Un enfoque semiformal para la introducción a la programación

Jesús García Molina
Departamento de Informática y Sistemas
Universidad de Murcia
30071 Murcia
e-mail: jmolina@um.es

Resumen

Desde hace trece años, en nuestra Facultad se aplica un enfoque semiformal para la introducción a la programación, inspirado en el planteamiento expuesto en el libro *"Esquemas Algorítmicos Fundamentales"* de Scholl y Peyrin [14]. En este trabajo, tras analizar la evolución de la enseñanza de la programación y establecer los objetivos a alcanzar, describiremos nuestro enfoque, mostrando las aportaciones más importantes en relación a la propuesta original.

1. Introducción

Por lo general, un primer curso de programación se organiza en dos partes (dentro de una asignatura anual o como dos asignaturas cuatrimestrales). La primera parte se ocupa de introducir a los alumnos en la programación y la segunda de enseñar conceptos fundamentales de programación, como son la recursividad, las estructuras de datos dinámicas y los algoritmos de ordenación. Nos referiremos a estas dos partes como IP (Introducción a la Programación) y FP (Fundamentos de Programación).

Cuando hace catorce años tuve que enfrentarme a la docencia de IP, organicé la enseñanza de acuerdo a un enfoque clásico: conceptos de un lenguaje de programación procedural y ejemplos que ayudaban a su comprensión. Tras la experiencia del primer año comprendí que ese no era el camino para enseñar a programar. Entonces comencé a recopilar y analizar los trabajos más significativos que se habían publicado sobre métodos para enseñar a programar, entre los que destacaría: la propuesta de Dijkstra de un enfoque formal, ilustrada en su libro "*A method of programming*" [5]; el estudio sobre la naturaleza de la programación del grupo Anna Gram [3] y el libro *"How to Solve it by Computer"* de Dromey [7] inspirado en el clásico trabajo de Polya.

El resultado de aquel análisis fue una nueva organización docente siguiendo un enfoque semiformal basado en las ideas de Anna Gram tal y como son reflejadas en el texto *"Esquemas Algorítmicos Fundamentales"* (en adelante EAF) de Scholl y Peyrin [14].

Como miembro de tribunales de plazas de TEU y CEU he conocido muchos proyectos docentes para IP en los que se planteaba el enfoque no formal de *"conceptos del lenguaje más algoritmos para ilustrarlos"*. Por otra parte, en las últimas ediciones de las JENUI se han presentado diferentes propuestas para IP cuyo interés estaba más centrado en el paradigma a elegir que en el método a seguir. También he observado como en los últimos años se han publicado numerosos libros en los que la programación se sigue enseñando a través de ejemplos en un determinado lenguaje, pero es raro encontrar libros que propongan métodos novedosos para introducir a los alumnos en el arte de programar, al estilo de los mencionados arriba. Durante la preparación de esta ponencia he conocido el libro "*Touch of Class*" de B. Meyer [13] que ilustra cómo abordar un curso de IP/FP desde una perspectiva orientada a objetos pura. En el prólogo, Meyer comenta que "*actualmente, muchos departamentos de informática de todo el mundo están preocupados sobre cómo enseñar la introducción a la programación considerando los nuevos desafíos que se suman a los tradicionales*".

Por todo ello, pienso que todavía hoy es necesario un análisis sobre cómo diseñar un curso IP, aunque para muchos parezca un problema resuelto. Ahora que han pasado trece años desde que planteé el mencionado enfoque semiformal y hace ocho años que no imparto la materia, me he

atrevido a escribir este trabajo cuyo objetivo es describir el enfoque y justificar su interés. Como es lógico, los compañeros que han participado en la docencia de la asignatura a lo largo de estos años han enriquecido la propuesta original con sus aportaciones, y ahora nos encontramos preparando un libro a partir de un texto previo [10] que se ha utilizado como texto-guía de la asignatura.

El trabajo se ha organizado de la siguiente manera. En el siguiente apartado se realiza un análisis de la evolución de la enseñanza de la programación. En el tercero se establecen los objetivos de un curso de introducción a la programación. En el cuarto se describen las principales ideas en EAF. A continuación se describe nuestro enfoque, indicando las aportaciones con respecto a EAF y finalmente se presentan las conclusiones y la bibliografía.

2. La evolución de la enseñanza de la programación

Dentro de la evolución de la programación podemos distinguir tres grandes etapas. En la primera etapa la programación se concibe como un arte y las principales herramientas del programador son el lenguaje elegido y sus habilidades personales.

La segunda etapa arrancaría con la conocida reunión de Garmisch de 1968 en la que se reconocen las dificultades de programar y la existencia de una "crisis del software". Los trabajos de Dijkstra sobre *programación estructurada* y de Hoare sobre *tipos de datos* abrieron el camino a una visión más formal de la programación, al tiempo que ofrecieron mecanismos de abstracción para dominar la complejidad. Otras ideas influyentes de esta segunda etapa son: la técnica del *refinamiento por pasos sucesivos* de Wirth, y los conceptos de *ocultación de información* de Parnas, y de *jerarquía de tipos* de Dahl. Mientras que en la primera etapa, la abstracción procedural era el principal mecanismo para reducir la complejidad y organizar los programas, en esta segunda etapa surgen las abstracciones de datos o tipos abstractos de datos (TAD).

La tercera etapa está caracterizada por la aparición de un conjunto importante de tecnologías de desarrollo de software, la definición de estándares y la importancia del modelado de software. Esta etapa se inicia con el reconocimiento a principios de la pasada década del paradigma orientado a objetos (OO) como el más adecuado para producir software de calidad, esto es, extensible y reutilizable. Este paradigma encuentra sus raíces en la mencionada idea de Dahl, expuesta en un trabajo que describía los conceptos básicos subyacentes al primer lenguaje OO, Simula [4]. Después vinieron otros lenguajes OO como Smalltalk, Eiffel, C++ y Java, y a partir de ellos la eclosión de la denominada tecnología de objetos.

Si nos fijamos en cómo ha ido cambiando la forma de introducir a los alumnos en la programación, notamos la influencia de las visiones que iban emergiendo en el ámbito de la investigación y la práctica del desarrollo de software, de modo que también podemos distinguir tres etapas. Una etapa inicial en la que la enseñanza se limitaba a describir los elementos de un lenguaje de programación concreto junto con un conjunto de ejemplos que ilustraban su sintaxis y semántica. Con la programación estructurada arrancó una nueva etapa caracterizada por el reconocimiento de Pascal como lenguaje más adecuado para enseñar a programar y el uso del refinamiento por pasos sucesivos cómo principal técnica de diseño de programas. A finales de los ochenta este esquema era utilizado en la mayoría de centros universitarios de todo el mundo. Nos referiremos a esta forma de enseñar la programación como *enfoque no-formal*.

Dos principios básicos de la programación estructurada eran la exigencia de que el flujo de ejecución del programa coincidiese con la secuencia de instrucciones del código, como forma de mejorar la comprensión de los programas, y la creación de los programas aplicando el refinamiento por pasos sucesivos, como medio de dominar la complejidad. Pero también propugnaba algo más profundo que pasó inadvertido para muchos: el *cálculo de programas*, esto es, la corrección debe guiar el proceso de creación del programa a través de una derivación formal a partir de la especificación. En 1989, Dijkstra criticó duramente el estado de la enseñanza de la programación, ya que en su opinión se estaban formando programadores con una visión de la programación más cercana al arte que al de una disciplina sujeta a principios que

permitiesen ejercerla con rigor. En su libro *"A Method of Programming"* [5], Dijkstra presentó un método formal de enseñanza basado en dos principios básicos: "*enseñar un lenguaje imperativo claro y sencillo, cuya semántica es definida por reglas de prueba, y que la principal tarea del estudiante no es escribir programas, sino dar una prueba formal de que el programa cumple la especificación"*. Esta propuesta inició un intenso debate sobre el grado de formalismo en un primer curso de programación. Van Amstel [2] describió muy bien los tres enfoques que era posible aplicar:

- **No-formal**: Basado en mostrar ejemplos escritos en Pascal; apropiado para una formación en otra disciplina.
- **Semiformal**: Se utiliza el concepto de invariante para deducir el algoritmo, pero no se aplica una derivación formal; apropiado para estudiantes de ingeniería en informática.
- **Formal**: Se aplica el planteamiento de Dijkstra; apropiado para estudiantes de matemáticas e informática (visión más teórica).

Por otra parte, inspirados en el clásico libro de Polya, "*How to solve it?",* a principios de los ochenta surgieron algunas propuestas que consideraban la programación como una tarea de resolución de problemas. De todas ellas, la más elaborada fue la descrita por Dromey en su libro "*How to solve it by computer"* [7], en el que primero enseña un conjunto de heurísticas y técnicas de programación y luego las aplica a un conjunto de problemas clasificados en varias categorías.

Otro trabajo importante de la década de los ochenta fue el realizado por el grupo Anna Gram, creado por la AFCET (Asociación Francesa de Informática) para identificar los principales razonamientos utilizados en la construcción de programas [3]. En 1989, Scholl y Peyrin, miembros de Anna Gram, publicaron el libro *"Esquemas Algorítmicos Fundamentales"* [14] en que proponen un enfoque para la introducción a la programación basado en su experiencia en el grupo, y que será la base de nuestra propuesta.

El éxito del paradigma OO en la pasada década ha provocado, cómo es lógico, que surjan propuestas para introducir la programación a través de los conceptos OO. En [11] se clasifican estas propuestas en tres enfoques: *débil*, *híbrido* y *fuerte*, según el grado en que se aplican los conceptos OO. En [13], Meyer ilustra cómo aplicar el enfoque fuerte, con una enseñanza en la que los principios de la OO se combinan con la utilización de la técnica del *Diseño Por Contrato* [12] para formar a los alumnos en la construcción guiada por la corrección. Esta propuesta de Meyer podríamos catalogarla como un *"enfoque OO puro semiformal"* Sin duda este texto ejercerá mucha influencia en los próximos años.

En la actualidad, existe un debate sobre la conveniencia de aplicar un enfoque OO, en cualquiera de sus alternativas. Con esta discusión hemos entrado en una nueva etapa en la evolución de la enseñanza de la programación. Si la segunda estaba caracterizada por la programación estructurada, ahora la orientación a objetos se ha erigido como una alternativa. Cabe destacar que la última propuesta curricular de ACM/IEEE no se ha pronunciado sobre el debate *Orientación a Objetos vs. Programación Estructurada*, sino que entiende que ambos enfoques pueden ser adecuados, incluso comenzar con el paradigma funcional.

¿Qué ha pasado con la discusión sobre el enfoque formal? El debate parece cerrado con la derrota de este enfoque. En nuestro país no conozco ninguna universidad en la que los alumnos aprendan a programar siguiendo los postulados planteados por Dijkstra. En [8] aparece un análisis sobre los contenidos de IP en 33 titulaciones de 25 universidades españolas, referidos al curso 2001/02. Se observa que en la mayoría de universidades se sigue un enfoque no-formal. Sólo en la Universidad Politécnica de Cataluña y en la nuestra se sigue un enfoque semiformal. Esto también coincide con mi experiencia como miembro de tribunales de oposiciones, en las que con frecuencia el perfil de la plaza corresponde a una introducción a la programación.

En [8] también se puede observar que sólo un 12% de titulaciones arrancan con OO desde el principio y que un 30% comienzan con un lenguaje imperativo pero en el segundo cuatrimestre introducen un lenguaje OO (en ocasiones C++ sirve como lenguaje imperativo y OO). Es de destacar que Pascal (o Modula-2) se sigue empleando en 12 de las 25 universidades y ADA en 6.

3. Objetivos de un primer curso de introducción a la programación

El diseño de una asignatura de IP debe suponer que los alumnos llegan sin ningún conocimiento sobre programación, aunque muchos de ellos han adquirido algunas nociones en el bachillerato o de forma autodidacta, o bien porque han estudiado una especialidad de informática en formación profesional, en el caso de las titulaciones técnicas.

En [9] se establece que el objetivo básico de un primer curso de programación es: "*proporcionar a alumnos sin ninguna experiencia en programación los mecanismos necesarios para enfrentarse a la creación de programas para problemas pequeños, en el marco de expresividad de un paradigma de programación, enseñándoles un lenguaje de programación y los conceptos, métodos y técnicas que les permitan abordar los problemas, llegando de un modo riguroso desde la especificación al programa correcto"*.

En cuanto a IP, este objetivo hay que acotarlo a que el alumno sea capaz de escribir sus primeros algoritmos a través de las composiciones secuencial, condicional e iterativa utilizando los tipos de datos y estructuras de datos básicas. Será en la segunda parte (FP) en la que se produzca el desplazamiento desde los algoritmos a los programas pequeños.

Las cuestiones que hay que resolver son la elección del paradigma y del lenguaje, la selección de los conceptos, métodos y técnicas, y encontrar el modo de inculcar al alumno el razonamiento riguroso y la preocupación por la corrección.

En cuanto a la elección del paradigma propugnamos la conveniencia del paradigma imperativo estructurado frente a la tendencia creciente de introducir un lenguaje OO, por las razones expuestas en [11]. El lenguaje podría ser Pascal, Modula o Ada, por ser lenguajes que reflejan bien el paradigma. Creo que Ada es la elección acertada por disponer de genericidad, mecanismo que facilitará en FP la definición de estructuras de datos genéricas. Para este fin, algunos profesores defienden el uso de Java, utilizando declaraciones de tipo Object. En [11] se discuten los inconvenientes de esta decisión. Otros profesores propugnan el empleo de C con la intención de pasar a C++ en el segundo cuatrimestre, y así disponer de clases y genericidad. Creo que esta decisión es adecuada para unos estudios en los que sólo hay una asignatura de programación, como es el caso del resto de ingenierías, pero no es conveniente en el caso de informática, ya que C no es un lenguaje claro y sencillo que favorezca la legibilidad y la aplicación de los principios de la programación estructurada.

Una introducción a la programación puede realizarse con diferentes grados de formalismo, tal y como hemos señalado en el apartado anterior al distinguir los enfoques formal, semiformal y no-formal. En [6] varios expertos en programación comentaron la propuesta de Dijkstra de un enfoque formal [5] y aunque reconocieron el interés de los métodos formales, argumentaron una serie de objeciones al planteamiento: i) la dificultad de obtener, escribir y manipular las especificaciones formales de los problemas, ii) la propuesta supone un alejamiento de la práctica real de desarrollo de software, ya que se olvidan habilidades y abstracciones muy importantes, y los alumnos no ejecutan sus programas, y iii) los métodos formales no son el único medio de introducir el rigor, como señala Winograd, *"no hay duda de que la enseñanza debe introducir a los estudiantes en el pensamiento riguroso, pero de ahí a encumbrar a la manipulación de las abstracciones formales como objetivo primordial media un abismo"*. No obstante, se reconoce la utilidad de saber escribir especificaciones aplicando la teoría de conjuntos y la lógica de predicados, y que precondiciones, postcondiciones e invariantes son conceptos muy valiosos para los programadores.

Estas críticas ponen de manifiesto el interés de un enfoque semiformal que tenga en cuenta algunos conceptos básicos del enfoque formal, como son el invariante de bucle y las especificaciones, pero sin exigir al alumno la derivación formal de los programas. Totalmente de acuerdo con esta visión, considero que un enfoque semiformal, al estilo de EAF, es el camino acertado, ya que permite introducir un razonamiento riguroso para el diseño de algoritmos, al tiempo que supone una aproximación más cercana a la experiencia real del desarrollo de software.

Los conceptos, técnicas y métodos que habría que enseñar en IP serían: escritura de especificaciones (pre y postcondiciones, e

invariante); distinguir entre especificación e implementación de una operación; razonamiento inductivo para diseñar algoritmos iterativos; el concepto de abstracción y en particular la abstracción operacional; el concepto de esquema algorítmico y conocer esquemas básicos de recorrido y búsqueda; el refinamiento por pasos sucesivos como medio de descomposición de programas; conocer los tipos de datos básicos, que se presentarían como un TAD (dominio de valores y conjunto de operaciones), y los tipos estructurados array, tupla y secuencia; conocer de un modo intuitivo el problema de la eficiencia.

Este conjunto de herramientas conceptuales, combinado con un método de razonamiento riguroso para crear los algoritmos constituye la base del enfoque semiformal propuesto.

4. El enfoque de EAF

La principal idea del enfoque de Scholl y Peyrin es la utilización del concepto de invariante para el diseño de algoritmos iterativos sobre secuencias. El invariante se define mediante funciones recurrentes. En EAF se discuten en detalle un buen número de algoritmos que se escriben en una notación algorítmica estructurada muy cercana a Pascal, pero con propiedades interesantes como son que: i) incluye el *análisis de casos* como única composición alternativa, ii) dispone de una composición iterativa con punto de salida intermedio, y iii) incluye el tipo de dato "*secuencia*".

Un aspecto muy interesante es que las acciones (procedimientos y funciones) se introducen al principio, como parte del léxico de un algoritmo junto con las variables. Se muestra la necesidad de la parametrización y la influencia positiva de las acciones sobre la corrección, legibilidad, estructuración y reutilización. Cada acción es especificada con su pre y postcondición.

Hay un buen tratamiento del análisis de casos como medio de descomposición algorítmica, que consiste en dividir un dominio de datos en varios subdominios especificados mediante expresiones lógicas mutuamente excluyentes; cada subdominio o "caso" lleva asociado la ejecución de un conjunto de acciones.

Incluye la definición de esquemas algorítmicos para los problemas clásicos de recorrido y búsqueda de secuencias. El diseño de estos algoritmos se realiza a partir de un análisis iterativo basado en el concepto de invariante, planteando un proceso que guía la construcción del algoritmo, aunque no se profundiza todo lo deseable. Primero las secuencias se representan como listas de acceso secuencial y posteriormente mediante tablas.

Introduce la técnica de las *secuencias intermedias* para el manejo de la complejidad en los problemas sobre secuencias y para la organización de los programas: sobre una secuencia elemental S_0 formada por elementos de tipo T_0 (por ejemplo una secuencia de caracteres) se define una secuencia más abstracta S_1 cuyos elementos son de tipo T_1 (por ejemplo, una secuencia de longitudes de palabras en vez de la secuencia de caracteres original). Este proceso de abstracción se repite hasta obtener una secuencia intermedia adecuada para expresar la solución.

Los algoritmos sobre tablas se tratan después de los algoritmos sobre secuencias, de modo que el alumno es capaz de valorar la potencia del acceso indexado frente al acceso secuencial.

Finalmente se utilizan problemas numéricos de sucesiones y series para generalizar el estudio previo de algoritmos iterativos que manejaban secuencias de elementos explícitos, almacenados en una estructura de datos secuencia o tabla, a iteraciones en las que la secuencia se construye paso a paso, por recurrencia o por cálculo.

4.1. Análisis iterativo

Para el caso de un algoritmo de recorrido de una secuencia el proceso propuesto en EAF para construir el algoritmo iterativo sería:

1. Expresar la postcondición, post, mediante una serie de igualdades del tipo $v_i = f_i(S)$, de modo que

$$\text{post} = \{ v_i = f_i(S) \} \qquad \forall i,\ 1 \le i \le n$$

donde v_i es una variable que describe el resultado del programa, n es el número de variables, S representa la secuencia a recorrer y f_i es una función recurrente que aplicada sobre S nos devuelve el valor que tendrá al final de la ejecución la correspondiente variable v_i. Tendremos una variable v_i por cada una de las magnitudes que queremos calcular.

2. Establecer el invariante sustituyendo S por P_{iz} en la postcondición

$$INV = \{v_1 = f_1 (P_{iz}),..., v_n = f_n (P_{iz})\}$$

En cada instante del proceso iterativo se cumple $S= P_{iz} \bullet P_{dr}$, siendo P_{iz} la parte de la secuencia ya tratada, P_{dr} la parte que queda por tratar y el símbolo $\bullet$ denota la concatenación. El invariante expresa la relación existente entre la parte ya recorrida y las variables implicadas en la iteración. El primer elemento de P_{dr} es el siguiente elemento a tratar.

3. Definir cada función f_i por inducción

$$f_i ([\,]) = valorInicial$$
$$f_i (S \bullet e) = g (f_i (S), e), \qquad \forall i : 1 \le i \le n$$

donde g es la función de tratamiento y valorInicial es el valor de la función para la secuencia vacía.

4. Deducir la inicialización, la condición de finalización y el cuerpo del bucle a partir de las funciones recurrentes y de sus relaciones.

5. Un enfoque basado en EAF

Para alcanzar los objetivos expuestos en el apartado anterior, en el curso 91-92 planteé un enfoque semiformal inspirado en EAF. En mi opinión, el texto EAF tenía carencias importantes como no establecer la relación entre el invariante y el razonamiento inductivo o que el análisis iterativo sólo se aplica a unos pocos algoritmos, que además son muy sencillos. Además era un texto susceptible de mejoras en cuanto a claridad y organización. Creía necesario mejorar el enfoque y escribir un nuevo texto, a partir de las ideas subyacentes en EAF.

Organicé el curso siguiendo el orden de los primeros once capítulos de EAF, completado con dos capítulos finales, uno sobre lenguajes de programación y otro sobre tipos de datos. El curso constaba de los siguientes temas: *Máquinas que ejecutan algoritmos*; *Secuenciación y análisis de casos*; *Parametrización de acciones; Composición iterativa y concepto de secuencia; Esquemas de recorrido y búsqueda; Secuencias intermedias; Tratamiento secuencial de tablas; Sucesiones, recurrencia e iteración; Lenguajes de programación y Tipos de datos*. El penúltimo capítulo está destinado a definir con precisión qué es un lenguaje de programación y a introducir conceptos como la notación BNF y los programas traductores, también se aprovecha para establecer una correspondencia entre la notación algorítmica y el lenguaje Pascal. El último capítulo describe los tipos de datos de Pascal y su relación con los tipos vistos en la notación de EAF, describiendo cómo materializar secuencias de números o caracteres a través de la entrada por teclado o con ficheros secuenciales.

El texto [10], fruto del trabajo de todos los que hemos participado en la asignatura desde el curso 91/92, refleja cómo hemos adaptado EAF con un doble propósito: mostrar con claridad al estudiante el proceso de creación que hay detrás de cada algoritmo tratado en el curso y facilitar el aprendizaje de los conceptos, métodos, técnicas y esquemas que constituyen su caja de herramientas para construir algoritmos. Nuestras principales aportaciones en relación a EAF son:

1. Se presenta el razonamiento inductivo basado en el concepto de invariante como una técnica de diseño iterativo. En realidad se trata del razonamiento que hay detrás de las funciones recurrentes, pero en EAF este aspecto tan importante no es tratado. En nuestro caso, se explica con claridad y se aplica a un buen número de ejemplos no triviales y no incluidos en EAF.
2. Se especifican con mayor rigor todos los problemas, estableciendo la pre y postcondición.
3. Se describe con minuciosidad todo el razonamiento que hay detrás de cada algoritmo, analizando la diversas alternativas.
4. Las secuencias intermedias se aprovechan para describir la técnica del diseño descendente o refinamiento por pasos sucesivos, lo que supone un acercamiento al concepto de tipo abstracto de dato.
5. Se estudia la búsqueda binaria y se compara con la búsqueda secuencial, aprovechando esta comparación para mostrar la ventaja del acceso indexado e introducir el concepto de eficiencia de un algoritmo.

6. Se describen las tablas multidimensionales y se generalizan los esquemas algorítmicos de recorrido y búsqueda estudiados para tablas.
7. Se añaden los mencionados capítulos sobre lenguajes de programación y tipos de datos. El lenguaje utilizado (Pascal) aparece de forma natural como una notación ejecutable, y se establece una correspondencia entre la notación algorítmica y Pascal. Se introduce el concepto de TAD cuando se describen los tipos básicos simples y estructurados.
8. En el primer capítulo se presenta al ordenador como una máquina que procesa algoritmos. Se analiza la naturaleza de un algoritmo y se estudian con detalle los conceptos de *proceso* y *estado computacional*.

Es preciso destacar el importante papel que juega en nuestro enfoque el razonamiento inductivo. Para cada problema sobre una secuencia, partimos de un ejemplo de secuencia y aplicamos el razonamiento inductivo: "*cuál sería el resultado si la secuencia tuviese cero o un único elemento*" y *"supuesto que conozco la solución para el segmento de la secuencia a la izquierda del elemento i-ésimo, ¿qué operaciones debo aplicar sobre ese elemento para seguir conociendo la solución?"*. Las funciones recurrentes se escriben una vez se ha escrito el algoritmo, como una forma de formalizar el razonamiento inductivo realizado. El conocimiento de estas funciones es muy útil para introducir la recursividad, en el segundo cuatrimestre como un mecanismo del lenguaje que permite escribir directamente funciones recurrentes en vez de transformarlas en un algoritmo iterativo.

El primer año que apliqué este enfoque, influenciado por la corriente que exigía un visión más formal de la enseñanza de la programación, seguí el proceso de análisis iterativo planteado en EAF, y observé que la obtención primero de las funciones a partir de una especificación formal, dificultaba a los alumnos la comprensión del razonamiento aplicado. Comprendí que lo importante era el razonamiento inductivo en sí mismo y no escribir funciones recurrentes que pueden llegar a ser complicadas.

Los alumnos adquieren destreza en la aplicación del razonamiento inductivo y comprenden bien el concepto de invariante. Así, dado un invariante, los alumnos son capaces de escribir el algoritmo iterativo; por ejemplo, sin conocer ningún algoritmo de ordenación, pueden idear uno de ellos, como el de *selección* o *inserción*, dado el invariante. Por lo general, encontrar el invariante es una actividad difícil que requiere el mismo ingenio que encontrar la solución, pero el razonamiento inductivo es una herramienta conceptual muy útil para el diseño de algoritmos y que conlleva la identificación del invariante.

Estas ideas han guiado el desarrollo de las clases teóricas y el estilo de nuestro libro. Si analizamos [10] es evidente tanto la inspiración en el EAF, como que se trata de un libro diferente: mucho más claro y pedagógico, en el que se explican con rigor y detalle todos los algoritmos y que refleja todas las aportaciones mencionadas.

5.1 Clases Prácticas

Como es lógico, la parte práctica es fundamental en un curso de IP, ya que no es posible adquirir destrezas en programación a partir de los conocimientos teóricos y de la lectura de programas escritos por otros, sino que programar, como cualquier actividad constructiva, exige una práctica, en este caso que el alumno se enfrente a la creación de programas.

A alumnos sin una experiencia en programación, les resulta muy difícil el paso del enunciado del problema al programa. La programación les supone la entrada en un universo de ideas y pensamiento totalmente nuevo y es necesario proporcionarles de forma cuidadosa los elementos que le permitirán afianzar esas ideas y adquirir destreza en esa forma de razonar que permite crear algoritmos. Por ello, es necesario que primero les expliquemos la resolución de muchos problemas, para transmitirles el tipo de razonamiento que lleva del problema al esbozo de la solución. Luego, serán ellos los que deban enfrentarse de forma individual a ejercicios propuestos.

Hay que evitar crear programadores compulsivos, ansiosos por sentarse delante de la máquina para escribir directamente el código y establecer con el ordenador una batalla estúpida hasta obtener el programa que se supone calcula el resultado deseado, a través de un ciclo "escribo-ejecuto-modifico". Para inculcar el rigor en los

alumnos, se les debe exigir que expresen su solución en una notación algorítmica no ejecutable, y sobre ella analicen la corrección, como paso previo a escribir el programa.

Hay dos posibles estrategias: que el alumno conozca el lenguaje de programación desde el principio o introducirlo sobre la novena semana del curso. En ambos casos se debe obligar a los alumnos a que escriban primero la solución con la notación. La primera estrategia es necesaria en el caso de una asignatura cuatrimestral. Con una asignatura anual, como es nuestro caso, es más conveniente la segunda estrategia, ya que permite que durante los tres primeros meses los alumnos escriban los algoritmos con la notación de EAF, y luego, una vez conocen el lenguaje, traduzcan sus algoritmos y los ejecuten. Durante esos meses se realizan seminarios para la resolución de problemas propuestos, con grupos de treinta alumnos. En estos seminarios se busca la participación de los alumnos para construir el algoritmo paso a paso, se analizan alternativas y se justifican las decisiones. Los algoritmos se ejecutan utilizando una tabla que muestra la evolución de los valores de las variables a lo largo de la ejecución. De esta forma se pretende que los alumnos adquieran el hábito de pensar sobre la solución en vez de escribir código directamente, y que comprendan el papel de los lenguajes de programación frente a una notación.

6. Conclusiones

Se ha presentado un enfoque semiformal para un curso de IP, basado en el planteamiento expuesto en EAF. El interés del enfoque se ha justificado definiendo con antelación los objetivos del curso. Con las aportaciones que hemos realizado sobre EAF, creemos que se alcanzan todos los objetivos establecidos en el apartado 3. En los años que me encargué de la asignatura la valoración de los alumnos fue positiva y los resultados fueron buenos, así en el último año que la impartí, aprobaron un 70% de los alumnos matriculados, con exámenes que requerían un buen dominio de los conceptos y técnicas explicados en clase. Este trabajo complementa [11] y puede ser útil para quienes se enfrentan al diseño de un primer curso de programación. Tiene el interés añadido de presentar el enfoque EAF, que a pesar de aportar ideas muy valiosas, es poco conocido en nuestras universidades.

Este trabajo será complementado en un futuro con otro en el que se explicará, mediante varios ejemplos, cómo se aplica el razonamiento inductivo para crear algoritmos iterativos no triviales, y se describirá un proceso ideado a tal efecto; también se analizará como este enfoque facilita la comprensión de la recursividad.

Referencias

[1] ACM/IEEE, *Computing Curricula 2001*, March, 2000.

[2] J.J. van Amstel, *Teaching Programming at various levels of fomality*, IFIP, 1985.

[3] Anna Gram, *Raissoner pour programmer*, Dunod, 1986.

[4] O. Dahl, E. Dijkstra, C.A.R. Hoare, *Structured Programming*, Academic Press, 1972.

[5] E.W. Dijkstra y W.H.J. Feijen, *A Method of Programming*, Academic Service, 1984.

[6] E.W. Dijkstra, *On the cruelty of really teaching computing science*, en "*A debate on teaching computer science"*, P. Denning, Comm. ACM, vol. 32, num. 12, 1989.

[7] R. G. Dromey, *How solve it by computer?,* Prentice-Hall, 1982.

[8] J.L. Fernández Alemán, Proyecto Docente TEU, Universidad de Murcia, 2002.

[9] J. García Molina, Proyecto Docente CEU, Universidad de Murcia, 1991.

[10] J. García Molina et al., *Introducción a la Programación*, ICE, Universidad de Murcia, Ed. Diego Marín, 1999.

[11] J. García Molina, *¿Es conveniente la orientación a objetos en un primer curso de programación?*, JENUI'01, Palma de Mallorca, 2001.

[12] B. Meyer, *Construcción de Software Orientada a Objetos*, Prentice-Hall, 1997.

[13] B. Meyer, *Touch of Class*, www.inf.ethz.ch/~meyer/down/touch/

[14] P. Scholl y J. Peyrin, *Esquemas Algorítmicos Fundamentales*, Masson, 1989.