Enfoque diacrónico para la enseñanza de la programación imperativa

L. Fernández Muñoz (a), R. Peña (b), F. Nava (c), Á. Velázquez Iturbide (c)

(a) Dept. LPSI. Universidad Politécnica. 28071 Madrid e-mail: setillo@eui.upm.es

(b) Facultad de Documentación. Universidad de Alcalá. 28871 Madrid e-mail: rpr@uah.es

(c) Escuela Superior de Ciencias Experimentales. Universidad Rey Juan Carlos. 28933 Madrid e-mail: {f.j.nava,a.velazquez}@escet.urjc.es

Resumen

Presentamos una alternativa para enseñar la programación imperativa no orientada a paradigmas; proponemos un enfoque diacrónico basado en la exposición justificada de cada constructor de los lenguajes de programación a través de su evolución histórica, motivada por los conceptos recurrentes que subyacen a los mecanismos particulares de cualquier paradigma.

Especificamos los objetivos que debe perseguir esta docencia, así como los contenidos que debe cubrir. Para alcanzar los objetivos empleamos herramientas de desarrollo y visualización del software, sobre un lenguaje flexible que se adecua sintáctico-semánticamente al avance de la exposición de la materia. La adecuación del lenguaje a los conceptos impartidos minimiza el tiempo dedicado al aprendizaje de su sintaxis y los recursos pedagógicos empleados ayudan a asentar los conocimientos.

La estructura propuesta elimina la dificultad que supone el cambio de paradigma, encontrada en experiencias que presentan el paradigma procedimental en primer curso y el orientado a objetos (OO) en segundo, y la sobrecarga inicial de conceptos que implica empezar directamente con el paradigma orientado a objetos.

1. Introducción

En el momento actual existe un fuerte debate en torno a la planificación de la enseñanza de la programación imperativa en los primeros cursos de los estudios de informática. En un trabajo previo [6] hemos analizado las experiencias docentes publicadas, tropezando con inconvenientes que impiden seleccionar, entre ellas, el enfoque adecuado.

Entendemos que la discusión sobre la planificación no debe centrarse en el paradigma a presentar en primer lugar, o en el lenguaje empleado para las prácticas, sino en los objetivos perseguidos y en el modo de conseguirlos, por lo que realizamos una nueva propuesta docente.

Este artículo se inicia presentando muy brevemente, en su apartado 2, los problemas que plantean otras propuestas docentes. El apartado 3 describe el enfoque diacrónico: sus objetivos y la planificación del contenido a impartir. El apartado 4 describe la metodología adecuada para la consecución de estos objetivos.

2. Revisión de las experiencias docentes publicadas

Durante muchos años el paradigma procedimental ha sido el adoptado para el primer acercamiento a la programación. En los últimos veinte años la mayoría de los centros de enseñanza universitaria de Informática ha integrado conceptos de OO en cursos superiores de sus planes de estudio. Pero los alumnos presentan dificultades al cambiar de paradigma, viviendo como una ruptura la exposición al nuevo, lo que se ha llamado el "problema del desplazamiento de paradigma".

Para evitarlo, algunos autores han optado por experimentar el primer acercamiento a la programación directamente desde el paradigma

OO. Este enfoque conlleva la necesidad de exponer muchos y novedosos conceptos al alumno antes de poder ponerlos en práctica y asentarlos, por tanto, supone una sobrecarga de conceptos que puede resultar desbordante.

Algunas de las propuestas docentes que empiezan con OO se basan en la manipulación de universos virtuales, aprovechando el supuesto carácter "intuitivo" del paradigma orientado a objetos. Este tratamiento conlleva una sobresimplificación que genera expectativas irreales.

Otras experiencias de OO desde primero posponen la programación, basando el desarrollo del curso en análisis y diseño. Pero el lenguaje determina el pensamiento, y resulta imposible diseñar sin disponer de un lenguaje subyacente. La OO es intuitiva porque utiliza los mecanismos habituales en la adquisición del conocimiento, pero un diseño eficiente (tanto en el momento del desarrollo, ejecución como en el mantenimiento) no suele ser paralelo a la percepción informal del mundo real y, por tanto, un alumno de primer curso no está mejor preparado para comprender este paradigma que otros.

Otras propuestas de OO en primero empiezan directamente con programación. Usan patrones, frameworks y bibliotecas para fomentar la abstracción, encapsulación, modularización, y reutilización. Los objetivos son buenos, pero sin duda, el empleo de estas herramientas requiere del dominio de muchos conceptos de los que el principiante no dispone.

En las propuestas que se centran en desarrollo de interfaces gráficas de usuario, además del problema que acabamos de mencionar, se maneja un modelo de eventos globalizado -applets- o una arquitectura sin eventos, que no tienen envergadura para resolver problemas reales y posiblemente fomentarán hábitos inadecuados.

Paralelamente, las experiencias presentadas proponen tres soluciones para el lenguaje de soporte de las prácticas. La primera emplea un lenguaje distinto para cada uno de los paradigmas a presentar. Tiene el inconveniente de que es necesario desviar, continuamente, la atención de los conceptos presentados hacia la sintaxis del nuevo lenguaje.

La solución de emplear un lenguaje OO desde el principio implica necesariamente el empleo de conceptos desconocidos por el alumno.

La tercera vía emplea lenguajes híbridos, que en principio, parecen resolver los problemas de las anteriores soluciones, pero generan confusión en el alumno a la hora de ubicarse en uno u otro paradigma, ya que el programa "compila" y "funciona" aunque el diseño no se adecue al paradigma.

3. Enfoque diacrónico

3.1. Justificación y presentación

Partimos de que "las teorías científicas son entidades que se extienden o perduran en el tiempo, que permanecen a través del cambio. Ello supone que el estudio puramente sincrónico que las considera como entidades estáticas, *congeladas*, constituye sólo una primera aproximación que se debe completar con un análisis diacrónico que dé cuenta del carácter persistente de estas entidades" [5].

Esta premisa de la filosofía de la ciencia, en el mundo de la programación, se reescribe en los siguientes términos: "cada paradigma se construye sobre los que lo han precedido, añade algo nuevo al arsenal de herramientas del programador [...] y refleja un planteamiento de diseño" [11]. En particular, no existen dos paradigmas enfrentados, estructurado y objetos; existe un único modelo imperativo cuya evolución genera nuevos enfoques persiguiendo lo mismo: la mejor gestión del software. En palabras de Lewis [9] "El enfoque OO no abandona los conceptos que admiramos en el enfoque procedimental, los aumenta y los fortalece".

La transición entre paradigmas es natural y evolutiva pero algunos alumnos lo encajan como un cambio total. Nuestra experiencia en la docencia de ambos paradigmas demuestra que el problema del desplazamiento no es global. Los alumnos aventajados en la programación procedimental asumen con naturalidad el nuevo paradigma, entendiéndolo como algo lógico y casi esperado. Este hecho queda patente con la frase de Alan Kay, creador de Smalltalk, cuando

evaluó SIMULA-67: "el impacto fue tan grande que fue la última vez que pensé en términos de subrutinas y estructuras" [10].

La conjunción de la dualidad entre la dificultad inicial mayoritaria y la naturalidad del cambio para alumnos aventajados, conduce a un nuevo planteamiento: el problema del desplazamiento afecta a los alumnos no aventajados. Agrupamos en esta denominación, los que superan la asignatura del paradigma procedimental y usan correctamente los constructores del lenguaje, cómo, cuándo y dónde usarlos, pero carecen de una visión más profunda y general de la programación. No han asumido el porqué de los constructores del lenguaje, sus implicaciones y sus carencias. Desde una perspectiva más general, tampoco han asumido que "ciertos conceptos fundamentales son recurrentes a través de toda la disciplina... son ideas significativas, cuestiones, principios y procesos que ayudan a unificar una disciplina académica profundamente" [1]. El informe “Computing curricula’91” ha establecido los siguientes 12 conceptos recurrentes: ordenación en el espacio y tiempo, enlace, niveles de abstracción, completitud, evolución, complejidad, compromisos, seguridad, modelos, reusabilidad y eficiencia.

Tradicionalmente, los primeros cursos de programación se centraban en la exposición de los mecanismos concretos de un lenguaje, no en los conceptos recurrentes de la Informática. "Un concepto recurrente es más fundamental que cualquiera de sus ejemplarizaciones. Se sustenta en sí mismo como fundamental, persistente a lo largo de la historia de la computación, y es muy probable que permanezca en un futuro" [1].

La incorporación de la OO en los planes de estudio evidenció las carencias de estos cursos de introducción a la programación, reflejadas por el problema del desplazamiento del paradigma, al incluir nuevas terminologías y objetivos de diseño: evolución, reusabilidad, jerarquización, acoplamiento, encapsulación, etc. La primera respuesta a este problema fue invertir el orden de exposición, trayendo otras carencias en su lugar: la sobrecarga y, en algunos casos, el desconcierto o la simplificación excesiva de la programación OO en base a su carácter intuitivo, e incluso, la ausencia de la exposición de la programación procedimental.

Entonces, el problema del desplazamiento radica en la ruptura del contenido de la exposición: con el cambio del paradigma, la terminología es completamente nueva (abstracción, encapsulación, modularización, jerarquización...), el lenguaje también es nuevo o inadecuado y los objetivos de ambos paradigmas parecen diferentes. El exponente más dramático de esta situación se recoge en [7]: "algunos profesores comienzan su primera clase con *olviden cualquier cosa que sepan. OO es radicalmente diferente*. Es ilógico". No hay nada que desaprender, la OO no supone una revolución de conceptos, sino una evolución.

Tras detectar esta situación, hemos modificado nuestros cursos de OO, comenzando con un profundo análisis del grado de concreción de los conceptos recurrentes que exhibe el porqué, implicaciones y carencias de los mecanismos del lenguaje procedimental (sistema de tipos, registros, subprogramación, módulos, etc). Para fortalecer el carácter diacrónico desarrollamos un nuevo lenguaje OO, con la misma sintaxis y semántica del lenguaje empleado en primero, para los conceptos asumidos de la programación estructurada, pero que introduce los mínimos cambios necesarios para el nuevo paradigma. El aprovechamiento de los alumnos, sus resultados académicos, y las encuestas de evaluación de la docencia mejoraron notablemente.

En base a esta experiencia, en este trabajo proponemos una tercera vía, un enfoque diacrónico basado en la exposición justificada de cada concepto de la programación a través de su evolución histórica motivada por los conceptos recurrentes que subyacen a los mecanismos particulares de cualquier paradigma de la programación.

Esta manera de exponer la programación en los cursos de introducción mejora la comprensión y mitiga el impacto de la transición al paradigma OO. Los cambios propuestos afectan no tanto al índice del curso como a sus objetivos, a la terminología y al lenguaje y naturaleza de las prácticas. Un ejemplo de esta idea es que el concepto de enlace (estático o dinámico) no surge al presentar la sobrecarga y

el polimorfismo, que es hasta donde se suele postergar su introducción, sino que ya está presente al ligar los nombres de constantes y variables a su valor. En nuestra propuesta, el concepto de enlace, al igual que el resto de conceptos recurrentes, debe ser resaltado desde sus primeras concreciones, y en todas y cada una de ellas. Con esto nos diferenciamos del enfoque procedimental en primero y OO en segundo, ya que presentamos cada constructor como una justificación conjugada de los mismos conceptos recurrentes, independientemente del paradigma al que pertenezca.

3.2. Objetivos

El enfoque diacrónico posibilita el cumplimiento de los siguientes objetivos para mitigar las desventajas de las propuestas anteriores:

- Una exposición escalonada y paulatina de los conceptos de la programación en sintonía con la madurez del aprendiz, evitando la sobrecarga inicial, fluyendo a través de la programación estructurada, modular, basada en objetos, orientada a objetos, etc. "El aprendizaje debería estar basado en el conocimiento previo de los programadores" [4]. "La construcción del conocimiento se forja recursivamente sobre el conocimiento que los estudiantes ya tienen" [2];
- Anulación del impacto del "desplazamiento de paradigma" de programación estructurada a OO gracias a un hilo conductor: los conceptos recurrentes; "la estimación de la penetración de estos conceptos y la facultad para aplicarlos en un contexto apropiado es un indicador de la madurez de los graduados" [1].
- Herramientas adecuadas para la participación en el desarrollo temprano de aplicaciones motivadoras (potentes y con interfaces gráficas de usuario), pero evitando la exposición a conceptos más complejos que sus capacidades.

La figura 1 presenta un esquema de la evolución de los enfoques discutidos.

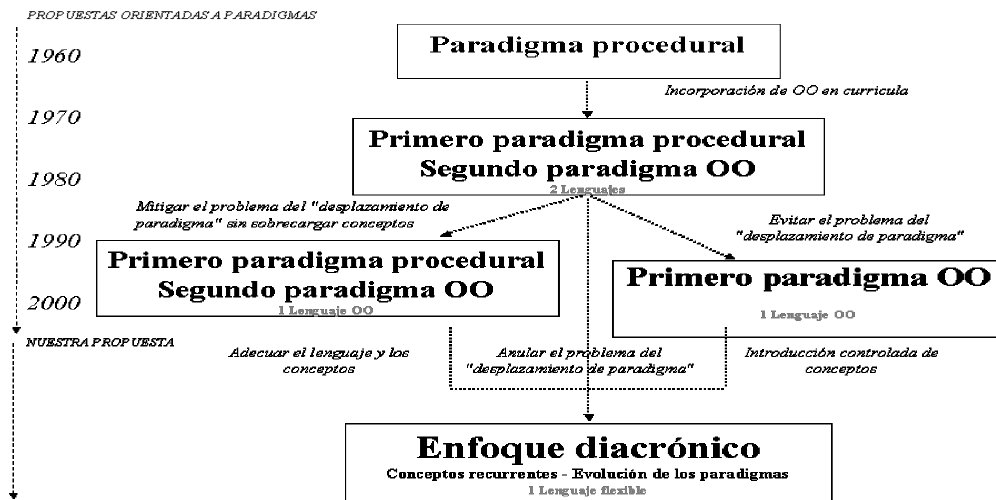

Figura 1 Evolución de los enfoques de la enseñanza de los paradigmas

3.2. Planificación

Los contenidos expuestos deben asegurar la comprensión de los conceptos recurrentes de la Informática y establecer su grado de concreción en cada mecanismo de la programación. "En el diseño de un currículo concreto, estos conceptos recurrentes deben comunicarse de manera efectiva; es importante notar que el uso apropiado de los conceptos recurrentes es un elemento

esencial en la implantación del currículo y cursos" [1].

Proponemos una previa exposición de estos conceptos mediante analogías con el mundo real. Por ejemplo: complejidad y evolución de las normas de tráfico; modelos formales en el "lenguaje de un semáforo"; enlace estático entre un vehículo y su propietario o dinámico con su conductor. A través de toda la exposición de la materia se vuelve a incidir en el papel de los conceptos recurrentes. Un ejemplo es exponer el concepto de seguridad desde el sistema de tipos, pasando por las precondiciones y poscondiciones de las sentencias de control de flujo de ejecución y de los subprogramas, las invariantes de los bucles, hasta llegar al interfaz de los módulos, tipos abstractos de datos y clases.

Como hemos comentado anteriormente, exponer el concepto de enlace estático y dinámico presente entre las constantes, variables o expresiones y sus tipos, valores..., pasando por el enlace entre los tipos genéricos a sus tipos concretos, hasta el enlace en la sobrecarga y el polimorfismo con mensajes a métodos.

O exponer los conceptos de niveles de abstracción, el control de la evolución, la posible reusabilidad y la resolución de compromisos, desde las primeras oportunidades en la solución de los problemas con registros, subprogramación, programación modular, etc.

Por tanto, la terminología del alumno de primer curso debe incorporar términos propios del diseño como patrones, modularidad, encapsulación, acoplamiento, cohesión, abstracción, jerarquización, legibilidad, fiabilidad... y saber evaluarlos en cada mecanismo de la programación.

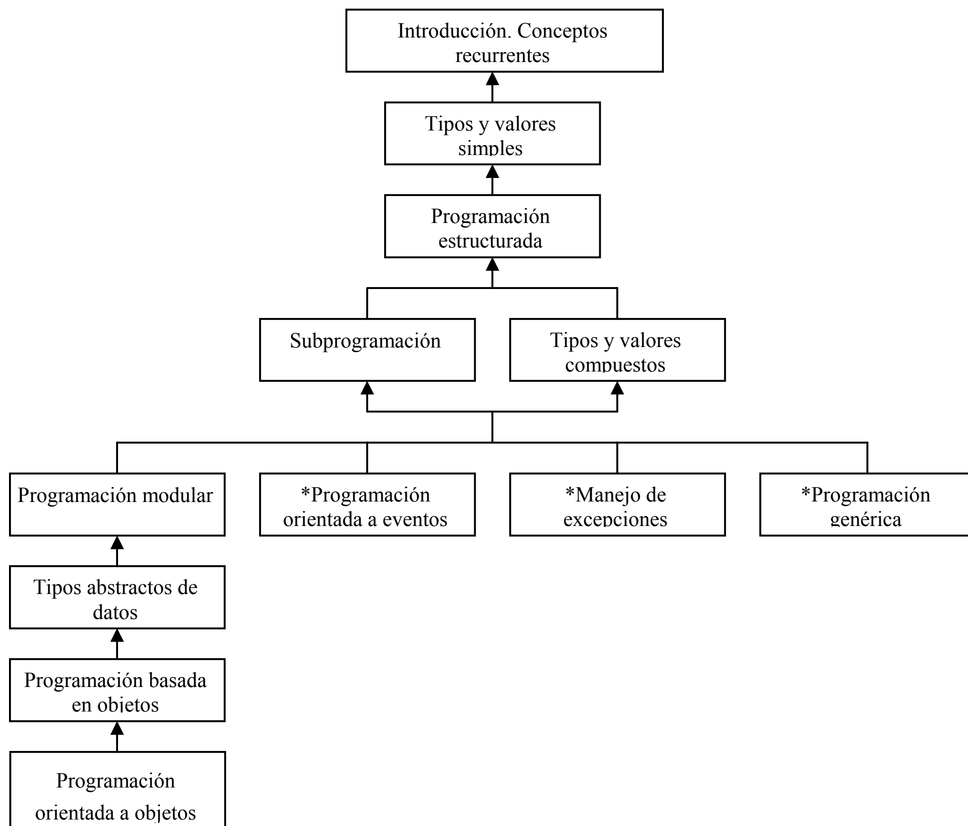

Figura 2.Prerrequisitos entre los núcleos teóricos del enfoque diacrónico.

Acorde al carácter diacrónico basado en la filosofía de la ciencia y el constructivismo de la psicología de la programación, nuestra planificación contiene un conjunto de núcleos cerrados de conceptos en que cada uno viene a resolver los inconvenientes evidenciados y subrayados en las soluciones de programas construidos con los conceptos anteriormente disponibles.

Los 12 núcleos considerados son:

- Introducción: conceptos recurrentes: datos y procesos;
- Tipos y valores simples: constantes y variables; expresiones, precedencia y asociatividad;
- Programación estructurada: sentencias de control de flujo de ejecución;
- Subprogramación: parámetros, Entrada/Salida, recursividad;
- Tipos y valores compuestos: registros y tablas; estructuras de datos dinámicas; ficheros;
- Programación genérica*: plantillas y concreción de tipos;
- Programación modular: interfaz e implantación; acoplamiento y cohesión;
- Manejo de excepciones*: elevación, delegación y captura;
- Programación orientada a eventos*: elevación, delegación y captura;
- Tipos abstractos de datos: múltiple instanciación
- Programación basada en objetos: clases y objetos; sobrecarga.
- Programación orientada a objetos: herencia y polimorfismo.

Esta planificación incluye núcleos opcionales (marcados con'*') que se introducirán en mayor o menor grado, dependiendo de la profundidad deseada en cada curricula. De modo que la figura 2 debe interpretarse como una tabla de prerrequisitos entre núcleos que posibilitan diferentes secuencias de exposición y su distribución, en dos o tres cursos, según convenga en un centro o titulación concreta.

4.Metodología

4.1. Lenguaje flexible

El lenguaje de programación debe ser ejecutable, permitir el desarrollo de aplicaciones motivadoras, pero flexible para incorporar nuevos mecanismos y eliminar otros correspondientes a conceptos subsumidos con el avance del curso. Por ejemplo, este lenguaje debe mantener la tabla de precedencia de operadores, las mismas sentencias de control de flujo de ejecución..., pero debe inhibir los registros y los subprogramas libres al introducir la programación basada en OO, etc.

Por tanto, debe cambiar las palabras reservadas correspondientes a la organización de la estructura de los programas, pero mantener inalterable las reglas sintáctico-semánticas de aquellos conceptos asumidos en los nuevos enfoques: precedencia de operadores, sentencias de control de flujo de ejecución, etc.

4.2. Prácticas

"La teoría dominante de la enseñanza hoy día, llamada constructivismo, afirma que el conocimiento se construye activamente por el estudiante, no se adquiere pasivamente desde los libros de texto o en las aulas" [2]. Así "enfatizar las actividades de lectura y depuración, simultáneamente a las de programación puede iluminar concepciones" [8].

"El aprendizaje cognitivo de la taxonomía de Bloom es útil en la estructuración del principio del currículo informático. Cada nivel de la jerarquía está subsumido por el siguiente nivel, de modo que, funcionalidades más altas exigen habilidad en los niveles inferiores" [3]. Las etapas en dicha taxonomía son:

- Conocimiento: el recuerdo de material previamente aprendido
- Comprensión: la habilidad para extraer el significado del material
- Aplicación: la habilidad para usar el material aprendido en una situación nueva y concreta
- Análisis: la habilidad para descomponer un material en sus componentes y comprender su estructura organizativa
- Síntesis: la habilidad para poner partes juntas formando un todo
- Evaluación: la habilidad de juzgar el valor de un material para un propósito determinado.

La vertiente práctica de nuestra propuesta respeta la aproximación constructivista a través de

los niveles de aprendizaje para cada núcleo teórico de la planificación. Por ejemplo, para la docencia de las estructuras de control de ejecución:

- Conocimiento: sintaxis y semántica de las estructuras
- Comprensión: dados unos pequeños bloques de código, extraer el control de flujo de ejecución y sus efectos sobre el entorno (variables, periféricos, etc.)
- Aplicación: escribir las estructuras que satisfacen unos requisitos específicos
- Análisis: dados unos bloques de código, extraer su organización para comprender los requisitos que satisfacen
- Síntesis: dada una aplicación incompleta, dotarla de la funcionalidad deseada
- Evaluación: contrastar soluciones alternativas, con estructuras y sin estructuras –por ejemplo con sentencia *goto*- y, por otro lado, presentar soluciones “inadecuadas” con 100 líneas de código de estructuras anidadas, evidenciando las bondades y carencias de las estructuras de control de flujo de ejecución en términos de los conceptos recurrentes: complejidad, ordenación en el espacio, evolución, niveles de abstracción, etc.

  Para la consecución de los objetivos consideramos vitales los dos últimos niveles.
- Síntesis: el alumno incorpora código en aplicaciones motivadoras, con interfaces gráficos de usuario, bibliotecas, sistemas de comunicaciones, etc. Todo ello, resaltando la aplicabilidad del material aprendido en el “mundo real”, pero, evitando la exposición y manejo de conceptos avanzados
- Evaluación: permite al alumno reconocer, en cada núcleo, el hilo conductor en nuevas concreciones de los conceptos recurrentes y justifica, visualizando las limitaciones, la necesidad de evolucionar a nuevos constructores del lenguaje.

4.3. Herramientas

Las herramientas necesarias para la concreción del enfoque diacrónico responden, paralelamente, a los niveles cognitivos de Bloom:

- Intérpretes para la visualización del software que apoyen los niveles de conocimiento y comprensión: enlace estático de tipos a expresiones, orden de evaluación de operadores en las expresiones; orden de ejecución de las estructuras; comunicación por el paso de parámetros; desencadenamiento de instancias y mensajes en objetos; etc.
- Depuradores para el nivel de aplicación.
- Herramientas CASE e ingeniería inversa para extraer la organización de software en el nivel de análisis: diagramas de jerarquías de estructuras, subprogramas, módulos, clases… adecuados al núcleo que se está impartiendo.
- Entornos de desarrollo con bibliotecas, compilador,… y herramientas que bloqueen la edición del código de una aplicación excepto en los ámbitos a codificar, para la fase de síntesis: codificar el cuerpo de un subprograma, la implantación de un módulo o un TAD, etc.
- Métricas del software para el nivel de evaluación que exhiban la complejidad y consumo de memoria; acoplamiento y cohesión de módulos, TAD’s o clases; etc. de las soluciones implementadas.

Todas estas herramientas deben configurarse con un lenguaje evolutivo que incorpore e inhiba reglas sintáctico-semánticas, adecuándose a la secuencia de núcleos expuestos en la planificación.

5. Conclusiones

Con la incorporación de la OO en los planes de estudio han surgido inconvenientes, tanto en su asimilación por parte de los alumnos, como en la localización de recursos adecuados por parte del profesor (disponibilidad de textos, ejemplos que encajen en una sesión docente, incluso elección del lenguaje).

El enfoque, procedimental en primero y OO en segundo, evidenció el problema del "desplazamiento de paradigma". Posteriormente, el enfoque contrapuesto, OO en primero, trajo la sobrecarga inicial de conceptos para los alumnos y, en algunos casos, la inadecuación de los lenguajes a los paradigmas.

Hacemos una propuesta docente, no orientada a paradigmas, que elimina los inconvenientes de las anteriores, utilizando un enfoque diacrónico, que consiste en la exposición justificada de cada constructor de los lenguajes de programación, a través de su evolución histórica, motivada por los

conceptos recurrentes que subyacen a los mecanismos particulares de cualquier paradigma, de modo que cada nuevo concepto a presentar surge para resolver las limitaciones de los anteriores. Reclamamos la necesidad de usar herramientas pedagógicas que apoyen la docencia, basándose en un lenguaje eminentemente evolutivo. Este lenguaje incorpora y elimina, según el avance de la materia, las reglas sintáctico-semánticas de los mecanismos de la programación.

Por tanto, nuestra propuesta modifica:

- objetivos, en concordancia con el enfoque diacrónico de la filosofía de la ciencia, concretados en los conceptos recurrentes de la Informática;
- planificación y lenguaje, acorde al constructivismo de la psicología de la programación, donde la adquisición del conocimiento se construye sobre conocimientos previos;
- prácticas y herramientas, acordes a los niveles cognitivos de la taxonomía de Bloom, que propugna la lectura del código, la depuración, el análisis, la implantación de aplicaciones y la evaluación de soluciones.

Agradecimientos

Este trabajo se ha financiado con el proyecto TIC2000-1413 de la CICYT.

Referencias

[1] ACM/IEEE. Computing curricula 1991. http://www.computer.org/education/cc1991/eab1.html

[2] Ben-Ari, M. *Constructivism in computer science education.* 30 SIGCSE Technical Symposium on Computer Science Education, 1998

[3] Buck, D.; Stucki, D. *Design early considered harmful: Graduate exposure to complexity and structure based on levels of congnitive development.* SIGCSE 2000 3/00

[4] Détienne, F. *Software-design cognitive aspects.* Springer 2002

[5] Díez, J.A.; Moulines, C.U. *Fundamentos de la Filosofía de la Ciencia.* Ariel, 1997

[6] Fernández Muñoz L., Peña R., Nava F., Velázquez Iturbide A. *Análisis de las propuestas de la enseñanza de la programación orientada a objetos en los primeros cursos.* JENUI 2002.

[7] Fernández, A.; Rossi, G. *An Effective approach to learning Object-Oriented technology* ECOOP 98 xxii+573pp

[8] Fleury, A.E. *Programming in Java: student-constructed rules.* SIGCSE 2000, 3/00

[9] Lewis, J. *Myths about OO and its pedagogy.* SIGCSE 2000, 245-49

[10] Sterwart, M.K. *The natural history of objects* C++. Report nov 1992 (suplemento) pp.12-13

[11] Stroustrup, B. *El lenguaje de programación C++* Addison Wesley, 1998