Del diseño a la implementación del Software: una metodología de cohesión de disciplinas

Cristina Cachero
Dpto. Lenguajes Sistemas Informáticos
Universidad Alicante
03072 San Vicente Raspeig
ccachero@dlsi.ua.es

Otoniel López
Dpto. Física y Arquitectura de Computadores
Universidad Miguel Hernández
03202 Elche
otoniel@umh.es

María José Durá
Dpto. Orientación
IES Victoria Kent
03202 Elche
majosedm@telefonica.net

Resumen

En la actualidad es común que enseñanza de la Programación Orientada a Objetos se realice atendiendo principalmente a criterios sintácticos (basados en un lenguaje de programación determinado) y deje de lado aspectos conceptuales. Este hecho causa dificultades a los alumnos a la hora de abstraer los conocimientos adquiridos y relacionarlos con los que asimilan en otras materias como la Ingeniería del Software. Desde este artículo se defiende la idea de que esta falta de interconexión explícita en los temarios de ambas asignaturas crea en los alumnos lagunas de conocimiento y menoscaba su motivación a la hora de aprender técnicas de diseño, que no perciben como verdaderamente útiles para la codificación de software. Para paliar este problema, presentamos una experiencia de trabajo donde la enseñanza y discusión de los conceptos relacionados con la Programación Orientada a Objetos se ha apoyado en la notación UML. Los resultados demuestran, desde nuestro punto de vista, que el uso de UML como apoyo desde el inicio de la formación del alumno en conceptos de OO no sólo permite que éste sea capaz de comunicarse mediante especificaciones diagramáticas estándares, sino que le ayuda a percibir la importancia del diseño en el código implementado.

1. Introducción

El área de Programación en la carrera de Ingeniería Informática tiene un papel preponderante en la formación del alumno, sobre todo en los primeros años de carrera. En nuestra universidad, este área gestiona un 20 % de la docencia troncal/obligatoria impartida en primer curso y un 28 % de la docencia troncal/obligatoria de segundo curso. Es precisamente en segundo donde los alumnos tienen su primer contacto con la Programación Orientada a Objetos (POO), que se imparte como asignatura obligatoria de las tres ingenierías y tiene una carga lectiva de 2,25 créditos teóricos y 2,25 créditos prácticos. Esta asignatura se complementa entre otras con Lenguajes y Paradigmas de Programación (donde se introducen todos los paradigmas) y Herramientas de Programación (donde el alumno se familiariza con entornos de desarrollo y depuración, herramientas de control de versiones, estándares de nomenclatura y documentación, etc.), también obligatorias de segundo curso.

Para la enseñanza de las principales técnicas de POO, esta asignatura, al igual que ocurre en otras universidades, se imparte en base a un lenguaje determinado. En nuestro caso, dicho lenguaje es C++, que, defendido como lenguaje de aprendizaje para el paradigma OO por autores como [11], marca además una continuidad con el lenguaje aprendido por los alumnos en primer curso. Pensamos que este enfoque implícitamete transmite al alumno uno de los más extendidos mitos de la programación, que es el de *Soy un programador. ¿Por qué debería preocuparme por el diseño?*, en lugar de transmitirle que *todo el mundo que escribe código también diseña código, sea bien o mal,*

consciente o inconscientemente [10]. Además, nuestra experiencia en las aulas indica que el uso de un lenguaje determinado como vehículo de comunicación, si bien a corto plazo agiliza la capacidad de implementación del alumno en dicho lenguaje, implícitamente condiciona el modo en que el alumno interioriza los distintos conceptos de la asignatura, lo cual menoscaba su capacidad para aplicar dichos conceptos a nuevos lenguajes o relacionarlos con asignaturas impartidas con posterioridad en la ingeniería.

Es por ello que en este artículo defendemos la idea de que es necesario hacer visible el proceso de diseño desde las primeras etapas de formación del alumno en conceptos relacionados con la Orientación a Objetos (OO). El uso de técnicas de diseño permite la introducción de una metodología de programación basada en la abstracción de datos. De este modo se ayuda a romper la inercia del alumno al seguir utilizando las técnicas aprendidas en primer curso, fundamentadas en la abstracción funcional y el diseño descendente para la resolución de problemas (metodologías top-down)[11]. Además, pensamos que hacer visible este proceso ayuda a explicitar la relación entre las asignaturas de Programación y las de Ingeniería del Software, y de este modo facilita la adquisición de una visión global de la disciplina.

A continuación presentamos una propuesta metodológica que, basándose en esta idea, utiliza como base para la discusión de los principales conceptos de OO la notación UML. Para ello, en la sección 2 presentamos el contexto en el que se desarrolla la asignatura, para a continuación, en la sección 3, describir los fundamentos metodológicos de nuestra propuesta y cómo se adapta a este contexto. En la sección 4 mostramos los resultados de la evaluación que el alumnado ha hecho de nuestra aproximación. Por último, la sección 5 muestra las conclusiones y las líneas de trabajo que estamos siguiendo para continuar mejorando la asignatura.

2. Situación Actual

En la actualidad existe una disparidad notable de criterios respecto al mejor modo de abordar la enseñanza de la POO, criterios que abarcan desde el lenguaje a utilizar para ejemplificar los conceptos hasta cuándo se debe introducir al alumno en este paradigma. Respecto al primer punto, abundan las propuestas [2, 8] en las que se aboga por el uso de más de un lenguaje de programación para ilustrar los distintos conceptos. De este modo se pretende demostrar al alumno que, independientemente del ritmo vertiginoso al que los lenguajes de programación aparecen y desaparecen, los conceptos subyacentes siguen siendo válidos. Otros [9, 7] acuden a lenguajes diseñados por ellos mismos y que incluyen bajo una sola sintaxis los conceptos considerados más relevantes. No obstante, las circunstancias de la práctica docente (escaso número de horas de clase, amplitud de los temarios y necesidad de dar respuesta a la demanda de profesionales formados en determinadas tecnologías por parte de la empresa) hacen que en la mayoría de los casos se acabe impartiendo la materia en base a un lenguaje *popular* y de alto nivel como C++ o Java, incluso reconociendo que dichos lenguajes no son perfectos. Por otra parte, y refiriéndonos a la cuestión de cuándo abordar la enseñanza de este paradigma, frente a autores que proponen introducirlo desde el primer contacto del alumno con la programación, existen otros que abogan por hacerlo tras haber profundizado en la programación estructurada e incluso en los Tipos Abstractos de Datos.

No es nuestro propósito contribuir a esta polémica, sino señalar la necesidad de, sea cual sea el enfoque adoptado, tratar de manera explícita su relación con la Ingeniería del Software. En efecto, la mayor parte de los programas y bibliografía consultados se centran en discutir los distintos conceptos de la POO en relación a su implementación en uno u otro lenguaje, y dejan implícita la discusión respecto a por qué se adopta una u otra configuración de clases y objetos en dicha implementación. Nuestra experiencia docente demuestra que de este modo el alumno tiende a centrarse en concep-

tos sintácticos y no profundiza en los conceptos teóricos de la asignatura. Múltiples ejemplos en nuestro trabajo diario avalan esta teoría: desde un uso automatizado, basado en mimetismo de ejemplos de código similares dados en clase, de palabras reservadas de C++ como *virtual* hasta una falta de comprensión de por qué determinados tipos de generalización (implementados mediante herencia) son seguros y cuáles, aunque también permitidos en los lenguajes de programación, sí pueden causar sin embargo problemas de mantenimiento y reutilización. En general, el alumno asume la idea de que *si compila, entonces está bien.*

Es por ello que en este artículo proponemos una metodología que, partiendo de la situación actual de la asignatura, introduce al alumno en consideraciones relativas al diseño desde las primeras fases de enseñanza del paradigma OO. Es importante destacar que desde este artículo no abogamos por la sustitución del uso de ejemplos de código para ilustrar los conceptos específicos, sino que, al contrario, optamos por una complementariedad entre conceptos teóricos, consideraciones de diseño y ejemplos prácticos que favorezcan la interiorización correcta de conceptos por parte del alumno.

A continuación presentamos dicha metodología, así como nuestra experiencia en las aulas.

3. Metodología Propuesta

No cabe duda de que las técnicas de enseñanza están evolucionando de una manera muy rápida en los últimos años. Hasta hace poco tiempo, la docencia de cualquier especialidad científica o técnica era mayoritariamente transmitida mediante la exposición oral por parte del profesor en las llamadas clases magistrales. El principal problema de este tipo de clase es que constituye un método de transmisión unidireccional entre profesor y alumno [1, 4]. Las clases de laboratorio, más innovadoras, tampoco carecen de problemas prácticos, entre los que destacamos el gran número de alumnos (682 matriculados en la asignatura durante el curso 2003-2004) y la escasez de medios en dichos laboratorios, que hacen difícil un buen aprendizaje y una tutorización personalizada de los problemas y dificultades particulares de cada alumno. Este problema ha sido parcialmente paliado por la inclusión en la práctica diaria del uso de otros medios para la docencia, entre los que destacamos las distintas versiones de campus virtual presentes en la actualidad en la práctica totalidad de las universidades españolas.

Nuestra propuesta intenta potenciar el aprendizaje del alumno mediante una combinación sincronizada de los distintos tipos de técnicas. Así, durante las 15 semanas que dura el cuatrimestre, cada clase de teoría se ve apoyada por una clase práctica donde se revisan e implementan los conceptos teóricos comentados. Por otro lado, en las clases teóricas se combina lección magistral con ejercicios que los alumnos resuelven y entregan en clase. Con el fin de integrar el diseño en esta dinámica, hemos hecho de UML el vehículo mediante el cual, por un lado, se plantean y discuten los conceptos teóricos en clase, y por otro se especifican los ejercicios que el alumno debe implementar en un lenguaje de programación determinado (en nuestro caso C++) durante las clases prácticas.

Toda nuestra labor docente se ve apoyada por la web de la Escuela Politécnica y, especialmente, por el Campus Virtual de nuestra universidad, donde el alumno recoge todos los apuntes y transparencias necesarios, consulta la información relativa a objetivos, temario, evaluación o bibliografía, revisa las preguntas frecuentes de la asignatura y los enlaces de interés actualizados y, sobre todo, realiza tutorías electrónicas que complementan la tutorización presencial: en el año 2003-2004 se ha contabilizado una media de 20 tutorías semanales, con picos importantes en fechas de examen y entrega de prácticas.

3.1. *¿Por qué esta opción metodológica?*

Esta propuesta está enmarcada en el "Modelo Tecnológico", uno de los cuatro modelos metodológicos que distinguen diversos autores [3]. El modelo tecnológico se fundamenta en las teorías eficientistas y conductistas del

aprendizaje. Los aspectos característicos de este enfoque disciplinar se sintetizan en una programación basada en objetivos operativos, la puesta en práctica de secuencias cerradas de actividades vinculadas a los objetivos y la realización de un diagnóstico previo y final del nivel de aprendizaje del alumno que sirve de evaluación. Dentro de este modelo, y como bien dicen Coll *et al.* [5], aprender un contenido implica atribuirle un significado, construir una representación o un *modelo mental* del mismo. Cuando hablamos de la actividad mental del alumno nos referimos por tanto al hecho de que este construye significados, representaciones o modelos mentales de los conocimientos a aprender.

En esta metodología destacan dos factores fundamentales a tener en cuenta:

- *El factor visual en el proceso de enseñanza-aprendizaje.* Nuestra propuesta facilita el aprendizaje al permitir que los alumnos tengan una perspectiva visual de los conceptos teóricos [6, 12], al mostrarles ese modelo mental mediante diagramas de clases (con notación UML). Así, los alumnos aprenden a manejar relaciones entre clases o conceptos como herencia, polimorfismo, generalización, y en general todos aquellos relacionados con la OO de manera visual, abstrayéndolos de la sintaxis específica de un lenguaje determinado.

- *La secuenciación de los contenidos.* En este tipo de metodología es importante que los conceptos se expongan primeramente de una forma genérica y visual, y posteriormente se lleven a la práctica, en nuestro caso mediante su implementación en un lenguaje de programación específico. Esta forma de secuenciación de los contenidos está totalmente en consonancia con los principios del aprendizaje significativo, el cual consiste en comenzar por los elementos más generales e ir introduciendo paulatinamente los elementos más específicos[13]. Pensamos que una correcta secuenciación de los contenidos es fundamental para conseguir que el alumno consiga fijar esos conceptos más rápidamente.

3.2. Puesta en práctica de la metodología

Como ejemplo ilustrativo del modo en que la metodología propuesta se ha llevado a cabo durante el curso 2003-2004 presentamos la temporalización de las primeras sesiones de la asignatura que culminaron con la implementación de la primera de las dos prácticas que el alumno debe entregar en la asignatura.

En esta programación, las primeras tres semanas de prácticas se dedicaron a impartir un seminario de C++ orientado a que los alumnos se familiarizasen con el entorno de desarrollo (compilador g++ bajo Linux) y herramientas que tenían que utilizar (makefile, gdb, tar, doxygen, etc.). Durante la cuarta sesión, y tras haber impartido en clase el tema correspondiente a *clases, objetos y métodos*, se les entregó el enunciado de la primera práctica, que giraba en torno al diagrama de clases que puede ser visto en la Fig. 1, y que constituía un subconjunto de las clases necesarias para implementar el conocido juego de *Hundir la Flota*. En esta cuarta clase se hizo que los alumnos se centraran en la clase *Casilla*, e implementaran una versión compilable de la misma. Esa misma semana en clase se vieron las relaciones básicas entre objetos (asociación, composición y dependencia), y esos contenidos teóricos permitieron en la sesión 5 de prácticas abordar la construcción de la clase *Tablero*. Durante dicha construcción se planteó una discusión sobre por qué la relación semántica existente entre ambas clases había hecho conveniente implementar primero la clase *Casilla*. También se plantearon distintas alternativas para implementar la relación entre ambas clases (mediante un array estático, mediante un puntero simple o mediante un doble puntero) y qué ventajas e inconvenientes tenía cada aproximación. La sexta sesión se dedicó a la depuración del código, y la séptima semana fue puesta como fecha límite para la entrega de la práctica en el servidor de prácticas del departamento. Paralelamente, en las clases de teoría ya se estaba impartiendo el tema de genericidad, necesario para implementar la segunda práctica a partir

de la octava sesión.

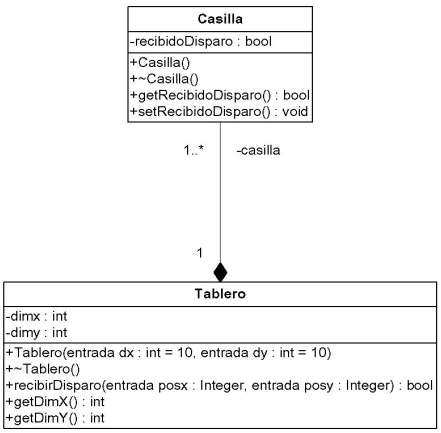

Figura 1: *Diagrama de clases de Tablero*

Dado el gran número de alumnos, para esta primera práctica proporcionamos el el fichero de configuración de *Doxygen* a partir del cual se obtiene la documentación (ver Fig. 2) y el *makefile* con el que se debía compilar la práctica, y que permitió controlar el nombre de los ficheros fuente. La práctica fue posteriormente corregida mediante un *script* automático, donde sustituimos el *main* del alumno por nuestros propios programas principales, desde los que instanciábamos sus clases y comprobábamos aspectos como índices fuera de rango o la correcta implementación de los destructores. Pensamos que este modo de corregir ayudó al alumno a percibir la importancia de la encapsulación, las posibilidades de reuso del código creado, y la necesidad de ajustarse a las especificaciones (interfaces de clase), ya que un error en las mismas impedía que su práctica se ejecutase correctamente. Con el fin de incentivar al alumno, esta práctica puntuó un 10 % de la nota final de la asignatura, y era además totalmente reutilizable para la implementación de la segunda práctica.

La elección de un juego tan conocido por parte de los alumnos como es el *Hundir la Flota*, unido a la gran simplicidad de esta primera práctica, sin duda contribuyó a que el alumno perdiera el miedo a la asignatura, y fue determinante para que un 88 % de los alumnos (602 de 682 matriculados) la entregasen. Estos resultados contrastan con los obtenidos durante el año anterior, donde una sola práctica al final de curso y una menor sincronización entre ambas partes de la asignatura había causado que más del 50 % de los alumnos abandonasen la asignatura en el mes de diciembre.

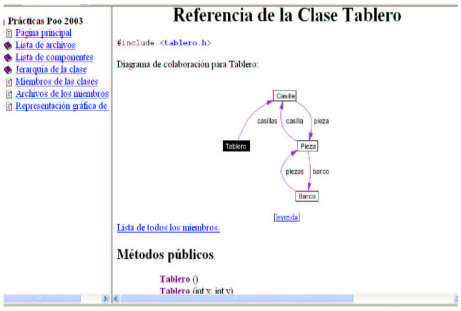

Figura 2: Documentación obtenida con Doxygen

En esta experiencia es importante destacar lo positivo del uso de la herramienta de documentación *Doxygen*. Esta herramienta no sólo es uno de los vínculos de unión explícitos entre nuestra asignatura y la asignatura de *Herramientas de Programación* (donde se profundiza, entre otros aspectos, en las distintas herramientas de documentación existentes), sino que tiene como ventaja adicional la generación de un diagrama de clases como parte de la documentación. De este modo, por un lado el alumno percibe nuevamente la importancia de manejar la notación UML, y por otro se facilita enormemente el contrastar, de una manera visual, si la solución implementada por el alumno coincide con el diseño original, así como averiguar el por qué de las posibles diferencias.

3.2.1. Problemas encontrados

A pesar de lo positivo de la experiencia, nos hemos encontrado con algunos problemas a la hora de la puesta en marcha de esta metodología, que pasamos a detallar a continuación:

- Bajo nivel en el manejo de la sintaxis de C++. El nivel de programación (en cual-

quier paradigma) de los alumnos que acceden a esta asignatura es en general muy limitado. Es por ello que algunos alumnos se han quejado de que se pierden en cuestiones sintácticas relacionadas con los ejemplos presentados en clase, y eso les dificulta entender los conceptos semánticos de la asignatura.

- Confusión por parte de algunos alumnos entre relaciones entre clases y el concepto de *clave ajena* presente en las bases de datos relacionales. Para aquellos alumnos que han cursado previamente alguna asignatura de Bases de Datos Relacionales, se ha detectado que en general resulta más fácil pensar en términos de claves ajenas que pensar en punteros entre objetos.

- Dificultad en la comprensión del papel que no sólo el tipo sino también la cardinalidad de las relaciones y el uso o no de polimorfismo juega en la elección de un tipo u otro de implementación de las relaciones entre clases. Éste es sin duda uno de los puntos más críticos de la asignatura, cuya solución pasa desde nuestro punto de vista por la familiarización del alumno con un mayor número de ejemplos que le permitan comprender los criterios aplicables.

- Masificación de los grupos de práctica. Este hecho dificultó el seguimiento individualizado de los progresos y problemas particulares de cada alumno.

4. Evaluación de la metodología

A los profesores de la asignatura nos interesaba especialmente evaluar el grado de satisfacción del alumnado con respecto a este sistema de trabajo, para lo cual, y nuevamente apoyándonos en los recursos ofertados por el Campus Virtual, se realizó la siguiente encuesta a un conjunto de 40 alumnos correspondientes a un grupo de prácticas aleatorio.

- ¿Qué te ha resultado más intuitivo para la comprensión de los conceptos de Herencia, Polimorfismo, Generalización y Relaciones entre clases?

  1. Ejemplos de código en C++ 2. Trazas de ejecución en C++ 3. Diagramas UML 4. Diagramas UML + trazas de ejecución C++

- ¿Cuánto tiempo te ha llevado comprender la transformación del diagrama de clases en código en C++?

  1. menos de una semana 2. entre una semana y un mes 3. más de un mes 4. aún no lo entiendo

- Una vez comprendido el diagrama de clases, ¿consideras que es útil para visualizar la estructura del código y las relaciones entre clases? ¿Y como vehículo de comunicación entre programadores?

  1. sí, considero que es útil para programar y para comunicarse entre programadores 2. creo que es útil como vehículo de comunicación, pero no para programar 3. creo que es útil para programar, pero no para comunicarme con otros programadores 4. creo que no es útil

- ¿Qué crees que es lo más importante que has aprendido en esta asignatura?

  1. Sintaxis C++ 2. Conceptos teóricos de OO 3. Conceptos teóricos de OO y modo de expresarlos en UML 4. Conceptos teóricos de OO, modos de expresarlos en UML y relación entre sintaxis UML y sintaxis C++

- Una vez finalizada la asignatura, ¿crees que te resultaría difícil implementar un diagrama de clases distinto en C++?

  1. sí, mucho 2. bastante 3. poco 4. nada

- ¿Crees que, conociendo la sintaxis de otro lenguaje de programación (e.j. Java), te resultaría difícil implementar la práctica en ese otro lenguaje?

  1. sí, mucho 2. bastante 3. poco 4. nada

- ¿Cuándo crees que debería enseñarse la relación entre el diagrama de clases y la sintaxis C++?

1. al principio del curso 2. tras haber dado los conceptos teóricos básicos de la OO 3. al mismo tiempo que se van viendo ejemplos de código de cada concepto 4. tras haber visto ejemplos de código de todos los conceptos de OO (herencia, polimorfismo, genericidad, etc)

- ¿Te parece interesante el hecho de aprender conceptos y herramientas que vas a volver a utilizar en asignaturas de cursos superiores?

  1. sí, mucho 2. bastante 3. poco 4. nada

Una vez recopilados los datos, la encuesta muestra cómo a un 52 % de los encuestados el uso de diagramas UML les ha facilitado la comprensión de los conceptos de la asignatura frente a un 39 % que han seguido mejor los ejemplos de código. Pensamos que parte de este porcentaje se debe a las dificultades inherentes a la abstracción de conceptos, y que por tanto se podría mejorar mediante una ejemplificación más exhaustiva. Esta apreciación se corrobora si observamos que el 40 % de los alumnos piensa que lo más importante aprendido en esta asignatura han sido los conceptos de OO y su modo de implementarlos en un lenguaje como C++, frente a sólo un 6 % que considera que lo más importante aprendido es la sintaxis de C++.

Un 88 % de los alumnos encuestados han comprendido la transformación entre diagramas de clases y código en un periodo inferior a un mes, lo cual se adecúa al tiempo dedicado a la primera práctica de la asignatura.

Respecto a la apreciación de UML, un 76 % de los alumnos consideran que UML es un vehículo útil para programar. El 24 % restante considera que es útil como vehículo de comunicación, pero no para programar. Pensamos que este porcentaje podría nuevamente mejorarse con más ejemplos resueltos de implementaciones parciales de sistemas a partir de diagramas.

Un 61 % de los encuestados consideran que la implementación de un nuevo diagrama UML les causaría pocas o ninguna dificultad, frente a un 73 % que considera que implementar el mismo sistema en un lenguaje distinto les resultaría poco o nada dificultoso. Ese 12 % de diferencia, atribuible a las dificultades que han encontrado los alumnos a la hora de entender e implementar las relaciones entre clases, nos hace pensar que aún es necesario hacer un esfuerzo adicional en la discusión interactiva en teoría acerca de la influencia de las características de estas relaciones en la implementación final de código.

Un 73 % de los encuestados coinciden en señalar la necesidad de sincronización entre teoría y práctica por la que hemos apostado en la asignatura. Sorprende constatar cómo sin embargo un 18 % preferiría ver todos los conceptos relacionados con UML al principio de curso, antes de haber abordado dichos conceptos en la asignatura. Pensamos que este dato es debido al distinto nivel con que llegan los alumnos a la asignatura, que hace que aquéllos que ya tienen conocimientos avanzados de programación en C++ prefieran abordar directamente aquella parte que aún no conocen, i.e. la notación UML.

Por último, el hecho de que un 97 % de los encuestados consideren positiva la experiencia de conocer conceptos que serán retomados en cursos posteriores avalan, desde nuestro punto de vista, la aceptación por parte de los alumnos de este enfoque, a pesar de las dificultades añadidas con las que se tienen que enfrentar.

5. Conclusiones

Desde este artículo se ha defendido la idea de que la enseñanza de Programación Orientada a Objetos sin referencias explícitas a los criterios de diseño que deberían dirigir la definición de las clases y relaciones perjudica el proceso de aprendizaje. Por ello, se ha presentado una propuesta metodológica que, sustentada en teorías pedagógicas actuales, se adapta al número de alumnos y medios disponibles en nuestra universidad y complementa las definiciones teóricas y los ejemplos de código con representaciones visuales en notación UML. De este modo se contribuye a que el alumno desarrolle su capacidad de abstracción respecto a la sintaxis específica de un lenguaje deter-

minado. Una ventaja adicional del manejo de esta notación es que se desambigua la especificación de ejercicios y prácticas, y se proporciona una herramienta útil de comunicación entre los propios alumnos. La experiencia en clase y los resultados de la evaluación por parte de los alumnos demuestran la viabilidad del enfoque, a pesar de las dificultades añadidas con las que, somos conscientes, se tienen que enfrentar dichos alumnos.

Nuestra hipótesis, aún no contrastada, es que esta enseñanza temprana además va a influir de manera positiva en la asimilación de conceptos en las asignaturas correspondientes al área de Ingeniería del Software, al permitir enlazar los conceptos de análisis con experiencias de implementación previas. El resultado esperado es una mejor percepción de la importancia de un buen análisis y diseño en la construcción del software. La corroboración o no de esta hipótesis dirigirá sin duda la evolución de la asignatura en próximos años.

Referencias

[1] V. Benedito. *Introducción a la didáctica*. Barcanova, 1987.

[2] T. Budd. *Object-Oriented Programming. 3rd Edition*. Addison Wesley, 2001.

[3] P. Cañal, A. Lledó, F.J. Pozuelos, and G. Travé. Investigar en la escuela: elementos para una enseñanza alternativa. *Cuadernos de Pedagogía*, (294):74–75, 2000.

[4] J.L Castillejo. *Pedagogía Tecnológica*. Ceac, 1987.

[5] C. Coll, J. Palacios, and A. Marchesi. *Desarrollo psicológico y educación*. Alianza, 1991.

[6] F. Hernández. Repensar la educación en las artes visuales. *Cuadernos de Pedagogía*, (312):53, 2002.

[7] L. Irún and V. Fuentes. Vega: un lenguaje para la enseñanza de la programación. In M.Ñicolau and F. Virgós, editors, *IV Jornades sobre l'ensenyament universitari de la informàtica (JENUI)*, pages 416–422. Universitat d'Andorra, 05 1998.

[8] F. Llopis. Propuesta de metodología para un curso universitario introductorio a la programación. In M.Ñicolau and F. Virgós, editors, *IV Jornades sobre l'ensenyament universitari de la informàtica (JENUI)*, pages 427–534. Universitat d'Andorra, 05 1998.

[9] B. Meyer. *Object-Oriented Software Construction. 2nd Edition*. Addison-Wesley, 1997.

[10] M. Page-Jones. *Fundamentals of Object Oriented Design in UML*. Addison-Wesley, 1999.

[11] J.M. Ribó. Hacia una renovación de la enseñanza de la programación básica. In M.Ñicolau and F. Virgós, editors, *IV Jornades sobre l'ensenyament universitari de la informàtica (JENUI)*, pages 403–410. Universitat d'Andorra, 05 1998.

[12] G. Rose. *Visual Methodologies*. Sage, 2001.

[13] J. Gimeno Sacristán. *Autoconcepto, Sociabilidad y Rendimiento escolar*. MEC, 1976.