiva de arrays. Pasamos pues a la siguiente, el tratamiento de las posiciones de array.

Al igual que en las etapas anteriores, lo primero fue pensar, discutir y decidir cual sería la nomenclatura más adecuada para alcanzar los principios propuestos: sencillez tanto para el usuario como para el programador y eficiencia docente.

Siguiendo un proceso similar al descrito para definición intensiva, llegamos a la conclusión de que, al igual que el símbolo asociado a las definiciones de array eran las llaves, para sus posiciones serían los corchetes “*[ ]* ”.

¿Y cómo representar la posición y el array al que pertenece? En este punto el acuerdo alcanzado fue bastante más rápido y sencillo. La mejor opción sin duda alguna era la ya conocida:

nom_array [posición]

Cumple con todos los requisitos a la perfección: es intuitivo, muestra toda la información necesaria en un formato muy sencillo y no es más complicada de implementar que las anteriores nomenclaturas.

Si lo es sin embargo el tratamiento sobre el término *posición*. La primera idea de utilización en cuanto a posiciones de array, es la que surge de forma inmediata: para referirnos a la posición hacemos uso de un número (entero, positivo e incluido el 0, es

decir, los requisitos exigidos para la utilización de posiciones de array). No obstante, esta situación conduce irremediablemente a otra: ¿y si queremos referenciar dicha posición mediante una variable?. Bajo este escenario si que surgieron enormes dificultades de código e implementación, por la sencilla razón de que en un mismo término debíamos utilizar no una variable sino dos. Ejemplo: *a[i]*.

En cuanto al manejo teníamos pues estas dos opciones recién comentadas. Pero, ¿y en cuanto a su uso? Realmente qué es lo que hacemos con las posiciones de array. La respuesta es sorprendentemente sencilla: leer (extraer) su valor almacenado o modificarlo (insertar un nuevo valor). Surge así una nueva ramificación de casos ya incluida en el esquema principal de este bloque y que podemos volver a ver en la *Fig. 4.17*.

Antes de comenzar a describir la implementación de esta etapa, vamos primero a comentar que supone para NWTJava v2.0 los conceptos de "*leer valor*" y "*modificar valor*". Para ello, vamos a sumergirnos directamente en la interfaz de la aplicación y gozar así de una perspectiva de primera mano.

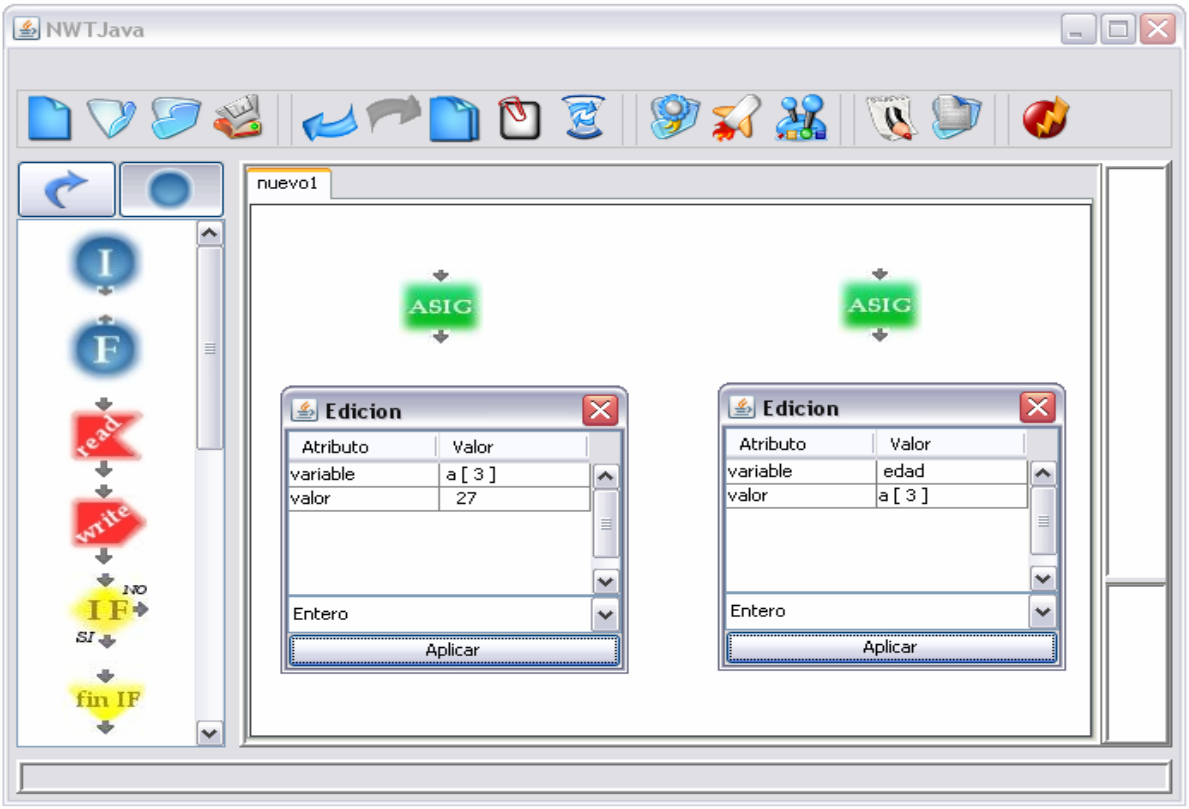


Fig. 4.18

Como podemos intuir, varía sensiblemente la instrucción que pretendemos realizar con el rellenado de la primera *Tabla de Edición* con la de la segunda. Con la situada a la izquierda, lo que pretendemos es modificar el valor almacenado en la posición 3 del array "*a*". Ello lo hacemos introduciendo el valor "*27*" en la variable *a[3]*. Muy distinto es el fin de tabla situada a la derecha. Lo que hacemos en ésta es leer, extraer el valor contenido en dicha posición del mismo array y guardarlo, en este caso, en la variable "*edad*". Como vemos, son dos conceptos muy diferentes y que esperamos hayan quedado muy claros con la figura ofrecida.

Una vez comentadas las ideas iniciales, los problemas, disquisiciones y soluciones finales, las aplicamos mediante la implementación. Para ello, volvemos a dividir el proceso en las fases de validación, ejecución y traducción debido a que es el orden lógico del tratamiento de diagramas de usuario. Además nos sirve también de itinerario o guía, pues un proceso no tiene lógica sin antes pasar por su predecesor. Como vemos, los dos principales bloques vistos hasta ahora, siguen caminos muy similares, y la explicación es sencilla: ambos están muy interrelacionados. Veremos un cambio de paradigma ya en los tercer y cuarto bloque, pero hasta entonces, continuamos con el de aplicación de arrays.
Para explicar la batería de pruebas que sufren las posiciones de array, podríamos comentarlas una a una, pero aún a riesgo de parecer demasiado simplista, creemos que la mejor y más clara explicación la podemos dar mediante diagramas.

En primer lugar, vemos el asociado a la validación, donde los métodos que entran en juego son el *validar* y su familia de métodos auxiliares cuya función es la de comprobar los datos con que el usuario rellena los campos de cada pieza:

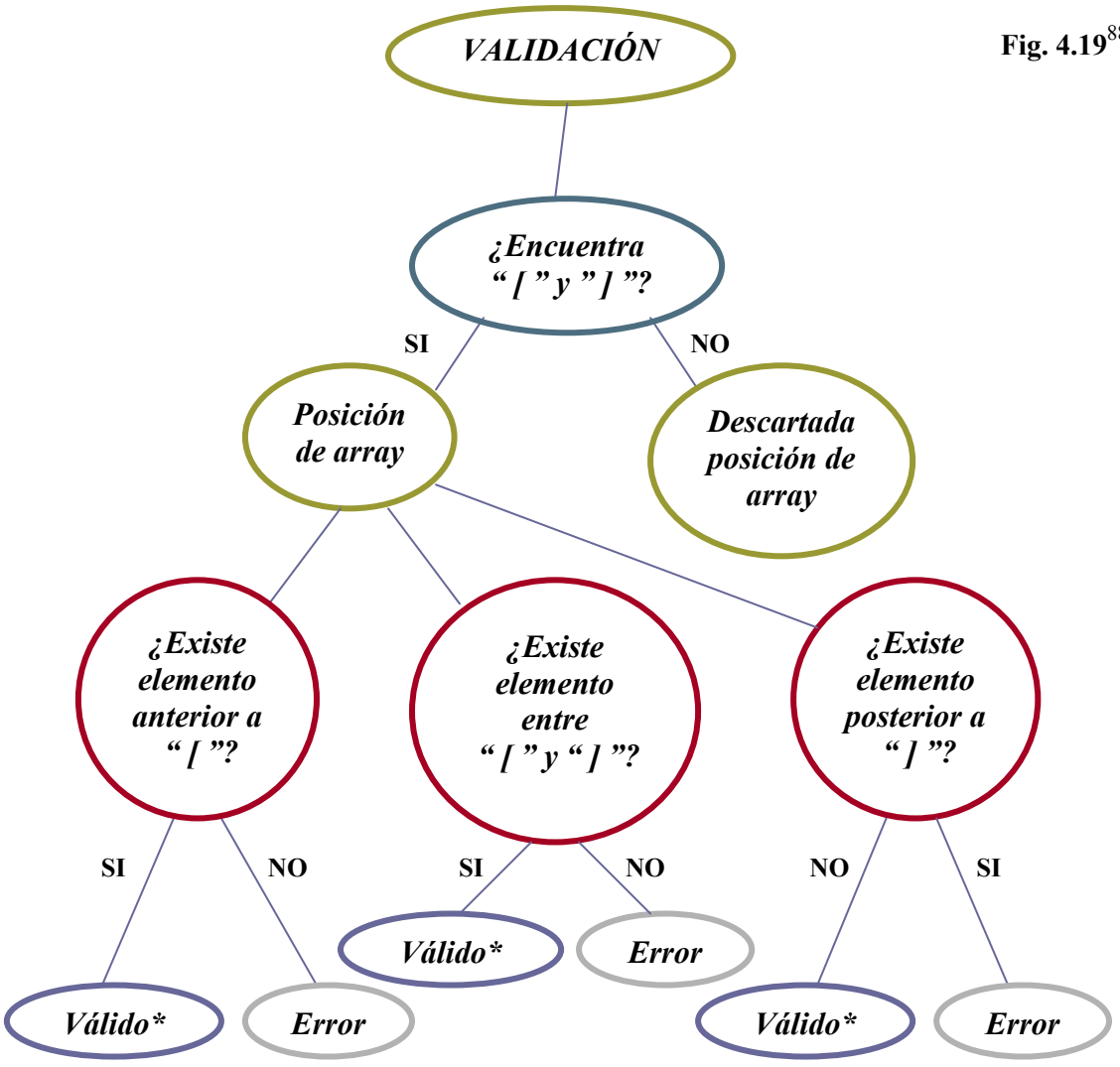


Fig. 4.19[88]

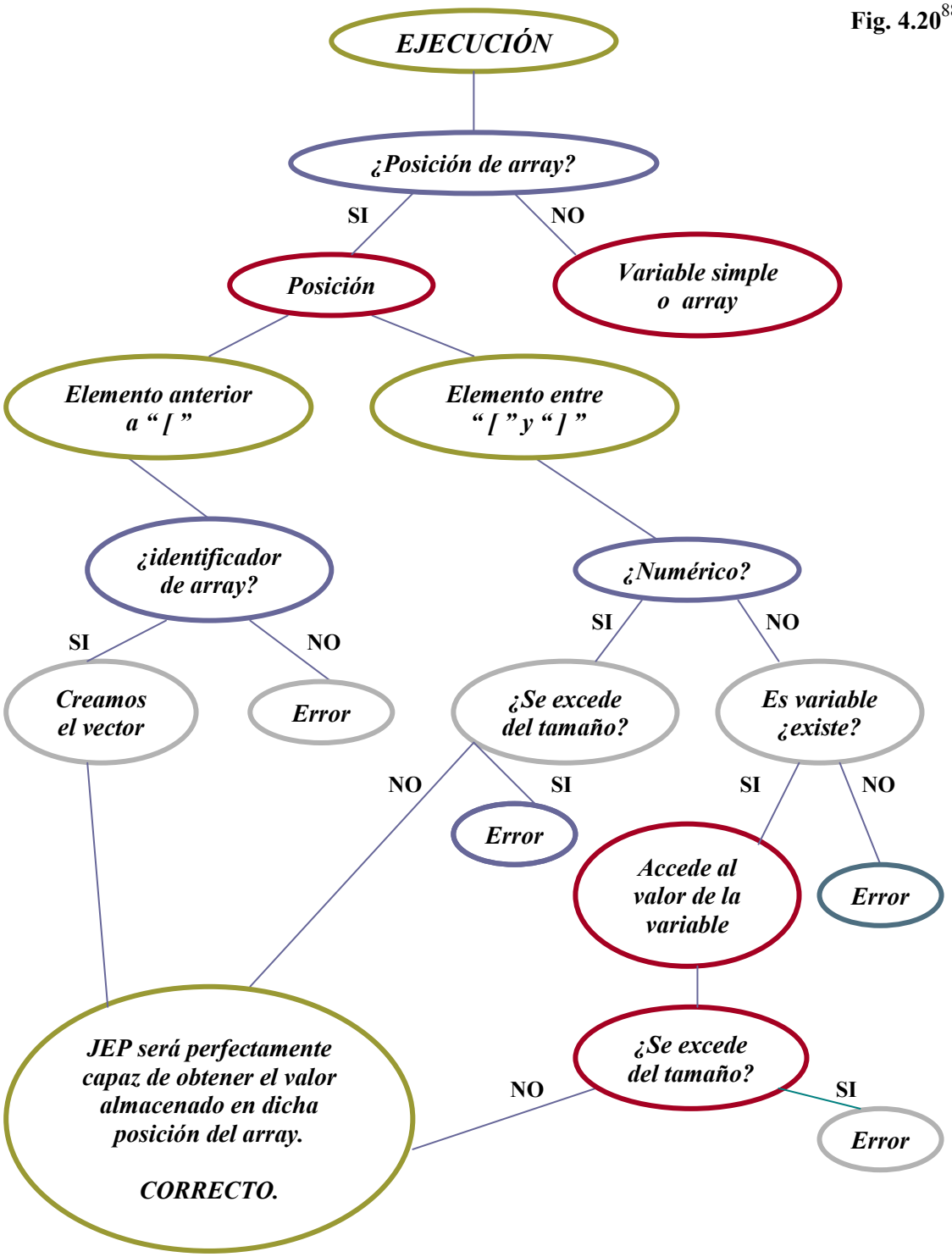
[88] Tras este diagrama se esconden infinidad de cuestiones y matices. Sin embargo no haremos hincapié en ellos debido a que podemos cometer el error de excedernos en una inmensidad de explicaciones que nos conducen al mismo fin.

* Al poder tratarse tanto de valores numéricos como no numéricos ("*a[3]*" vs. "*a[i]*"), el análisis de si son correctos o no dichos elementos delega en el proceso de ejecución. A este nivel, con "*a[i]*" no podemos saber si "*i*" existe como variable o no. Sin embargo, en ejecución que es donde se gestiona el almacén de variables (atributo *Vector variables*) sí tenemos las herramientas necesarias para poder discernir si la utilización de "*i*" es correcta o no. La misma situación ocurre con el nombre del array, en este caso "*a*".

  Otro caso que ilustra esta misma filosofía es la de, por ejemplo "*a[5]*" cuando en realidad el array "*a*" creado es de longitud 2. Es en *Ejecutor* cuando, creando el vector "*a*", NWTJava es capaz de apreciar la intención de acceder a una posición incorrecta. Ni que decir tiene que esta situación genera un error en tiempo de ejecución de la que el usuario es debidamente informado.

Pasamos pues a ver el diagrama de ejecución, donde *Ejecutor* y sus métodos asociados a cada una de las instrucciones o iconos realizan el siguiente tratamiento:

Fig. 4.20[88]


EJECUCIÓN
¿Posición de array?
SI
NO
Posición
Variable simple o array
Elemento anterior a " ["
Elemento entre " [" y "] "
¿identificador de array?
¿Numérico?
SI
NO
SI
NO
Creamos el vector
Error
¿Se excede del tamaño?
Es variable ¿existe?
NO
SI
SI
NO
Error
Accede al valor de la variable
Error
JEP será perfectamente capaz de obtener el valor almacenado en dicha posición del array.
CORRECTO.
¿Se excede del tamaño?
NO
SI
Error

Por último, y para finalizar ya este bloque, nos queda comentar cómo traducimos cada una de estas estructuras a lenguaje Java, mostrándolas así en el *Área de Traducción*.

Para ello, en la clase *PluginINTERM*, que es con la que vamos a trabajar (puesto que el empleo de arrays no es permitido a *NIVEL BÁSICO*), y dependiendo de la instrucción o pieza a traducir, nos vamos al bloque de código correspondiente. De la pieza tratada, analizamos el contenido de cada uno de sus atributos. Para verlo más claramente, vamos a utilizar un ejemplo: el atributo "*valor*" de la pieza *ASSIGN*. Si en el análisis detectamos que contiene llaves, deberemos traducir un array a la nomenclatura utilizada por Java para arrays. Si contiene corchetes, la misma situación pero para posiciones de array, y si lo que contiene es el nombre de una variable, deberemos comprobar si dicha variable almacena un array o no, y algo muy importante, si dicha variable esta inicializada o no.

Una vez seleccionado el camino y alcanzada la conclusión de qué es lo que debemos traducir, actuamos en consecuencia. No es lo mismo traducir a Java la inicialización de una "*variable simple*" que la de un array, o si el array viene definido de forma intensiva o extensiva, etc. Vemos a continuación, para dar ya por finalizado este bloque 3, ejemplos de traducciones mostradas en el correspondiente *Área de Traducción*, con el fin de dar así una idea de lo que el código implementado en *PluginINTERM* es capaz de hacer:

double b;
b = 6.7; → Inicialización de "*variable simple*" de tipo *Real*.

int[] a;
a = { 7 , 8 , 9 , 10 }; → Declaración de array intensivo de tipo *Entero*.

double[] a;
a = new Double[3]; → Declaración de array extensivo de tipo *Real*.

4.2.4 Bloque Subrutinas

Con este bloque es con el que entramos de lleno en el ámbito de la *programación estructurada*. Aquí, como ya dijimos en *Capítulo 2: ESTADO DEL ARTE*, distinguimos entre *procedimientos* y *funciones*. Pues bien, el modo en que NWTJava permitía al usuario programar sus diagramas lo asociamos directamente con los procedimientos (explicaremos porqué). Sin embargo, ahora introducimos un nuevo elemento, no solo estructural, sino también conceptual, y que supone para NWTJava v2.0, respecto de la primera versión, un enorme salto cualitativo. Hablamos por supuesto de las funciones.

Hagamos antes de meternos en la materia, un pequeño inciso.

Atrás han quedado los bloques primero y segundo. En ellos, la mayor carga de trabajo recaía sin lugar a dudas, en la codificación pura y dura, lo que comúnmente denominamos "*picar código*". El trabajo conceptual, aunque arduo en la toma de decisiones determinantes y finales, partía de unas ideas claras y sencillas: ampliar el número de tipos de variable ya existentes e incluir el uso y manejo de arrays.

Si hacemos un cálculo aproximado de tiempos, podemos decir, sin miedo a equivocarnos, que a la toma de decisiones sobre papel se dedicó aproximadamente un 10% del tiempo total, mientras que a la codificación el restante 90%. Ejemplos que reflejan ésto lo tenemos en:

- Tener claro que a los tipos *Entero* y *Real* había que sumarles los *Booleanos* y *Cadenas* **vs.** la cantidad de comprobaciones codificadas para detectar si los valores introducidos correspondían al tipo seleccionado o no.

- Saber que debíamos definir dos modalidades de definiciones de *array* **vs.** implementar las comprobaciones necesarias para dilucidar si la nomenclatura utilizada por el usuario es la correcta o no.

- Tener claro la imposibilidad de operar entre variables de tipo *Entero* y *Booleanos* **vs.** la cantidad de pruebas codificadas en *Ejecutor* para la detección de dichas variables, su clasificación y la toma de las decisiones correspondientes.

Y un largo etcétera donde, si analizamos el código implementado en la aplicación, podremos observar que la inclusión de nuevas líneas de código y la modificación de otras muchas son el fiel reflejo de este 90% del tiempo empleado.

Sin embargo, y a partir de este nuevo bloque, el de las *subrutinas*, esta tendencia cambia drásticamente. Pasa a cobrar mayor peso específico lo conceptual, hasta el punto de situarse la balanza en un 65/35%, donde el segundo porcentaje corresponde al tiempo y volumen asociado a la codificación.

Una vez comentada esta situación, con la que pretendíamos dar una idea del tipo de trabajo realizado en cada uno de los bloques, pasamos a centrarnos en éste tan fundamental.

Como hemos mencionado anteriormente, en la programación estructurada se distinguen, valga la redundancia, dos estructuras: *procedimientos* y *funciones*. Como también hemos dicho, asociamos al tipo de codificación empleado hasta ahora por el usuario con la primera de dichas estructuras, los procedimientos, ¿por qué?, analizamos la definición de *procedimiento*:

"*Rutina al que se le asigna un cierto nombre y que realiza unas determinadas operaciones*."

Si nos damos cuenta, responde a la perfección con el modelo de diagramas o grafos de nuestra herramienta. Son rutinas de operaciones enlazadas unas con otras, de forma lineal, y a la que se le asigna un nombre, en nuestro caso, el que el usuario le proporcione al diagrama para almacenarlo en disco.

Sin embargo, ¿qué podemos decir acerca de las *funciones*? Lo primero de todo, que se pueden ver como un tipo de *procedimiento*, pero con la cualidad especial de que puede devuelve un valor como resultado de su ejecución.

Entonces, ¿es posible utilizar *funciones* en un *procedimiento*? Por supuesto, de hecho es el paradigma fundamental de la programación estructurada. Y ésta es la idea inicial de la que partimos. Sin embargo, y llegados a este punto, vamos a utilizar nosotros nuestra particular forma de ver este asunto.

Vamos pues a distinguir en este nuestro proyecto, entre *Programa Principal* (PP) y las *Subrutinas* (SR)[89]. Como acabamos de describir, lo que hasta ahora podía codificar un usuario en el *Área de Trabajo* de la interfaz de NWTJava era sencillamente un *procedimiento*, una rutina de operaciones. Sin embargo ahora incluiremos la posibilidad de intercalar bloques de código independientes en dicha rutina principal.

Para ir profundizando, diremos que estas subrutinas corresponden a otros diagramas también codificados por el usuario, que incluso en determinadas circunstancias, pueden funcionar como PPs.

El funcionamiento general de este bloque lo podemos describir de la siguiente forma:

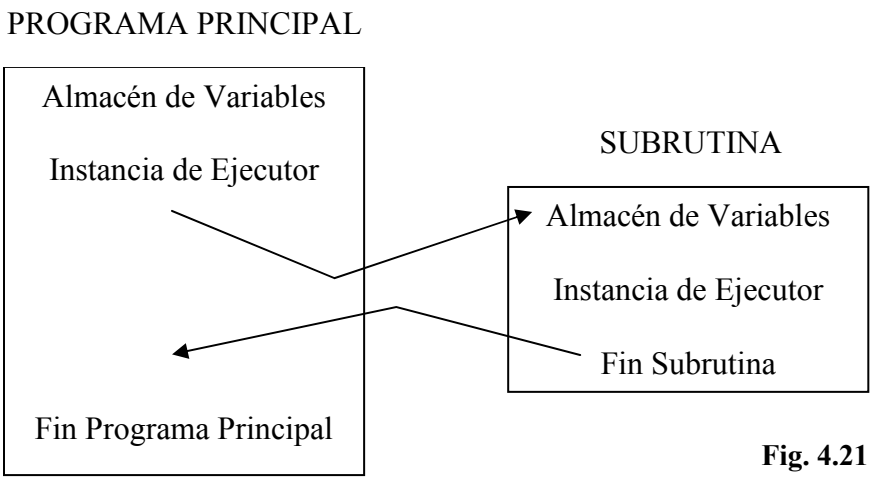


Fig. 4.21

Vamos a comentar de forma breve este diagrama para a continuación indagar más en el proceso de organización e implementación.

La primera instrucción o pieza del PP es, como en todo diagrama, la de INICIO. En esta pieza, nunca ha sido necesario invocar su *Tabla de Edición* pues no tenía asociada ningún parámetro como pueda ser *FOR* o *READ*. Pues bien, a pesar de lo que más tarde comentaremos, esta pieza *INICIO* sigue sin recibir parámetro alguno. El siguiente paso será la creación y construcción del diagrama por parte del usuario, del que derivará la creación de su correspondiente *Almacén de Variables* y el proceso de ejecución. Sin embargo, en dicho transcurso tendrá lugar la aparición de una nueva fase. Digámoslo de la siguiente forma: un subproceso. Al llegar al nuevo punto creado en este bloque, representado por el nuevo icono *SUBRUTINA*, la ejecución del PP se detendrá y entrará en un estado stand-by. Entrará pues en funcionamiento un nuevo proceso de creación de *Almacén de Variables* y ejecución independiente al del PP. En este caso corresponderá al del diagrama que juega el papel de SR. Una vez finalizado éste, se proporcionará la

[89] A partir de ahora, y para abreviar, utilizaremos las siguientes siglas: *PP* para designar al *Programa Principal* y *SR* para *Subrutina*.

señal necesaria para recuperar la ejecución del PP desde el punto de parada, y retomar desde ahí.

Dos cuestiones a destacar son:

- Ambos procesos de ejecución son independientes, pero podrán intercambiar información en dos momentos muy concretos: el paso de valores en forma de parámetros del PP a la SR, y el paso de un valor devuelto como resultado de la SR al PP.

- La otra consideración es que, las variables del PP no serán visibles desde las SRs y viceversa, es decir, cada programa tendrá su propio almacén de variables.

Si nos damos cuenta, el proceso, por compararlo con un lenguaje real conocido, es prácticamente análogo al de la utilización de un método en un programa Java. El PP invoca a un método pasándole una serie de parámetros con la intención de recibir a cambio un valor determinado (o no, dependiendo de si dicho método retorna un valor o no).

¿Cuáles fueron las fases seguidas para implementación de este bloque? Pues las mencionadas a continuación:

1. Creación del nuevo icono *SUBRUTINA*.

2. Proporcionarle su funcionalidad a nivel de interfaz gráfica (implementación de su *Tabla de Edición*, aportarle sus pares de valores,…).

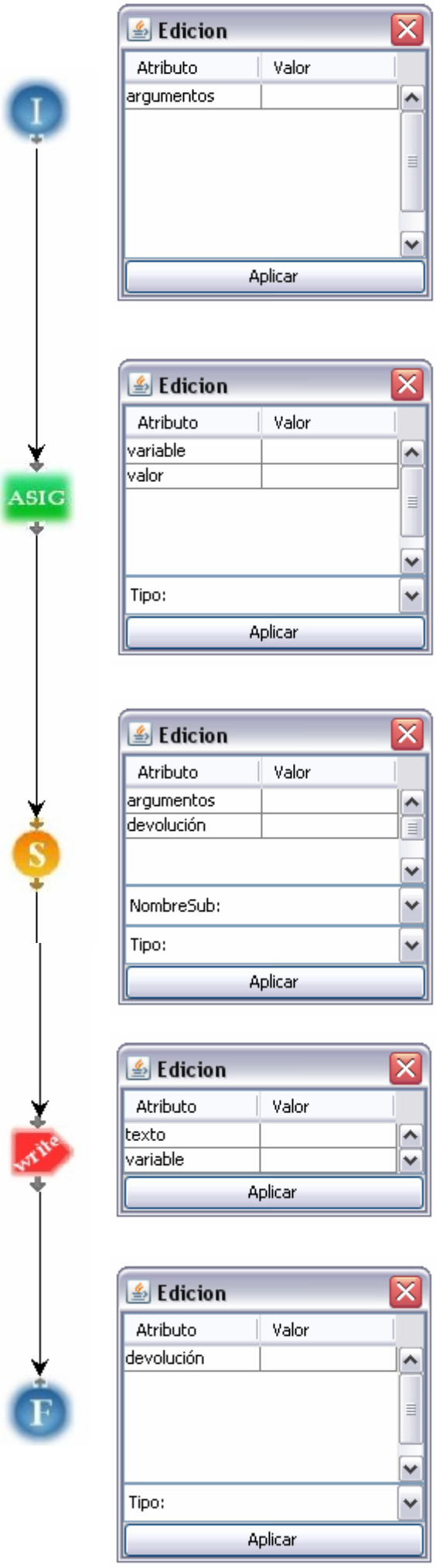
3. Redefinir los iconos de *INICIO* y *FINAL* (junto con el de *SUBRUTINA*, los verdaderos protagonistas de este bloque). Esta redefinición tiene lugar únicamente en niveles distintos de *BÁSICO*.

4. Añadir la funcionalidad en su conjunto y a nivel de ejecución.

5. Traducción.

Pero ésto puede aún parecernos confuso o lejano. Por ello explicamos ahora que es lo que realmente buscamos, y ya no tanto a nivel general, sino en el caso particular de NWTJava v2.0.

Vamos por tanto a ver un diagrama explicativo en el que mostramos la simulación de un programa NWTJava en el rol de PP. En él adelantaremos aspectos como el modelo del nuevo icono *SUBRUTINA* o la inclusión de nuevos parámetros en los iconos de *INICIO* y *FINAL*, que como ya comentamos, al menos para el caso *INICIO*, anteriormente a la aparición de este bloque, no presentaban atributos.



- *0 argumentos*: el diagrama podrá funcionar tanto como PP como SR. Es como un método Java, puede tener o no parámetros.
- *1 o más argumentos*: el diagrama sólo podrá funcionar como SR.

- *NombreSub*: desplegable con los diagramas abiertos en NWTJava v2.0 (comentaremos porqué).
- *Argumentos*: 0, 1, 2 o más. El mismo número, tipo y orden que exija el icono *INICIO* del diagrama indicado en *NombreSub*, el cual actuará como SR.
- *Devolución*: 0 ó 1 variables. Del mismo tipo que devuelva el icono *FINAL* de la SR.
- *Tipo*: debe ser el mismo que el de la variable que devuelva el icono *FINAL* de la SR.

- 0 ó 1 variables. Aquí no se selecciona *tipo*, pues *FINAL* se limita a devolver la variable que se le introduzca en *devolución*.

Fig. 4.22

A continuación vamos a reproducir un ejemplo práctico en el que podremos observar un PP, una SR y las posiciones que tienen uno respecto al otro. Lo haremos mostrando los iconos y su respectivo rellenado de *Tablas de Edición* (mediante pseudocódigo). Así acabaremos de entender la mecánica de trabajo que buscamos con las subrutinas y nos meteremos de lleno en su implementación.

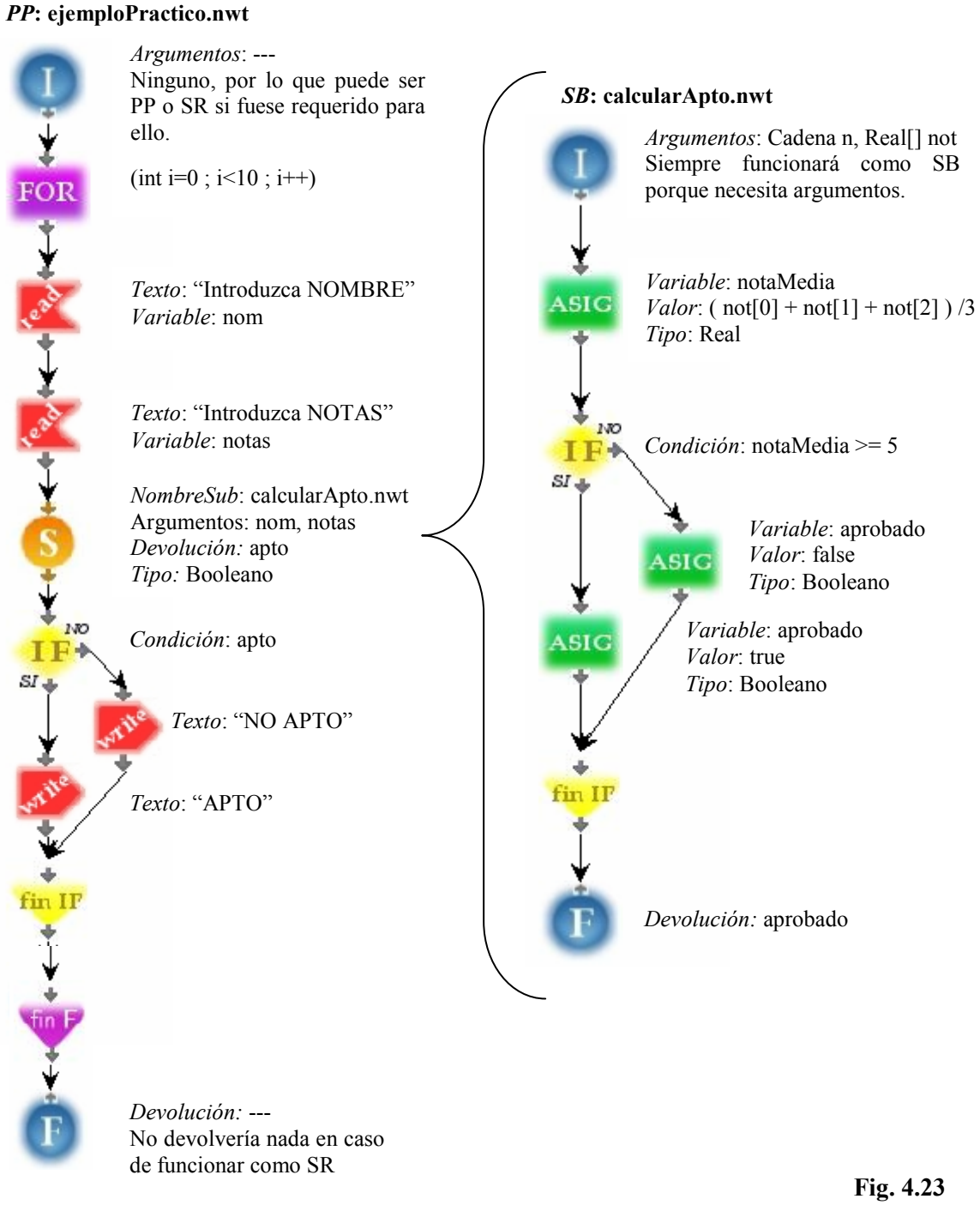


Fig. 4.23

1. Creación del nuevo icono *SUBRUTINA*.

Este nuevo requisito, a diferencia de las explicadas en los anteriores bloques, requería de una pieza exclusiva. El motivo es que, al igual que un *FOR* o un *IF*, una subrutina también tiene asociada una estructura de líneas de código.

Al igual que para la modificación de los iconos ya existentes, el programa de diseño utilizado para la creación de esta nueva pieza fue *GIMP Portable*. Y como muestra de que en este proyecto, ninguna decisión tomada ha sido arbitraria o azarosa, sino que muy al contrario todas han sido meditadas, justificadas, y revisadas una y otra vez tenemos el proceso de creación de este icono. ¿Cuál debía ser su forma?, ¿el color?, ¿puertos de entrada y salida? A pesar de lo que pueda parecer, todas ellas eran decisiones críticas, pues debíamos ser coherentes con un modelo ya establecido.

Una subrutina, tal y como la interpretamos en NWTJava v2.0, supone un punto, en un diagrama creado, que da acceso a otro diagrama (el que actúa como SR). Pero no sólo eso, también lleva asociado el final. ¿Qué iconos son los asociados a principio y final de un diagrama? La respuesta es: los de *INICIO* y *FINAL*. Pues bien, teníamos en este planteamiento la respuesta en cuanto a su forma: circular. De esta manera, las piezas asociadas a principios y finales de diagramas serían todas circulares. Pero lógicamente, la función indicativa o limitadora de los iconos *INICIO* y *FINAL* dista mucho de la del icono *SUBRUTINA*. El color debía por tanto ser diferente. Como requisito teníamos el no poder repetir alguno de los ya utilizados. Además, entendemos que esta estructura marca un punto de inflexión en cuanto a que, no pertenece al grupo de estructuras condicionales, ni de bucle, y señala un punto novedoso e importante desde el punto de vista del alumno neófito (el empleo de subrutinas), por lo que debía en algún sentido, llamar la atención. Un color que cumplía con estos requisitos era el naranja. Por tanto, ya teníamos forma y color, faltaba el número de puertos de entrada y salida. Esta decisión estaba más encauzada y dirigida que las anteriores. Al igual que un método de Java, nuestras subrutinas requieren una entrada, que le suministre los datos o parámetros requeridos, y una salida, por la que es capaz de devolver un resultado o valor. Por tanto la pieza resultante de este proceso fue:

Fig. 4.24

Sin embargo, este icono presenta un hándicap respecto a todos los anteriores. Y es que únicamente debe ser utilizado a partir del *NIVEL INTERMEDIO*, es decir, no debe poder ser elegido ni tan siquiera ser visible a *NIVEL BÁSICO*. Éste hecho resulta fundamental para no entremezclar niveles de aprendizaje y sus conceptos, evitando así una fuente de confusión para los alumnos. Para ello creamos, en la clase *NWTJava*, el método *ponerIconoSubr*, el cual es llamado junto al *ponerEtiqueta* en el método *main*. Lo que hace es

ver el nivel seleccionado, y si es diferente de *BÁSICO*, incluye el icono *SUBRUTINA* en el panel de iconos y en el menú de *Insertar*.

2. Proporcionarle su funcionalidad a nivel de interfaz gráfica.

Una vez construida la pieza, debíamos poder utilizarla, y por supuesto, manejar la información requerida por este icono e introducida por el usuario.

Para ello, lo primero que debíamos hacer era implementar la definición del icono o nodo[90] en la clase *UserObject* y asociarle el conjunto de atributos necesarios (*nombreSub*, *argumentos*, *devolución* y *tipo*).

En segundo lugar, fue necesario modificar en *editAtt* de *NWTJava la Tabla de Edición* para el caso de *SUBR*, pues como podemos ver en ella,

Fig. 4.25

presenta no un *JComboBox* sino dos. El segundo y nuevo *JComboBox* es el asociado al parámetro "*NombreSub*", donde el usuario introduce el nombre del diagrama que será SR. Al respecto de este parámetro haremos un inciso. Al igual que ya justificamos con "*Tipo*", la utilización de una lista predeterminada de valores minimiza la posibilidad de errores por parte del usuario. Recordemos lo que ya dijimos acerca de este hecho:

- *"…el usuario podría no saber que tipos son los admitidos…"*
- *"Además, aún sabiéndolo existiría una gran fuente de error en esa libre escritura (ej: ¿en mayúsculas?, ¿sólo la inicial?, ¿todo en minúsculas?, ¿qué pasaría si en lugar de "Entero" escribiese "Entierro"?...)."*

Lo mismo ocurría a la hora de escribir los nombres de los diagramas que queremos que funcionen como SR. ¿Se respetarían mayúsculas y minúsculas?, ¿se debería escribir la extensión del fichero[91]? Por ello, se decidió suministrar automáticamente dicha lista. Pero llegados a este punto, surgió otra cuestión: ¿Qué diagramas debían aparecer en el desplegable? Opciones posibles eran: todos los ficheros .nwt, los de una determinada carpeta, etc. Opciones entre las que elegimos, por considerarla la más útil y práctica la que comentamos a continuación.

[90] Denominación de icono o pieza a nivel interno.

[91] .nwt

Aparecerían únicamente los diagramas que estuviesen ya abiertos en la aplicación. De este modo nos aseguraríamos diversos aspectos. Por un lado, el de manejar una lista muy concreta y específica (en oposición a la de mostrar todos los diagramas de una carpeta, o todos los .nwt). Y en segundo lugar, esta forma encauzaba directamente a que, el usuario, antes de ejecutar el PP, controlase las SRs a utilizar. Así, obligábamos a que para utilizar un diagrama como SR, el alumno antes lo tuviese que abrir, y además, validarlo. Así forma, sino fuese un diagrama correcto, se centraría primero en el estudio y corrección de éste, facilitando tanto su labor como la nuestra desde un punto de vista funcional y educativo. En todo caso, si al intentar validar el PP sin antes haberlo hecho con las SRs pertinentes, saldría por pantalla un mensaje aclaratorio informando de que, antes de validar el PP, debe hacerlo con las SRs, para así trabajar con unas subrutinas ya válidas.

Pero todavía quedaba un aspecto por resolver, y éste era el de cómo obtener la lista de programas abiertos. Existe un atributo en NWTJava v1.0 que proporciona esta información. Es "*Hastable diagramas*" de *Datos*, pero contemplemos como referencia a tan solo uno de los diagramas:

"*{GraphInfo@11d3226=org.apache.crimson.tree.XmlDocument@63f09e, GraphInfo@b3cac9=org.apache.crimson.tree.XmlDocument@14a55e5}*"

Como vemos, esta nomenclatura no proporciona ninguna información útil al usuario sobre el diagrama al que desea referirse. Para solucionar esta contingencia, creamos un nuevo atributo en la clase *Datos*. Funcionaría como una lista, que además sería dinámica y permitiría incluir los diagramas abiertos en tiempo real, y como no, eliminarlos al cerrarlos. Este nuevo atributo es "*Vector filenames*". Su utilización, de la que acabamos de comentar unas pinceladas, se basa en que, al abrir un nuevo diagrama, lo añadimos pero ¿desde dónde?, ¿y cómo? Lo añadimos desde la clase *Datos* que es la primera que recibe la información de "diagrama abierto", y lo hacemos no incluyendo cualquier identificador, sino el de su nombre asociado e impuesto por el usuario, al cual se accede invocando al atributo *fichero* de su correspondiente objeto *GraphInfo*[92]:

" *filenames.addElement(gi.fichero);* "

Para eliminarlo realizamos el proceso inverso. Pero esta labor ya no es competencia de *Datos*, sino de *GraphManager*, la cual contiene el método *close*, que como hace intuir su nombre, se encarga del cierre de diagramas.

" *filenames.remove(gi.fichero);* "

De esta forma conseguimos tener siempre una lista totalmente actualizada a tiempo real, invocada desde el instante en que el usuario la requiere mediante la apertura del *JComboBox* "*NombreSub*".

[92] Objeto que almacena la información asociado a un gráfico abierto en la aplicación.

3. Redefinir los iconos de *INICIO* y *FINAL*

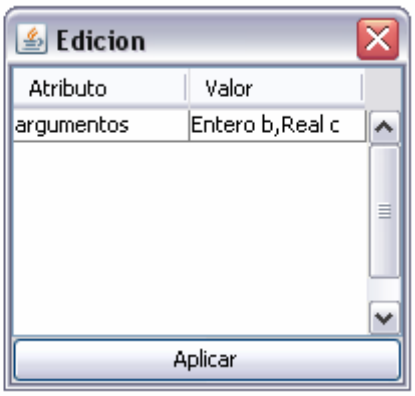
Estas dos piezas, junto con la de *SUBR*, son las protagonistas de este bloque. Muy importante es tener en cuenta que esta redefinición tiene lugar únicamente a niveles distintos de *BÁSICO*.

El motivo de su redefinición es que ahora, estos iconos van a funcionar también como puntos de enlace entre PPs y SRs. Por lo tanto serán los encargados del traspaso de información entre unos y otros.

De forma rápida y sencilla podemos decir que el PP, mediante el icono *SUBR* transmite la información necesaria a la SR, la cual es captada por su pieza *INICIO*. Una vez ejecutada la SR, es la pieza *FINAL* la encargada de devolver la información requerida por el PP.

Para ello, hubo que decidir que clase de datos, y en que cantidad, podrían comunicarse PPs y SRs. Como ya hemos dicho antes, son dos los puntos donde se produce una cesión de información. El primero, PP → SR es con los iconos *SUBR* → *INICIO*. El parámetro ideado para esta función en el icono *SUBR* fue el de "*argumentos*", por lo que *INICIO* debía, por lo pronto, tener un nuevo parámetro para recoger dicha información. Para no marear y confundir a los alumnos con un ingente cantidad de términos y conceptos, y debido a que la función de los dos parámetros tratados es la de pasar y recoger la misma información, este nuevo parámetro de *INICIO* también se denominó "*argumentos*".

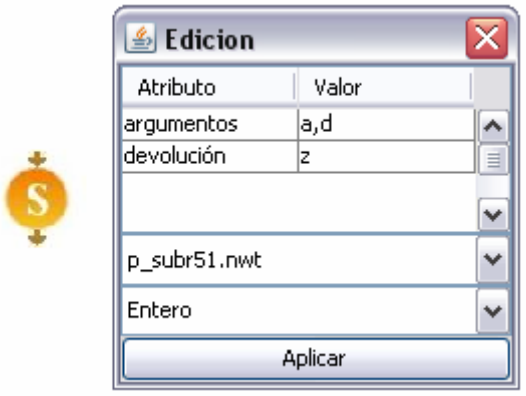
Nos falta ahora comentar la estructura de los datos manejados en esta operación. Si comparamos esta situación con la de un método Java, vemos como el PP corresponde al código de un programa, mientras que la SR al método. Pues bien, un método, en su definición expresa entre paréntesis los parámetros que requiere, tanto en tipo, como en número y orden. Igualmente ocurre en NWTJava v2.0: la SR, con su pieza *INICIO* a la cabeza (a modo de los paréntesis de un método), indica qué datos son los requeridos para su correcta ejecución, también expresados en tipo, número y orden. Es en su parámetro "*argumentos*" donde muestra esta información.



En esta *Tabla de Edición* se puede observar perfectamente. *INICIO* presenta en su atributo "*argumentos*" la información que requiere. En este caso dos variables, en primer lugar una de tipo *Entero* y después otra *Real*.

Fig. 4.26

Para el correcto funcionamiento del diagrama general (diagrama que funciona como PP unido a los que actúan como sus SRs), el icono *SUBR* debe cumplir con los requisitos exigidos por *INICIO*. Siguiendo el ejemplo:



Vemos como *SUBR*, para invocar la SR, cuyo nombre es "*p_subr51.nwt*", le pasa como argumentos las variables "*a*" y "*d*", que en este caso serán *Entera* y *Real* respectivamente.

Fig. 4.27

Y mediante esta figura enlazamos con el segundo de los procesos de intercambio de información: *FINAL* → *SUBR*, correspondiente a SR → PP. Podemos observar como el PP, mediante el icono *SUBR* exige a la SR p_subr51.nwt la devolución de un valor que almacenará en la variable "*z*", El tipo de "*z*" debe ser *Entero*. De esta función se encargará la pieza *FINAL*, la cual obtendrá tras la ejecución de la SR un valor, que será el que proporcionará al PP. Para tal efecto, creamos y le asociamos a *FINAL* el atributo "*devolución*".

Fig. 4.28

Para la implementación de estos atributos, baste decir que se los asociamos a sus respectivas estructuras modificando los métodos *initAtts* de *UserObject* y *toArray* de *AdapterNode* (encargados ambos de la gestión de atributos), y que reestructuramos sus *Tablas de Edición* en *editAtt* de *NWTJava*.

El resto de cambios sujetos a estas dos estructuras las vamos a incluir en el siguiente punto, donde comentaremos el funcionamiento de la ejecución una vez incluido este *Bloque de Subrutinas*.

4. Añadir la funcionalidad en su conjunto y a nivel de ejecución.

E aquí la fase que supuso mayor complicación para el desarrollo e implementación de este bloque.

Comenzaremos diciendo que, para este punto, independientemente de lo programado hasta ahora y de la batería de comprobaciones implementada para la validación de los elementos de este bloque, fue necesario: crear una nueva clase (*VariablesSub*), tres métodos en la clase *Ejecutor* (*beginNoBas*, *subr* y *fin*) y realizar una profunda modificación en dos de los métodos esenciales de esta misma clase (*ejecuta* y *ejecutaPaso*).

Fundamental resulta entender que el proceso de ejecución utilizado hasta ahora ha sido puramente lineal (se ejecuta pieza a pieza en un orden descendente de principio a fin)[93]. Tras la implementación de este bloque, ésto sigue siendo así, pero ahora de modo genérico, es decir, para todos aquellos casos en que no se utilicen SRs.

Sin embargo, con la inclusión de subrutinas dependientes de un diagrama principal, la ejecución deja de ser lineal. Para el correcto funcionamiento de este subtipo de ejecución, nos vimos obligados a realizar una serie de modificaciones. Las comentamos:

Para empezar, creamos el método *beginNoBas*. Éste es asociado a *INICIO*, por lo que ya son dos los métodos relacionados con esta pieza (*begin* y *beginNoBas*). Tomamos la decisión de crear este nuevo método y no implementar su contenido en el ya existente porque existían las suficientes diferencias entre sus funcionamientos como para hacerlo. El primero, *begin*, no requiere parámetros y se limita simplemente a dar paso a la siguiente pieza, la pieza "*hijo*". Este funcionamiento es el ideal para el *NIVEL BÁSICO*. Sin embargo, dista mucho de lo requerido para niveles superiores, de ahí el nuevo nombre con el sufijo "*NoBas*" (No Básico). Como ya vimos en el anterior punto (*Redefinir los iconos de INICIO y FINAL*), el icono *INICIO* tiene ahora un atributo ("*argumentos*") por lo que el método debe recibir parámetros. Pero no sólo eso, ahora esta pieza deja de ser un mero indicador de principio de diagrama para convertirse en un elemento fundamental en el tratamiento y manejo de subrutinas.

Como dijimos, los argumentos que recibe *INICIO* por parte del usuario pueden ser 0 (el diagrama puede funcionar tanto como PP como SR que no requiere parámetros) y 1 ó más (donde obligatoriamente el diagrama debe funcionar como SR). Pues bien, si el caso dado es este último, *beginNoBas* lo primero que debe hacer es procesar esos parámetros, que si recordamos van a ser una serie de variables acompañadas de su tipo correspondiente (Ver *Fig. 4.26*).

En segundo lugar creamos el método *subr*, asociado como su propio nombre indica a la recién creada pieza *SUBR*. Es el encargado de ejecutar las funciones de dicho icono y es la estructura que comunica al PP con la SR a la que invoca. Este método será el que transmita la información necesaria del PP a la SR y viceversa. Veremos su funcionamiento.

[93] Entendemos "lineal" como una sucesión de iconos que, aún a pesar de los por las bifurcaciones debidas a los ifs y a los bucles, pertenecen al mismo diagrama.

El tercer método creado es el *fin*, correspondiente a la pieza *FINAL*. A diferencia de *INICIO*, esta estructura no tenía método asociado pues simplemente marcaba el final del diagrama y una vez llegado a ella, finalizaba la ejecución. Ahora sin embargo, ya no tiene porque marcar este punto, puede también funcionar como fin de una SR y tener que devolver el valor de la variable solicitada al PP.

Y para acabar de describir los elementos de reciente creación, la nueva clase *VariablesSub*. Su documentación cita literalmente:

```
/**
*Clase que implementa la estructura de datos utilizada
*para el intercambio de parámetros entre PP y SR.
*/
```

Es decir, esta clase es la encargada del manejo de la estructura de datos que contendrá las variables que se intercambien PPs y SRs. Será el puente que comunique a ambos tipos de diagramas. De hecho será el único nexo de unión, y en breve comentaremos porqué. Esta clase contendrá los métodos necesarios para añadir las variables necesarias a la estructura de datos y para devolver sus valores cuando sean requeridos. Para más información diremos que dicha estructura de control es un *Hashtable*[94], y que las variables contenidas en ella vienen definidas por su valor y su clave. Esta clave, que funciona a modo de contraseña, es con la que clasificamos dichas variables, y nos sirve para recuperarlas sin miedo a confundir las de una SR con las de otra. Para ello, la clave que hemos utilizado tiene la siguiente estructura:

"*Nombre de la subrutina_Número de orden*"

Hemos comentado hace unas líneas que la estructura de datos (*Hastable*) de esta clase es el único nexo de unión entre la ejecución del PP y la de las SR, y es cierto debido a que, para estos casos, la ejecución deja de ser un proceso lineal. Pasa ahora a ser un proceso fragmentado en partes paralelas, en la que: comienza la ejecución del PP; llegados al punto de la SR, esta ejecución se detiene para comenzar un nuevo proceso totalmente independiente en el que se ejecuta la SR; una vez finalizada esta ejecución, retomamos la del PP; y así sucesivamente con tantas SRs como haya. Y el único enlace existente entre estos procesos independientes es el *Hastable*, el almacén de variables cuyos valores presentan los dos sentidos de circulación ya mencionados: PP → SR y SR → PP.

Analicemos el funcionamiento de cada uno de estos elementos, y el proceso en su conjunto, mediante un ejemplo:

Como muy bien sabemos ya, tras la construcción del diagrama por parte del usuario, y la correcta validación del mismo (pasando este proceso por las fases de *retrieveAtt* y *validar*), llegamos al punto donde se pulsa el botón "*Ejecutar*". Es a partir de aquí donde la clase *Ejecutor* toma el control del proceso.

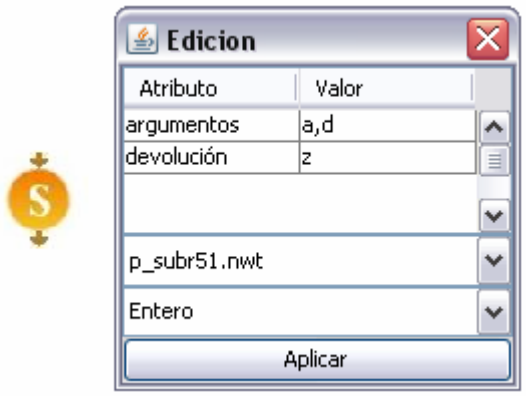
[94] Representa una colección de pares de clave y valor organizados en función de la clave. Para más información ver Referencia [18].

El primer método que entra en acción es el *run*, el cual inicializa una serie de valores necesarios e invoca al método que a partir de ahora dirigirá las operaciones: *ejecuta*. La forma en que trabaja este importantísimo método es la siguiente: a él llega, en forma de parámetro, un array de *Strings* donde cada posición corresponde, en orden, a cada una de las piezas utilizadas en el diagrama actual (el que se encuentra en ejecución). Recorriendo este array, va pieza por pieza invocando cada uno de sus respectivos métodos (la primera pieza es un *INICIO* por lo que llama al método *begin*, si la segunda es un *ASSIGN* invoca al método *assign*, y así sucesivamente hasta llegar a la última pieza del diagrama que será un *FIN*).

Supongamos que el diagrama actual posee una subrutina. No es necesario que tenga más para la explicación, puesto que si la hubiese, el proceso sería perfectamente extrapolable.

Tras haber el *ejecuta* ejecutado un cierto número de piezas, llegamos a una cuyo identificador es "*subr*". Se invoca su método, con sus atributos correspondientes como parámetros.

- el primero de éstos sobre el que se trabajamos es "*argumentos*". Recordemos que en el campo "*argumentos*" el usuario introducía las variables (sus nombres) que debían ser utilizadas en la SR.



En este ejemplo, "*a*" y "*d*".

Fig. 4.29

Comprobamos el número de ellas (0, 1 ó más), y una vez hecho ésto, las recorremos para ir realizando el siguiente tratamiento con cada una de ellas: mediante los métodos adecuados de la familia *buscaVariable*, comprobamos si existe. Si no es éste el caso, estamos ante una situación de error en la que el usuario intenta pasar a una SR una variable que no ha sido creada con anterioridad. El mensaje de error correspondiente es:

"*Una VARIABLE utilizada para invocar una SUBRUTINA, no existe.*"

Si existe, lo que hacemos es: añadirla al *Hastable* de la clase *VariablesSub*, con la clave necesaria para localizarla y recuperarla:

"*Nombre de la subrutina_Número de orden que ocupa dentro los parámetros*"

- en segundo lugar pasamos a ejecutar la SR. Como ya hemos comentado, ésto supone iniciar otro proceso de ejecución independiente, por lo que una vez terminado, y con objeto de volver al punto donde interrumpimos el primero de los proceso, debemos guardar una serie de elementos sin los cuales no sería posible recuperarlo.

  Transcribimos literalmente los comentarios del código correspondiente, que explican perfectamente qué elementos y porqué, son los que se debemos almacenar:

```
/* Pasamos a ejecutar la subrutina cuyo nombre se ha
indicado en el JComboBox de la tabla de edición.

Pero antes, debemos guardar el almacén de variables del
diag.princ. y el de bucles for anidados, pues lo 1º que
hace el método ejecuta() es un clear de ambos, y sin
ésto, al volver al diag.princ., no podríamos seguir
utilizando las variables ya creadas y propias de este
diagr.

Hacemos lo mismo con el puntero de programa, pues al
ejecutar la subrutina, el pp que queda al finalizar no
concuerda con el pp por el que debe continuar el diag.
princ. */
```

  El siguiente paso es establecer la SR como diagrama actual, que es el tipo de diagramas que NWTJava puede ejecutar.

- Bien, pues ya estamos inmersos en el nuevo diagrama. Al tratarse de un proceso independiente, se ejecuta tal cual y con toda normalidad. Como si se tratase de un diagrama totalmente independiente y autónomo.

  La primera pieza, como no, es *INICIO*. Por tanto, nos disponemos a analizar su atributo ("*argumentos*"). ¿En qué nivel de aprendizaje nos encontramos? Desde luego en el *BÁSICO* no, puesto que sino no podríamos estar utilizando subrutinas. Por tanto el método asociado a esta pieza no es *begin* sino *beginNoBas*. Aquí realizamos un proceso muy similar al que hace *SUBR* con su atributo "*argumentos*". De hecho, los dos parámetros "*argumentos*" son con los que ambas piezas se comunican y constituyen el único nexo de unión entre PP y SUBR. La diferencia estriba en que ahora lo que analizamos es si las variables introducidas por el usuario en la pieza *INICIO* de la SR concuerdan en número, tipo y orden con las introducidas en SUBR (mismo funcionamiento que con los parámetros de un método en Java: deben coincidir número, tipo y orden con lo esperado por la definición del método).

  Si todo concuerda, el proceso sigue adelante. Y para ello, lo que debemos hacer, es recuperar los valores de las variables que se le han

pasado a *INICIO*. ¿Cómo? Para ésto precisamente fue para lo que creamos la clase *VariablesSub* y su atributo *Hastable*. Es este el momento en que mediante la clave necesaria recuperamos el valor asociado a cada una de las variables.

Fig. 4.30

Si todo es correcto, las variables "*a*" y "*d*" serán de tipo *Entero* y *Real*, pues son el orden y los tipos que requiere la SR. Tras haber *INICIO* recuperado los valores del *Hastable*, "*b*" tiene el valor de "*a*" y "*c*" el de "*d*".

La ejecución por tanto tendrá lugar sin problemas.

- Y será al acabar la SR cuando lleguemos a su pieza *FINAL*.

Fig. 4.31

Durante el transcurso de la ejecución de la SR se habrá creado y utilizado la variable "*f*", que será la que devolverá la SR. Además, vemos como "*f*" es de tipo *Entero*. Esta variable seguirá pues el camino inverso de aquellas que recogió la SR del PP. Para ello, el método *fin* la añadirá al almacén de variables de intercambio. También con su correspondiente clave, la cual cambia ligeramente su identificación con respecto a las invocadas por la subrutina. Como ahora la variable va a

ser sólo una, y en sentido de devolución, la clave, aunque respetando la nomenclatura creada para la misma, será:

"*Nombre de la subrutina_dev*"

donde con "*dev*" queremos hacer ver que es la variable que devuelve dicha subrutina.

Pero todavía queda un último asunto por tratar. ¿Concuerdan la variable que devuelve la SR con la que espera el PP? (ver las Tablas de Edición de los iconos *SUBR Fig. 4.30* y *FINAL Fig. 4.31* para ponerse en situación).

El PP espera una variable, cuyo valor será almacenado en "*z*", y de tipo *Entero*. ¿Corresponde con el tipo de la que devuelve la SB? Sí, por lo que en este caso no hay error.

Una vez acabado este proceso, únicamente resta un aspecto para volver a ejecutar el PP, y es el de recuperar toda la información necesaria, y que ya almacenamos, para continuar con la ejecución del PP desde el mismo punto en que la interrumpimos.

- Una vez hecho todo ésto, y como última fase del proceso, el PP continúa ejecutándose normalmente hasta llegar a su correspondiente pieza *FINAL*, con la que finaliza.

5. Traducción.

La implementación de la traducción con respecto a los elementos de este bloque fue un proceso muy delicado. Debido a que la utilización de éste se da, de momento, a *NIVEL INTERMEDIO*, todo lo aquí concebido se refleja en la clase *PluginINTERM*.

Para entender el funcionamiento del código encargado de este apartado, debemos ser conscientes de que cada proceso de ejecución inicializa también uno de traducción. No lo inicializa exactamente *Ejecutor*, pero si son dos procesos totalmente asociados. Este hecho lo podemos ver en el método *connEtoM2* perteneciente a la clase *NWTJava*, donde dos de sus líneas de código, precisamente una seguida de la otra, inicializan una el proceso de ejecución y la otra el de traducción. Por tanto, el problema que aquí se nos presentaba es que la traducción seguía fielmente a las ejecuciones del PP y de las SRs asociadas, con la contrariedad todavía mayor, de que cada vez que establecíamos un nuevo diagrama como diagrama actual (recordemos que este paso era fundamental para poder ejecutar dicho diagrama), la traducción se reinicializaba.

Lógicamente, el resultado mostrado en el *Área de Traducción* no era el correcto. Para solucionar esta cuestión, tuvimos que ir al método con el que etiquetábamos a un diagrama determinado como el actual. Este era *setCurrent* de *Datos*. Para que todo funcionase correctamente tuvimos que modificarlo, le

añadimos un parámetro más. Éste consiste en un *boolean* que vale "*true*" si su llamada se realiza desde el método *subr* de *Ejecutor*. Éste es el caso en que se va a ejecutar una SR, y lo que hacemos presenta dos líneas de acción:

1. no permitir que se reinicialice la traducción (así no perdemos lo traducido hasta ahora), y

2. anular la traducción correspondiente al diagrama SR (de esta forma, no se incluye en el *Área de Traducción* los iconos pertenecientes a la SR).

Una vez llegados a este punto, en el bloque de *PluginINTERM* perteneciente al icono *SUBR*, se recoge el contenido de todos sus parámetros y se representan de la siguiente forma:

"*tipo devolución = nombre_subr(argumento1, argumento2,...);* "

Para nuestro ejemplo sería:

" *int z = p_subr51(a , d);* "

Esta línea Java incluye ya toda la ejecución de *p_subr51.nwt*, como si fuese exactamente igual a la ejecución de un método.

En caso de que la SR no devolviese nada, la instrucción reflejada sería:

"*nombre_subr(argumento1, argumento2,...);* "

Y con lo descrito hasta ahora damos por finalizado el punto número 5, y con ello, el apartado de las fases seguidas para la implantación de este bloque. Ahora si podemos decir con todas las garantías, que NWTJava v2.0 introduce a los alumnos en los principios de la programación estructurada. Pasamos pues al último y definitivo bloque.

4.2.5 Bloque Ámbito de Variables

Como ya mencionamos al inicio del anterior bloque, a lo largo de la implementación, el tiempo y volumen de codificación fue variando en relación al trabajo de creación de ideas, conceptos y su planificación. Comentábamos que la carga asociada a la codificación en los dos primeros bloques alcanzó una cota del 90%. Pues bien, para este cuarto y último bloque, dicho porcentaje fue totalmente opuesto. Prácticamente todo el trabajo realizado para la implementación de los ámbitos de variable fue conceptual. Ideando, discurriendo y reflexionando sobre cual sería la forma más apropiada de acometer su solución final. En el transcurso de esta fase, surgieron dificultades, pero sólo una pequeña parte de ellas fueron resueltas tras ver los resultados sobre el código. Por ello es por lo que la relación de trabajo realizado en este bloque corresponde a un 10/90% donde el primer porcentaje corresponde al tiempo y volumen asociado a la codificación.

Una vez ilustrado estos datos, pasamos a describir la implementación propiamente dicha de este bloque.

Su objetivo es el de proporcionar ámbitos de validez para las variables. Hasta este punto, todas las variables utilizadas en un programa han sido globales, es decir, una vez declaradas eran visibles desde cualquier punto del diagrama y ocupaban una posición de memoria (en nuestro almacén de variables) fija.

Sin embargo, lo que buscamos ahora es que estas variables dejen de ser globales y posean las siguientes características:

- Una vez declaradas en el interior de un bloque, valgan solo allí.
- Que solamente sean visibles dentro de ese bloque, y su valor se pierde al finalizar éste.
- Que se creen al entrar a ejecutarse el bloque, en su stack correspondiente (como veremos más adelante, implementaremos una pila de ámbitos).

Pues bien, para alcanzar y conseguir estos requisitos, lo primero que vimos es que la estructura del almacén de variables heredado de NWTJava v1.0 no era adecuada. Se trataba de un vector lineal, de dimensiones 1x(el número de variables empleadas en el diagrama). De esta forma, su funcionamiento consistía en ir introduciendo indiscriminadamente todas y cada una de las variables creadas, una detrás de otra. Todas estas variables eran manejadas, por tanto, al mismo nivel, y siempre que se requería una búsqueda, dicho vector era recorrido entero, independientemente de si la variable pertenecía a una estructura de bucle, a una condicional o había sido creada fuera de cualquiera de éstas.

Vamos a ilustrar dicha estructura mediante una figura, y a continuación pasaremos a comentar y razonar el nuevo modelo seguido, el cual es la base y el elemento sobre el que pivota todo el funcionamiento de este bloque:

Vector *variables*

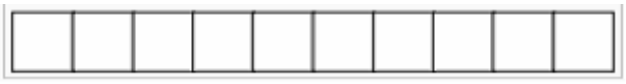


Fig. 4.32

Estructura origen del almacén de variables. Todas las variables eran introducidas a un mismo nivel, en un mismo ámbito (ámbito global).

Nuevo Vector *variables*

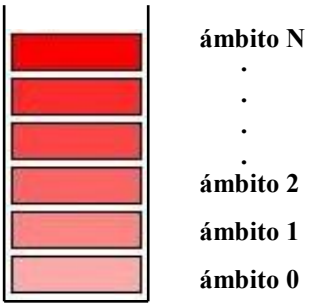


Fig. 4.33

Y es en esta *Fig. 4.33* donde podemos analizar la evolución y reestructuración del almacén de variables. Este almacén, atributo de la clase *Ejecutor* cuyo identificador es "*variables*", pasa de ser un vector unidimensional a otro bidimensional. Ahora es un vector de vectores, donde cada uno de ellos representa un ámbito diferente. Cada ámbito de estos, asociado a los diferentes niveles de profundidad que alcanza un diagrama con su correspondiente anidamiento de bloques de código, es el encargado de almacenar sus variables asociadas.

El funcionamiento es el de una pila, donde vamos almacenando tantos sub-almacenes locales como sea necesario.

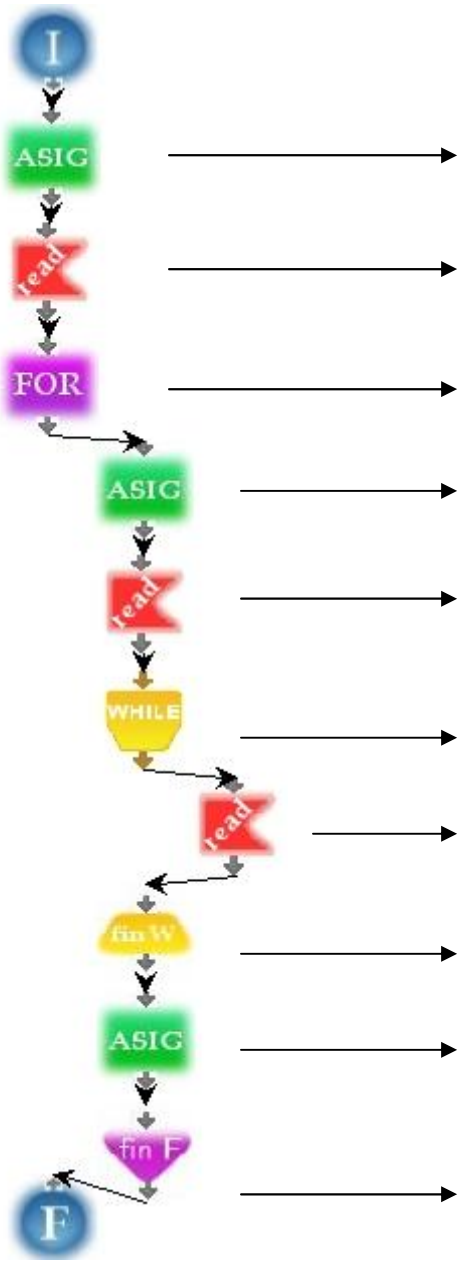
Ahora, y una vez conocida la nueva estructura del almacén, lo que vamos a hacer es describir su funcionamiento, al tiempo que comentaremos también la implementación seguida para lograr que esta maquinaria alcance un óptimo rendimiento.

Son cuatro los procesos básicos que vamos a describir, y que nos van a servir para entender a la perfección el proceso seguido por este bloque.

- Añadir nuevo ámbito local.
- Eliminar ámbito local.
- Insertar variable.
- Resto: Búsqueda de variables y Eliminación de variables.

Pero antes de meternos en materia vamos a, mediante un diagrama, mostrar un ejemplo de lo que sería la ejecución de un programa desde el punto de vista de este último

bloque. Éste no se encarga de nada referente a tipos de variable, arrays, invocación de subrutinas… De lo que se encarga es de gestionar el manejo de ámbitos de variable. Y por ello, podemos dar a entender su funcionamiento básico mediante el siguiente esquema:



Ámbito	Nombre Variable	
0	a	
0	b	
1	i	
1	c	
1	a	(modificará el valor de la variable "*a*" creada ya en un ámbito más global)
2	-	
2	d	
1	-	
1	d	(nueva variable d, perteneciente a un ámbito más general que la anterior variable d)
0	-	

Fig. 4.34

Como podemos observar, el modelo presenta una serie de patrones que se repiten dependiendo del tipo de estructura utilizado. Por ejemplo: siempre que utilizamos un icono de estructura de bucle (ídem para estructuras condicionales), creamos un nuevo ámbito de variables, el cual será local y asociado a dicho bloque. El proceso inverso tiene lugar cuando llegamos al correspondiente icono de fin de bucle, donde eliminamos el correspondiente ámbito, y con él, sus respectivas variables. Pero ésto lo iremos viendo y analizando poco a poco.

- Añadir nuevo ámbito local.

Como ya hemos comentando, el vector *variables* (el atributo de la clase *Ejecutor* que funciona como almacén de variables) es un vector de vectores, donde todos y cada uno de estos vectores actúa como un mini-almacén asociado a un ámbito.

A cada uno de estos vectores lo denominamos *ámbito* y lo identificamos mediante su posición en *variables*, es decir: *ambito[0]*, *ambito[1], ambito[2]*, …donde el correspondiente al [0] es el ámbito base, por así decirlo, el más global de todos.

Pues bien, sabido ésto ya podemos decir que todo el proceso de añadir nuevos ámbitos tiene lugar, exclusivamente, en el método *ejecuta* de *Ejecutor*. Si recordamos, este método se encarga de ir ejecutando pieza a pieza. Y lo que es todavía más importante, en orden. Un orden que nos proporciona *resumeGrafo* de *GraphManager* y que es fiel al establecido por el usuario mediante la dirección de los ejes.

De esta forma, podemos llevar una especie de contador que nos permita saber cuando debemos crear un nuevo ámbito, cuando eliminarlo, y por supuesto, en cuál nos encontramos.

Pero, ¿cuál es la filosofía a seguir para la creación de éstos? Los ámbitos locales deben estar asociados a sub-bloques de código, que por sus características, van a poder utilizar variables particulares incluso con el mismo nombre o identificador que otras variables de otros bloques. Y estos sub-bloques, para nuestro caso van a ser las estructuras condicionales y las iterativas. Por tanto, y a groso modo, podemos decir que cada vez que *ejecuta* descubre que va a invocar una pieza iniciadora de uno de estos sub-bloques, crea un ámbito local.

Son: *FOR*, *WHILE*, *REPEAT* e *IF*.

Por eso, si volvemos a la *Fig. 4.36*, vemos como al comenzar la ejecución nos encontramos en el ámbito más general posible, el [0], pero al llegar al *FOR*, *ejecuta* interpreta que va a comenzar a ejecutar un sub-bloque de código y añade uno nuevo. A lo largo de la ejecución de este sub-bloque, se encuentra con otra estructura de este tipo, en este caso un *WHILE*, por lo que vuelve a crear otro ámbito aún más local que el anterior (*ambito[2]*).

Y así sucesivamente.

- Eliminar ámbito local.

Sin embargo, también se dará la ocasión en que dichos sub-bloques finalicen, y por tanto, deba terminar su ejecución. Es este el momento en que su ámbito asociado ya no tiene razón de seguir existiendo. Debe ser eliminado. ¿Cuándo ocurre ésto? Desde el instante en que *ejecuta* distingue que, la pieza alcanzada simboliza el fin de una de estas estructuras o sub-bloques.

Por tanto, si el icono que toca ejecutar es: *END_FOR*, *END_WHILE*, *UNTIL* o *END_IF*, eliminamos el último ámbito creado.

Gracias al orden de las piezas, elaborado por *resumeGrafo*, y a la estructura de pila *FIFO* (*First In – First Out*) del almacén de variables (podríamos decir también: de ámbitos), nos aseguramos que el último creado, el más local, va a ser también el primero en ser eliminado[95].

- Insertar variable.

Nos falta ahora ver como insertamos las variables en sus correspondientes ámbitos de pertenencia. Al igual que en los dos anteriores puntos el método encargado era *ejecuta*, aquí lo es la familia de métodos *buscaVariable*, de la que ya hemos hablado en más de una ocasión.

Para la implementación de este aspecto, tan fundamental para el desarrollo no sólo de este bloque sino del funcionamiento general de la herramienta, el primer camino que tomamos fue erróneo.

Una vez ideado y planeado este primer plan de ataque, con el que íbamos a acometer esta tarea, nos percatamos de que no era el camino adecuado. Éste problema nos supuso varios días de trabajo infructuoso. Afortunadamente nos sirvió para entender la verdadera magnitud del problema al que nos enfrentábamos, y así poder encauzar debidamente la resolución del mismo.

Esta primera implementación consistía en dividir cada método *buscaVariable* en dos:

- *buscaVariable*
- *añadeVariable*

El primero se encargaría de la primera parte de los antiguos *buscaVariable*, cuya función era la de recorrer el almacén en busca de la variable pasada como parámetro. Pasaba de recorrer el vector *variables* a recorrer el vector de vectores *variables*.

Su búsqueda comenzaba desde el último ámbito creado. Si la encontraba, la retornaba. Si no era ese el caso, buscaba en el siguiente ámbito, y así sucesivamente hasta llegar al [0], al más general. En caso de no encontrarla en ninguno de ellos, el método devolvía *null*, y era entonces, y sólo entonces, cuando *añadeVariable* entraba en acción. Su función sería la de, al no existir dicha variable, crearla y añadirla al ámbito correspondiente. ¿Y cuál sería este ámbito? Siempre el último, pues es el asociado al bloque que se está ejecutando.

Sin embargo, este proceso era muy complejo. Implicaba una profunda modificación de todos los métodos de la familia *buscaVariable*. Y no sólo eso, también suponía reformar todo el código adyacente a sus llamadas, puesto que,

[95] Un hecho que podemos comprobar recurriendo de nuevo a la Fig. 4.34.

tanto los parámetros como lo devuelto por dichos métodos cambiaba drásticamente.

Por estos motivos, nos vimos obligados a plantear una nueva solución. Y la encontramos. Una infinitamente más sencilla que la anterior y que suponía únicamente la inclusión de dos pequeños cambios en cado uno de los métodos *buscaVariable*. Además, ya no sería necesario modificar ni sus invocaciones ni el código contiguo a éstas.

Esta solución estructuró los métodos *buscaVariable* de la siguiente forma (estructura general):

```
Variable buscaVariable(String nom_variable) {

    recorrerVariables(nom_variable);

    if ( no encontrada la variable → NO EXISTE ){

        crea la variable "nom_variable" (con su
        correspondiente valor por defecto. Ej: Real → 0.0
        Booleano → false);

        nueva forma de añadir variables;

    }

}
```

El primero de estos dos cambios es la nueva forma de buscar variables en el almacén. De ésto se encarga el método recién creado y diseñado para ello: *recorrerVariables* Antes, el *buscaVariable* presentaba un bloque de líneas de código encargadas de la búsqueda de las variables en el antiguo vector lineal. Ahora es este nuevo método el que recorre el vector de vectores *variables*.

Y el segundo es el que hemos querido mostrar con la línea "*nueva forma de añadir variables*". Anteriormente, la instrucción con que se añadía una nueva variable al almacén era:

```
variables.add(nom_variable);
```

Ahora es:

```
( (Vector)( variables.lastElement( ) ) ).add(nom_variable);
```

Se la añadimos al último ámbito del almacén.

- Resto: Búsqueda de variables y Eliminación de variables.

Hemos denominado a éste último punto como *Resto* debido a que, por el orden natural del índice de puntos propuesto, faltarían por explicar los procesos que incluye: *Búsqueda de variables* y *Eliminación de variables*.

Sin embargo, debido al momento en el que nos encontramos, ni siquiera es necesario hacerlo, y vamos a ver muy rápidamente porqué.

Si nos percatamos, el proceso de Búsqueda de variables ya lo hemos comentado, sólo que estaba inmerso en el punto de Insertar variable. Si volvemos la vista atrás, podemos ver como es la primera parte de los métodos *buscaVariable*. Es *recorrerVariables* el que realiza la búsqueda, devolviendo la variable si la encuentra, o *null* si por el contrario no existe.

Algo parecido ocurre con la *Eliminación de variables*, aunque este proceso ha quedado un poco más oculto en uno de puntos citados. Dicho punto es el de *Eliminar ámbito local*, donde ni tan siquiera hemos hecho hincapié en la eliminación de variables puesto que ésta se realiza automáticamente con la destrucción del ámbito al que corresponden.

Y con ésto hemos descrito ya la implementación de los procesos fundamentales seguidos por este cuarto y último bloque. Sin embargo, llegados a este punto nos encontramos con una dificultad más, que podemos describir como: de índole esencial.

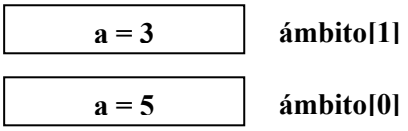
Este problema consistía en que *JEP* no era capaz de respetar los ámbitos de variables.

El motivo por el cual ocurre ésto es el siguiente: con todas y cada una de las variables, y con objeto de que las reconozca *JEP* (para el empleo de las mismas en las operaciones), debemos utilizar la siguiente instrucción:

myParser.addVariable(nombre, valor);

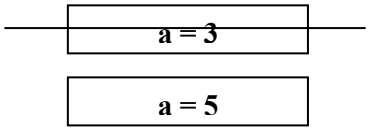
¿Qué hace *JEP* mediante esta instrucción? Añade una variable con su correspondiente par nombre-valor a su propio almacén, pero, y e aquí donde viene el problema, este almacén es también lineal, por tanto se nos da la siguiente problemática que vamos a comentar con ayuda de un ejemplo:

Tenemos 2 ámbitos:



JEP, en el momento que le indiquemos que asocie a "*a*" el valor 3, va a pisar el antiguo, en este caso, el valor 5, sin respetar ámbitos.

Por eso es necesario, que aunque eliminemos el último ámbito,

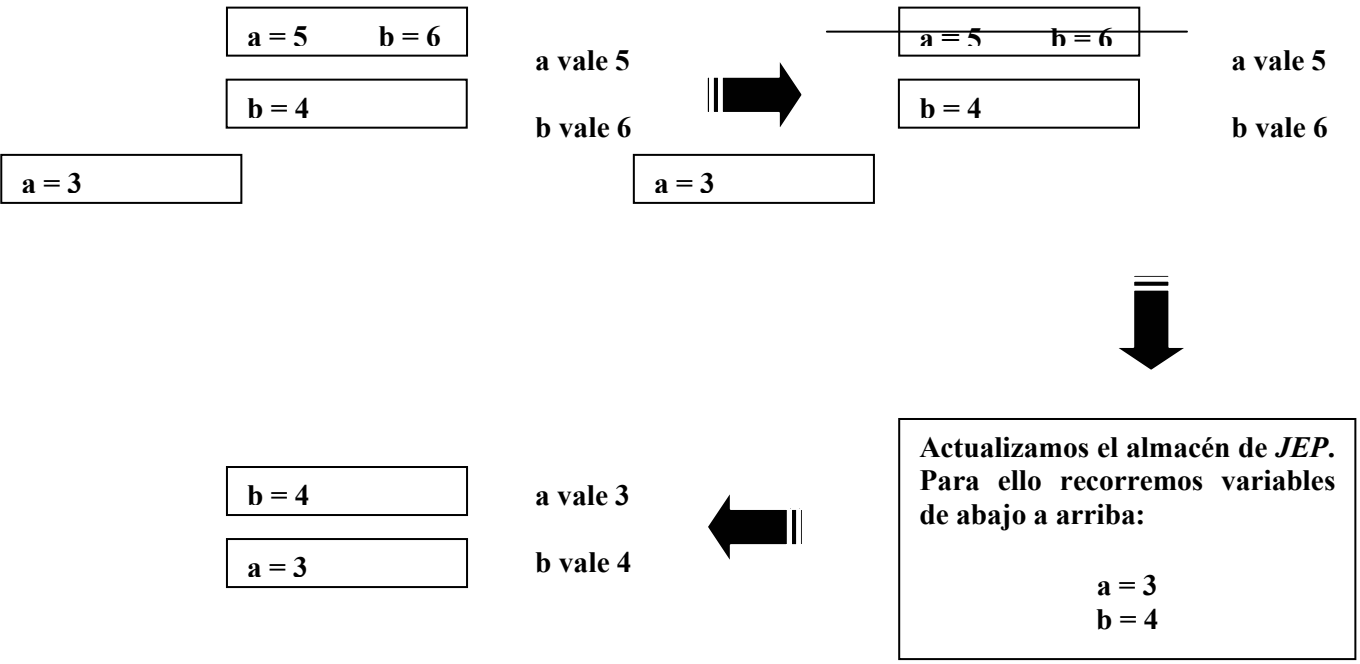


recorramos el resto y vayamos actualizando los valores de las variables, pues en este caso, aunque eliminemos el último ámbito, para *JEP*, "*a*" sigue valiendo 3.

Y para este propósito fue para el que creamos el método *actualizarVariables*. Una vez eliminamos el último ámbito, *actualizarVariables* recorre los restantes con el fin de ir actualizando en *JEP* las variables que quedan.

Con el siguiente ejemplo, mucho más extenso que el caso recién comentado, el proceso por el que hacemos que *JEP* "respete ámbitos" quedará mucho más claro y notorio:

Inciso: las variables representadas dentro de los ámbitos son las almacenadas en nuestro particular almacén *variables*, mientras que las que quedan fuera representan a las variables almacenadas por *JEP*.



Y de esta forma conseguimos que tanto las variables manejadas por nosotros como por *JEP* siempre coincidan plenamente en todas sus características. Podemos decir que hemos conseguido que el funcionamiento de *JEP* sea análogo al de utilizar y respetar ámbitos de variable.

Y con ésto hemos acabado la descripción de la implementación seguida para los cuatro bloques básicos y el quinto especial. Es con ello con lo que conseguimos la obtención y consecución de todos y cada uno de los requisitos propuestos.

Sólo nos queda, y para dar por finalizado el tema de la implementación, comentar que nos vimos obligados a generar una nueva *DTD* adaptada a los requerimientos del *NIVEL INTERMEDIO*, diferente en algunos aspectos a la ya existente para NWTJava v1.0.

Estas diferencias básicamente se refieren a la utilización de nuevas estructuras (*SUBR*) y la inclusión de nuevos atributos en las piezas ya existentes ("*tipo*" en *ASSIGN*, *READ*, *FOR*, *FINAL*, "*argumentos*" en *INICIO*, etc.).

Por lo tanto, podemos decir que actualmente coexisten dos DTDs: la propia de NWTJava v1.0, la cual adopta la segunda versión para el *NIVEL BÁSICO*, y la de reciente creación, que como ya hemos comentado se encarga de hacer respetar el conjunto de reglas gramaticales asociados al *NIVEL INTERMEDIO*.

Por tanto, y para respetar las citadas reglas gramaticales, a partir de las cuales se deben construir los archivos de NWTJava, NWTJava v2.0 utilizará una *DTD* u otra dependiendo del nivel seleccionado por el usuario. Estas *DTDs*[96] reciben el nombre de: *nwtdtd.dtd* y *nwtdtdINTERMED.dtd*.

Podemos ver ambas en el *APENDICE A* de esta memoria.

[96] Referenciamos el tutorial para la construcción de DTDs del que hicimos uso: [19].

Capítulo 5:

Conclusiones y Trabajos Futuros

5.1 CONCLUSIONES 160
5.2 TRABAJOS FUTUROS 162

5.1 Conclusiones

En este proyecto hemos ampliado una herramienta multimedia e interactiva, desarrollando así NWTJava v2.0, cuyo objetivo primordial es el de servir de apoyo a la enseñanza de la programación estructurada bajo las siguientes directrices:

- Focalizar y hace hincapié en los conceptos básicos más universales de la programación estructurada y en la lógica propia de esta disciplina.
- Basarse en paradigmas de actuación hasta cierto punto conocidos por el alumno.
- Permitir un alto grado de interactividad y de operatividad.
- Proporcionar realimentación al alumno, permitiéndole así observar los resultados de sus acciones.
- Mantener un cierto enfoque lúdico.

Siendo conscientes de ésto, podemos ahora analizar las conclusiones dirimidas, producto todas ellas del trabajo realizado durante estos meses. Para un mejor estudio de las mismas, vamos a plasmar el análisis bajo dos perspectivas: trabajo realizado y resultados obtenidos.

En primer lugar, destacar que uno de los mayores retos al que nos tuvimos que enfrentar fue el de tener que trabajar sobre un código preexistente, un esqueleto y unas estructuras ya construidas y formalizadas. Ello nos obligaba en primer lugar, a buscar soluciones que fuesen lo más compatible posible con el modelo ya establecido. Como ya hemos comentado debidamente en el capítulo de diseño e implementación, alcanzar esta solución no siempre era posible. Ello daba lugar, lógicamente, al consiguiente inconveniente de tener que desmontar las obsoletas estructuras y construir unas nuevas, adaptadas tanto a la base que la soportaba como a lo requerido por el objetivo y las circunstancias. El superar este reto ha sido pues, uno de los mayores logros de este proyecto.

En la primera fase del proyecto, análisis global de la herramienta NWTJava v1.0 (de la cual podemos encontrar su descripción en el apartado 1.4), el objetivo fundamental fue el de introducirse todo lo posible en la aplicación NWTJava original: conocer y dominar su manejo, estudiar sus procesos internos, analizar los pilares en que se sustentaba su razón de ser y su funcionamiento, etcétera. Ello nos permitió conocer, de forma pormenorizada cuales eran sus puntos fuertes y débiles, sus fortalezas y debilidades, y como no, su manejo externo y su funcionamiento interno.

Con objeto de comprobar la validez de las conclusiones alcanzadas a priori, corregirlas en lo necesario, y descubrir aquellas nuevas que se nos podrían haber escapado, realizamos un experimento de campo en el contexto de una asignatura de programación

de primer curso de ingeniería con alumnos voluntarios. Lo realizamos en formato de seminario y lo denominamos TSIOCA (ver *Capítulo 3: EXPERIMENTACIÓN*).

A raíz de los resultados obtenidos tras su realización, y aportando ahora la visión a posteriori de la que contamos en este capítulo, extraemos las siguientes conclusiones.

De la observación de los alumnos durante el experimento, y de sus opiniones personales, se puede concluir que, en efecto, NWTJava ayuda a clarificar los conceptos y mejora la comprensión de la programación, proporcionándoles un apoyo útil durante las primeras etapas del aprendizaje. Sin embargo, su objetivo principal no era el recién comentado, sino el de realizar un estudio acerca de posibles mejoras a incluir en NWTJava v2.0. Aunque en su respectivo momento, la opinión sobre el logro de este objetivo fue de "muy satisfactorio", no podíamos asegurar su éxito hasta llegado este punto.

No obstante, nos hizo también encontrar aspectos a pulir y sobre los que seguir trabajando. La principal es que NWTJava v2.0 puede quedarse corta rápidamente en un entorno universitario, en el que los temarios avanzan con rapidez y los alumnos tienen mayor capacidad de aprendizaje. La conclusión inevitable es que el ámbito ideal de aplicación de NWTJava v2.0 estaría en las primeras semanas de las asignaturas de programación. También en cursos preuniversitarios e incluso en institutos.

Posteriormente realizamos un estudio sobre las experiencias y tecnologías que enfocan y abordan la enseñanza de la programación estructurada.

Frente al método de docencia más extendido hoy día, basado en enseñar a programar con un lenguaje de programación particular, utilizando su sintaxis y su semántica, nuestro enfoque: utilizar un lenguaje algorítmico lo bastante general como para ser capaces después, de utilizar lo aprendido en el uso posterior de cualquier lenguaje de programación concreto.

El análisis de las mismas parece respaldar el valor y eficacia de las técnicas y estructuras escogidas. De hecho, todos y cada uno de los bloques en que hemos estructurado la elaboración de esta nueva versión tienen sus pilares en la combinación de las conclusiones extraídas tanto en experimento de campo como en el camino recorrido por estas tecnologías alternativas al modelo estándar o habitual.

A raíz de esta situación, la metodología de trabajo a seguir era clara: potenciar aun más si cabía las fortalezas y trabajar duramente sobre las debilidades para convertirlas también en fortalezas.

De esta forma, y bajo toda esta casuística (docencia, programación estructurada, implementación sobre un código y unas estructuras preexistentes,…), es como se ha desarrollado NWTJava v2.0.

En resumen, y como conclusión final, podemos decir que todo lo visto hasta el momento, unido a las opiniones de los alumnos en las que califican a NWTJava como una herramienta muy útil, más la crítica de que necesitaba profundizar más en sus conceptos y añadir nuevas estructuras, nos hace indicar que NWTJava v2.0 puede ser una aplicación, que junto con un correcto modo de empleo, e integrado en un sistema de

enseñanza, al menos alternativo, puede resultar muy beneficioso para la docencia y el aprendizaje de la programación estructurada. Hacemos indicar también que su mayor eficacia se alcanzaría en las primeras etapas del aprendizaje, es decir, el ámbito ideal de aplicación de NWTJava v2.0 estaría en las asignaturas de programación del primer curso universitario.

5.2 Trabajos futuros

Y para que esta herramienta siga progresando y evolucionando, en pos de una mejor y más eficaz enseñanza de la programación, es para lo que proponemos la siguiente línea de trabajos futuros.

El objetivo es siempre seguir una línea ascendente de calidad y eficacia en la enseñanza de este mundo de la programación, cuyos resultados redunden en un modelo docente basado en la sencillez y en una estructura progresiva de aprendizaje con mayor rendimiento y eficacia.

Dos son los caminos de avance propuestos. El primero de ellos incide directamente sobre la aplicación actual, mejoras que afectan a lo ya creado y estructurado. El segundo, plantea dos posibles alternativas de desarrollo para una tercera versión de NWTJava. Lo vemos:

1. Para hacer de NWTJava v2.0 un entorno de desarrollo más completo se podrían implementar las siguientes mejoras en la aplicación:

- Añadir la posibilidad de agregar cuadros de texto en el *Área de Trabajo* con el fin de actuar éstos como comentarios del diagrama.
- Hacer accesible el *Manual de Usuario* desde la aplicación. Se podría acceder a él fácilmente desde una nueva entrada situada en el menú principal: "*Ayuda*".
- Ampliar el número de lenguajes reales y concretos a los que poder traducir el diagrama creado.
- Asociar diferentes sucesos con animaciones multimedia tales como sonidos (de acierto y error), imágenes animadas que ofrezcan determinada información en momentos concretos, etc…
- Creación de animaciones flash en las que mostrar virtualmente la creación de algunos diagramas, muy útiles a modo de ejemplo y guía para los usuarios.

2. Las dos alternativas planteadas a priori para el diseño, desarrollo y creación de la tercera versión de NWTJava son:

- Recuperamos el fragmento de texto en el que justificábamos cuál era el salto cualitativo que debíamos dar de NWTJava original a NWTJava v2.0:

"*...¿cuál es el siguiente paso que debemos dar? Acudiendo a la evolución histórica sufrida por el mundo de la programación, observamos la siguiente sucesión:*

Programación mediante procedimientos → ***Programación estructurada*** → ***Programación orientada a objetos***

La lógica pues, era aplastante. Tras la concepción de los elementos básicos de la programación en la primera versión, tocaba incluir en esta segunda los referentes a la programación estructurada."

Por tanto, y siguiendo la dicha lógica, tras el *Nivel Intermedio* concebido para la programación estructurada, surge la posibilidad de ampliar la herramienta con un *Nivel Avanzado* dedicado a la programación orientada a objetos.

Esta nueva funcionalidad se utilizaría única y exclusivamente después de que los alumnos hubiesen superado satisfactoriamente los niveles *Básico* e *Intermedio* (siempre y cuando el profesor encargado no indicase lo contrario).

Sin embargo, esta opción, que a priori era indiscutible, ha ido perdiendo fuerza por dos motivos:

- La aparición en estos últimos años de multitud de sencillas y potentes herramientas que parecen cumplir la función de enseñar POO con muchas garantías. Ejemplos tales como *Squeak* y su lenguaje puro orientado a objetos *Smalltalk*[97] o *Alice*, un software que simula un innovador ambiente en tres dimensiones en el que el usuario puede, a modo de juego interactivo, recrear una animación mediante el uso de objetos, son aplicaciones diseñadas con el objetivo de ser una primera exposición para estudiantes de la programación orientada a objetos.

- El haber contemplado una segunda opción más que factible, de la cual hablamos a continuación.

[97] Podemos encontrar enlaces para conocer la herramienta Squeak y su lenguaje de programación asociado Smalltalk en las referencias [20] y [21].

- Centrarse y especializarse en la enseñanza de la programación estructurada, construyendo una rica y sofisticada biblioteca con multitud de funciones, las cuales serían utilizadas por los usuarios en la resolución de ejercicios y en la construcción de sus propios programas.

  Esta idea iría encaminada a aprovechar todavía más el potencial que puede llegar a desplegar el tercero de los bloques principales que hemos descrito en la implementación del programa: la opción de utilizar subrutinas (funciones).

  Además, estas nuevas subrutinas no tendrían que estar creadas con NWTJava, sino que podrían crearse en Java directamente y ser llamadas mediante objetos subrutinas (imitando de esta manera la forma en que la API de Java nos ofrece métodos que están implementados en código nativo).

Apéndice A

DTDs

newtdtd

```
<!-- edited with XMLSPY v5 rel. 2 U (http://www.xmlspy.com) by
Patricia (uc3m) -->
<!-- DTD para la definicion de datos en Newt-->
<!-- Entidad para definir dos atributos comunes a todos los elementos:
nombre (identificador del elemento que se guardarÃ¡ en las clases
Java) y ref (referencia al objeto Java asociado) -->
<!ENTITY % coreattr "nombre ID  #REQUIRED
       ref CDATA #REQUIRED
       ">
<!-- Elemento de inicio del diagrama: contiene el programa entero y el
elemento de final -->
<!ELEMENT begin ((read | write | assign | if | for | while | repeat)+,
end)>
<!ATTLIST begin
      %coreattr;
>
<!ELEMENT end EMPTY>
<!ATTLIST end
      %coreattr;
>
<!-- Representa la instrucciÃ³n bÃ¡sica de entrada de datos: lee una
lista no nula de variables de cualquier tipo -->
<!ELEMENT read EMPTY>
<!ATTLIST read
      %coreattr;
      variable CDATA #REQUIRED
      texto CDATA #REQUIRED
>
<!-- Representa la instrucciÃ³n bÃ¡sica de salida de datos: escribe
una lista no nula de variables de cualquier tipo -->
<!ELEMENT write EMPTY>
<!ATTLIST write
      %coreattr;
      variable CDATA #REQUIRED
      texto CDATA #REQUIRED
>
<!-- AsignaciÃ³n de un valor a una variable, el valor se representa
como una expresiÃ³n -->
<!ELEMENT assign EMPTY>
<!ATTLIST assign
      %coreattr;
      variable CDATA #REQUIRED
      valor CDATA #REQUIRED
>
<!-- Sentencia condicional: su condiciÃ³n viene dada por su hijo
espresiÃ³n; el elemento "else" hijo es opcional  -->
<!ELEMENT if (((read | write | assign | if | for | while | repeat)+,
else?, endif)|(else?,(read | write | assign | if | for | while |
repeat)+,endif))>
<!ATTLIST if
      %coreattr;
      condicion CDATA #REQUIRED
>
```

```
<!-- contiene el bloque de instrucciones que se ejecutan si falla la
condiciÃ³n del if -->
<!ELEMENT else ((read | write | assign | if | for | while |
repeat)+,endelse)>
<!ATTLIST else
      %coreattr;
>
<!ELEMENT endelse EMPTY>
<!ATTLIST endelse
      %coreattr;
>

<!-- Indica el fin de una sentencia if: se incluye como elemento
porque tendrÃ¡ que existir un onjeto Java asociado en el diagrama  -->
<!ELEMENT endif EMPTY>
<!ATTLIST endif
      %coreattr;
>
<!--  Bucle for: sus atributos inicial y final establecen los valores
correspondientes del contador de iteraciones -->
<!ELEMENT for ((read | write | assign | if | for | while | repeat)+,
endfor)>
<!ATTLIST for
      %coreattr;
      variable CDATA #REQUIRED
      inicial CDATA #REQUIRED
      final CDATA #REQUIRED
      inc CDATA #REQUIRED
>
<!--  Final de un bucle for -->
<!ELEMENT endfor EMPTY>
<!ATTLIST endfor
      %coreattr;
>
<!-- Sentencia while (igual que las anteriores)  -->
<!ELEMENT while ((read | write | assign | if | for | while | repeat)+,
endwhile)>
<!ATTLIST while
      %coreattr;
      condicion CDATA #REQUIRED
>
<!-- Fin de la sentencia while  -->
<!ELEMENT endwhile EMPTY>
<!ATTLIST endwhile
      %coreattr;
>
<!--  Sentencia repeat/until:de nuevo, por consistencia con el
diagrama el final de la sentencia se modela con un elemento until
aparte. -->
<!ELEMENT repeat ((read | write | assign | if | for | while |
repeat)+, until)>
<!ATTLIST repeat
      %coreattr;
>
<!--  En este caso el until lleva asociada la condiciÃ³n del bucle -->
<!ELEMENT until EMPTY>
<!ATTLIST until
      %coreattr;
      condicion CDATA #REQUIRED
>
```

newtdtdINTERM

```
<!-- edited with XMLSPY v5 rel. 2 U (http://www.xmlspy.com) by
Patricia (uc3m) -->
<!-- DTD para la definicion de datos en Newt-->
<!-- Entidad para definir dos atributos comunes a todos los elementos:
nombre (identificador del elemento que se guardará en las clases Java)
y ref (referencia al objeto Java asociado) -->
<!ENTITY % coreattr "nombre ID  #REQUIRED
       ref CDATA #REQUIRED
       ">
<!-- Elemento de inicio del diagrama: contiene el programa entero y el
elemento de final -->
<!ELEMENT begin ((read | write | assign | if | for | while | repeat |
subr)+, end)>
<!ATTLIST begin
      %coreattr;
       argumentos CDATA #REQUIRED
>
<!ELEMENT end EMPTY>
<!ATTLIST end
      %coreattr;
       devolución CDATA #REQUIRED
       tipo CDATA #REQUIRED
>
<!-- Representa la instrucción básica de entrada de datos: lee una
lista no nula de variables de cualquier tipo -->
<!ELEMENT read EMPTY>
<!ATTLIST read
      %coreattr;
      variable CDATA #REQUIRED
      texto CDATA #REQUIRED
      tipo CDATA #REQUIRED

>
<!-- Representa la instrucción básica de salida de datos: escribe una
lista no nula de variables de cualquier tipo -->
<!ELEMENT write EMPTY>
<!ATTLIST write
      %coreattr;
      variable CDATA #REQUIRED
      texto CDATA #REQUIRED
>
<!-- Asignación de un valor a una variable, el valor se representa
como una expresión -->
<!ELEMENT assign EMPTY>
<!ATTLIST assign
      %coreattr;
      variable CDATA #REQUIRED
      valor CDATA #REQUIRED
      tipo CDATA #REQUIRED
>
<!-- Sentencia condicional: su condición viene dada por su hijo
espresión; el elemento "else" hijo es opcional  -->
<!ELEMENT if (((read | write | assign | if | for | while | repeat |
subr)+, else?, endif)|(else?,(read | write | assign | if | for | while
| repeat | subr)+,endif))>
```

```
<!ATTLIST if
      %coreattr;
      condicion CDATA #REQUIRED
>
<!-- contiene el bloque de instrucciones que se ejecutan si falla la
condición del if -->
<!ELEMENT else ((read | write | assign | if | for | while | repeat |
subr)+,endelse)>
<!ATTLIST else
      %coreattr;
>
<!ELEMENT endelse EMPTY>
<!ATTLIST endelse
      %coreattr;
>
<!-- Indica el fin de una sentencia if: se incluye como elemento
porque tendrá que existir un onjeto Java asociado en el diagrama  -->
<!ELEMENT endif EMPTY>
<!ATTLIST endif
      %coreattr;
>
<!--  Bucle for: sus atributos inicial y final establecen los valores
correspondientes del contador de iteraciones -->
<!ELEMENT for ((read | write | assign | if | for | while | repeat |
subr)+, endfor)>
<!ATTLIST for
      %coreattr;
      variable CDATA #REQUIRED
      inicial CDATA #REQUIRED
      final CDATA #REQUIRED
      inc CDATA #REQUIRED
      tipo CDATA #REQUIRED

>
<!--  Final de un bucle for -->
<!ELEMENT endfor EMPTY>
<!ATTLIST endfor
      %coreattr;
>
<!-- Sentencia while (igual que las anteriores)  -->
<!ELEMENT while ((read | write | assign | if | for | while | repeat |
subr)+, endwhile)>
<!ATTLIST while
      %coreattr;
      condicion CDATA #REQUIRED
>
<!-- Fin de la sentencia while  -->
<!ELEMENT endwhile EMPTY>
<!ATTLIST endwhile
      %coreattr;
>
<!--  Sentencia repeat/until:de nuevo, por consistencia con el
diagrama el final de la sentencia se modela con un elemento until
aparte. -->
<!ELEMENT repeat ((read | write | assign | if | for | while | repeat |
subr)+, until)>
<!ATTLIST repeat
      %coreattr;
>
<!--  En este caso el until lleva asociada la condición del bucle -->
<!ELEMENT until EMPTY>
```

```
<!ATTLIST until
      %coreattr;
      condicion CDATA #REQUIRED
>
<!ELEMENT subr EMPTY>
<!ATTLIST subr
      %coreattr;
       devolución CDATA #REQUIRED
       tipo CDATA #REQUIRED
       argumentos CDATA #REQUIRED
        nombreSub CDATA #REQUIRED
>
```

Apéndice B

Test Previo

Apellidos	
Nombre	
E-mail	

Instrucciones para rellenar el test

El siguiente es un test básico para comprobar las capacidades que tenéis en cuanto al desarrollo de algoritmos simples y al control de las estructuras de control de flujo en las que se basa todo proceso de programación.

No se trata de evaluar vuestros conocimientos sobre Java, sino vuestra forma de pensar y de estructurar los problemas. Por ello, no queremos que escribáis los algoritmos en Java, sino en una especie de PSEUDOCÖDIGO. Las reglas no son en absoluto estrictas, basta con que queden claros los puntos fundamentales del programa: las variables a usar, las condiciones en los bucles (índices, incremento de los mismos,..) etc…

Un ejemplo de lo que podría ser la resolución de uno de estos problemas:

Hacer un programa que pida al usuario su edad y muestre por pantalla si es o no mayor de edad.

```
     Mostrar (¿Edad?)
     Leer (x)                        - donde x es la edad del
usuario. -

     Si (x<18)
     Mostrar (Menor de edad)
     Fin de Si

     Sino
     Mostrar (Mayor de edad)
     Fin de Sino

                          FIN
```

NOTA:
En aquellos ejercicios en los que se pide una salida por pantalla de varios caracteres, cada uno de ellos debe imprimirse por separado en una línea distinta de código.
En cada uno de los casos siguientes escriba en el espacio que se le ofrece un programa que realice las acciones indicadas:

1. Indicar por pantalla si dos números contenidos en las variables N1 y N2 son

iguales o distintos.
(2 puntos).

2. **Un cubo tiene una capacidad de 6 litros. Indicar si dada una cantidad de líquido en litros introducida por el usuario el cubo se desbordaría o no.**
   (3 puntos).

3. **Escribir por pantalla la secuencia de caracteres: 7 3 5 7 3 5 7 3 5 7 3 5 7 3 5.**
   (2 puntos).

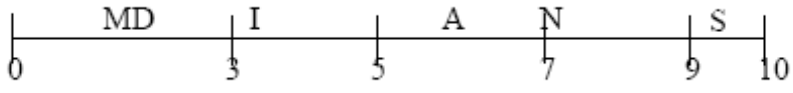
4. **Escribir por pantalla todos los números múltiplos de tres hasta el 99 (incluido).**
   (4 puntos).

5. **Imprimir por pantalla la frase "hola" para siempre (infinitas veces).**
   (2 puntos).

6. **Se tiene una variable N que contendrá un número cuyo valor se desconoce inicialmente. En un solo bucle conseguir lo siguiente: el programa va pidiendo el valor de N al usuario, y termina cuando éste introduce un valor mayor que 5.**
   (4 puntos).

7. **Se pide al usuario que introduzca el valor (entero) inicial de una variable por teclado. En un solo bucle realizar lo siguiente: hasta que la variable no sea mayor que 10, pedir al usuario un número, y sumárselo a la variable.**
   (4 puntos).

8. **Devolver por pantalla la nota "alfabética" correspondiente a una nota numérica que proporcione el usuario, con las siguientes categorías:**
   (4´5 puntos).



9. **Obtener los nombres de los tres alumnos con mejor nota en un examen dada una lista de 200 alumnos.**
   NOTA: suponer que la nota de cada alumno se puede conocer como "Nota_Ai". Dónde "i" es una variable que puede tomar un valor entero.
   (6 puntos).

10. **Convertir a decimal un número en binario que se obtiene leyendo por teclado dígito a dígito (cada cero ó uno en una línea de lectura); y que termina cuando el usuario introduce un -1.**
    NOTA: Se empieza introduciendo la cifra menos significativa.
    (8 puntos).

11. **Escribir por pantalla todas las posibles combinaciones de 3 letras.**
    NOTA: suponer que se puede operar con letras como si se tratase de variables enteras. Pe: a+1=b, a<b, etc…
    (10 puntos).

12. **Ordenar alfabéticamente un array de 100 palabras de cuatro letras.**
    (12 puntos).

Cuestionario inicial

1. **¿Es el 1º año que estudias programación? Sino es así, ¿cuál es tu experiencia anterior?**

2. **¿Cuántos exámenes relacionados con la programación has tenido? ¿A cuántos te has presentado? ¿Cuántos has aprobado?**

3. **¿Piensas que con tus conocimientos deberías haberlos aprobado?**

4. **¿Qué te ha parecido el *Test inicial*? (sencillo, muy complicado, algunos problemas eran accesibles y otros no…)**

5. **Si un "1" corresponde a "*pienso que estoy totalmente perdido*" y un "5" a "*todo lo que se ha dado en clase lo manejo sin dificultades*", ¿qué nota te pondrías?**

6. **¿Tienes clara la diferencia entre "programar" y "utilizar un lenguaje de programación"? ¿Podrías dar una breve definición de cada concepto?**

7. **¿Por qué razón te inscribes en este seminario?**

8. **¿Qué piensas sobre los métodos pedagógicos (o forma de enseñar) con los que se imparten las clases de programación? ¿Cómo los mejorarías?**

Apéndice C

Test Final

Apellidos	
Nombre	
E-mail	

Instrucciones para rellenar el test

El siguiente es un test básico para comprobar las capacidades que tenéis en cuanto al desarrollo de algoritmos simples y al control de las estructuras de control de flujo en las que se basa todo proceso de programación.

No se trata de evaluar vuestros conocimientos sobre Java, sino vuestra forma de pensar y de estructurar los problemas. Por ello, no queremos que escribáis los algoritmos en Java, sino en una especie de PSEUDOCÖDIGO. Las reglas no son en absoluto estrictas, basta con que queden claros los puntos fundamentales del programa: las variables a usar, las condiciones en los bucles (índices, incremento de los mismos,..) etc…

Un ejemplo de lo que podría ser la resolución de uno de estos problemas:

Hacer un programa que pida al usuario su edad y muestre por pantalla si es o no mayor de edad.

```
     Mostrar (¿Edad?)
     Leer (x)                        - donde x es la edad del
usuario. -

     Si (x<18)
     Mostrar (Menor de edad)
     Fin de Si

     Sino
     Mostrar (Mayor de edad)
     Fin de Sino

                          FIN
```

NOTA:
En aquellos ejercicios en los que se pide una salida por pantalla de varios caracteres, cada uno de ellos debe imprimirse por separado en una línea distinta de código.

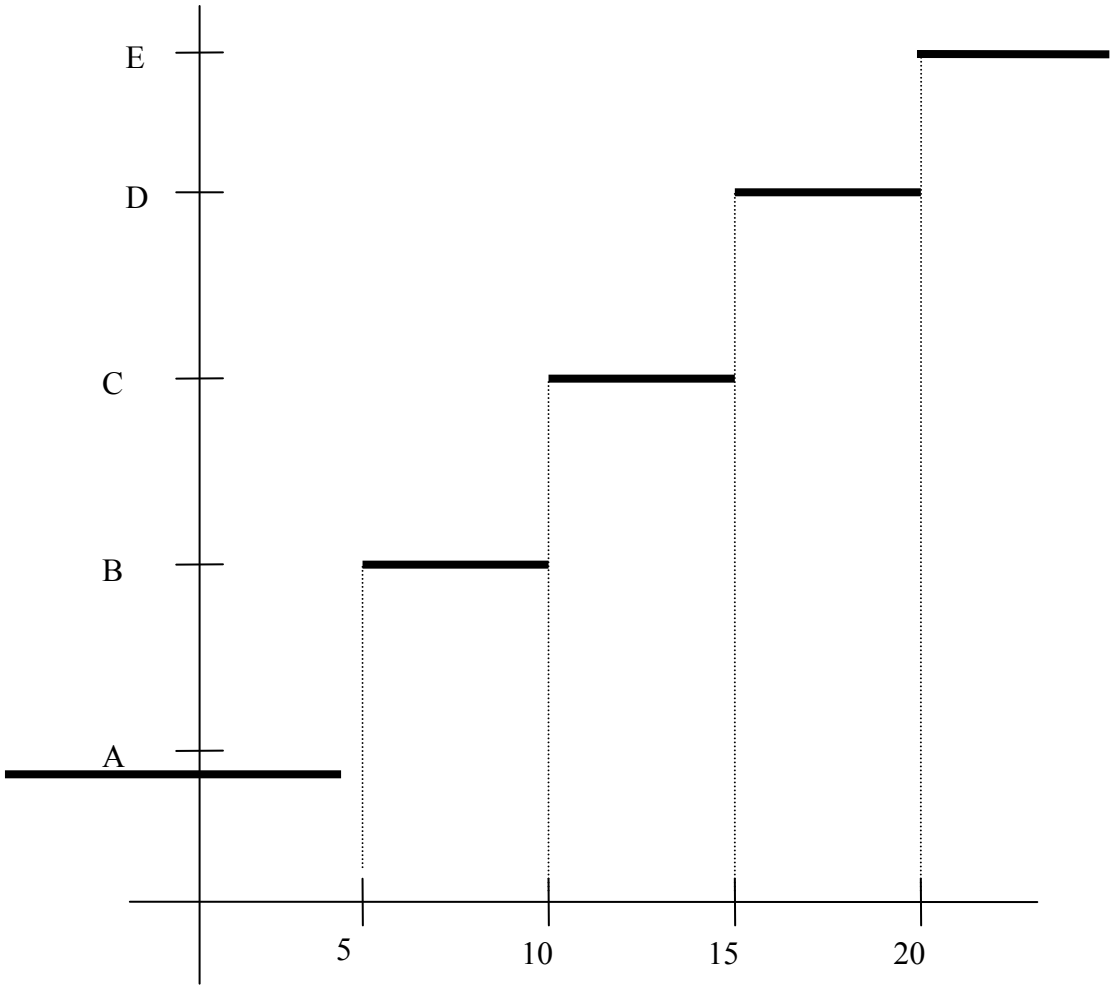
En cada uno de los casos siguientes escriba en el espacio que se le ofrece un programa que realice las acciones indicadas:

1. Imprima por pantalla, en orden decreciente, todos los múltiplos de 6 entre e1 1 y 1000.
(3 puntos).

2. Implemente un programa que repita por pantalla todas las líneas que vaya introduciendo el usuario, hasta que la línea introducida sea la palabra "salir".
(3 puntos).

3. Implementar un programa que imprima por pantalla todos los números mayores que 0 y menores que uno que el usuario introduzca por teclado. Este proceso deberá repetirse indefinidamente. Si el número no está en esos márgenes, el programa hará nada y pedirá otro número.
(4 puntos).

4. Programar una función de cuantificador digital con 5 niveles de cuantificación, de forma que el valor a cuantificar se introduce por teclado, y la salida de la función debe seguir la gráfica siguiente:
(4´5 puntos).



5. Imprimir por pantalla todas las posibles combinaciones de 10 números (del cero al nueve) tomados de tres en tres.
(10 puntos).

6. Imprimir por pantalla todos los número primos entre 1 y 100.
(14 puntos).

Cuestionario final

1. **¿Crees que tras el seminario y la utilización de NWTJava, tus conocimientos sobre programación han aumentado?**

2. **Entonces, ¿crees que ha sido positivo para ti el que lo hayas realizado? (SI, NO, ¿por qué?).**

3. **Piensa en ti antes del seminario, y ahora. ¿Crees que podrías resolver un mayor número de ejemplos, ejercicios y problemas de otras asignaturas?**
   ¿Lo has intentado? Si la respuesta es SI, ¿qué resultados has obtenido?

   En cuanto a la aplicación NWTJava,

4. **¿te ha servido para ver más claramente qué hace y cómo funciona cada una de las estructuras de programación (IF, WHILE, FOR,…)?**

5. **¿Te ha sido útil para organizar y aclarar la estructura que sigue un programa (declarar variables, ejecutar bucles, establecer condiciones,…)?**

6. **En líneas generales, ¿te ha resultado útil para el aprendizaje?**

   Ahora nos centramos en sus prestaciones,

7. **¿Qué te ha parecido su interfaz gráfica (sencilla, demasiados botones, no esta bien estructurada…)?**

8. **¿Y su manejabilidad?**

Bien, pues ahora,

9. **¿Puedes elaborar una pequeña lista de ventajas e inconvenientes de NWTJava?**

Y ya para terminar,

10. **¿Crees que la forma de trabajo que hemos seguido todos en el seminario favorece el aprendizaje?**

11. **¿Piensas que sería bueno para posteriores promociones el repetir este seminario?**

12. **Qué os ha parecido (puntos fuertes, puntos débiles, críticas, esperabais algo diferente, alguna decepción…)**

13. **¿Y en cuanto a los profesores? (Podéis ensañaros cuanto queráis)**

Apéndice D

Ejercicios del Módulo

Ejercicio 1

Escribir por pantalla la siguiente secuencia de palabras:

hola adiós ciao arrivederci hola adiós ciao arrivederci hola adiós ciao arrivederci hola adiós ciao arrivederci

Las palabras deberán aparecer en la pantalla una debajo de otra, en columna, y sólo podréis emplear en vuestro programa 4 instrucciones de imprimir por pantalla (ni una más ni una menos)

Pistas:

- Si he de imprimir 16 palabras y debo emplear sólo 4 instrucciones de imprimir (write), necesitaré algún tipo de bucle...
- Si sé de antemano cuántas pasadas ha de dar el bucle, ¿qué tipo de bucle necesito?
- ¿Dónde debo colocar las cuatro instrucciones write?
- ¡No olvidad colocar el elemento que señala el fin del bucle en el diagrama!!! ☺

Ejercicio 2

Crear un programa que vaya pidiendo al usuario que introduzca números por el teclado, uno tras otro, de forma que el programa pueda irlos imprimiendo por pantalla pero multiplicados por 10. El programa termina cuando el usuario introduce un 0.

Deberéis hacer el programa con un único bucle y, además, fuera del bucle no puede haber nada más que los elementos de inicio y de fin de programa.

Pistas:

- Fijaos que necesitáis un bucle, porque las mismas acciones (leer por teclado, multiplicar e imprimir el resultado) van a repetirse varias veces.
- El bucle no puede ser un for, puesto que no sabéis a priori cuántas veces habrá que repetir las mismas acciones (habrá que estar repitiendo hasta que el usuario decida parar)
- Si todo tiene que estar dentro del bucle, quiere decir que llegáis al inicio del mismo sin saber cuánto vale ninguna variable de nada, por lo tanto al inicio del bucle no estáis en condiciones de comprobar ninguna condición. Pero al final sí, por lo tanto... ¿qué bucle debéis aplicar? ☺

Ejercicio 3

Alguien en algún lugar ha hecho una tabla en la que relaciona temperaturas en grados centígrados con su equivalente “en palabras”. Por ejemplo, una temperatura de 5 grados es considerada como “fría”, mientras que una de 20 como “cálida”. La tabla completa es como sigue:

GRADOS	SENSACIÓN TÉRMICA
De –50 a –10	Muy fría
De –9 a 9	Fría
De 10 a 20	Templada
De 21 a 30	Cálida
De 31 a 50	Muy cálida

Se trata de que hagáis un programita que reciba por teclado una temperatura numérica y escriba por pantalla la sensación térmica correspondiente.

Ejercicio 4

Supongamos que tenemos una matriz de 5 filas por 10 columnas, y queremos recorrerla entera, fila a fila. Se trata de que construyáis un programita que lo haga, y que para cada elemento saque por pantalla su fila y su columna. Vamos, que la salida del programa debería ser algo como esto:

Elemento:
1
1
Elemento:
1
2
Elemento:
1
3
.
.
.

Y así sucesivamente hasta recorrer la matriz entera.

Pistas:

- Hay que recorrer todas las filas, y dentro de cada fila, todas las columnas. Esto suena a un par de bucles, muy probablemente uno dentro del otro.
- Como se sabe exactamente cuántas pasadas ha de hacer cada bucle (uno de ellos 5 pasadas y el otro 10), ¿qué tipo de bucle conviene utilizar?

Ejercicio 5

Vamos a hacer un pequeño programa que lleve la cuenta de cuántas cajas hay en un almacén, y haga salir por pantalla un mensaje cuando el almacén se llene (en el almacén caben 50 cajas) Con más detalle, lo que hará es irle pidiendo al operario encargado del almacén (usuario del programa) que introduzca por teclado el número de cajas que van llegando al almacén, vaya llevando la cuenta de cuantas cajas van ya, y cuando el número de cajas supere la capacidad del almacén deje de permitir al operario meter más cajas y avise de que el almacén está lleno. Tened en cuenta que al principio el almacén está vacío.

Pistas:

- Si el programa ha de estar pidiéndole al usuario continuamente que introduzca el número de cajas que van llegando, necesitaré que la instrucción de leer del teclado esté dentro de un bucle.
- Si el bucle ha de repetirse una y otra vez hasta que se cumpla una determinada condición, esto es, no sé a priori cuántas pasadas va a tener que dar el bucle, un FOR no me vale...
- ¿El mensaje de aviso debe imprimirse dentro o fuera del bucle?
- Antes de entrar en el bucle tendréis que darle un valor inicial a la variable que lleva la cuenta del número de cajas en el almacén, ¿no?
- Necesitaréis una variable para llevar la cuenta de cuantas cajas hay ya, y otra diferente para almacenar temporalmente las cajas que llegan cada vez (cantidad que se introduce por teclado)

Ejercicio 6

Hacer un programita que imprima por pantalla todos los números múltiplos de cuatro que haya desde 1 hasta un tope superior introducido por el usuario. Una vez que lo haya hecho, le pedirá al usuario un nuevo número para repetir la operación. Seguirá pidiendo números e imprimiendo todos los múltiplos de 4 desde 1 hasta ese número hasta que el usuario le meta un 0.

Pistas:

- Parece que vamos necesitar dos bucles: uno para pedirle al usuario una y otra vez que introduzca un nuevo tope superior, y otro dentro de éste para imprimir los múltiplos de 4 por pantalla.
- El bucle externo no puede ser un FOR, porque no sabemos a priori cuántas pasadas va a tener que hacer (depende de cuándo le de la gana al usuario de meter un 0) ¿Qué bucle nos puede servir entonces?
- El bucle interno sí que puede ser un FOR: sabemos que dará pasadas hasta alcanzar el tope que el usuario haya introducido.
- IMPORTANTE: ¿Cuál ha de ser el incremento de este bucle FOR para que sólo me tenga en cuenta los múltiplos de 4?

Otra cosa a tener en cuenta: la impresión de los números por pantalla se realiza dentro del bucle FOR, pero... ¿dónde debéis poner la cajita que se encarga de pedir al usuario el tope superior (una cajita READ)?

Ejercicio 7

Se trata de extraer los dos números mayores de una lista de números. Como NWTJava no admite arrays, lo que se hará es que el usuario vaya introduciendo números por teclado, hasta que introduzca un 0. En ese momento el programa sacará por pantalla los dos números mayores introducidos hasta el momento.

Pistas:

- Obviamente, necesitaremos un bucle que pida los números sucesivamente al usuario (ya deberíamos saber a estas alturas que un FOR no nos vale ;-)
- Se puede hacer con un WHILE pero... ¿Alguien sabría decir por qué en este caso un REPEAT UNTIL (equivalente al DO WHILE) puede ser más apropiado?
- Deberemos almacenar en dos variables los dos números mayores introducidos hasta el momento. Pregunta: ¿qué hacemos con los demás?
- Dentro del bucle habrá que comparar el último valor introducido con los dos almacenados, y actuar en consecuencia...
- Ojito: hay que quedarse con los DOS valores mayores. A poco que lo penséis, os daréis cuenta de que no basta con un par de IFs...

PRIMITIVA

Diseñar un programa para rellenar una columna de LA PRIMITIVA.
Reglas: rellenar 6 números que van desde el 1 hasta el 49 incluidos.
Si el número no pertenece a ese rango de valores, imprimir por pantalla "*número erróneo*" y seguir con la ejecución.

PROCESO METALÚRGICO

Realizar una aplicación que pida al técnico especializado introducir las temperaturas de diferentes muestras de hierro. El programa debe indicar si el hierro se encuentra en estado sólido, líquido o gaseoso.

Datos:

pto. de fusión: 1535 ºC (a temperatura menor o igual ➔ sólido)
pto. de ebullición: 2750 ºC (a temperatura mayor ➔ gaseoso)

La aplicación concluye cuando la temperatura introducida es de -273ºC o inferior.

EMBALSE

Un embalse que suministra AGUA a las poblaciones cercanas, tiene una capacidad inicial de 1000 hm cúbicos.

En el día de hoy, la capacidad ocupada es de 500 hm3.

Cada día se realiza una medición. Los días con precipitaciones se le suma la cantidad de agua recogida (ej: 23 hm3) y el resto se le resta la consumida (ej: -12 hm3).

Si en algún momento se supera el 75% de la capacidad, finaliza el programa advirtiendo de que se deben abrir las compuertas de la presa. Si por el contrario se baja del 25%, también se finaliza pero esta vez avisa de que hay que aplicar restricciones al consumo de agua.

DESFIBRILADOR

Diseñamos el software de un desfibrilador para tratar un infarto de miocardio. Lo 1º que debe hacer el desfibrilador es medir el nº de pulsaciones por minuto (nosotros haremos el papel de corazón, en el sentido de que, el desfibrilador pregunta y nosotros respondemos con dicho número).

Si las pulsaciones superan las 180, aplica 3 descargas (la forma en que aplica 1 de ellas es sacando por pantalla la palabra DESCARGA). Tras este proceso, vuelve a repetirlo tantas veces como sea necesario (hasta que el ritmo cardiaco se normalice, es decir, baje de 180).

Pliego II
Planos

Pliego III
Manual de Usuario

NWTJava
Manual de Usuario

Contenidos

1. SOBRE MANUAL DE USUARIO NWTJAVA V2.0 195

2. EMPEZANDO 195

3. NIVEL ESTRUCTURADO: CONCEPTOS Y MANEJO 199

3.1 Tipos de variable 199

3.2 Manejo de arrays 204

3.3 Subrutinas 211

3.4 Ámbitos de variable 216

1. Sobre *Manual de Usuario NWTJava v2.0*

Este *Manual de Usuario NWTJava v2.0* pretende ser continuación del elaborado con motivo de la primera versión. Con él, usted como usuario podrá instruirse sobre las nuevas funcionalidades añadidas a esta segunda versión de NWTJava. Por ello le advertimos de que si ésta es su primera toma de contacto con la aplicación, acuda en primer lugar al manual desarrollado en el anterior proyecto.

El objetivo o finalidad con el que elaboramos este manual es el de abordar las nuevas características surgidas en esta nueva versión, así como las modificaciones surgidas entre ambas versiones con motivo de la evolución de la misma.

Por ello, le recomendamos encarecidamente que para su inicialización en el manejo de la herramienta, se apoye primero en el *Manual de Usuario de NWTJava*, y una vez conocidos los principios básicos sobre los que se sustenta esta aplicación, acuda a este segundo para incorporar a su conocimiento las mejoras de la misma.

Para ayudarle en el entendimiento de los nuevos elementos encontrados en esta segunda versión, frecuentemente introduciremos breves comentarios sobre el concepto teórico al que hace referencia. Para distinguir con mayor facilidad la explicación de estos conceptos, utilizaremos el siguiente símbolo:

También utilizaremos un segundo símbolo asociado a comentarios varios, cuyo fin será el de aclararle aspectos de diversa índole.

* Estos símbolos han sido extraídos del anterior manual con el objetivo de aportar continuidad a los contenidos.

2. Empezando

Nada más iniciar la aplicación observamos ya el primer cambio respecto a la anterior versión. Consiste en la aparición de una ventana que le da a elegir, de entre tres posibilidades, el *Nivel de Aprendizaje* con el que quiere seguir avanzando en sus conocimientos.

En la figura siguiente se muestra la vista principal de la aplicación con la recién comentada ventana de elección de nivel. Es lo primero que vemos nada más iniciar la aplicación:

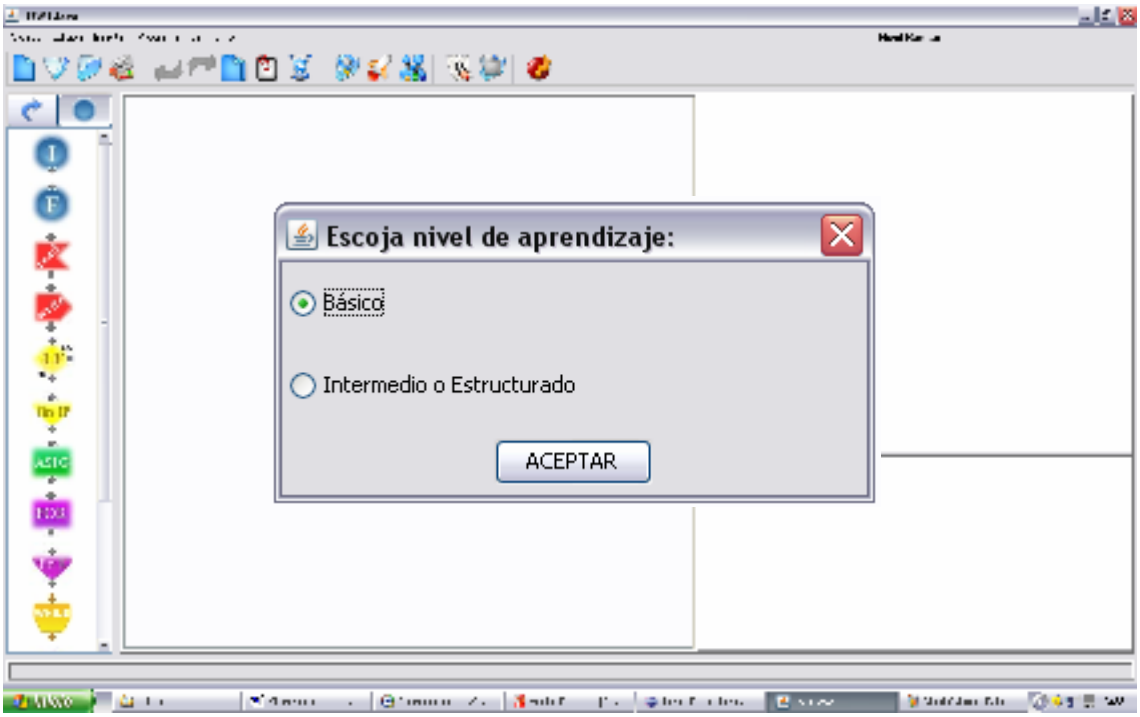


Fig. 1

Niveles de Aprendizaje

Acudiendo a la clasificación educativa de la programación, observamos la siguiente sucesión de niveles de aprendizaje:

Programación mediante procedimientos → ***Programación estructurada*** → ***Programación orientada a objetos***

Por ello, el objetivo docente de NWTJava v2.0 se encuentra dividido y estructurado de la siguiente forma:

- *Básico.*
- *Intermedio o Estructurado.*

(El alcanzar un tercer nivel será estudiado en la siguiente versión).

Usted como alumno, deberá recorrerlos progresivamente para adquirir todos aquellos conocimientos para los que NWTJava v2.0 ha sido diseñado.

En todo momento usted será capaz de ver el nivel seleccionado mediante la etiqueta situada en la esquina superior derecha de la ventana principal.

Cada uno de estos niveles está diseñado para que usted conozca y maneje conceptos y estructuras propias de cada nivel de aprendizaje.

Así tenemos, entre otros ejemplos:

- Básico → bucles y estructuras condicionales.
- Intermedio o Estructurado → subrutinas y ámbitos de variable.

Otro elemento novedoso, es la inclusión de un nuevo panel. Situado en la esquina inferior derecha de la interfaz, recibe el nombre de *Área de Detalles* y su función presenta distintas vertientes.

La primera es llevar la contabilidad sobre el número de estructuras utilizadas en el diagrama. Muy útil para cuando la extensión del mismo no permite visualizarlo de un solo vistazo. Digamos que funciona a modo de un contador de líneas de código como pueda haber en cualquier editor. La segunda y más importante es, que a través de los datos mostrados, usted busque una cierta competitividad consigo mismo, e intente abordar las soluciones desde la perspectiva de la optimización de código. Es decir, una vez analizado el funcionamiento del diagrama, la estructura del mismo y los datos que arroja este panel, busque alcanzar los mismos resultados pero rebajando el número de instrucciones empleadas, lo que en programación denominamos: optimización.

Optimización

La optimización de software es una rama de la Ingeniería de Software que trata de convertir programas existentes en otros programas que realicen las mismas tareas en menos tiempo, con menos requerimientos de memoria, o en general empleando los recursos de forma óptima.

La información mostrada por dicho panel presenta la siguiente estructura: en un diagrama con una pieza *INICIO*, dos *ASSIGN*, una *WRITE*, una *FOR* con su correspondiente *END_FOR*, y una última *FINAL*, la información mostrada por el *Área de Detalles* es la siguiente:

```
N° de instrucciones FOR    : 1
N° de instrucciones ASIG   : 2
N° de instrucciones WRITE  : 1
```

Fig. 2

Muestra el número de ocurrencias de estas estructuras que son las que aportan información de interés. Usted, como alumno, puede estar preguntándose porqué el número de apariciones del resto de iconos no resulta relevante. La explicación es la siguiente: no aportan ninguna información. Por ejemplo *INICIO* y *FINAL*, sabemos que éstos sólo van a aparecer una vez en un diagrama que sea válido. Otras como puedan ser *END_FOR* o *UNTIL* serían redundantes, pues tienen el mismo número de apariciones que el icono que abre el correspondiente bucle, es decir, las ocurrencias del icono *FOR* son las mismas que la del *END_FOR*.

Antes de pasar a los apartados donde ya profundizamos en las nuevas características de la versión, comentar también que las piezas o iconos han sufrido una transformación en cuanto a diseño, pero son totalmente equivalentes a sus predecesoras. A continuación, exponemos las modificaciones con el fin de mostrar sus paridades y evitar posibles confusiones:

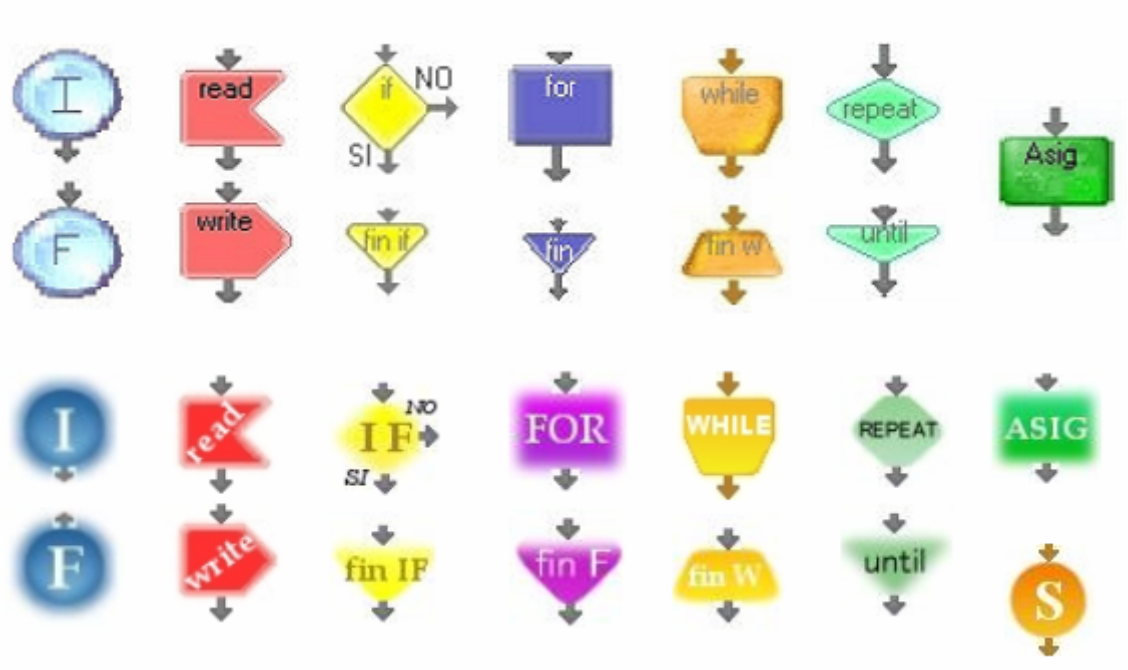


Fig. 3

A continuación, los apartados en los que se describen los nuevos conceptos, y manejo de los mismos, de NWTJava v2.0.

3. Nivel Estructurado: conceptos y manejo

Son cuatro grandes conceptos los que se pretenden incorporar en el recién creado *Nivel Intermedio o Estructurado*.

Corresponden a:

- Tipos de variables.
- Manejo de arrays.
- Subrutinas.
- Ámbito de variables.

Comenzamos a verlos.

3.1 Tipos de variables

Comenzamos pues con el primero de los citados conceptos. Para una mayor comprensión, vemos su definición desde un punto de vista general.

Tipos de variables

Un tipo de variable es una restricción impuesta a la misma variable para la interpretación, manipulación y representación del dato o valor que almacena. Los Tipos de variables varían de unos lenguajes de programación a otros, pero los más comunes son los tipos simples (enteros, caracteres, lógicos, etc.), los compuestos (cadena de caracteres, estructuras de datos, etc.), los de clases, etc.

Son cuatro los tipos de variable que se pueden manejar en el *Nivel Intermedio o Estructurado* de NWTJava v2.0.

- **Entero**
  - Variables de este tipo se utilizan para almacenar datos numéricos enteros comprendidos entre -2147483468 y +2147483467.
  - Ejemplo: valor = 1000.
  - Su valor por defecto es *0* (cero).

- **Real**

  - Variables de este tipo pueden almacenar números en coma flotante y con signo, es decir, permiten representar números decimales. Su precisión aproximada es de 16 dígitos.
  - Ejemplo: valor = -0.0001.
  - Su valor por defecto es *0.0* (cero punto cero).

- **Cadena de caracteres**

  - Se trata de un tipo capaz de almacenar una sucesión de caracteres alfa-numéricos, siempre y cuando su valor se delimite entre "" (comillas).
  - Ejemplo: valor = "HolaMundo"; valor = "Calle Mayor Nº-72".
  - Su valor por defecto es "*Cadena por defecto*".

- **Booleanos**

  - Únicamente tiene el siguiente rango de valores: *true* (verdadero) y *false* (falso). Su utilización es muy frecuente para determinar el flujo de los programas mediante instrucciones condicionales, como es el caso de la pieza *IF*.
  - Ejemplo: valor = true.
  - Su valor por defecto es *false*.

Es obligatorio declarar el tipo cuando se crea una variable. Además, el valor asignado a ésta debe concordar con las características y limitaciones inherentes a dicho tipo. Por ejemplo, si creamos una variable de tipo *Booleano*, no podemos asignarle un valor de 8, o si en la variable almacenamos el valor "HolaMundo", ésta no puede ser de tipo *Entero*. Estas situaciones de error son controladas y detectadas durante el proceso de validación, el cual, y al final del mismo, informará de esta ocurrencia con el siguiente mensaje de error:

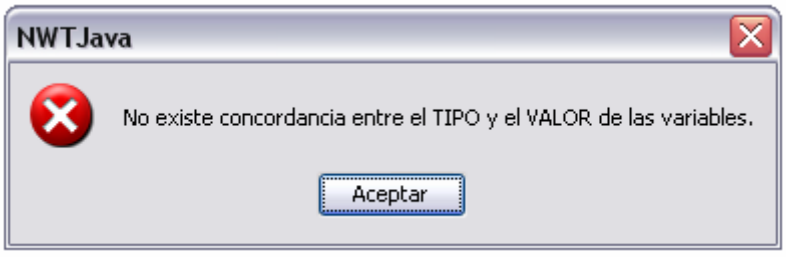


Fig. 4

Como hemos comentado, el tipo se lo asociamos a una variable en su creación. Acompaña al identificador (nombre de la variable) y al valor en la definición de ésta.

Pues bien, no en todas las piezas o iconos se puede crear variables. De hecho, las únicas tres en lo que ésto es posible son: *ASSIGN*, *READ* y *FOR*.

En la primera de ellas, como su nombre hace referir, realiza una asignación. Es la pieza por excelencia para crear variables. La instrucción *READ* tiene una función prácticamente análoga a la anterior. Se diferencia en que el valor de la variable creada lo aporta el usuario en tiempo de ejecución. Y la estructura de bucle *FOR* crea la variable que jugará el rol de contador de iteraciones.

Todas ellas presentan por tanto, en su *Tabla de Edición*, el nuevo campo *TIPO*.

Excepciones

El campo TIPO también lo incluyen en su Tabla de Edición otras piezas como son SUBR y FINAL, pero debido a que su función no es la de crear variables, explicaremos la función que desempeña este nuevo atributo para estos iconos en su debido tiempo.

Describimos pues como es el proceso de asociación de variables con sus correspondientes tipos.

Empezaremos por crear un diagrama. Este diagrama manejará obligatoriamente variables, pues son el elemento base sobre el que operan las instrucciones declaradas en dicho programa.

Para el correcto diseño del diagrama creado, debemos conferir a las instrucciones los datos requeridos por las mismas a través de sus atributos. Para definir estos valores, pulse el botón derecho del ratón sobre la pieza elegida (cualquiera de las tres mencionadas con anterioridad) y elija el comando *Editar*, o bien vaya al menú editar de la barra de menús, y pulse *Editar*. Tomando como modelo el icono *ASSIGN*, aparecerá la siguiente ventana de edición:

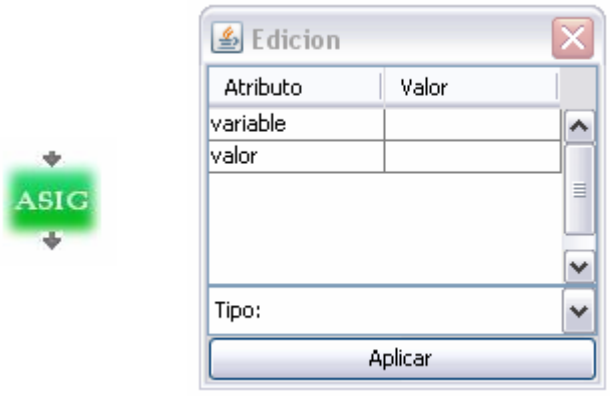


Fig. 5

Es entonces cuando tomamos contacto por primera vez con el atributo *TIPO*. La sucesión de pasos más lógica a continuación es la de, en primer lugar rellenar el campo variable con el nombre que se deseé asociar a la variable. Introducir su valor, pulsar la tecla intro para asociar dichos datos a la pieza en cuestión, y pinchar con el ratón sobre la flecha del campo *TIPO*.

E aquí la peculiaridad de este parámetro. No sigue el mecanismo de los anteriores (escribir mediante teclado la entrada correspondiente). Al pincha en la citada flecha lo que ocurre es que se despliega una lista en la que puede encontrar todos y cada uno de los tipos existentes. Lo vemos:

Fig. 6

Una vez desplegado este abanico de opciones, usted podrá seleccionar el tipo que desea asociar a la variable que está creando y definiendo.

IMPORTANTE: Recuerde que dependiendo del valor de la variable, el tipo seleccionado puede no ser válido.

El proceso de selección de tipo finaliza con la acción de pulsar el botón *APLICAR* de la correspondiente *Tabla de Edición*. Una vez realizada esta sucesión de pasos, la variable ha sido totalmente creada y definida, estando de esta forma totalmente dispuesta a ser utilizada. Resta por tanto el repetir este proceso con todas y cada una de las variables que desea utilizar en el diagrama.

En cuanto a los errores que usted, como alumno y usuario puede cometer, indicar que se clasifican en cuatro actuaciones. Es función del proceso de validación el controlar, detectar y comunicar las dos primeras, mientras que de las restantes se encarga el proceso de ejecución.

Dicha clasificación es la siguiente:

- **Validación**

  - No selecciona tipo alguno para la variable.

    Mensaje de error asociado:

    "El TIPO de las variables debe ser Entero , Real , Cadena o Booleano."

  - El valor y el tipo de la variable no son compatibles.

    Mensaje de error asociado:

    "No existe concordancia entre el TIPO y el VALOR de las variables."

- **Ejecución**

  - El valor que aporta a una variable en tiempo de ejecución, mediante una instrucción READ, no es compatible con el tipo de la misma.

    Mensaje de error asociado:

    "El valor introducido en el READ no concuerda con el tipo de la variable.

    - a NIVEL BASICO, debe de ser tipo ENTERO.

    - a NIVEL INTERMEDIO o AVANZADO, de tipo ENTERO, REAL, CADENA o BOOLEANOO dependiendo de lo introducido en el menú TIPO."

  - Error al intentar operar con dos o más variables de tipos incompatibles.

    Ejemplo: sumar una variable *entera* con otra *booleana*.

    Mensaje de error asociado:

    "Error debido a uno de los siguientes motivos:

    - ...
    - realizar una OPERACIÓN NO PERMITIDA para el TIPO de datos utilizado,
    - ..."

Por último, explicar que en el proceso de traducción, la nomenclatura Java asociada a la creación y definición de variables es la siguiente:

Tipo identificador;
identificador = valor;

Donde la equivalencia entre tipos NWTJava v2.0 y tipos de variable Java es:

NWTJava v2.0	Java
Entero	**int**
Real	**double**
Cadena de caracteres	**String**
Booleano	**boolean**

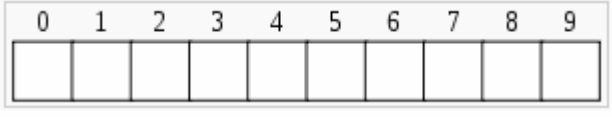
3.2 Manejo de arrays

Hasta el momento, usted sólo ha tenido que trabajar con algunas variables en cada uno de los diagramas construidos. Sin embargo, en más de una ocasión deberá manipular conjuntos más grandes de valores. Por ejemplo, para calcular la estatura media de un grupo de 20 personas, necesitará conocer esos 20 valores. Podría utilizar 20 variables diferentes, pero ¿qué pasaría si fueran 100 o más los valores que tuviera que registrar? Además de resultar una tarea muy engorrosa, el tamaño del código se vería enormemente incrementado. ¿Y si pudiese almacenar todos esos valores en una única variable?.

En este apartado, usted aprenderá a almacenar conjuntos de valores en una estructura de datos denominada *array*.

Arrays

Referido a la programación, un array es un conjunto o agrupación de variables del mismo tipo cuyo acceso se realiza por índices. Su tamaño o longitud se establece en su creación.



En nuestro caso, además le informamos de que serán estáticos (una vez creados, no será posible modificar su tamaño) y unidimensionales (1xLongitud del array).

Como puede extraer de la definición de array, las variables almacenadas deben ser homogéneas, es decir, del mismo tipo, y por tanto, los arrays con que trabajará en NWTJava v2.0 serán:

- *Enteros*: almacena variables o valores de tipo *Entero*.
- *Reales*: almacena variables o valores de tipo *Real*.
- *Cadenas de caracteres*: almacena variables o valores de tipo *Cadena de caracteres*.
- *Booleanos*: almacena variables o valores de tipo *Booleano*.

Para crear y manejar un array, son tres las operaciones que se pueden realizar:

- Crearlos (definirlos)
- Rellenarlos (aportarle los valores)
- Manejas sus posiciones
  - Leerlas (extraer contenido)
  - Modificarlas (cambiar contenido)

La forma de crear un array es prácticamente análoga a la de crear cualquier variable común: debe asignarle un *nombre* o identificador, declararle un *tipo*. La distinción entre ambas clases de variable se encuentra en su argumento *valor*. Mientras que las variables, llamémoslas "simples", almacenan un único valor o dato, los arrays lo hacen con una sucesión de ellos. Por este motivo, la forma en que debe aportarle ese conjunto de valores, rellenarlos, requiere una nomenclatura específica.

La creación de un array, junto con el almacenado de sus valores, puede hacerse de las siguientes formas:

1. Inicializando el array en su propia declaración (definición extensiva).
2. Declarándolo sin inicializar, únicamente le aporta el tamaño (definición intensiva).

Si desea iniciar un array con otros valores diferentes a los predeterminados, puede hacerlo mediante la primera de las definiciones: *definición intensiva*.

Responde a la siguiente filosofía (lo vemos con un ejemplo):

{ 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 }

La nomenclatura seguida para formar esta estructura es la siguiente: entre llaves "{}" se introducen tantos elementos como valores desee almacenar. El elemento separados que debe colocar entre cada uno de los valores es la coma ",".

Si por el contrario, su objetivo es el de reservar una serie de posiciones, vacías (sin almacenar valores por el momento), debe recurrir a la *definición extensiva*. La función de este tipo de definición es precisamente esa, crear un array de un determinado tamaño, pero vacío, sin todavía almacenar datos en sus posiciones.

Analizamos su nomenclatura con un ejemplo en el que creamos un array vacío de tamaño 3:

{ } 3

Tan simple y sencillo como puede observar. Recuerde que no debe haber elemento alguno antes de la primera llave "{", y tampoco entre ellas, de esta forma se indica que lo que quiere es construir un array vacío. Tras la última llave "}" debe situar el tamaño deseado. Para cumplir con la reglas gramaticales de esta nomenclatura, la longitud debe ser indicada mediante un número entero y positivo (no es válida la opción de crear un array de tamaño cero).

Cualquier otro tipo de escritura que no siga las normas establecidas, el proceso de validación resultará incorrecto, informando de esta situación mediante el siguiente mensaje de error:

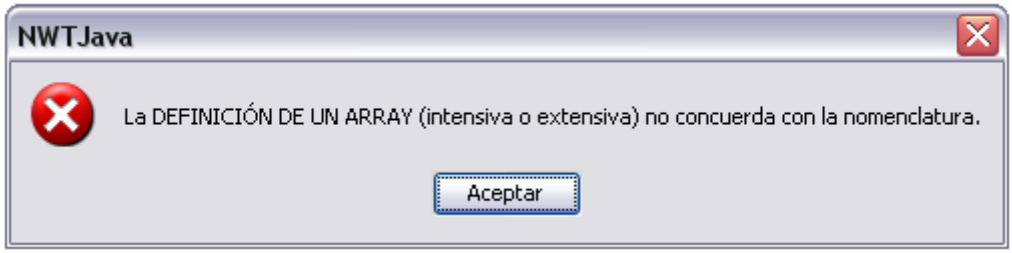


Fig. 7

Debemos recalcar que en NWTJava v2.0, en el momento en que se cree una variable cuyo valor digamos que es "vacío", se le asignará automáticamente un valor por defecto. Este valor por defecto varía en función del tipo, es decir, no es el mismo valor por defecto para las variables de tipo *Entero* que para las de tipo *Cadena de caracteres*. Ésto lo podrá entender usted rápidamente en cuanto vea la siguiente sucesión de ejemplos.

Vamos a crear a continuación un array mediante definición extensiva, de tamaño 2, para todos y cada uno de los tipos de variable. Estos arrays irán acompañados del que sería su representación por pantalla:

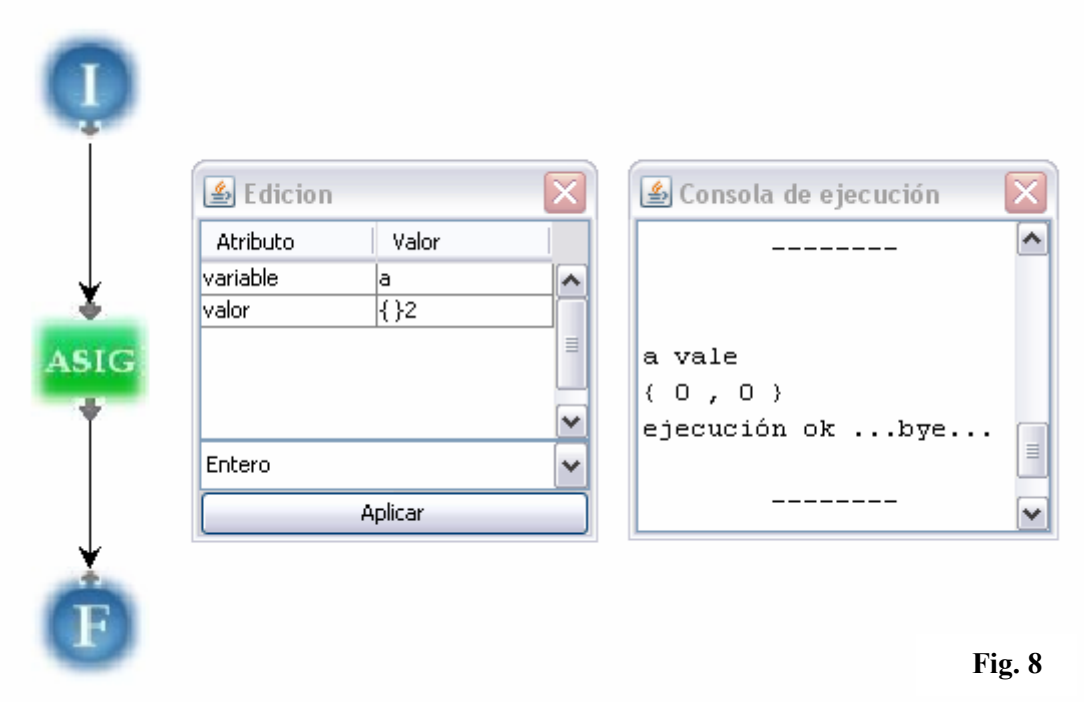


Fig. 8

El array de tamaño 2, creado mediante definición intensiva en la *Tabla de Edición* del *ASSIGN* está vacío, es decir, no hemos almacenado valor alguno en él. Sin embargo, y como acabamos de comentar, NWTJava v2.0 asigna automáticamente a estas posiciones valores por defecto, en este caso, de tipo *Entero*. Al representar su contenido por pantalla, vemos que en cada posición almacena el valor 0 (cero), que es el de por defecto para el tipo *Entero*.

Para los restantes es:

- **Tipo Real** ⟶ **{ 0.0 , 0.0 }**
- **Tipo Cadena** ⟶ **{ "Cadena por defecto" , "Cadena por defecto" }**
- **Tipo Booleano** ⟶ **{ false , false }**

Los arrays vacíos que usted cree podrá rellenarlos mediante lo que conocemos como manejo de las posiciones de array. Es una de las muchas acciones que puede realizar mediante la manipulación de estas posiciones.

Las dos operaciones básicas que puede hacer con ellas son:

1. Leerlas (extraer contenido)
2. Modificarlas (cambiar contenido)

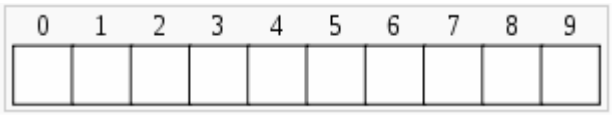
Para hacer referencia a una de estas posiciones se utiliza el nombre del array seguido de un subíndice entre corchetes "[ ]". Este subíndice puede ser tanto un valor numérico como el nombre de una variable que indique su valor, siempre y cuando no sobresalga de los límites del array indicado.

Límites de un Array

Los límites de un array son tanto inferior como superior. La primera posición está referenciada por el subíndice 0 (cero), mientras que la última posición viene indicada por subíndice resultante de la siguiente operación:

tamaño del array - 1

Si se toma como referencia la figura del array, se puede observar que es de tamaño 10, que su primera posición es la número 0 (cero) y la última la 9 (nueve).



Por éste motivo, la posición indicada entre corchetes, nunca debe exceder de los límites del array referenciado puesto que en caso contrario, se devolverá mensaje de error.

A continuación le mostramos una serie de ejemplos en los que intervienen posiciones de array, sus elementos almacenados y las operaciones de *leer* y *modificar* ya mencionadas:

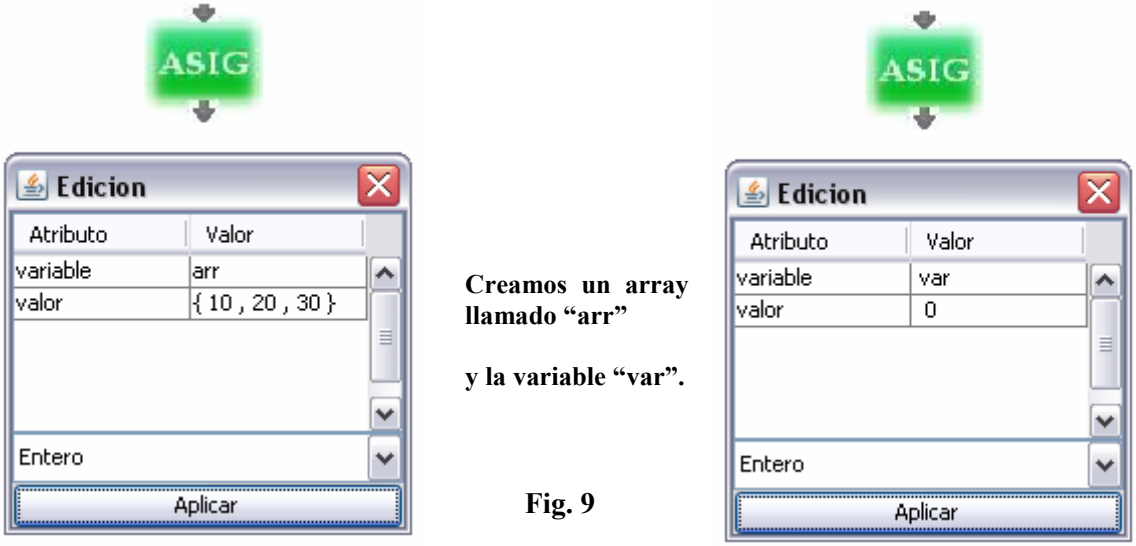


Fig. 9

Pues bien, en la *Tabla de Edición* situada a la derecha, es usted testigo de una operación de ***modificar*** el valor de una posición del array "*arr*". En este caso de la posición [1], por lo que si representásemos el array completo por pantalla, se mostraría el siguiente resultado:

{ 10 , 555 , 30 }

Fig. 10

Fig. 11

Por el contrario, y partiendo de nuevo con los resultados iniciales, con esta nueva *Tabla de Edición* de la izquierda, puede observar una operación de ***leer*** el contenido de una posición de array. Consiste en extraer el valor almacenado en dicha posición. En este caso guardamos este elemento en la variable que ya habíamos creado anteriormente, "*var*". De esta forma, el contenido del array queda inalterado pero el valor de la variable ha sido modificado. Ahora valen:

var: 20
***arr*: { 10 , 20 , 30 }**

Para terminar, aprecie el caso de que para acceder a la posición [0] del array "*arr*" (una vez más partiendo de los valores inicialmente mostrados), también puede utilizar la operación:

arr[var]

Los fallos relacionados con el empleo y manejo de arrays se dividen también entre los detectados en la validación y en la ejecución (al igual que ocurre también con Java por poner un ejemplo significativo). Éstos son los más comunes:

- **Validación**

  - Errores relacionados con la nomenclatura de las definiciones.

    Mensaje de error asociado:

    "La DEFINICIÓN DE UN ARRAY (intensiva o extensiva) no concuerda con la nomenclatura."

  - Errores relacionados con el apartado anterior (Tipos de variables)

    Se dan cuando alguno de los elementos almacenados en un array no corresponde con el tipo asociado a dicho array.

    Mensaje de error asociado:

    "No existe concordancia entre el TIPO y el VALOR de las variables."

- **Ejecución**

  Abarca una variedad de errores entre los que se encuentran:

  - Excederse de los límites del array.

    Mensaje de error asociado:

    "Fuera de los límites de un array."

  - Hacer un uso fraudulento con las posiciones de array

    Ejemplos de este punto son cuando se desean realizar operaciones con posiciones de array cuyo tipo son incompatibles; utilizar variables entre los corchetes cuyo valor no cumple con los requisitos de referenciar a una posición correcta (ejemplo: *arr[var]* cuando el valor de *var* es *-0.11* o *true*); etcétera.

    "Se desea acceder a una posición de un array que no existe."

    "Se desea acceder a una:

    - ...
    - posición de array incorrecta,
    - ..."

    Éstos y otros mensajes de error podrán encontrarse al final de este manual.

3.3 Subrutinas

Entramos ahora en este importante apartado donde NWTJava v2.0 introduce un nuevo y fundamental concepto de la programación estructurada. Pero antes, aportamos las siguientes definiciones:

Procedimientos y funciones

En la programación estructurada se distinguen, valga la redundancia, dos estructuras: procedimientos y funciones.

Un procedimiento es un conjunto de instrucciones que realizan unas operaciones dadas y que no devuelven un resultado

Una función es un lo mismo pero devolviendo un resultado (y por lo general, recibe una lista de parámetros de entrada).

Entonces, ¿puede usted llamar a una subrutina (sea *procedimiento* o *función*) desde otro lugar del código.? Por supuesto, de hecho es el paradigma fundamental de la programación estructurada. Sin embargo, NWTJava v2.0 utiliza una particular forma de tratar este asunto.

Esta aplicación utiliza los términos *Programa Principal* o *Rutina*, y *Subrutina*. Lo que hasta ahora podía codificar en el *Área de Trabajo* de la interfaz de NWTJava era sencillamente un *procedimiento*, una rutina de operaciones enlazadas de forma lineal. Sin embargo ahora tiene la posibilidad de invocar bloques de código independientes desde dicha rutina principal → *Subrutinas*.

Existen dos modos de afrontar la construcción de diagramas que emplearan subrutinas. La primera crear el que actuará como programa principal (PP) y a continuación las subrutinas (SR) que le sean necesarias, o crear en primer lugar una serie de SRs y después ir utilizándolas a medida que un PP las requiera.

Para que usted, usuario, entienda de forma más lógica e intuitiva la concepción y manejo de SRs, en este manual optamos por la segunda vía: crear una SR y a continuación el PP que hará uso de ella.

Lo primero que debe entender es que una SR es un diagrama más, como los que ha venido construyendo hasta ahora. Con dos particularidades: puede o no requerir parámetros para su ejecución (un PP nunca debe requerir parámetros) y puede o no devolver un resultado (en este caso, un PP también podría o no devolver un resultado).

¿Qué son y como va a recibir los parámetros? Los parámetros son variables que requiere la SR para poder ejecutarse. Necesita que se los pase el PP que la está invocando y además en el mismo orden, cantidad y tipo que establece la SR.

¿Cómo conseguimos ésto en NWTJava? Mediante la pieza INICIO del diagrama que actuará como SR.

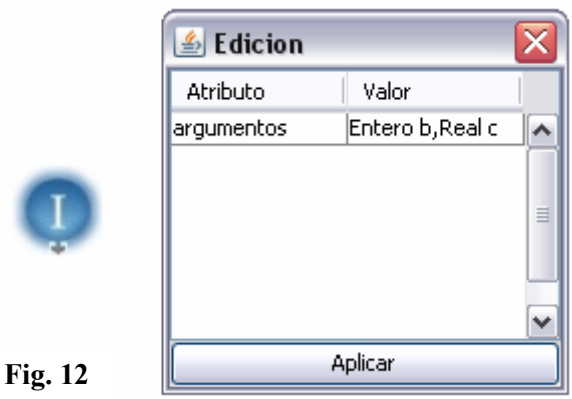


Fig. 12

Como puede observar, en la *Tabla de Edición* de *INICIO* se encuentra el atributo "*argumentos*". Estos argumentos son los parámetros que requiere la SR para poder ser ejecutada. Sin estas variables, la SR no dispone de los datos necesarios para su ejecución.

En este caso, puede ver como demanda dos variables, una de tipo *Entero* y otra *Real*, y además, en ese mismo orden.

Esta es la forma en que iniciamos la construcción de una SR. El siguiente paso se limita a codificar aquella labor que usted desee que realice la SR (una suma entre estas variables, que dibuje un cuadrado cuya superficie sea la primera variable por la segunda, etc.).

Y finalmente, tratar el resultado que quiera que devuelva. Este puede no ser ninguno, ejemplo: en el recién mencionado caso de la suma entre las variables pasadas como parámetros, sería lógico que la SR retornara el resultado de la operación, mientras que en el de dibujar el cuadrado, la SR se limita a dibujarlo por pantalla, pero no tiene porqué devolver ningún valor.

De éste aspecto se encarga la pieza *FINAL*. Analice su *Tabla de Edición*:

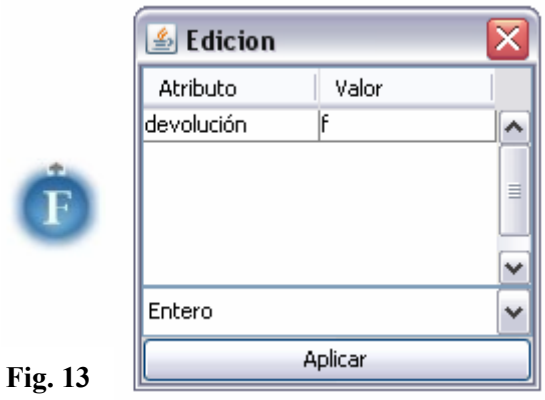


Fig. 13

Como puede observar, en este caso el atributo es "devolución". Hace mención a la variable que almacena el valor que desea devolver. También debe seleccionar el tipo. En esta ocasión, éste es un mecanismo de seguridad, para que a la hora de marcar el tipo en la pieza *SUBR* (de la que hablaremos a continuación), tenga una fácil referencia de qué tipo es la variable que retorna la SR.

Pasamos pues al desarrollo del PP. Debe tener en cuenta, al empezar, dos aspectos fundamentales.

- No debe requerir ningún parámetro de entrada. Esto es debido a que al actuar como PP, ningún diagrama podrá proporcionárselos, la ejecución comienza desde su pieza *INICIO*.

- Antes de validar el PP debe asegurarse de que las SRs a las que invoca han sido correctamente codificadas. Para ello debe, antes de comprobar la validación del PP, hacerlo con todas y cada una de sus SRs asociadas y, guardarlas en memoria. De esta forma, se asegura NWTJava v2.0 de no iniciar el proceso de ejecución del PP hasta no resultar válidos todos y cada uno de los participantes en ella.

Por lo tanto, y recapitulando, el campo "*argumentos*" de *INICIO* debe estar en esta ocasión, siempre vacío. A continuación, construya el diagrama normalmente.

Y es en medio de este proceso cuando, si lo considera necesario, invocará una SB (pueden ser uno o tantas como disponga, pero el tratamiento totalmente análogo). Para este efecto es para lo que ha sido creada la nueva pieza *SUBR*.

Fig. 14

Analicemos la *Tabla de Edición* y cada uno de los atributos rehuidos para propiciar su aprendizaje y manejo:

Fig. 15

Nombre de Subrutina*:* nada más hacer clic sobre la flecha de este campo, podrá ver como se despliega una lista compuesta por todos y cada uno de los diagramas abiertos en la aplicación. Usted debe seleccionar aquel que desea invocar como SR.

Argumentos: es aquí donde debe introducir las variables con las que quiere que trabaje la SB. Recuerde, deben coincidir en número, tipo y orden con lo requerido por la SR (ver el atributo "*argumentos*" de su pieza *INICIO*).

Devolución: la variable que usted introduzca en este campo será la encargada de almacenar el valor devuelto por la SR. Si dicha SR no devuelve ningún valor, no debe rellenar este atributo, al igual que el de "*tipo"*.

Tipo: es el de la variable "*devolución*". En muchas ocasiones será este el momento en que cree dicha variable, por ello es necesario declarar su tipo. Recuerde que el seleccionado debe coincidir con el tipo marcado en la pieza FINAL de la SR, de lo contrario caerá en una contradicción y le será informado debidamente.

Finalmente, tras este icono seguirán los que usted haya creído oportuno hasta llegar a la instrucción FINAL del PP.

A continuación, y para mayor claridad y comprensión, le mostramos un ejemplo gráfico de lo que sería la codificación de un PP, una SB y la relación que mantienen entre ellos:

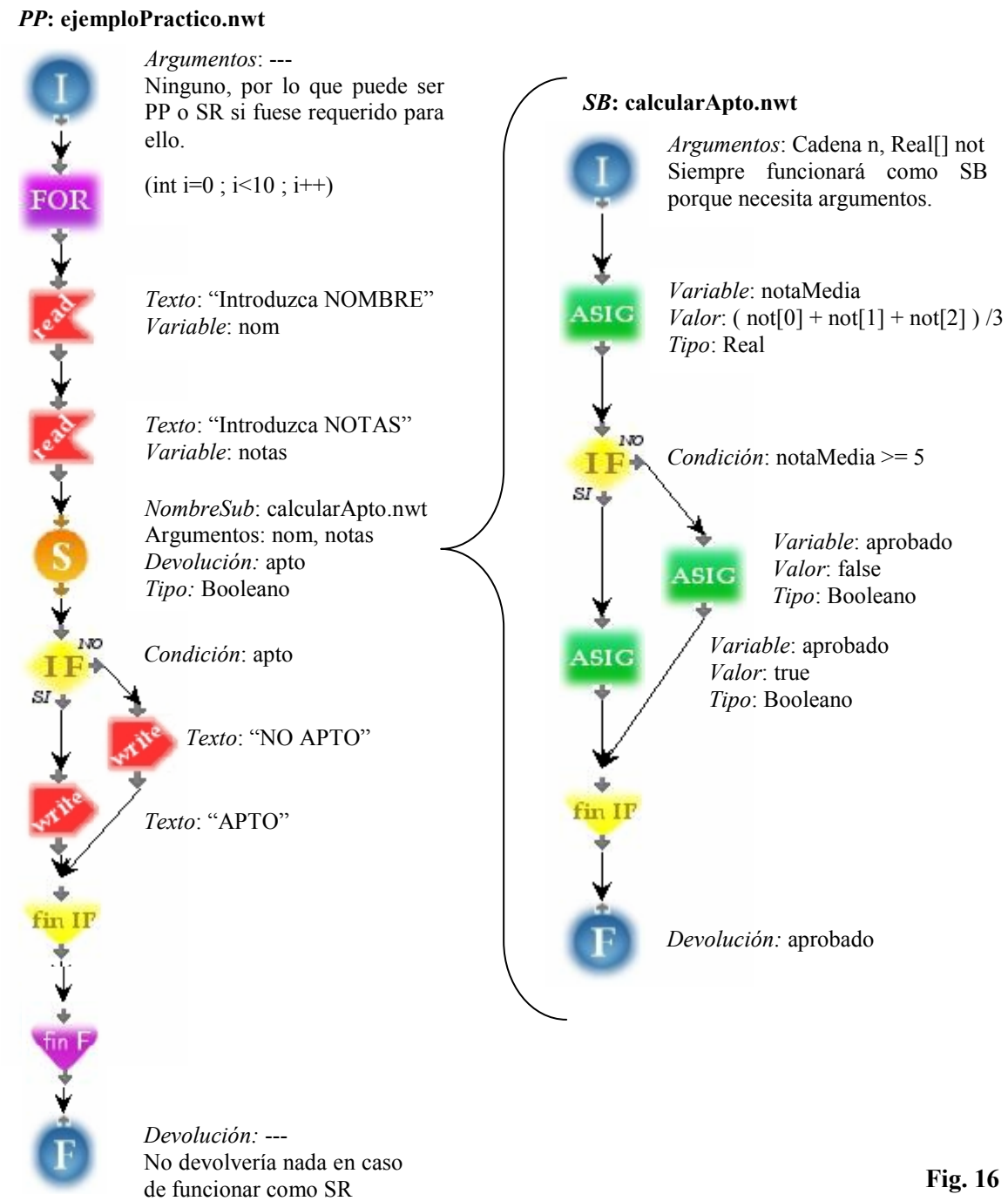


Fig. 16

Como no, los errores relacionados pueden ser detectados en el proceso de validación como ejecución. Veamos los más comunes:

- **Validación**

  - Relacionados con los atributos.

    Mensajes de error asociados:

    "Debe seleccionar NOMBRE DE SUBRUTINA."

    "NOMBRE DE LA VARIABLE utilizada en DEVOLUCIÓN no válido." Error que tiene lugar cuando en el campo "*devolución*" de la pieza SUBR se introduce un dato no válido, como por ejemplo, que almacene resultado devuelto por la subrutina en *43*.

    "ARGUMENTO de SUBRUTINA no válido." Se da cuando en el campo "*argumentos*" de la pieza SUBR se introducen argumentos que no cumplen con lo que se espera en dicho campo (nombres de variables).

- **Ejecución**

  - Asociados a la comunicación entre PP y SR.

    Mensajes de error asociados:

    "ARGUMENTO pasado a una SUBRUTINA incorrecto." Se diferencia de *"ARGUMENTO de SUBRUTINA no válido."* en que el argumento pasado si cumple con las reglas de validación, pero no con las de ejecución, como por ejemplo, que se haya introducido como un nombre de variable, pero esta no exista.

    "El número de ARGUMENTOS pasados a una SUBRUTINA no es el adecuado."

    "La VARIABLE que pretende devolver una SUBRUTINA no concuerda en TIPO con lo que debe devolver dicha subrutina."

  - Si antes de ejecutar el PP no ha validado las SRs.

    Mensaje de error asociado:

    "Debe VALIDAR las SUBRUTINAS invocadas."

3.4 Ámbitos de variable

Pasamos pues al último de los cuatro grandes conceptos pretendemos incorporar en el recién creado *Nivel Intermedio o Estructurado*.

Ámbito de variables

El ámbito de una variable es la parte de un programa donde dicha variable puede ser referenciada. De esta forma, el ámbito de una variable puede ser un módulo, un procedimiento/función o una estructura de control de flujo.

Se distinguen dos modalidades de ámbito:

Global: cuando la variable es accesible desde todos los procedimientos y funciones del programa.

Local: la variable existe desde su declaración en el interior de un bloque hasta el final del mismo.

Pues bien, una vez conocido el concepto y sus modalidades, podemos decir que en la antigua versión, todas las variables creadas eran de ámbito global. En NWTJava v2.0 cambia la filosofía.

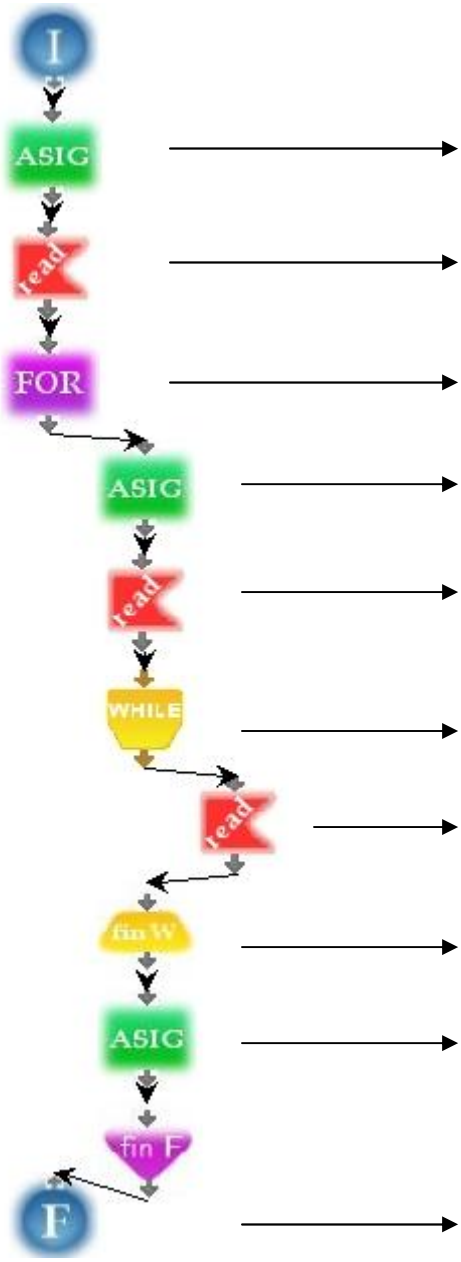
Respecto a este hecho debe conocer:

- Una variable creada en cualquier sitio del diagrama, siempre y cuando sea fuera de un bloque de código particular (bucles, estructuras condicionales, subrutinas), está disponible para todo el código del diagrama desde su declaración hasta el final del mismo.

- Una variable declarada en uno de estos bloques, es una variable local del mismo. Las variables pasadas como parámetros a una subrutina, también son variables locales de la misma.

- Una variable local existe y se puede manipular desde el momento en que se crea hasta el final del bloque donde se creó. Y es únicamente accesible dentro del bloque al que pertenece.

Para finalizar con este aparatado, le mostramos un diagrama a modo de ejemplo con el que podrá visualizar y entender básico:



Ámbito	Nombre Variable	
0	a	
0	b	
1	i	
1	c	
1	a	(modificará el valor de la variable "*a*" creada ya en un ámbito más global)
2	-	
2	d	
1	-	
1	d	(nueva variable "*d*", perteneciente a un ámbito más general que la anterior, la cual desapareció al finalizar su bloque)
0	-	

Fig. 17

Como puede observar, el ejemplo presenta una serie de patrones que se repiten dependiendo del tipo de estructura utilizado. Por ejemplo: siempre que utilizamos un icono de estructura de bucle (ídem para estructuras condicionales), se un nuevo ámbito de variables, el cual será local y asociado a dicho bloque. El proceso inverso tiene lugar cuando llegamos al correspondiente icono de fin de bucle, donde eliminamos el correspondiente ámbito, y con él, sus respectivas variables.

Los errores que puede cometer asociados a este apartado se reducen a uno:

"VARIABLE no inicializada en el ÁMBITO correspondiente."

Y se produce cuando pretende utilizar una variable local fuera del ámbito que le corresponde. Por ejemplo, si la variable "var" es destruida con su respectivo ámbito al haber finalizado su bloque correspondiente, pero después pretende acceder a ella. Por supuesto, este error también sale siempre que se encuentre una variable no inicializada, independientemente de si ha existido en otro ámbito o no.

Pliego IV
Presupuesto

PRESUPUESTO PARA LA REALIZACIÓN DEL PROYECTO
Desglose de Tareas

Tarea	Descripción	Duración (días laborables jornada 4h)
T 1	**Estudio de NWTJava v1.0.**	
T 1.1	Análisis teórico.	45
T 1.2	Experimentación (TSIOCA): • Preparación. • Clases. • Corrección de tests y ejercicios. • Contestación a dudas vía e-mail. • Elaboración de las conclusiones.	50
T 2	**Estudio del estado del arte en cuanto a herramientas software diseñadas para el aprendizaje de la programación estructurada.**	15
T 3	**Diseño e implementación de NWTJava v2.0.**	
T 3.1	Interfaz gráfica: • Rediseño de componentes para mejorar su calidad (iconos). • Aportación de nuevos elementos (etiquetas, nuevos paneles). • Redimensionado de componentes para comodidad del usuario (tamaño de ventanas, paneles).	25
T 3.2	Núcleo de la aplicación.	

T 3.2.1	Bloque 1: Tipos de variable.	
T 3.2.1.1	Diseño	5
T 3.2.1.2	Implementación	35
T 3.2.1.3	Pruebas y correcciones	10
T 3.2.2	Bloque 2: Arrays	
T 3.2.2.1	Diseño	10
T 3.2.2.2	Implementación	40
T 3.2.2.3	Pruebas y correcciones	7
T 3.2.3	Bloque 3: Subrutinas.	
T 3.2.3.1	Diseño	30
T 3.2.3.2	Implementación	12
T 3.2.3.3	Pruebas y correcciones	15
T 3.2.4	Bloque 4: Ámbito de variables.	
T 3.2.4.1	Diseño	20
T 3.2.4.2	Implementación	5
T 3.2.4.3	Pruebas y correcciones	5

T 4	**Realización de pruebas de verificación técnica de la aplicación y corrección de errores.**	20

T 5	**Redacción de la documentación perteneciente a la memoria.**	60

Costes por tiempo

Tarea	Tipo profesional (Ver *Tabla 2*)	Horas	Total Costes
T1	1	380	
T2	1	60	
T3	2	876	
T4	2	80	
T5	1	240	
		1636	
Total horas personal tipo (1)		680	10.104,8 €
Total horas personal tipo (2)		956	9.971.1 €
Total Costes por tiempo según la tarifa por hora calculada en la *Tabla 2*		**20.075,9 €**	

Costes Materiales			
Componente	Coste Total	Coste Asociado	Total Coste Material
Conexión Internet ADSL	280 €	75 €	
Intel Core Duo 1,7 GHz	999 €	150 €	
Softwares utilizados	1239 €	399 €	
Documentación	130 €	110 €	
	2.648 €	734 €	**3.382 €**
TOTAL COSTES			
Material		3.382 €	
Personal		20.075,9 €	
		23.457,9 €	

Categoría	**Descripción**	**Sueldo Bruto Anual**	**Coste Anual para la empresa**	**Coste por hora**
1	Analista	21.969 €	24.879 €	14,86 €
2	Programador Senior	15.442 €	17.658 €	10,43 €

Tabla 2: Coste salarial según categoría profesional.

Nota: Las categorías profesionales y los correspondientes salarios se han obtenido a partir del XVI Convenio Colectivo Estatal de Empresas Consultoras [*http://www.comfia.net/archivos/tic/XVI_Convenio_Colectivo_Texto.pdf*] y las últimas tablas actualizadas, publicados por el Ministerio de Trabajo y Asuntos Sociales.

El presupuesto total de este proyecto asciende a VEINTITRES MIL CUATROCIENTOS CINCUENTA Y SIETE EUROS CON 90 CÉNTIMOS.

Fdo**:**

José Francisco González Ruiz. Ingeniero Técnico de Telecomunicaciones.

Referencias.

[1] http://sle.mty.itesm.mx/investigadores/raul_valente_ramirez

[2] http://es.wikipedia.org/wiki/Algoritmo_divide_y_vencer%C3%A1s

[3] "How Computers Work".
White, Ron.
(PC COMPUTING) Ziff-Davis Press, 1993.

[4] Epistemología.
http://es.wikipedia.org/wiki/Ciencia_del_conocimiento
http://www.monografias.com/trabajos/epistemologia2/epistemologia2.shtml

[5] Proyecto Fin de Carrera: "NWTJava, un micro-entorno de desarrollo para el aprendizaje de la programación".
Autor: Patricia Fernández Garrido.
Tutor: Dr. José Jesús García Rueda.
Leganés, 2006.

[6] Jean William Fritz Piaget.
http://www.biografiasyvidas.com/biografia/p/piaget.htm

[7] "Esquemas algorítmicos fundamentales".
Scholl P. C. y Pieryn J. P.
Masson S.A., 1991.

[8] "Desafío a la mente".
Papert, Seymour.
Ediciones Galápago, 1985.

[9] Constructivismo.
http://www.cop.es/colegiados/M-00407/CONSTRUCTIVISMO.HTM

[10] "Cambios metodológico-didácticos y evaluación del impacto de los mismos en un curso introductorio a los conceptos de algorítmica y programación".
Ferreira Szpiniak, Ariel. y Rojo, Guillermo.

[11] "Enseñanza de la programación".
Revista Iberoamericana de Tecnología en Educación y Educación en Tecnología.
Ferreira Szpiniak, Ariel. y Rojo, Guillermo.

[12] "SL: un lenguaje para la introducción a la algoritmia".
Segoviano Silvero, Juan.
Centro Nacional de Computación, Universidad Nacional de Asunción.
http://newton.cnc.una.py/id115.htm

[13] Manual de usuario: “Introducción al lenguaje SL”.
Segoviano Silvero, Juan.
Centro Nacional de Computación, Universidad Nacional de Asunción.
http://www.cnc.una.py/sl/libro-sl.pdf

[14] Documentación de clases de Swing.
http://java.sun.com/j2se/1.4.2/docs/api/javax/swing/package-summary.html

[15] Documentación de clases de jgraph.
http://www.jgraph.com/pub/api/

[16] JEP - Java Math Expression Parser.
http://www.singularsys.com/jep/

[17] Visión general del concepto “array”.
http://es.wikipedia.org/wiki/Array

[18] Documentación de clase de Hashtable.
http://java.sun.com/j2se/1.4.2/docs/api/java/util/Hashtable.html

[19] Tutorial para la construcción de DTDs.
http://www.zvon.org/xxl/DTDTutorial/General_spa/contents.html

[20] Web oficial de Smalltalk.
http://www.smalltalk.org/main/

[21] Magníficos enlaces para conocer la herramienta Squeak y su lenguaje de programación asociado Smalltalk.
http://www.squeak.org/
http://www.squeak.org/Smalltalk/