Enseñando a programar: un camino directo para desarrollar el pensamiento computacional

Teaching Programming: A Direct Way to Develop the Computational Thinking

Patricia Compañ-Rosique
Universidad de Alicante. España
patricia.company@ua.es

Rosana Satorre-Cuerda
Universidad de Alicante. España
rosana.satorre@ua.es

Faraón Llorens-Largo
Universidad de Alicante. España
faraon.llorens@ua.es

Rafael Molina-Carmona
Universidad de Alicante. España
rmolina@ua.es

Resumen

Está ampliamente aceptado que es fundamental desarrollar la habilidad de resolver problemas. El pensamiento computacional se basa en resolver problemas haciendo uso de conceptos fundamentales de la informática. Nada mejor para desarrollar la habilidad de resolver problemas usando conceptos informáticos que una asignatura de introducción a la programación. Este trabajo presenta nuestras reflexiones acerca de cómo iniciar a un estudiante en el campo de la programación de computadores. El trabajo no detalla los contenidos a impartir, sino que se centra en aspectos metodológicos, con la inclusión de experiencias y ejemplos concretos, a la vez que generales, extensibles a cualquier enseñanza de programación. En general, aunque se van desarrollado lenguajes cada vez más cercanos al lenguaje humano, la programación de ordenadores utilizando lenguajes formales no es una materia intuitiva y de fácil comprensión por parte de los estudiantes. A la persona que ya sabe programar le parece una tarea sencilla, pero al neófito no. Es más, dominar el arte de la programación es complejo. Por esta razón es indispensable utilizar todas las técnicas y herramientas posibles que faciliten dicha labor.

Palabras clave

Programación, resolución de problemas, aprendizaje.

Abstract

It is widely accepted that developing the ability to solve problems is essential. Computational thinking is based on problem solving using basic concepts of computing. An introductory course to programming is a direct way to develop the ability to solve problems using computer concepts. This paper presents our thinking about initiating students into the field of computer programming. This work does not detail the contents to be taught, but focuses on methodological aspects, including experiences and specific examples, which are general and extensible to any programming course. Although programming languages are been developed to be increasingly closer to human language, computer programming using formal languages is not intuitive and easy to be understood by our students. It may seem a

simple task for an experienced programmer, but it is not for a neophyte. Moreover, mastering the art of programming is complex. For this reason it is essential to use all possible techniques and tools that facilitate this work.

Keywords
Programming, problem solving, learning.

Introducción

La enseñanza/aprendizaje es un proceso bipolar, donde en un extremo se encuentra la enseñanza, cuyo protagonista principal es el profesor y en el otro el aprendizaje, cuyo protagonista principal es el alumno. Aunque está claro que ambos términos no son lo mismo, son dos caras de una misma moneda y por tanto indisolubles. De ahí que prefiramos hablar de enseñanza+aprendizaje (con signo más, en positivo, con carácter de integración) ya que se complementan y realimentan (Llorens, 2009).

Todo profesor se plantea como objetivo que sus estudiantes aprendan, y para ello son diversas las metodologías próximas a su asignatura que año tras año pone en práctica. Todo ello con una única y complicada finalidad, conseguir que sus estudiantes adquieran los conocimientos, habilidades, destrezas, …, que dicha materia conlleva.

El profesor universitario, es seleccionado y se caracteriza por ser un experto en su área. Pero ello no necesariamente conlleva que sepa cómo enseñar (López, 2009). Si buscamos el término en un diccionario, un docente es una persona que enseña (Miró, 2008), no obstante, para nosotros un docente es aquella persona que además trata de aumentar sus conocimientos no sólo sobre la materia que imparte, en este caso programación de ordenadores, si no sobre la forma de impartir esa materia para que los estudiantes puedan asimilar más y mejor. Es realmente importante estar motivado por la educación, cuestionándose curso tras curso las metodologías utilizadas y reflexionando acerca de cómo mejorar la docencia, siguiendo un ciclo de mejora continua.

La programación de ordenadores es una materia básica en cualquier curriculum de Informática. Además, se trata de una asignatura con la que muchos estudiantes no han tenido contacto previo y por tanto les causa bastante preocupación (Miliszewska & Tan, 2007). A la hora de impartir una asignatura de programación de computadores y mucho más si es la primera con la que el estudiante se encuentra en su titulación, las reflexiones se centran en: ¿cómo se enseña?, ¿por dónde se empieza?, ¿qué paradigma se debe utilizar?, ¿qué lenguaje de programación se debe emplear?, ¿cómo se deben orientar los ejercicios?, y lo peor ¿cómo se imparten las clases teóricas? Acerca del paradigma de programación con el cual empezar se ha debatido reiteradamente en distintos foros educativos (García, 2001), (Gómez et al., 2001), (Wiedenbeck, 1999), (Van et al, 2003), (Robins et al., 2003). No vamos a entrar en ello, pues independientemente del paradigma en el que se basen, los estudiantes tendrán que escribir algoritmos que resuelvan determinados problemas. También se han realizado estudios acerca de los conceptos y tópicos de programación que a los estudiantes les cuesta más comprender (Milne & Rowe, 2002). Para las preguntas anteriores se encuentran respuestas que en ocasiones sirven, soluciones que funcionan muy bien con algunos estudiantes y no tan bien con otros. Así que año tras año el profesorado se vuelve a plantear esas mismas preguntas y vuelve a encontrar otras respuestas, y de nuevo, vuelta a empezar, aunque no del todo, pues la experiencia es un grado.

En el informe sobre educación presentado a la UNESCO en 1996 y coordinado por Jacques Delors, se indica que los cuatro pilares en los que se basa la educación son:

- Aprender a conocer, actividad más tradicional de la enseñanza a través de la transmisión de conocimientos del profesor al alumno, aunque complementada con nuevos aspectos.
- Aprender a hacer, visión práctica de la misma, mediante la capacitación del estudiante para enfrentarse a determinadas tareas.
- Aprender a vivir juntos, desarrollando la comprensión del otro y los valores del pluralismo y la percepción de las formas de interdependencia sin renunciar a las propias ideas.
- Aprender a ser, supone el desarrollo de la personalidad, de la autonomía personal, del juicio y de la responsabilidad.

Estos pilares se pueden resumir en educación inteligente: una posible formación en el uso de las herramientas de la comunicación haría que entre el educador y el educando se genere el verdadero saber según lo definía Ortega y Gasset (2002), maneras de conocer, de comportarse y de orientarse o como él mismo lo resumía, el saber a qué atenerse en la vida. El pensamiento computacional es hoy en día un tema de mucha actualidad (Llorens, 2015) y tal como dice Jeannette Wing (2006) es una habilidad fundamental para todo el mundo, no solo para los ingenieros en informática. Desde nuestro punto de vista, nada mejor para un estudiante de Informática que una asignatura de introducción a la programación para desarrollar esta habilidad. Es importante incidir en que la forma de impartir la asignatura, repercutirá drásticamente en el desarrollo del pensamiento computacional en los estudiantes.

El profesor de cualquier área se debe enfrentar muchas veces, al desconocimiento, por parte del estudiante, de la materia que imparte. Ello complica todavía más el proceso de aprendizaje, al carecer, el estudiante, de un hábito adecuado de estudio y de aprendizaje. En esta tarea se encuentra como cualquier otro docente el profesor de una asignatura introductoria a la programación de computadores. El profesor se tropieza con una serie de dificultades entre las que se puede citar en primer lugar el tiempo: el profesor no dispone del tiempo necesario para transmitir todo lo que considera que el estudiante debe saber. Otra dificultad importante a tener en cuenta es que se trata de una materia que generalmente los estudiantes no han cursado con anterioridad en las enseñanzas previas obligatorias y por tanto, desconocen la mecánica para su aprendizaje. Y añadido a todo ello, el hecho de que se trata de una asignatura que requiere que los problemas se aborden desde un pensamiento computacional, donde las soluciones deben ser representadas como secuencias de instrucciones y algoritmos.

Es decir, que la esencia del pensamiento computacional es pensar como lo haría un científico informático cuando nos enfrentamos a un problema.

La enseñanza, y su dual el aprendizaje, atraviesa distintas fases que no hay que obviar. Es más, hay que aprovechar cada una de ellas para conseguir el resultado final, el aprendizaje por parte del estudiante. Estas fases o etapas se han concretado en:

- Oír: Asistir a la explicación que el maestro hace de una facultad para aprenderla.
- Ver: Percibir algo con cualquier sentido o con la inteligencia.
- Hacer: Ejecutar, poner por obra una acción o trabajo

Las descripciones citadas anteriormente se pueden encontrar en cualquier diccionario. Sin embargo, la manera de entender estas fases, a juicio de los autores, presenta una serie de matices. Estas singularidades junto con las ideas de cómo entender la programación ya publicadas en Satorre, Compañ y Llorens (2004) han servido de base para el desarrollo de este trabajo. A continuación se explican estas particularidades en los siguientes apartados.

Oír *versus* Escuchar

Una de las fases del proceso de enseñanza requiere de la impartición de clases magistrales. Una de sus posibles acepciones sería Instrucción o conjunto de los conocimientos teóricos o prácticos que da a los discípulos el maestro de una ciencia, arte, oficio o habilidad.

La lección magistral sigue siendo el método más usado en didáctica universitaria a pesar de las críticas que ha recibido: reducir las fuentes de información a la palabra del profesor, favorecer la pasividad del alumno, uniformidad en el ritmo de aprendizaje, etc. Por otra parte presenta algunas ventajas: proporciona información de forma rápida y económica, da seguridad al alumno, facilita la comprensión de temas complejos, sintetiza diversas fuentes de información, etc.

El desarrollo de una buena lección consta de cuatro fases:

1. Preparación y diseño: formulación de objetivos, organización de los contenidos, preparación de actividades para el alumnado. Esta fase sería previa a la actuación en el aula.
2. Introducción: ganar la atención de la audiencia, establecer relaciones con el grupo, despertar el interés, motivar hacia la tarea, presentación de objetivos, resumen general introductorio, etc.
3. Cuerpo: estructuración del contenido, mantenimiento de la atención y del interés, velocidad y ritmo adecuados, expresividad, etc.
4. Conclusión: el objetivo es intensificar la retención, énfasis en las ideas principales, preguntas, resumen, etc.

Existe una cierta tendencia, debido entre otros factores a la escasez de tiempo, a centrarse en la fase 3, es decir, en el contenido y no dedicar el tiempo necesario a las otras fases. Es fundamental que el estudiante esté motivado. Se puede distinguir dos tipos de motivación: motivación interna (desea aprender porque le gusta) y motivación externa (estudia por obligación) (Boffill & Miro, 2007). No podemos dejar el aprendizaje exclusivamente en manos de la motivación externa (la nota y el título). Es imprescindible que haya un mínimo de motivación interna para que haya éxito en el aprendizaje. Tal y como argumenta Jenkins (2001), si los estudiantes no están motivados, no aprenderán.

Aunque el profesor de una materia siga exactamente las cuatro fases que conlleva el desarrollo de una lección esto no implica la correcta asimilación por parte del estudiante de los conocimientos tratados, ya que, como reza el apartado "oír no significa escuchar". Siguiendo las enseñanzas del gran filósofo romano Lucio Séneca: "Largo es el camino de la enseñanza por medio de teorías; breve y eficaz por medio de ejemplos". Esto nos lleva a pensar en clases más participativas, con numerosos ejercicios, con "trucos", con comentarios sobre costumbres y hábitos de programación, con

comparaciones con situaciones reales con las que están habituados y conocen bien, nada que ver con la teoría de libro. Al igual que hace Dunican (2002), tratamos de buscar analogías entre conceptos de programación con ejemplos de la vida real.

Un buen método para que reflexionen sobre cualquier estructura o tipo de datos, es colocar frente a ellos la sintaxis de dicha estructura y esperar que la interpreten, que intenten entender cómo funciona y para qué, para posteriormente explicar cómo y dónde se utiliza.

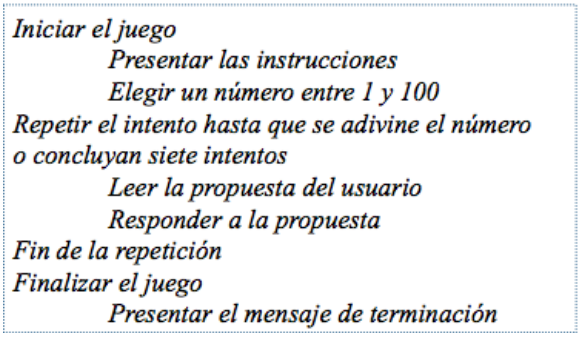

Figura 1. Ejemplo en lenguaje algorítmico

Un ejemplo de esta metodología es mostrarles un fragmento en un lenguaje algorítmico cercano al lenguaje natural o en cualquier otro lenguaje y preguntarles ¿Qué hace? Por regla general, los estudiantes responden explicando el cómo, es decir, diciendo con otras palabras lo mismo que se les presenta en ese fragmento. En el caso de la figura 1, la respuesta que nos gustaría oír es: "adivinar un número de 1 a 100 teniendo siete oportunidades", sin embargo la respuesta habitual es "se pide un número, se recoge, se compara con el que es, si no coincide se pide otro, … y así sucesivamente". Deben organizar y analizar lógicamente la información, no leer literalmente lo que ven.

Otro método puede ser el de plantear un ejercicio en el que seguro necesitarán una estructura o tipo de datos que no han visto todavía y ver cómo resuelven esa carencia. Por ejemplo, tras ver únicamente tipos de datos simples se les pide un ejercicio en el que almacenen las edades de 100 individuos y calculen la edad media. Sin casi pensar, empiezan a escribir y lanzan la siguiente pregunta: ¿se pueden poner puntos suspensivos? Ante la negativa al uso de puntos suspensivos, etc., se inicia un debate para encontrar una solución y ello conlleva a la introducción del concepto de tipos de datos estructurados. El hecho de explicar cualquier concepto, estructura, etc., sin cuestionarse el porqué, les hace creer en ello como si de un acto de fe se tratase, porque lo dice el profesor y es así. Por contra, el obligarles a pensar, a resolver un problema que les lleve a un nuevo concepto les ayuda a entender el porqué es así y cuales son sus ventajas y aplicaciones. Se trata de emplear la relevancia o aplicabilidad de algo como técnica pedagógica (Sheard & Hagan, 1997).

Si a todo lo comentado anteriormente, se une que se trata de una asignatura de introducción a la programación de computadores, surgen más dificultades: ¿cómo se explica teóricamente un algoritmo?, ¿mediante un resumen? Además, la elección del lenguaje de programación a utilizar es una de la decisiones importante a tomar cuando se va a iniciar al estudiante en el campo de la programación de computadores. Los

debates sobre esta cuestión son continuos, tanto entre el profesorado de dicha materia como entre profesorado de materias relacionadas, estudiantes, profesionales del sector, etc. Posiblemente, hay tantos puntos de vista y enfoques distintos como áreas de interés en la programación.

En lo que sí coincide el profesorado de la materia que se está tratando en este trabajo, es en que sea cual sea el lenguaje utilizado, el conocimiento del lenguaje no es el objetivo en sí, sino la herramienta para expresar el algoritmo.

Es obvio que el profesor está obligado a emplear un lenguaje de programación, pero lo que es fundamental es impedir que los estudiantes se centren en las particularidades del lenguaje, en sus defectos, en sus carencias, en sus bugs, etc., y desvíen la atención sobre lo importante. Esto es muy complicado, es una tarea muy difícil y que implica un determinado nivel de abstracción que los estudiantes no suelen tener. Normalmente ese nivel lo alcanza un programador cuando ya tiene cierta experiencia (Jenkins, 2002) .

En los ejercicios escritos que presenta nuestro alumnado para ser evaluado, ejercicios con su propuesta de solución a un determinado problema, intentamos que entiendan que el profesorado no actúa como un compilador, sino que mira el contenido, la forma de construir la solución, las estructuras empleadas, el flujo lógico de su propuesta, etc. Año tras año insistimos en ello para que se centren en lo importante, esto es, en la construcción de la solución y dejen de lado cuestiones sintácticas que no alteran el espíritu, el fondo de la solución, como por ejemplo, si detrás de un "*scanf*" deben colocar una instrucción que vacíe o no el buffer, o procedimientos similares y específicos de cada lenguaje de programación.

Con todos estos ejercicios queremos que aprendan a razonar, a pensar computacionalmente, automatizando soluciones mediante el pensamiento algorítmico, esto es, estableciendo una serie de pasos ordenados para llegar a la solución.

Ver *versus* Observar

La visualización de programas se define como el uso de técnicas gráficas para ayudar a entender el funcionamiento de un programa de ordenador (D'Auriol et al., 2001). En el trabajo de Lemieux y Salois (2006) se hace un estudio bastante detallado acerca de las distintas categorías en las que se pueden incluir los diferentes tipos de visualización de software. Jeliot (Ben-Ari, 2001), (Moreno et al., 2004), o SRec (Pérez & Velázquez, 2009) son algunas de las propuestas existentes. La utilización de algún sistema de visualización de programas puede ayudar al alumnado a mejorar su comprensión acerca de la ejecución de un programa. De todas formas no hay que pensar que la utilización de este tipo de sistemas va a ser la solución a todos los problemas y que el aprendizaje será más eficaz.

Normalmente se entiende que una representación gráfica de un programa, y más si es con animaciones, siempre es mejor que el código fuente escrito en texto. Se dice que una imagen vale más que mil palabras, pero no siempre es así. Depende de los estilos de aprendizaje (Debdi et al., 2014) y de la percepción personal. Por ejemplo, supongamos que lees un libro que te gusta mucho, y un tiempo después estrenan una película en el cine basada en dicho libro. Vas a verla y sufres la gran decepción, ¿qué ha sucedido? Normalmente el problema ha sido que la película no transmitía todas las descripciones tal y como lo hacía el libro y que tu recreaste en tu imaginación.

Es importante, si se usa algún tipo de sistema de este tipo hacer una explicación teórica precisa de lo que dicha visualización pretende mostrar ya que las animaciones sólo son útiles con explicaciones personales que las acompañen.

Ya que hay que enseñar a "ver" la visualización tanto como a trabajar con ella, se requiere de un tiempo extra.

¿Es lo mismo ver que observar? Si se busca la definición de la palabra ver en un diccionario, aparece la palabra observar como si ambos términos fueran equivalentes. No se puede negar que ambos hacen referencia a percibir. Sin embargo, para nosotros hay una diferencia fundamental entre ambas palabras. Ver es una actitud pasiva, no hay que hacer ningún esfuerzo adicional aparte de mantener los ojos abiertos. Observar es más bien una actividad, hay que esforzarse en asimilar lo que se está viendo. Por tanto, si el sistema de visualización de programas sirve para observar, entonces es muy adecuada su utilización durante el proceso de aprendizaje.

Dado que el uso de sistemas de visualización se considera que es más eficaz para estudiantes con poco conocimiento de la materia, parece adecuado introducir este tipo de herramienta en una asignatura de introducción a la programación.

Deben aprender a ver a través de abstracciones, como los modelos y las simulaciones.

Hacer equivale a practicar

Practicar quiere decir poner en práctica algo que se ha aprendido.

Hay un antiguo proverbio chino que dice "*Escucho y olvido. Veo y recuerdo. Hago y comprendo."* Este sabio consejo se puede ampliar más. Tal y como dice el gran filósofo Séneca, *"Los hombres aprenden mientras enseñan".* La mejor manera de aprender una materia es teniendo que explicársela a otra persona. Cuando el alumno está diseñando un algoritmo, en realidad, le está enseñando a otro compañero "más torpe" (el ordenador, cuyas posibilidades de entendimiento son nulas) cómo realizar una tarea concreta, lo que implica la comprensión por parte del alumno de lo que intenta explicar, puesto que no se pueden transmitir conocimientos si no se posee el suficiente dominio de los mismos. Además hay que tener en cuenta todos los posibles casos y situaciones.

Es importante inculcar que la práctica hace maestros, que más vale dedicar dos horas a analizar un problema y resolverlo de manera incorrecta, que escuchar diez minutos al profesor y aceptar su perfecta solución al problema. Las soluciones finales aparecen claras y ordenadas y parecen sencillas. Pero el proceso de razonamiento y de búsqueda de la solución nunca es lineal. Se avanza y se retrocede, nos equivocamos y rehacemos el camino, en un complejo recorrido por el laberinto que nos ha llevado a la solución final. Únicamente cuando hemos encontrado una solución a nuestro problema, podemos volver la vista atrás y pasar a limpio la solución.

Uno de los objetivos a conseguir es intentar que sean capaces de estructurar la resolución del problema. Al contrario de lo que ellos opinan, el principal objetivo de un profesor de una asignatura de introducción a la programación no es que los programas o algoritmos, en este caso, estén perfectamente escritos, es decir, ajustándose perfectamente a la sintaxis marcada por el lenguaje de programación empleado (se trata de una asignatura introductoria). El objetivo es que el estudiante consiga estructurar la solución del problema. No es tan importante al principio ver si una u otra variable tiene uno u otro valor, sino componer la solución, estructurar el modo de resolver el

problema. Lo relevante es identificar, analizar e implementar posibles soluciones con el objetivo de logar la combinación más eficiente y efectiva tanto en el número de pasos de programa como en los recursos empleados.

Una manera de cumplir este objetivo es la utilización de "diagramas de flujo mudos" o "diagramas de cajas mudos", con la finalidad de indicar de un modo gráfico y sin contenido la estructura lógica de la solución para resolver un determinado problema.

Esta idea se puede entender mediante el siguiente ejemplo: rellenar de números consecutivos y en espiral, empezando por el 1, una matriz de dimensión $n \times m$ empleando un diagrama de cajas mudo (figura 2).

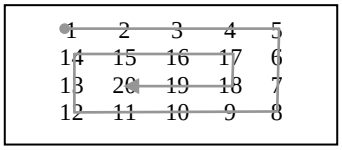

Figura 2. Matriz en espiral

En la solución se quiere reflejar que para resolver el problema hacen falta cuatro estructuras de repetición, una para cada sentido de la espiral (de izquierda a derecha, de arriba abajo, de derecha a izquierda y de abajo a arriba) dentro de otra estructura de repetición que permitirá repetir el ciclo hasta completar la matriz. La figura 3 muestra el diagrama de cajas mudo de esta solución.

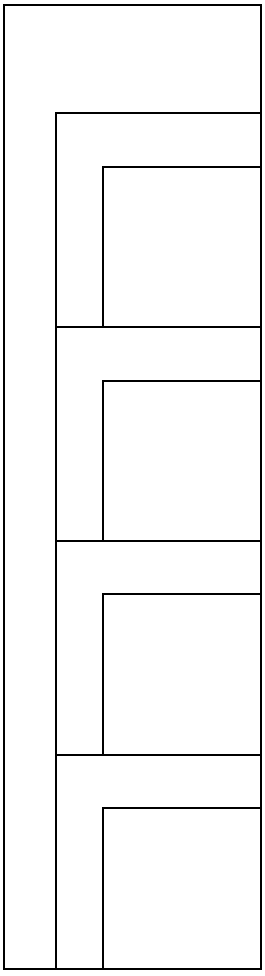

Figura 3. Diagrama de cajas mudo

Es obvio que la solución final en forma de programa estará llena de matices, pero si un estudiante es capaz de diseñar mentalmente la estructura anterior estará aplicando habilidades correspondientes a pensamiento computacional, estará diseñando un modelo para resolver un problema sin centrarse en detalles.

Sin embargo, la experiencia personal nos indica que el estudiante escribe primero la solución en un lenguaje de programación para posteriormente dibujar el diagrama mudo. Y a pesar de los esfuerzos no hay modo de modificar este mal hábito.

Al estudiante le cuesta ver el problema en su conjunto, de forma general, sin el detalle. Le dan más importancia a si algo termina en punto y coma o no, como si eso fuera lo trascendental. Para el estudiante esto tiene tanta importancia que los días previos al examen refleja su preocupación en preguntas del tipo: ¿cuánta nota se quita si no se pone ";" al final de una instrucción?, ¿descuenta mucho si en lugar de escribir la asignación con "←" la escribimos ":="?

Los estudiantes se esmeran en que las cosas funcionen, a cualquier precio, con estructuras incorrectas e incoherentes, con el número de parches que haga falta, pero que funcione, aunque falle en ocasiones, pues si esto ocurre incorporan un nuevo parche y a funcionar. Olvidan que lo que se pretende es que aprendan a generalizar este proceso de resolución de problemas para poder ser capaces de resolver una gran variedad de familias de problemas.

Para aumentar el atractivo de las prácticas, se les plantean ejercicios resueltos, ejercicios propuestos breves, juegos, ejercicios similares a casos concretos reales, etc. Continuando con las enseñanzas de los maestros, Platón decía: "*Ninguna enseñanza obligatoria puede permanecer en el alma … Al educar a los niños, enséñales como si fuera una especie de juego y podrás ver con más claridad las inclinaciones naturales de cada uno*".

El aprendizaje mediante juegos no es una técnica nueva, ya en 1938 Johan Huizinga (2000) en su libro "Homo Ludens" planteaba el juego como fenómeno cultural, concibiéndolo como una función humana tan esencial como la reflexión y el trabajo. A lo largo de los años se ha mostrado que es muy útil para motivar a los estudiantes mediante la creación, confección o utilización de juegos (Colomina et al., 2004).

Otra competencia que nos parece fundamental que desarrollen es el trabajo en grupo. Es por esta razón que realizamos un concurso de programación en el que los estudiantes forman grupos y desarrollan un proyecto de programación. Se establecen premios consistentes en aumentos en la puntuación para los equipos ganadores. Aprovechando que los ejercicios basados en juegos les motivan, proponemos ejercicios de este tipo. Por ejemplo, este último curso hemos propuesto tres proyectos, cada uno enfocado a valorar unas competencias determinadas. El primer ejercicio que propusimos consistía en un juego de adivinación de números con el que pretendíamos conocer si comprendían las estructuras básicas de programación, bucles, condicionales, así como la programación modular. Para realizar el segundo ejercicio era necesario haber asimilado el manejo de *arrays*. Se trataba de un juego estilo buscaminas. Con respecto al último ejercicio, nuestro objetivo era que manejasen *arrays* y registros por lo que les propusimos la realización de un trivial.

Hemos de destacar que la experiencia fue todo un éxito puesto que los estudiantes se esforzaron en realizar sus juegos, incluso mejorándolos empleando características que no se habían explicado en el aula.

A pesar de que se trata de estudiantes universitarios, no podemos olvidar que acaban de iniciar su vida universitaria. En el instituto estaban sometidos a un cierto control, pero en la Universidad tienen mucha más libertad. Si no asisten a clase, no se informa a los padres, puesto que se trata de personas adultas, y por tanto son responsables de sus actos. Gracias a la experiencia de estar varios años impartiendo la materia, también hemos detectado que si no se les estimula de alguna forma a trabajar de forma diaria, estudiarán sólo cuando tengan algún tipo de control para evaluarlos. Las evaluaciones guían sus pasos y condicionan su forma de trabajar. Es fundamental seguir un tipo de evaluación continua mediante la realización de pruebas que vayan asegurando que los estudiantes asimilan los conceptos y que cuando hayan terminado la asignatura, hayan adquirido las competencias correspondientes. En la actualidad esto resulta mucho más sencillo que hace unos años gracias a las herramientas tecnológicas vía Web. Utilizando las funcionalidades del Campus Virtual[1] resulta extremadamente fácil incorporar tests así como controles dónde se puede supervisar el aprendizaje del estudiante de manera simple (Campus Virtual, 2015). Esto es útil para ambos actores involucrados en el proceso: el profesor puede averiguar si sus estudiantes van siguiendo las clases y el estudiante puede saber si está asimilando los conceptos. De esta forma, al hacerlo de manera continua, tanto el profesor como el estudiante pueden reaccionar a tiempo por si se detectase algún problema. Por ejemplo, si el 70% de los estudiantes no supera los tests, evidentemente hay algún problema en la forma de impartir la asignatura y el profesor tendrá que plantearse otra estrategia. La figura 4 muestra un ejemplo de una sencilla pregunta de test relativa a la compresión de las estructuras iterativas.

Dado el siguiente fragmento de código en C, el valor de la variable x después del bucle es …

```
x=0;
for (i=1; i<=3; i++)
   for (j=1; j<3; j++)
          x++;
```

a. 3
b. 6
c. 9
d. 1

Figura 4. Pregunta tipo test

Infundiendo buenos hábitos

Para intentar que comprendan que lo importante en un algoritmo no es el hecho de que funcione (esta característica debe ser aplicable a un programa), sino que tenga una estructura correcta, un paso de parámetros adecuado, un buen empleo de las variables, pues ello llevará al funcionamiento final, solemos escribir en la pizarra

"La bentana hesta havierta"

Al leer esta frase entienden que aunque se consiga el cometido, es decir, transmita el mensaje de que la ventana está abierta, la frase no es ortográficamente correcta y si se tratase de una asignatura de lengua no se admitiría. Eso mismo pretendemos que hagan ellos con sus algoritmos, que construyan algoritmos correctos. Haciendo un símil, la

[1] Campus Virtual es un servicio de complemento a la docencia y a la gestión académica y administrativa, cuyo entorno es Internet y está dirigido tanto al profesorado como al alumnado y al personal de administración de la Universidad de Alicante.

gramática en lengua es como la estructura en un programa, tanto a nivel de algoritmo como a nivel de datos.

Como práctica, les recomendamos leer el artículo de Agustín Cernuda (1998) "Cómo no hacer una práctica de programación" y efectivamente, tal y como comenta el autor, les entusiasma y se ven reflejados en él, aunque no hemos podido constatar si repercute de manera positiva en sus ejercicios, pues la leen prácticamente al final del cuatrimestre, y por tanto al final de la asignatura.

Algunos estudiantes también tienen una costumbre curiosa que tratamos de erradicar, en unas ocasiones con más éxito que en otras. A pesar de que se les insiste en el que el sangrado en un programa es fundamental para entender la estructura del mismo, muchos estudiantes escriben el código sin sangrar (según ellos para ir más rápidos) y después (sobre todo si el profesorado se lo dice) lo sangran de manera más o menos correcta. No comprenden que si van aplicando el sangrado correcto, esto les ayuda en la propia labor de codificación haciendo el programa mucho más comprensible.

En este tipo de prácticas ellos descubren la importancia de cumplir las especificaciones, pero no antes de empezar a resolver el ejercicio, pues van directamente a escribir código y se olvidan de ello. Es después, cuando se construye el programa final y ven que no funciona, cuando aprenden que no seguir de manera rigurosa las instrucciones, les lleva a no poder enlazar su módulo con el resto. Y hay que tener en cuenta que los proyectos informáticos actuales son de tal complejidad que un único programador no lo puede hacer completamente y seguro que formarán parte de un equipo de desarrrollo.

La realidad es que cuando les pides a los estudiantes que lean con detalle los enunciados parece que les solicites algo extraordinario: se limitan a echarle un vistazo rápido, y corriendo a escribir código que es lo realmente importante a su modo de ver, sin análisis previo ni diseño ni nada.

Otro tema en el que procuramos insistir, no con mucho éxito la verdad, es la documentación. Nuestros estudiantes detestan escribir comentarios y no digamos elaborar la documentación externa, para ellos lo importante es el código. Sólo escriben la documentación al final, y eso porque se la exigimos como parte del ejercicio. Tratamos de inculcarles que si en un futuro trabajan en una empresa como programadores profesionales, será habitual que una persona tenga que ampliar o revisar el código que ha realizado otro programador, que a lo mejor ya ni siquiera pertenece a la empresa. Si el programa está bien documentado, le será más sencilla la tarea. Por bien documentado no entendemos sólo que tenga una buena explicación de lo que hacen los distintos módulos que forman el programa. Es importante poner comentarios en los "puntos oscuros" o conflictivos, entendiendo como tales aquellas líneas de programa que pasado un tiempo es fácil no acordarse de por qué las pusimos. Tampoco entienden que la documentación externa va ligada al programa, que si actualizan el fuente, deben actualizar la documentación, en caso contrario no sirve para nada.

Apoyándonos en la cita de Match Kapor "*los programadores trabajan en la forma en que los artesanos medievales construían catedrales: piedra por piedra*", se les insiste en que para aprender a programar, hay que hacerlo día a día, construyendo unos conceptos sobre otros. Esta materia, por sus características, requiere un esfuerzo desde el principio, paso a paso, de forma continuada y creciente, dónde los primeros

conocimientos son base de los que le siguen y estos a su vez son necesarios para los posteriores, y así sucesivamente. Por ejemplo, para entender el funcionamiento de un bucle, es fundamental saber diseñar estructuras condicionales.

Otra mala costumbre que nos parece adecuado combatir es el hecho de que sólo empleen el material que se les suministra en la asignatura, en nuestro caso las diapositivas y los enunciados de las prácticas. Hay que motivarles a investigar: buscando en libros, Internet, etc. Como ejemplo de esta técnica les pedimos que resuelvan algún de tipo de problema para el que el material proporcionado en la asignatura no es suficiente. Además, se les avisa que para resolver ese ejercicio tienen que investigar por su cuenta.

Conclusiones

Una asignatura de introducción a la programación de ordenadores es el vehículo perfecto para desarrollar habilidades de pensamiento computacional ya que implica la resolución de problemas haciendo uso de conceptos informáticos.

A nuestro entender, enseñar programación no consiste en enumerar una serie de estructuras de programación indicando para que sirve cada una de ellas. Es mucho más que eso, se trata de que el estudiante aprenda a pensar, a analizar una situación y a diseñar el método de resolución más adecuado, dejando al margen el lenguaje de programación. Se trata de un objetivo muy complejo. Para cualquier persona diseñar la solución a un problema requiere de un esfuerzo importante de abstracción, aún más si tiene que expresarla en forma de un algoritmo.

Es fundamental que los estudiantes comprendan que la habilidad de escribir programas correctos, eficientes, bien organizados y adecuadamente documentados es un requisito esencial para cualquier titulado en informática, porque programar es mucho más que conocer un determinado lenguaje. Hemos presentado en este trabajo nuestras reflexiones acerca de cómo iniciar con éxito a un estudiante en el campo de la programación de ordenadores**.** Aunque nuestra experiencia se basa en enseñar programación a futuros ingenieros en informática, la misma es válida para cursos de introducción a la programación tanto de niveles no universitarias como de universitarios de otras ramas.

Presentación del artículo: 27 de julio de 2015
Fecha de aprobación: 6 de septiembre de 2015
Fecha de publicación: 15 de septiembre de 2015

Compañ-Rosique, P. et al. (2015). Enseñando a programar: un camino directo para desarrollar el pensamiento computacional. *RED. Revista de Educación a Distancia. Número 46*. 15 de Septiembre de 2015. Consultado el (dd/mm/aa) en http://www.um.es/ead/red/46

Referencias

Ben-Ari, M. (2001). La visualización de programas en la teoría y en la práctica. *Novática* nº 150.

Bofill, P., y Miro J. (2007). Las fases del aprendizaje: Un esquema para el análisis y diseño de actividades de enseñanza/aprendizaje. - *XIII Jornadas de Enseñanza Universitaria de la Informática* (JENUI).

Campus Virtual http://si.ua.es/es/ite/servicios/campus-virtual.html. Fecha última visita junio 2015.

Colomina, O., Compañ, P., Satorre, R., Aznar, F., Suau, P., y Rizo, R. (2004). Aprendiendo mediante juegos: experiencia de una competición de juegos inteligentes. *X Jornadas de Enseñanza Universitaria de la Informática* (JENUI).

Cernuda del Río, A. (1998). Cómo NO hacer unas prácticas de programación. *VIII Jornadas de Enseñanza Universitaria de la Informática* (JENUI).

D´Auriol, B., Casas, C., Kumar, P., Draper, L., Esper, A., López, J. Molakaseema, R., Seelam S., Saenz, R., Wen, Q., y Yang, Z. (2001). Exploratory study of scientific visualization techniques for program visualization. En: V. N. Alexandrov et al. (Eds) ICCS 2001, LNCS 2074, pp 701-710. Springer-Verlag Berlin Heidelberg.

Debdi, O., Paredes-Velasco, M., y Velázquez-Iturbide, J.A. (2014). Relationship between Learning Styles, Motivation and Educational Efficiency in Students of Computer Science. En: J.L. Sierra-Rodríguez, J.M. Dodero-Beardo y D. Burgos (Eds) Proceedings from the *2014 International Symposium on Computers in Education (SIIE)*, pp 13-16. IEEE Xplore.

Dunican, E. (2002). Making the analogy: alternative delivery techniques for first year programming courses. En :J. Kuljis, L. Baldwin & R. Scoble (Eds). Proceedings from the 14th *Workshop of the Psychology of programming interest group*, Brunel University, 89-99.

Gal-Ezer, J., y Harel, D. (1998). What (Else) Should CS Educators Know? *Communications of the ACM*, vol. 41, nº 9.

García Molina, J. (2001). ¿Es conveniente la orientación a objetos en un primer curso de programación? *VII Jornadas de Enseñanza Universitaria de la Informática* (JENUI).

Gómez, A., González, J., y Lozano, A. (2001). Tres lenguajes distintos y un solo objeto verdadero: una propuesta de introducción a la programación. *VII Jornadas de Enseñanza Universitaria de la Informática* (JENUI).

Huizinga, J. (2000). *Homo Ludens*. Alianza Editorial. ISBN 978-84-206-3539-2.

Jenkins, T. (2001). The motivation of students of programming. *Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education* (ITiCSE).

Jenkins, T. (2002). On the difficulty of learning to program. *Proceedings of the 3rd Annual Conference of the LTSN Centre for Information and Computer Science*, 53-58.

Lemieux, F., y Salois, M. (2006). *Visualization techniques for program comprehension. A literature review*. Defence R & D Canada – Valcartier. Technical memorandum. DRDC Valcartier TM 2005-535.

Llorens, F. (2009). La tecnología como motor de la innovación educativa. Estrategia y política institucional de la Universidad de Alicante. *ARBOR Ciencia, Pensamiento y Cultura*, vol CLXXXV extra 2099, pág. 21-32.

Llorens, F. (2015). Dicen por ahí ... que la nueva alfabetización pasa por la programación. *ReVisión, Revista de investigación en Docencia Universitaria de la Informática*, vol. 8, nº 2.

López D. (2009). Investigar en educación: guía práctica. *XV Jornadas de Enseñanza Universitaria de la Informática* (JENUI).

Miliszewska, I., y Tan, G. (2007). Befriending computer programming: a proposed approach to teaching introductory programming. *Issues in Informing Science and Information Technology*, vol 4.

Milne, I., y Rowe, G. (2002). Difficulties in learning and teaching programming - views of students and tutors. *Education and Information Technologies*, vol. 7, nº 1, 55-56.

Miró, J. (2008). De la alquimia a la química. *ReVisión, Revista de investigación en Docencia Universitaria de la Informática*, vol 1 nº 2.

Moreno, A., Myller, N., Sutinen, E., y Ben-Ari M. (2004). Visualizing programs with Jeliot 3. *Proceedings of the Working Conference on Advanced Visual Interfaces*. ISBN:1-58113-867-9.

Ortega y Gasset J. (2002). *Misión de la Universidad y otros ensayos sobre educación y pedagogía*. Alianza editorial. ISBN: 9788420641225. Colección: OBRAS DE ORTEGA Y GASSET.

Pérez, A., y Velázquez, Á. (2009). Animación Automatizada de Técnicas de Diseño de Algoritmos. *XV Jornadas de Enseñanza Universitaria de la Informática* (JENUI).

Robins, A., Rountree, J., y Rountree, N. (2003). Learning and teaching programming: a review and discussion. *Computer Science Education*, vol. 13, nº 2, pp 137-172.

Satorre, R., Compañ, P., y Llorens, F. (2004). Un modo de entender la programación. *X Jornadas de Enseñanza Universitaria de la Informática* (JENUI).

Sheard, J., y Hagan, D. (1997). Experiences with teaching object-oriented concepts to introductory programming students using C++. *Technology of Object-Oriented Languages and Systems* (TOOLS-24), IEEE Technology, 310-319.

Van Roy, P., Armstrong, J., Flatt, M., y Magnusson, B. (2003). The role of language paradigms in teaching programming. *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*, 269-270.

Wiedenbeck, S. (1999). A comparison of the comprehension of object-oriented and procedural programs by novice programmers. *Interacting with Computers*, vol. 11, 255-282.

Wing, J.M. (2006). Computational Thinking. *Communications of the ACM*, vol. 49, nº 3.