Actas de las Jenui, vol. 2. 2017.
Páginas: 31-38

Explicando el bajo nivel de programación de los estudiantes

Francisco J. Gallego-Durán Rosana Satorre-Cuerda
Patricia Compañ-Rosique Carlos Villagrá-Arnedo
Departamento de Ciencia de la Computación e Inteligencia Artificial
Universidad de Alicante
03690 San Vicente del Raspeig (Alicante)
{fgallego,rosana,patricia,villagra}@dccia.ua.es

Resumen

En los últimos años hemos observado un deterioro de la calidad de los programas creados por estudiantes de cuarto curso. Cuando se les exige crear sistemas completos desde cero, su código muestra problemas de base: código espagueti, mala estructuración, repeticiones innecesarias, deficiente paso de parámetros, escasa comprensión del paradigma orientado a objetos, etc.

En este trabajo mostramos ejemplos de los problemas y analizamos sus causas. Deducimos que hay una responsabilidad en el proceso de enseñanza/aprendizaje aplicado. Construimos una explicación basada en las teorías psicológicas actuales sobre modelos mentales y conceptuales. Observamos que muchos detalles necesarios de bajo nivel son obviados o simplificados en la enseñanza de la programación. Con todo, formulamos una hipótesis: los estudiantes están modelizando erróneamente los conceptos base de bajo nivel.

Para enfrenar a los estudiantes con el problema y concienciarlos, organizamos el #CPCRetroDev: un concurso de programación de videojuegos para Amstrad CPC. Al disponer de sólo 4Mhz y 64KB de RAM, se ven forzados a mejorar el código para aprovechar los recursos. Las evidencias muestran que conseguimos concienciarlos. Sin embargo, no son suficientes para validar la hipótesis. Dada su potencial relevancia educativa, proponemos obtener más evidencias para contrastar su validez.

Abstract

During latest years we have appreciated a decay in the quality of fourth-year student's code. Whenever they are asked to develop complete systems from scratch, their code shows ground problems: spaghetti code, bad structuring, unnecessary repetitions, deficient parameter passing, lacking understanding of the object oriented paradigm, etc.

This work shows examples of these problems and analyses their causes. We deduce that the applied teaching/learning process is somewhat responsible. We construct an explanation based on present psychological theories on mental and conceptual models. We notice that low-level details are omitted or simplified when programming is taught. Hence, we formulate a hypothesis: students are miss-modelling low-level concepts.

In order to confront students with the problem and develop conciousness, we organized #CPCRetroDev: a videogame developing contest for Amstrad CPC. Having only 4Mhz and 64KB of RAM, they are forced to improve their code to leverage resources. Evidences show that students' awareness was developed. However, evidences are not enough to validate the hypothesis. Considering its educative relevance, we encourage getting more evidences to evaluate the hypothesis.

Palabras clave

Aprendizaje, Programación, Modelos Mentales

1. Enseñanza de la programación

Desde los comienzos de la programación, su enseñanza ha estado centrada en el pensamiento algorítmico: secuencias, bucles, variables, tipos, estructuras, etc. Dijkstra da buena prueba al presentar el resumen de años de experiencia:

> *[...]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* (Dijkstra. A Method of Programming [3])

Tras Dijkstra y su método formal de programación, la enseñanza en universidades está basada en las propuestas de ACM / IEEE [18]. En ellas se habla de lenguajes de alto nivel y de enseñar paradigmas como programación estructurada, orientada a objetos y funcional. De hecho, esta última revisión [18] se decanta

directamente por la orientación a objetos como base. Aún así, sigue sin haber consenso general sobre el paradigma para empezar a enseñar [5]. Las alternativas más populares siguen siendo programación estructurada y orientada a objetos, aunque la programación funcional también gana interés.

En 2004 el paradigma estructurado era el más impartido [7]. Pascal y/o Modula-2 eran los lenguajes iniciales en 12 de 25 universidades, ADA era preferido en 6, y 3 titulaciones (un 12 %) comenzaban con paradigma orientado a objetos. En 8 titulaciones la orientación a objetos se impartía tras la introducción estructurada.

La dicotomía entre estructurada y objetos se ha mantenido en el tiempo, variando los lenguajes. C/C++ y Java se han convertido en los más populares, aunque recientemente crece el interés en Python, Javascript y Scratch [4, 6, 10, 13, 15, 16, 19, 21]. Por el camino se ha hecho uso de lenguajes de pseudocódigo y especificación, pero han sido abandonados [6].

Durante esta evolución han habido constantes intentos de enseñar mejor. Sin embargo, los estudiantes siguen teniendo grandes dificultades para entender la programación [2, 15, 16]. Cernuda del Río describe muy gráficamente la sensación que tenemos los profesores:

> *Cuesta un tiempo entender los programas más simples: la iteración, los condicionales, o el simple intercambio de valor entre dos variables. Con un poco de suerte, un buen día algo hace* ***clic*** *y de repente todo empieza a cobrar sentido. [...]Estoy seguro de que los resultados académicos de miles de alumnos mejorarían si fuésemos capaces de transmitir esa habilidad elemental. Pero muchos alumnos llegan a ello (si es que llegan) por un camino largo, aparentemente ineficiente. ¿Realmente no hay una buena forma de contarlo?* (Cernuda del Río, A. Todavía no sabemos enseñar programación[2])

Muchas innovaciones actuales persiguen ese «***clic***». La mayoría son metodológicas: implementación de metodologías activas en el aula [9], uso de ambientación de rol y misiones para los problemas (C++/Java) [21], uso de herramientas para visualizar la ejecución (Javascript) [19], unificación de paradigmas y lenguajes entre asignaturas (Java) [6], aprendizaje a través de los errores de programación (Java) [4], creación de videojuegos en Scratch como introducción [15], o uso de Realidad Aumentada como elemento motivador (Scratch) [20].

Generalmente, estas innovaciones mejoran la motivación de los estudiantes. Las evaluaciones también mejoran tímidamente[1]. Sin embargo, las dificultades de aprendizaje siguen estando subyacentes, como indican Muñoz y otros:

> *La aplicación de conceptos básicos o el diseño de algoritmos que son relativamente simples para los docentes, parece ser algo difícil para el estudiante [13]. La mayor parte de esos problemas son originados por la complejidad de los conceptos tales como variables, estructuras repetitivas, arreglos, funciones de los lenguajes de programación [16]. Estas dificultades se manifiestan independientes del paradigma y/o lenguaje utilizado.* (Muñoz, R. y otros. Uso de Scratch [...]en fundamentos de programación [15])

¿Por qué conceptos aparentemente simples resultan difíciles a los estudiantes? ¿Son conceptos verdaderamente difíciles o quizá los enseñamos de forma ineficiente? ¿Hasta cuánto nos puede afectar la maldición del conocimiento [1]?

Comenzamos este trabajo hace cuatro años observando como estudiantes de cuarto curso producían código de mala calidad (sección 2). Realizamos entrevistas personales y análisis cualitativos, comprobando errores en la base. Apoyándonos en teorías psicológicas actuales, formulamos una hipótesis: los estudiantes están modelizando erróneamente conocimientos base (sección 3). Propusimos actividades para confrontarles con el problema y concienciarles, de las que hemos recabado evidencias (sección 4). Las evidencias muestran concienciación, pero no son suficientes para validar la hipótesis. Concluimos que la hipótesis es factible, pero es necesario profundizar para comprobar su validez. (sección 5). Animamos a buscar nuevas evidencias, pues creemos que la hipótesis puede tener un importante impacto educativo en caso de validarse.

2. Código de estudiantes

Las asignaturas de cuarto curso de Razonamiento Automático y Videojuegos 1 comenzaron a impartirse en 2010/2011. Ambas siguen un modelo basado en proyectos: los estudiantes desarrollan sistemas completos en grupos. Un motor de razonamiento para agentes autómomos y un videojuego 3D son los proyectos tipo. Partiendo de cero, y de forma autónoma, los estudiantes terminan implementando sus proyectos en C++.

Desde el comienzo observamos problemas para diseñar e implementar software desde cero. La mayoría de grupos se frustraban al no ser capaces de crear sistemas de tamaño medio por sí solos. Pronto nos encontramos explicando en clase conceptos de programación

[1] Es muy difícil aseverar que las mejoras son significativas. Los trabajos publicados usan distintas formas de evaluar; incluso en trabajos aislados, las innovaciones cambian la evaluación.

de cursos anteriores: una mayoría de estudiantes los necesitaba para solventar problemas en sus desarrollos. Se explicaban conceptos como el funcionamiento concreto de `#include`, diferencia entre herencia y composición, correcto uso de memoria dinámica, paso de variables por valor, referencia y referencia constante, o uso básico de punteros.

Con déficit de dominio sobre estos conceptos, diseñar e implementar sistemas elaborados desde cero resultaba casi imposible. Esto queda patente analizando fragmentos del código elaborado por los estudiantes. Por ejemplo, el listado 1 muestra parte de un constructor de un *Behaviour Tree* [14].

```
// Reserva de espacio para array de hijos
m_hijos = new Node*;
// ...
m_hijos[0]->tipo = NODO_DECISION;
m_hijos[1]->tipo = NODO_SECUENCIA;
// ...
```

Listado 1: Reserva errónea de espacio para un array. Se reserva un único elemento.

El error de reserva de memoria podría entenderse como un descuido. Sin embargo, muchos estudiantes demostraron no entender lo que sucedía. Los nodos del árbol realizaban tareas distintas a las asignadas. Al construirse muchos nodos seguidos, las distintas reservas se realizaban contiguas en memoria. Por tanto, `m_hijos[1]` de un nodo era en realidad `m_hijos[0]` del siguiente nodo. Los estudiantes seguían sin entenderlo tras varias explicaciones.

Cuestiones parecidas pueden ser observadas en el listado 2. Los estudiantes reservan un *array* de punteros a `Entity` con un único elemento. La primera impresión es de un código provisional que podría tener sentido más adelante. Sin embargo, el problema resulta patente tras preguntar a los estudiantes. Indicaron que siempre que reservaban con `new` utilizaban un *array*, porque no sabían hacerlo de otra manera.

```
Entity **sector;
sector=new Entity*[1];
sector[0]=entities[1];
```

Listado 2: Reserva para vector de un único elemento. Incluido en una función *main()*.

El listado 3 muestra otro problema sorprendente. En un sencillo cálculo temporal de una distancia *Manhattan* para un algoritmo *A** [14], los estudiantes utilizan tres variables estáticas. Dos de estas variables son totalmente temporales, y la última es devuelta mediante referencia constante. Además, los parámetros son pasados como referencias constantes a un tipo básico, algo que introduce costes y ningún beneficio. Los estudiantes explicaron que pensaban que pasar o devolver copias era «*malo*». No tenían una explicación de porqué era «*malo*»: tan sólo seguían esa regla porque les «*sonaba*».

```
const int& NodeP::estimate (const int &x,↩
    const int &y) {
  static int xd, yd, d;
  xd = x - xPos;
  yd = y - yPos;

  // Manhattan distance
  d = abs(xd) + abs(yd);

  return(d);
}
```

Listado 3: Devolución de referencia constante a un cálculo temporal de distancias en un nodo de *A**.

Otro caso muy habitual es el uso de código espagueti como en el listado 4. Se trata de la función de evaluación de las ocho celdas contiguas en un algoritmo de *pathfinding* [14] mediante *A**. Los despropósitos de este código están presentes en el de casi todos los estudiantes: repeticiones innecesarias de código, cálculos y llamadas, copiar y pegar, abuso de `if-else`, nula generalización, diseño y análisis, etc. Este código ha sido presentado como ejercicio a los estudiantes del curso 2016/17. Se les pedía analizarlo y mejorarlo reescribiendo lo que fuera necesario. Cincuenta y un estudiantes lo intentaron individualmente. Ningún estudiante utilizó variables para prevenir múltiples llamadas. Seis intentaron recorrer las casillas con un bucle, sin conseguir que funcionase. Diez factorizaron el contenido de los bloques `if-else` en funciones: sólo uno utilizó una función con parámetros para evitar repetir el código.

```
int pos = listM.front();
if (first == 0)
   posPrev = getPos();
if (first == 0 && pos == 3 && m->getC(((int↩
    )getPS()->getPos().X() / 2), ((int)↩
    getPS()->getPos().Z() / 2)+1) != 0) {
   listM.erase(listM.begin());
   listM.insert(listM.begin(), 4);
   listM.insert(listM.begin() + 1, 2);
   pos = listM.front();
}else if(first == 0 && pos == 3 && m->getC↩
    (((int)getPS()->getPos().X() / 2)-1, ((↩
    int)getPS()->getPos().Z() / 2)) != 0) {
   listM.erase(listM.begin());
   listM.insert(listM.begin(), 2);
   listM.insert(listM.begin() + 1, 4);
   pos = listM.front();
}
//
// ... 6 else if adicionales con misma ↩
    estructura y contenido
//
```

Listado 4: Código manifiestamente mejorable y potencialmente peligroso. Repeticiones innecesarias. Extraído de una rutina de *Pathfinding* con *A**.

Algunos fragmentos muestran graves problemas de comprensión. El listado 5 es un buen ejemplo. Se trata de un juego basado en celdas. Cada celda tie-

ne un valor de 0 a 31 representando el tipo de baldosa (*tile*) que contiene. Algunas son no-transitables. `colisiona()` determina si un personaje colisionará con una baldosa no-transitable en `x,y`. Los estudiantes comprueban todos los casos posibles uno por uno con un `if`. El bloque `else-if` muestra que no deducen que una baldosa no no-transitable, es transitable. Tampoco reparten los tipos de baldosa correlativamente, lo que dejaría una única comprobación de tipo menor-que. Además, utilizan un puntero a la celda que desreferencian 32 veces, en lugar de obtener su valor directamente (que es un `u8`, un *byte* sin signo).

```
bool colisiona(u8 x, u8 y) {
  u8* t = obtenerTilePtr(x, y);
  if(*t != 0) {
    if(*t==1 || *t==3 || *t==4 || *t==7 /* ↩
        6+ checks */ || *t==31) {
      return true;
    } else if (*t==2 || *t==5 || *t==6 /* ↩
        16+ checks */ || *t==30) {
      return false;
    }
  } else {
    return false;
  }
}
```

Listado 5: Función para comprobar colisiones con un mapa basado en celdas con 32 posibles baldosas (*tiles*).

Como es esperable tras analizar estos ejemplos algorítmicos, también hay serios problemas con la orientación a objetos. El listado 6 es un método que dibuja en pantalla (*renderiza*) las entidades de un juego. Un buen diseño orientado a objetos tendría una clase base `Entidad` u `Objeto` con un método virtual `render()`. Esto permitiría *renderizar* todas las entidades en un único bucle. En cambio, el código muestra tres tipos de objeto (jugador, enemigos y balas) por separado. Además, hay cuestiones incomprensibles como las variables temporales `iter` y `bala_aux` que son miembros de la clase `Game` en lugar de ser locales.

Lejos de ser casos aislados, encontramos cientos de ejemplos similares en el código de casi todos los estudiantes. Además, los profesores percibimos que estos problemas van en aumento. ¿A qué se deben? ¿Cómo podemos revertirlos?

```
void Game::render(){
  jugador->render();

  std::vector<enemigos*>::iterator it = ↩
      npcs.begin();
  while (it != npcs.end()) {
    (*it)->render();
    it++;
  }

  iter = balas.begin();
  while (iter != balas.end()){
    bala_aux = *iter;
    bala_aux->render();
    iter++;
  }
  renderizador->dibujar();
}
```

Listado 6: Clases especializadas sin clase base. Uso de variables auxiliares innecesarias y miembro de clase en lugar de locales.

2.1. Qué dicen los estudiantes

El análisis nos ha llevado a preguntar a los estudiantes. Queríamos conocer cómo razonan e implementan para indagar sobre el origen de los problemas. Además de las situaciones ya comentadas en la sección 2, hemos recibido muchas respuestas reveladoras. A continuación resumimos algunas de las más comunes:

Estudiantes preguntados por su código espagueti:

- «Siempre he programado así y mis programas funcionan bien.»
- «En la empresa X me han dicho que programo muy bien y está así.»
- «¿Por qué tengo que complicarme más la vida si así funciona?»

Preguntados por repeticiones de código:

- «Si un bucle hace lo mismo, ¿Qué más da cómo se implemente?»
- «Pero, ¿eso no lo optimiza el compilador?»
- «Si hago un bucle no puedo diferenciar las partes que cambian.»

Búsquedas lineales innecesarias:

- «No entiendo muy bien los iteradores. El código lo copié y adapté.»
- «¿De qué forma lo busco, si no sé donde está?»
- «Es que si uso punteros me pierdo. Por eso prefiero arrays y vectores.»

Orientación a objetos:

- «Lo hago todo en una sola clase porque es mejor.»
- «¿Para qué voy crear X clases si con un switch funciona igual?»
- «No uso patrones de diseño porque me parecen muy complicados. ¿Por qué C++ no es como lenguaje X que ya los tiene hechos?»

Estas respuestas evidencian problemas importantes: confunden funcionalidad con escalabilidad y usabilidad del código, desconocen el valor de una mejor estructuración, desconocen el funcionamiento compilador, tienen poca o nula experiencia generalizando, no reflexionan sobre coste/rendimiento, desechan lo que desconocen, su análisis de necesidades es muy limitado, no identifican muchos problemas, etc.

Sin embargo, otros detalles llamaron más nuestra atención. Evidencian problemas conceptuales de nivel

básico: bucles, punteros, modularización, uso del compilador, conceptos equivocados, ignorancia de detalles de ejecución... Estos problemas serían esperables en primeros cursos, pero no en cuarto. ¿Por qué no dominan estos detalles básicos? ¿Impiden estos detalles que comprendan correctamente el funcionamiento de sus programas? ¿Tiene esto relación con la calidad de su código?

3. Planteamiento de la hipótesis

Resulta evidente que los estudiantes no detectan estos problemas por sí mismos. De hecho, la mayoría ni siquiera entiende por qué son problemas. Esto indica que son conceptuales y de base. De ser simples errores o copia-pegas apresurados, serían capaces de reconocerlos y entenderlos. Además, los problemas son generales entre los estudiantes. Es lógico deducir que hay carencias en su proceso de enseñanza/aprendizaje.

Las carencias pueden explicar la mala calidad de su código. Pero, ¿dónde se originan? Nuestra hipótesis las sitúa en la falta de dominio de conceptos básicos. Conviene explicar detalladamente esta afirmación:

1. Los estudiantes son enseñados de inicio en lenguajes de alto nivel como C/C++ o Java.
2. Entre la ejecución del binario en un procesador y el código fuente hay un abismo de conceptos.
3. Las asignaturas de programación explican algunos, no todos, metafórica o teóricamente.
4. Predominan conceptos algorítmicos y del paradigma: variables, secuencias, condicionales, funciones, clases, encapsulación, etc.
5. Se introduce el uso básico del compilador y después se usan entornos de desarrollo (IDEs).
6. Otras asignaturas explican algunos conceptos de bajo nivel, en distintos contextos y semestres.

Los profesores vemos lógico este proceso: los detalles no son necesarios para programar y deben dominar los paradigmas cuanto antes. Aquí puede estar el origen del problema: obviamos el valor de los detalles porque nosotros ya los dominamos [1]. Sin embargo, los detalles son relevantes y necesarios para entender, aprender y mejorar en programación. Éstos forman el sustrato concreto del que abstraer los conceptos de programación. Construir abstracciones sin bases concretas no es posible. Cuando no hay, los detalles concretos son creados para formar abstracciones mentales.

3.1. Modelos mentales

En 1983, Johnson-Laird [11] describió las representaciones mentales que son utilizadas actualmente para explicar el proceso de aprendizaje en ciencias [12]. En su teoría describe los modelos mentales como aproximaciones análogas estructurales del mundo que los individuos elaboran. Según Johnson-Laird, son no-científicas, incompletas e inestables: su único compromiso es ser útiles para las predicciones del sujeto.

En ciencia se utilizan modelos conceptuales [8]: representaciones externas, precisas, completas y consistentes con el conocimiento científicamente compartido. Pueden materializarse como formulaciones matemáticas, analogías o artefactos materiales.

El aprendizaje [12] consistiría en un proceso de modelización, donde los estudiantes construyen modelos mentales nuevos o adaptan los previamente existentes. Conforme desarrollan modelos mentales más completos, precisos y cercanos a la realidad que modelan, más efectivo resulta el aprendizaje.

Los profesores tendemos a asumir que los estudiantes construirán modelos mentales muy similares a los conceptuales que les presentamos. Sin embargo, esto no sucede [8]. Como indica Norman [17] «*[...]idealmente debería haber una relación directa entre modelo conceptual y modelo mental. Generalmente, éste no es el caso.*» Por su parte, Johnson-Laird [11] deja claro que «*Ni los modelos mentales resultan copias perfectas de los modelos conceptuales, ni la modelización resulta* ***evidente*** *para los estudiantes.*»

Cabe resaltar que los modelos conceptuales no son necesariamente entendibles por sí solos. Muchas veces requieren conocimiento del dominio. Por ejemplo, no es posible entender las fórmulas de la mecánica newtoniana si no se conoce a qué se refieren o qué representan sus variables. Este problema afecta particularmente a los estudiantes [8]. Además, resulta especialmente difícil de apreciar por quienes dominan el modelo conceptual [1]. Lejos de percibir cuan complejo resulta modelizar, suele interpretarse que el modelo conceptual es intuitivo. Esta subestimación produce desconexión entre profesores y estudiantes.

Por otra parte, muchos modelos conceptuales en ciencias son completos y concisos. En cambio, conseguir completitud en programación resulta extremadamente extenso. Ser completo requiere una descripción minuciosa del lenguaje de programación, del funcionamiento del compilador, del código máquina o intermedio generado, y de la máquina que ejecuta. Sólo así podría considerarse completo, preciso y consistente [8]. Esto explica que la programación sea tan compleja como disciplina, con tanta disparidad de capacidades entre profesionales.

3.2. Modelización errónea en la base

Resumiendo los factores analizados podemos plantear una hipótesis coherente con ellos:

1. El aprendizaje de los estudiantes produce modelos mentales funcionales, inexactos e inestables.

2. Estos modelos resultan más efectivos cuanto más cercanos son a la realidad modelada.
3. La realidad a modelar en programación es compleja y extensa, con muchos niveles.
4. Enseñamos en lenguajes de alto nivel, simplificando u obviando detalles de niveles inferiores.

La figura 1 muestra los niveles que formarían un modelo conceptual en programación. En verde aparece el porcentaje aproximado de contenidos que se enseñan en asignaturas de programación. El foco está puesto en la creación de programas usando un lenguaje y un paradigma. Del lenguaje no da tiempo a explicarlo todo, por lo que se hace énfasis en las partes esenciales. Del compilador se explica qué es y cómo usarlo a nivel básico. El código máquina y la ejecución en la máquina se definen rápidamente en teoría, a modo informativo.

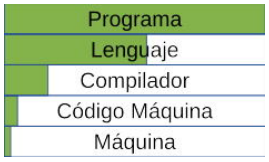

Figura 1: Niveles de un modelo conceptual en programación y enseñanza aproximada

La distribución de la figura 1 suele deberse a que la mayoría de profesiones, proyectos o desarrollos se realizan en los niveles superiores. Además, el bajo nivel se ve en asignaturas de arquitectura. Sin embargo, esto no tiene en cuenta las necesidades de un estudiante a la hora de construir el conocimiento. Aquí se sitúa la hipótesis de este trabajo:

> Los estudiantes están modelizando erróneamente los niveles base de la programación: desde compilador hacia abajo. (Hipótesis)

Esta hipótesis puede explicar los problemas observados a alto nivel. Se les enseña la parte verde de la figura 1, pero necesitan entender y visualizar el funcionamiento de muchos más detalles para poder modelizar todo el conjunto. Es probable que construyan modelos mentales tentativos hasta dar con alguno que funcione. En ese momento se produciría el «***clic***» referido por Cernuda del Río [2]. Este proceso podría ser causante frustración en muchos estudiantes.

Asumiendo que el aprendizaje produce modelos inexactos e inestables [11], deducimos dos cosas: 1) la parte enseñada (verde) se modela de forma inexacta, 2) la parte no enseñada se modela de forma autónoma, aleatoria y, probablemente, errónea. Es importante señalar que los niveles inferiores no quedan vacíos en los modelos mentales de los estudiantes. Son necesarios para entender el funcionamiento y programar. El modelado erróneo de estos cimientos puede ser la causa de que el edificio no se sostenga.

4. Acciones realizadas

El problema se detectó en las asignaturas de cuarto curso. Aunque la intervención ideal debería haber sido en primero, no era razonable aplicar fuertes cambios con una hipótesis de partida. Por tanto, se pensó alternativas que pudieran aplicarse en cuarto curso y que permitieran obtener evidencias iniciales.

Se optó entonces por actividades que obligasen a los estudiantes a enfrentarse a problemas de bajo nivel. En concreto, propusimos el desarrollo de videojuegos en máquinas con recursos limitados. Suponiendo la hipótesis como cierta, los estudiantes evidenciarían déficits y errores en sus modelos mentales con estos trabajos. Ganarían conciencia sobre el problema y el impacto en sus capacidades como profesionales. Conocer estos déficits les abriría el camino para paliarlos.

4.1. El concurso #CPCRetroDev

En 2013 convocamos la primera edición del Concurso de Programación de Videojuegos Retro de la UNIVERSIDAD #CPCRetroDev. Se pedía a los estudiantes que desarrollaran un videojuego para Amstrad CPC 464. Se trata de un ordenador de 1984, con un procesador *Zilog Z80* a 4 Mhz, 64 KB de RAM y una unidad de casete como almacenamiento.

Programar un videojuego es una actividad motivadora y compleja. Hacerlo en una máquina con recursos muy limitados obliga a trabajar los conceptos de bajo nivel y a entender lo que sucede en la máquina. Los estudiantes programan la máquina en C, utilizando un compilador cruzado. Para mejorar su código y aprovechar los recursos, deben prestar atención al código ensamblador que genera el compilador. Un código mal implementado en C genera un ensamblador más grande y lento, y en un Amstrad CPC 464 el resultado es perceptible. Esto permite que aprecien la diferencia entre distintas formas de programar y cambien de actitud.

El concurso se organiza con carácter internacional, permitiendo participar a cualquier persona del mundo. Al ser una máquina real, con una base de usuarios estable, participan desarrolladores experimentados. Además, se produce una edición física real en casete con todos los videojuegos presentados. Esto ayuda a motivar a los estudiantes: sus videojuegos no son una simple práctica, sino un producto real que muchos usuarios van a probar. También tienen la oportunidad de competir con desarrolladores reales, y de aprender de ellos. El concurso incentiva fuertemente la publicación del código fuente con licencias libres para favorecer el aprendizaje.

Todos los videojuegos presentados a concurso durante estos 4 años y su código fuente están disponibles en la web del concurso[2].

4.2. Indicios obtenidos

Los estudiantes reciben una encuesta antes y después de desarrollar los videojuegos. Por motivos de espacio, incluimos sólo las preguntas más relevantes:

1. ¿Cómo evalúas tu nivel de programación? (0-10)
2. ¿Qué deberías hacer para mejorarlo?
3. ¿Qué concepto crees que dominas menos?
4. ¿Cómo de difícil consideras ensamblador? (0-10)
5. ¿Crees necesario saber ensamblador? (S/N)
6. ¿Crees necesario dominar el compilador? (S/N)

Los resultados agregados de estos 4 años son bastante reveladores como muestra el cuadro 1. Resulta sorprendente su autoevaluación: se otorgan un notable (7,23) antes de desarrollar el videojuego, y un aprobado (5,92) después. Esto evidencia que se conciencian de sus limitaciones.

Respecto al bajo nivel, inicialmente piensan que es muy difícil y poco importante. Después lo ven menos difícil e importante. De hecho, muchos estudiantes nos han preguntado directamente por qué no se les enseña este conocimiento antes. El cambio de actitud resulta evidente.

	Antes	**Después**
1	7.23	5.92
2	1º Programar más 2º Nuevos lenguajes	1º Aprender bajo nivel 2º Programar más
3	1º Herencia 2º Memoria	1º Memoria 2º Punteros
4	9.07	7.66
5	22 / 169 si/no	93 / 98 si/no
6	82 / 108 si/no	150 / 41 si/no

Cuadro 1: Resultados agregados encuesta a 191 estudiantes en 4 años antes y después del #CPCRetroDev.

Muy curiosa es actitud respecto al compilador. Antes de empezar, muchos de ellos usaban el compilador a través de entornos de desarrollo. Después se interesan por la separación entre preprocesado, compilado y enlazado, así como el código generado y las distintas opciones de configuración.

Es importante indicar que esta actividad no resuelve sus problemas. De hecho, tras la encuesta creen que sus principales defectos son la gestión de memoria en general (no sólo dinámica) y el manejo de punteros. Sin embargo, si parece conseguir concienciación y cambio de actitud. También parecen coincidir con la hipótesis planteada: ven el origen de sus problemas en los niveles inferiores.

Aunque estos indicios apoyan la hipótesis, no son suficientes para validarla. Sólo son un primer paso que muestra su viabilidad. Confiamos en que este trabajo anime a recabar nuevas evidencias.

5. Conclusiones

En este trabajo hemos analizado el origen de la mala calidad del código en estudiantes de cuarto curso. Los estudiantes muestran muchos problemas conceptuales y de base. Además, no son conscientes de muchos de sus problemas y no les preocupa la calidad de su código en general. Su principal percepción es que si algo funciona, no está mal.

Al ser un caso general, hemos deducido que el proceso de enseñanza/aprendizaje debía tener déficits. Hemos recurrido a los modelos mentales y conceptuales de la psicología para entender mejor lo que sucedía. Todo este proceso nos ha llevado a formular la hipótesis de que *los estudiantes modelizan erróneamente los niveles base de la programación*. Esto es coherente con los conceptos que se obvian o simplifican en las asignaturas de programación.

Hemos descrito el concurso #CPCRetroDev como una actividad organizada para concienciar a los estudiantes de estos problemas. La actividad les exige programar un videojuego con recursos severamente limitados. Los indicios recabados mediante encuestas muestran que la actividad consigue cambiar su actitud. Sin embargo, la actividad no tiene un efecto de mejora apreciable en su nivel.

Nuestra conclusión final es que la hipótesis planteada es factible, pero no hay evidencia suficiente para validarla. Por este motivo, creemos importante compartir todo el proceso y proponer la obtención de más evidencias. En caso de que la evidencia recogida validara la hipótesis, podría tener importantes consecuencias en la forma en que enseñamos programación.

Nuestro próximo objetivo será enseñar programación a bajo nivel en primero. Elaboraremos contenidos en el orden inverso de la figura 1, empezando en código máquina y construyendo hasta llegar al lenguaje de alto nivel. Construiremos los conceptos en órden creciente de abstracción, empezando por los más concretos. Diseñaremos el experimento para obtener evidencias a corto, medio y largo plazo y estudiaremos la evolución de conocimientos y calidad del código. Mediremos la velocidad de adquisición de conceptos y el nivel de dominio, comparándolo en estudiantes enseñados con el temario clásico y con esta nueva aproximación. Publicaremos todos los materiales elaborados para permitir a otros profesores repetir las experiencias y obtener más evidencia sobre la validez de la hipótesis.

[2] `http://cpcretrodev.byterealms.com`

Referencias

[1] Susan A.J. Birch. When knowledge is a curse. *Current Directions in Psychological Science*, *14*(1), 25–29. 2005.

[2] Agustín Cernuda del Río. Todavía no sabemos enseñar programación. *ReVisión*, *10*(1), 13 – 15. 2017

[3] Edsger W. Dijkstra y W. H. Feijen. *A Method of Programming*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. 1988.

[4] Carlos Fernandez-Medina, Juan Ramón Perez-Perez, Mª del Puerto Paule-Ruiz, y Víctor Alvarez-Garcia. Aprendizaje de la programación guiado por los errores de compilación. En *Actas de las XX Jornadas de Enseñanza Universitaria de Informática, Jenui 2014*, (pp. 371–378), Oviedo, 2014.

[5] L. Fernández Muñoz, R. Peña, F. Nava, y Á. Velázquez Iturbide. Análisis de las propuestas de la enseñanza de la programación orientada a objetos en los primeros cursos. En *Actas de las VIII Jornadas de Enseñanza Universitaria de Informática, Jenui 2002*, (pp. 433 – 440). Cáceres, 2002.

[6] Jorge García, José C. Riquelme, Mariano González, y Isabel Nepomuceno. Diez años innovando en la enseñanza de los fundamentos de la programación: resultados y conclusiones. En *Actas de las XVIII Jornadas de Enseñanza Universitaria de Informatica, Jenui 2012*, (pp. 9–16). Ciudad Real, 2012

[7] Jesús García Molina. Un enfoque semiformal para la introducción a la programación. En *Actas de las X Jornadas de Enseñanza Universitaria de Informática, Jenui 2004*, (pp. 401 – 408). Alicante, 2004.

[8] Ileana Maria Greca y Marco Antonio Moreira. Mental models, conceptual models, and modelling. *International Journal of Science Education*, *22*(1), 1–11. 2000.

[9] Estrella Gómez Fernández, María José García García, y Gurutze Miguel Villalba. Aplicación de diversas metodologías activas en la asignatura de introducción a la programación. En *Actas de las XI Jornadas de Enseñanza Universitaria de Informática, Jenui 2005*, (pp. 205–211). Villaviciosa de Odón (Madrid), 2005.

[10] Jesús Ibáñez Martínez y Toni Navarrete Terrasa. Diseño y evaluación de la asignatura Programación I de acuerdo a las directrices del Espacio Europeo de Educación Superior. En *Actas de las XII Jornadas de Enseñanza Universitaria de Informática, Jenui 2006*, (pp. 461 – 468). Bilbao, 2006.

[11] Phillip N. Johnson-Laird. *Mental Models*. Cognitive science series. Cambridge University Press. 1983.

[12] Rosária Justi y Jan van Driel. The use of the interconnected model of teacher professional growth for understanding the development of science teachers' knowledge on models and modelling. *Teaching and Teacher Education*, *22*(4), 437 – 450. 2006.

[13] Antonio Jose Mendes y María José Marcelino. Tools to support initial programming learning. En *CompSysTech 2006- International Conference on Computer Systems and Technologies*, Bulgaria. 2006.

[14] Ian Millington y John Funge. *Artificial Intelligence for Games, Second Edition* (2nd ed.). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. 2009.

[15] Roberto Muñoz, Thiago S. Barcelos, Rodolfo Villarroel, Marría Marta, Becerra Carlos, Noel Rene, y Ismar Frango Silveira. Uso de scratch y lego mindstorms como apoyo a la docencia en fundamentos de programación. En *Actas de las XXI Jornadas de Enseñanza Universitaria de Informática, Jenui 2015*, (pp. 248–254). Andorra La Vella, 2015.

[16] Roberto Muñoz, Marta Barría, René Nöel, Eliana Providel, y Patricio Quiroz. Determinando las dificultades en el aprendizaje de la primera asignatura de programación en estudiantes de ingeniería civil informática. En J. Sánchez (Ed.), *Nuevas Ideas en Informátíca Educativa. Memorias del XVII Congreso Internacional de Informática Educativa, TISE*, (pp. 120 – 126). Santiago de Chile, 2012.

[17] Donald A. Norman. Some observations on mental models. *Mental models*, *7*(112), 7–14. 198.

[18] Joint Task Group on Computing Curricula . *Computer Engineering Curricula 2016: Curriculum Guidelines for Undergraduate Degree Programs in Computer Engineering*. New York, NY. 2016.

[19] Lluís Ribas Xirgo. Animación interactiva de algoritmos para cursos de introducción a la programación. En *Actas de las XVII Jornadas de Enseñanza Universitaria de Informática, Jenui 2011*, (pp. 503–510). Sevilla, 2011. Póster/Recurso docente.

[20] Natali Salazar Mesía, Cecilia Sanz, y Gladys Gorga. Experiencia de enseñanza de programación con realidad aumentada. En *Actas de las XXII Jornadas de Enseñanza Universitaria de Informática, Jenui 2016*, (pp. 213–220). Almería, 2016.

[21] Pilar Sancho Thomas, Pedro Pablo Gómez Martín, Rubén Fuentes Fernández, y Baltasar Fernández-Manjón. NÚCLEO, aprendizaje colaborativo escenificado mediante un juego de rol. En *Actas de las XIV Jornadas de Enseñanza Universitaria de Informática, Jenui 2008*, (pp. 453 – 460). Granada, 2008.