La complejidad de entender y enfrentar la formación de futuros desarrolladores de *software*

Ramón Ventura Roque Hernández*, Juan Manuel Salinas Escandón* y Adán López Mendoza*

Recepción: 2 de julio de 2014
Aceptación: 29 de octubre de 2014

***Universidad Autónoma de Tamaulipas, México.**
Correo electrónico: rvHernandez@uat.edu.mx; jmSalinas@uat.edu.mx; aLopez@uat.edu.mx
Se agradecen los comentarios de los árbitros de la revista.

Resumen. El desarrollo de *software* es una tarea que involucra complejidad proveniente del área de aplicación del producto final, de los factores técnicos y humanos y de la gestión del proceso. Los docentes formadores de desarrolladores, además de dominar este proceso, deben facilitar el conocimiento del área y propiciar actividades constructivas para ayudar al alumno en las competencias que le permitirán ser un profesionista preparado que contribuya a mejorar la competitividad de las empresas donde laborará.
Palabras clave: complejidad, educación, desarrolladores, *software.*

Understanding and Addressing the Complexity in the Training of Future *Software* Developers

Abstract. Software Development is a complex task due to several factors such as the application domain, technical and human issues, and the administration of the process.
Professors who teach in the Software Development field must be masters in the process as well as be able to provide students with the knowledge through an easier method. It is professors' work to foster constructive activities to involve the students in learning experiences that will help them to develop skills that will lead to improve the competitiveness of the companies where they will belong
Key words: complexity, education, developers, software.

Introducción

Los alumnos de las carreras profesionales de las instituciones de educación superior en donde laboramos como docentes se enfrentan al reto de aprender a desarrollar *software* para diferentes dominios de aplicación. Como lo refieren Anderson *et al.* (1984), el aprender a programar es una meta importante en nuestra sociedad y el comprender cómo adquirir este conocimiento tiene un enorme impacto educacional. Uno de los mayores problemas de nuestra sociedad de alta tecnología es capacitar nuevas y complejas habilidades técnicas como lo es la programación, con el objetivo de educar en esta disciplina a nuestra presente y futura fuerza laboral. Por ello, los maestros de estas asignaturas debemos facilitar esta tarea desde un punto de vista de simplicidad pedagógica. Es conocido que los programadores novatos enfrentan una tarea difícil, pues implica adquirir nuevo conocimiento complejo, estrategias relacionadas y habilidades prácticas. Por lo tanto, el contenido de cursos iniciales en programación debería ser sencillo y aumentado de manera sistemática conforme los estudiantes adquieran experiencia (Robins *et al.,* 2003).

Sin embargo, el desarrollo de *software*, por sí mismo, no es una tarea trivial; está reconocida por los expertos como una actividad cuya complejidad puede sobrepasar las capacidades humanas para manejarla. Produce con ello proyectos fallidos, de baja calidad, incompletos, que no satisfacen los requerimientos del usuario o que exceden el presupuesto considerado inicialmente. En situaciones críticas, las fallas de *software* han sido responsables, incluso, de la pérdida de vidas humanas. A este fenómeno se le denomina en la literatura *la crisis del software*. Según lo describe Glass (2002), por cada incremento de un 25% en la complejidad del problema, existe un incremento de 100% en la complejidad de la solución por medio de *software*.

Por otra parte, ser profesor de estas asignaturas no es algo que cualquiera pueda hacer bien sin preparación. De acuerdo con Chamillard y Braun (2002), uno de los mayores retos a los que se enfrentan los docentes al momento de enseñar a los alumnos de cursos de ingeniería de *software* es proveerles experiencias significativas y que sean útiles al momento de su egreso. Esas experiencias deberían abarcar todas las fases del proceso de desarrollo de *software* y ser tan realistas como sea posible e incluir la incertidumbre y el cambio continuo presente en cualquier proyecto real. A estas experiencias debería agregarse la necesidad de trabajar con otros individuos en un equipo, lo cual puede afectar la moral de algunos estudiantes y también representa un reto al docente al momento de evaluar y calificar. Es por esto que el docente, en su papel de facilitador, se convierte en un medio para que el alumno pueda alcanzar los objetivos del curso desarrollando competencias que, en el caso de las asignaturas afines, son numerosas.

Las actividades realizadas en el transcurso de un semestre académico deben promover que los ahora alumnos sean futuros profesionistas eficientes y de alto rendimiento para que contribuyan a la competitividad de las empresas donde laborarán. ¿Qué estrategias pueden utilizarse para colaborar con los alumnos en su aprendizaje del proceso de desarrollo de *software*? Según Claypool y Claypool (2005), muchos proyectos de *software* utilizados en el currículo de informática carecen del factor "diversión" que se requiere para involucrar a los estudiantes, así como de un realismo práctico que incluya otras disciplinas del área tales como Redes o Interacción Humano Computador. De igual manera debemos preguntarnos: ¿de qué forma los alumnos pueden construir más fácilmente su conocimiento en esta área?, ¿cómo contribuir a que el aprendizaje sea significativo, especialmente en un campo de conocimiento complejo?, ¿cómo se puede contribuir desde el aula a la formación de desarrolladores de *software* capacitados y competitivos? Este artículo aborda estas preguntas y reúne referencias de la literatura, así como algunas de nuestras experiencias como docentes de las materias con este perfil.

1. La complejidad del proceso

1. 1. Desde la perspectiva del desarrollador

El desarrollo de *software* es una actividad que incorpora complejidad de diversas áreas. Una aplicación no sólo debe implementar todos los requisitos funcionales, sino también los no funcionales como por ejemplo robustez, capacidad de respuesta, mantenibilidad, verificabilidad, escalabilidad, seguridad, soporte, monitoreo y recuperación de desastres con el propósito de satisfacer no sólo las necesidades inmediatas de las organizaciones, sino ser a la vez lo suficientemente flexible como para adaptarse a las crecientes y cambiantes necesidades futuras de la mismas (Kumaran, 2010). Es por ello que si un desarrollador de *software* debe realizar una aplicación para automatizar el proceso de generación de una nómina, es su obligación conocer todos los detalles del proceso de nómina con el que va a trabajar. Debe conocer, manejar, e implementar con algoritmia los conceptos de contabilidad, impuestos, leyes fiscales, prestaciones, deducciones, criterios gerenciales, recursos financieros y humanos, entre otros muchos, sin ser contador de profesión ni tener un perfil académico afín a esa área.

Además, debe conocer los métodos de desarrollo de *software*, ya que va a analizar, diseñar, implementar, probar y mantener una aplicación completa. Debe también conocer notaciones y paradigmas de análisis y diseño para documentar y comunicar procesos y resultados, debe conocer por lo menos un lenguaje de programación, un manejador de bases de datos, y las diferentes estructuras de datos que puede utilizar.

Por si esto fuera poco, además debe tener la habilidad de congeniar con los usuarios, pues los usuarios son piezas clave. ¿Cómo desarrollar una buena aplicación sin ellos si al final son quienes convivirán diariamente con ella? ¿Cómo un desarrollador puede carecer de una habilidad empática con los usuarios si son quienes le proporcionarán de primera mano la información del proceso que él pretende automatizar?

Por otra parte, también debe ser capaz de trabajar en equipo, pues los grandes sistemas rara vez se desarrollan por una sola persona. Riemenschneider *et al.* (2002) comprobó que el énfasis en la utilización de equipos para la organización del desarrollo de *software* incrementa la relevancia de la influencia de los colegas de trabajo comparada contra el empleo de herramientas de tecnologías de información enfocadas a la productividad individual. El trabajo colaborativo es,

por tanto, fundamental en todo equipo. Cada integrante debe encaminar su trabajo al mismo objetivo y debe ser capaz de colaborar responsablemente con sus compañeros.

Ahora bien, un buen desarrollador también debe ser versátil. Aunque se trate de un profesionista especializado, las áreas en las que puede trabajar aplicaciones pueden ser variadas. En el ejemplo planteado, después de automatizar una nómina, la siguiente tarea puede ser desarrollar un sistema de facturación, de control de clientes, gestión de inventarios, de operaciones, de pedidos, etc., pero también pueden surgir proyectos en otras áreas como medicina, producción, logística o electrónica. El carácter casi omnipresente del *software* le da al desarrollador un sinfín de posibilidades laborales.

Otra de las características distintivas de un desarrollador es su alta capacidad de adaptación y de rápido aprendizaje que le permiten comprender variados conceptos y aplicarlos en soluciones a problemas de diferentes áreas. Inclusive Gallivan (2004) sostenía la hipótesis de que dos atributos personales como lo son específicamente la tolerancia a la ambigüedad y la apertura hacia la experiencia se encuentran correlacionados con la habilidad para adaptarse a la innovación tecnológica.

Como es evidente, la construcción de *software* implica demasiadas habilidades que hacen muy complejo su proceso. Quien se dedica a esta labor, tiene como tarea cotidiana enfrentarse a esta complejidad y manejarla adecuadamente para obtener productos de alta calidad.

1. 2. Desde la perspectiva del formador de desarrolladores

Los maestros de las asignaturas relacionadas con la Ingeniería de *Software* y Programación no tenemos una labor menos sencilla. Además de idealmente poseer las habilidades ya mencionadas, debemos ser buenos facilitadores del conocimiento para nuestros alumnos y enfrentarnos a los devenires cotidianos del sistema educativo. La buena docencia tampoco es labor fácil. Se agrega un ingrediente más al coctel de la complejidad.

Los maestros de estas asignaturas tenemos que tomar ventaja de los recursos que están disponibles en un momento dado y propiciar situaciones que conduzcan a que los alumnos se enfrenten a escenarios muy parecidos a los que encontrará en el campo laboral. De acuerdo con Damian *et al.* (2006) tanto el enfoque instruccional como las estrategias de evaluación deberían integrar un entorno de trabajo que incluya las habilidades y estrategias requeridas y alinear los proyectos de clase con condiciones contemporáneas y auténticas de desarrollos de *software* globalizado.

Las actividades de la Ingeniería del *software* son disciplinas activas que conducen a productos específicos. Los saberes de esta rama ingenieril no son inertes, sino que su naturaleza es de aplicación constante. Si el maestro considera esto, puede diseñar las actividades de la clase con una dinámica centrada en la práctica, y no en la teoría memorizada, con el objetivo de que el alumno construya sus conocimientos y desarrolle sus habilidades.

1. 3. Desde la perspectiva de contenidos y competencias

En la carrera de Ingeniería en Sistemas Computacionales de los institutos tecnológicos en México o en la Licenciatura en Informática de las universidades estatales, dentro de las materias del área de desarrollo del *software* con énfasis en programación se encuentran: fundamentos de programación, programación orientada a objetos, estructuras de datos, tópicos avanzados de programación, programación web. Por otro lado dentro de las materias con énfasis en ingeniería del *software* se encuentran: fundamentos de ingeniería del *software*, ingeniería del *software* y gestión de proyectos.

Las materias de programación conducen a desarrollar en el alumno habilidades para programar aplicaciones utilizando diferentes herramientas. En estas materias, el alumno debe aprender principalmente (Tecnológicos, 2009): lenguajes de programación, paradigmas, sintaxis, estructuras, instrucciones, estrategias, técnicas, y estilos que sean útiles para solucionar problemas. Sin embargo, también son necesarias las competencias tales como capacidad de análisis y síntesis, organización, planificación, comunicación oral y escrita, búsqueda de información, toma de decisiones, auto-crítica, trabajo en equipo y autónomo, generación de nuevas ideas, entre otras.

En cada uno de los temas teórico-prácticos de estas materias obligadamente se deben resolver múltiples problemas y desarrollar programas relacionados con ellos. Esta tarea que pareciera repetitiva puede matizarse con algunas situaciones añadidas. No obstante, algo que está ausente en todas es la calidad. Según Hilburn (2000), se debería incorporar a las carreras de Ingeniería de Sistemas o Licenciatura en Informática un modelo que incluya una variedad de técnicas de aseguramiento de la calidad para el desarrollo de *software*.

2. Enfrentando la complejidad en la formación de futuros desarrolladores de *software*

En nuestra labor docente nos hemos percatado de que se obtienen buenos resultados si las intervenciones teóricas son reducidas. Hasta el momento utilizamos esporádicamente algunas diapositivas, pero con información muy sustanciosa, más gráfica y sólo con los conceptos más importantes, las cuales se comparten desde un sitio web con los alumnos,

quienes también las realizan para los temas en que participan y las distribuyen con el resto del grupo.

Durante las exposiciones, tratamos de no ser las únicas personas que hablan. Recurrentemente se abre la oportunidad de intervenciones sencillas, no en forma de pregunta directa, sino indirecta. Por ejemplo, si hay un término en inglés, se busca la colaboración para una traducción informal contextualizada. A veces nos asombramos porque hay palabras o frases, que surgen del grupo, que jamás las habríamos asociado con el tema. En algunos casos se logra una confianza pedagógica tal que la exposición es más bien un diálogo donde los alumnos comentan analogías con situaciones que ya conocen o proponen nuevas a la par de nuestra intervención.

Algunas veces se incorporan detalles sutiles en las mismas diapositivas que fomentan un intercambio de opiniones relacionadas con el tema. En otras, durante la exposición se menciona intencionalmente alguna frase con contenido académico engañoso o falso, y quienes están más atentos comienzan una participación a la que se incorpora el resto del grupo.

Con el tiempo hemos agregado a las tareas teóricas actividades tales como realizar mapas mentales, conceptuales, proponer ejemplos asociados a la vida real, buscar las diferencias (y las similitudes), hacer diagramas y acordeones (pequeños resúmenes o recordatorios de contenido).

Si el alumno se involucra, construye mejor sus propias aproximaciones a los temas de la materia, y es capaz de aplicarlas después. Esta participación en las secciones teóricas por parte de los alumnos les permite reflexionar, generar nuevas ideas, realizar críticas, desarrollar el sentido común, externar opiniones y coexistir con otras personas con puntos de vista diferentes. Todas ellas, habilidades importantes en un desarrollador de *software*.

Otro de los ejercicios que resulta muy útil para que el alumno contraste sus construcciones existentes es el estudio de las situaciones excepcionales. Esto se refiere a que, además de abordar los casos más comunes del uso de las instrucciones y estructuras, se analizan las excepciones a la regla general. Esto induce a un nuevo acomodo de las aprehensiones que el alumno ha realizado, y le permite aumentar los límites de su horizonte, ya que considera nuevos posibles escenarios de aplicación de ese conocimiento.

En cuanto a la parte práctica relacionada con el desarrollo de programas hemos encontrado muy útil combinar trabajo individual con trabajo presencial en equipo dentro del mismo periodo de evaluación. Esto va en concordancia a lo establecido en el *Manifiesto para el desarrollo ágil de software,* pues según Pressman (2009) el método más eficiente y efectivo para transmitir información hacia y dentro de un equipo de desarrollo es hacerlo conversando cara a cara.

Esta estrategia ha permitido que los alumnos discutan y colaboren entre ellos para aprender y reforzar ciertas áreas de conocimiento que de manera individual no habían sido tan exploradas. En ocasiones, el lenguaje del maestro no alcanza el sentido de los alumnos, pero es ahí donde resalta particularmente la comunicación entre los compañeros que hablan el mismo lenguaje y que comparten experiencias pedagógicas.

Los proyectos que se les solicitan varían en cuanto al nivel de dificultad, normalmente no se completan en corto tiempo, pero tampoco involucran semanas de trabajo exhaustivo. En ocasiones, van dirigidos hacia una situación en particular (como el desarrollo de *software* para situaciones escolares, empresariales, administrativas, etc.); son ellos quienes deciden el área de aplicación de su proyecto. En un principio, era sorprendente como los alumnos encontraban difícil el proceso de decidir el dominio de su proyecto. Si era individual, era un conflicto propio; si era en equipo, era compartido. Al final, siempre se llega a un acuerdo y se debe trabajar responsable y conjuntamente.

Esta forma de trabajar permite el intercambio de ideas entre compañeros, el desarrollo de liderazgo, la toma de decisiones, la tolerancia, la repartición inteligente y equilibrada de trabajo y la delegación de tareas.

En el *Manifiesto para el desarrollo ágil de software* de Beck *et al.* (2014) se menciona que se van descubriendo mejores formas de desarrollar *software* "haciéndolo" y ayudando a otros a "hacerlo"; a través de eso se valora a los individuos y sus interacciones sobre los procesos y sus herramientas; además se procura trabajar conjuntamente con una documentación mínima pero comprensible y con una colaboración con el cliente por medio de negociaciones. Todo esto si se tiene continuamente una apertura al cambio en el seguimiento del plan de desarrollo.

Por ello, hemos trabajado en el escenario de desarrollar un proyecto real en donde un usuario final expone sus necesidades, y cada equipo desarrolla una solución diferente para el mismo problema. Durante el desarrollo del proyecto hay revisiones periódicas. Al final, se presentan al usuario todas las soluciones y él retroalimenta a los equipos; valoriza el grado de cumplimiento de las necesidades que inicialmente planteó.

Muchas de las características tecnológicas que se les incorporan a los sistemas de los proyectos surgen como inquietud de los alumnos. Ellos pueden buscar la orientación del maestro, investigar y aplicar el conocimiento directamente a la solución en la que trabajan. Los estudiantes de esta manera analizan, diseñan, organizan, planifican, buscan, identifican y crean mientras trabajan con situaciones apegadas a la vida real.

Por otra parte, la práctica pedagógica tradicional en las materias de programación tiende a tomar la siguiente secuencia: se parte del análisis de un problema, luego pasa por una etapa de diseño de una solución y después llega a la implementación de la misma. Una práctica alternativa en la que también hemos participado es obtener diseños o análisis a partir de una implementación particular. Aunque a primera vista esto puede resultar descabellado, en la vida real existe este proceso con los sistemas heredados, que son sistemas antiguos ya implementados, poco documentados, a partir de los cuales se realizan procesos para obtener diseños o especificaciones con el objetivo de comprender mejor su estructura y tal vez re-escribirlos posteriormente.

También resulta de mucha ayuda que los alumnos vean la programación que realizan otras personas, de ahí surgen nuevas ideas y técnicas que ellos personalizan en sus aplicaciones. Una variante de esta práctica es mostrar fragmentos de programas que contienen errores, y solicitarles que los encuentren e indiquen la razón de los mismos, o que realicen las correcciones pertinentes. Esto agudiza su sentido común, y activa su lógica, al mismo tiempo que se ponen a prueba todos los conocimientos adquiridos. En estas prácticas el alumno se enfrenta a situaciones donde puede comparar sus estructuras mentales con otras diferentes, lo que conduce a un reacomodo positivo de conocimientos que tiende a redimensionar sus perspectivas.

En las asignaturas que impartimos, para las evaluaciones, no se depende únicamente de la calificación obtenida en el examen, sino que se toman en cuenta los proyectos, las actividades de clase, y se aplican dos exámenes: el teórico y el práctico. Aunque el teórico se resuelve sin consultar nada, en el examen práctico sí: apuntes, programas desarrollados anteriormente, libros e internet incluso, pueden escuchar música en sus audífonos si ellos lo prefieren. Esta práctica ha sido altamente criticada, pero ha dado buenos resultados. Los alumnos sienten más confianza, al mismo tiempo que están frente a una situación con una fuerte metáfora de la situación laboral en la vida real: cuando un desarrollador está trabajando en una aplicación, puede consultar lo que él desee, y al ritmo que él desee, pero a final de cuentas, él es responsable del producto final que debe entregar completo y correcto en una fecha límite previamente pactada. Esta manera de trabajar en la evaluación práctica también induce al alumno a organizar su material antes del examen. Él se debe asegurar que sus notas estén completas y correctas, y que pueda acceder fácilmente a ellas. No hacerlo equivale a desperdiciar tiempo en las consultas realizadas. En este proceso de organización, los alumnos revisan previamente el material con que cuentan y le dan una estructura que tiene sentido para cada uno de ellos, con lo cual se obtienen también ganancias pedagógicas.

3. Resultados prácticos

Con la finalidad de recabar evidencias de los resultados de la implementación de las prácticas de enseñanza mencionadas, se encuestó por correo electrónico a tres alumnos que en la actualidad laboran profesionalmente como desarrolladores de *software* y han cursado hasta el momento por lo menos cuatro materias universitarias relacionadas con esta área, las cuales han sido impartidas con este enfoque por los autores de este trabajo en la Licenciatura en Informática de la Facultad de Comercio, Administración y Ciencias Sociales de la Universidad Autónoma de Tamaulipas, México. La encuesta consistió en cinco preguntas abiertas:

a) ¿Cuáles son tus comentarios sobre la manera de enseñar a desarrollar *software* en las materias universitarias que has llevado en tu carrera?

b) ¿Cuál ha sido la experiencia más significativa en estas materias?

c) ¿Cuáles son los beneficios de lo que has aprendido?

d) ¿Cuáles son los conocimientos que has desarrollado en clase y que has aplicado en tu empleo actual?

e) ¿Cuáles son los conocimientos que no fueron abordados en clase pero que actualmente son útiles en tu empleo?

Los resultados resumidos se muestran en la tabla 1. Los tres estudiantes coincidieron en que han obtenido beneficios de la manera en la que aprendieron a construir *software*, y que los temas abordados en clase han sido útiles en su entorno laboral. También emitieron comentarios positivos del enfoque utilizado y aceptaron haber tenido un cambio en su apreciación y entendimiento.

Conclusiones

En esta investigación se han presentado experiencias y reflexiones referentes al aprendizaje y la construcción de conocimientos en el área de Programación e Ingeniería del *software* encaminada a la formación de profesionistas competitivos. Se podría pensar al leerlo que en las sesiones de clase todo es miel sobre hojuelas; sin embargo, no es así.

Algunos de los principales inconvenientes a los que nos hemos enfrentado cuando implementamos estas actividades en clases son el tiempo insuficiente en las sesiones, la falta de disponibilidad de los recursos tecnológicos, la poca cooperación y sentido de responsabilidad por parte de algunos estudiantes, la falta de madurez para aceptar nuevas actividades en una docencia tradicionalmente inflexible, los planes de estudio tan ambiciosos que parecieran no ajustarse jamás al tiempo hábil del calendario escolar, y la formación pedagógica tradicionalista que en ocasiones nos hace actuar por inercia.

A pesar de todo esto, debemos reflexionar acerca de la importancia de que el alumno alcance el conocimiento y se apropie de él, asimismo sobre la relevancia de que el futuro desarrollador de *software* conozca los retos que enfrentará en el campo laboral y se entrene para ser un profesionista preparado para resolver eficientemente los problemas del mundo que le rodea. En este proceso, el docente, al igual que el alumno, tiene un rol activo, y debe propiciar actividades grupales e introducir formas de enseñanza que involucren el factor "diversión" (Claypool y Claypool, 2005), pero que a la vez sean aterrizadas, prácticas y significativas que permitan al alumno confrontarse con el mundo real en situaciones que desencadenen procesos de construcción y reajuste de conocimientos. Siempre habrá inconvenientes que sean buenas razones para no adoptar una docencia constructiva, pero la razón de mayor peso para sí hacerlo siempre será cumplir con nuestra labor incesante de formar profesionistas preparados y éticos que contribuyan con su eficiencia a que las empresas sean más competitivas.

Análisis prospectivo

Los formadores de desarrolladores de *software* tenemos un gran reto en los años venideros: debemos encontrar formas novedosas y efectivas de seguir abordando contenidos, desarrollando habilidades y fomentando valores, pero también de enseñar a los alumnos a aprender a enfrentar la inevitable complejidad del cambio y a hacer más eficiente su ciclo de formación no solamente en las aulas sino en la vida misma.

Es necesario reflexionar sobre la tendencia de los sistemas de *software* a ser cada vez más grandes y complejos para satisfacer necesidades mayores cada vez. La aparición continua de nuevas tecnologías como plataformas, lenguajes y sistemas operativos obliga a que los profesionistas de esta área estén en constante capacitación. Sin embargo, no solamente los conocimientos técnicos son suficientes para solucionar problemas; también se requiere de una forma de pensamiento dispuesta a descubrir y valorar nuevas alternativas. ¿Cómo serán los sistemas de *software* del futuro? ¿Qué nuevos conocimientos técnicos se requerirán para su creación? Las respuestas concretas no las tenemos aún. Lo que sí sabemos es que la enseñanza, el aprendizaje y la creatividad serán elementos que dotarán de herramientas poderosas a los futuros desarrolladores de *software*.

Si nosotros somos incapaces de formar desarrolladores contestatarios al entorno de complejidad y cambio, ellos no serán capaces de seguir el acelerado ritmo del avance tecnológico de los próximos años. En este escenario, los futuros desarrolladores estarían en desventaja pues su productividad, eficiencia y creatividad se verían afectados negativamente mientras que nuestra práctica docente quedaría anclada en modelos tradicionalistas obsoletos. La importancia estratégica y económica para nuestra sociedad de impulsar una fuerza laboral creativa, eficiente, efectiva y productiva desde el aula resulta obvia en este contexto.

Tabla 1. Resultados de las encuestas realizadas a los alumnos.

	Comentarios sobre la manera de enseñanza del desarrollo de *software*	Experiencia más significativa en las materias con este enfoque	Beneficios de lo aprendido con este enfoque	Conocimientos desarrollados en clase y aplicados en su empleo actual	Conocimientos no vistos en clase que actualmente se usen en su empleo actual
Estudiante 1 (8° semestre)	Los métodos de enseñanza son buenos, aun cuando en el mismo semestre se utilizan varios lenguajes a la vez.	Obtuvo un panorama muy amplio de lo que se puede lograr en el desarrollo de aplicaciones.	Muchos. Gracias a lo aprendido, actualmente se encuentra laborando en el área de desarrollo de *software*.	Sintaxis, buenas prácticas como sangría en el código y depuración. Modelo MVC.	Herramientas y lenguajes para desarrollo web. Nuevas bases de datos como NOSQL. Herramientas de programación en equipo
Estudiante 2 (7° semestre)	Con esta metodología se ha avanzado mucho en comparación con generaciones anteriores.	Antes de llevar programación de esta manera, se le hacía muy difícil ver el código y tratar de comprenderlo.	Le ha beneficiado en muchas formas. Por ejemplo, ahora ve su futuro dedicándose al desarrollo de *software*.	Programación orientada a objetos y la sintaxis de los lenguajes estudiados.	Programación con micro controladores. Sistemas de control de versiones.
Estudiante 3 (5° semestre)	Interactivo, divertido. Se aprende a ritmo moderado. Se aplica rápidamente lo aprendido.	El aprendizaje profundo de los errores le ha permitido no cometer un mismo error varias veces.	Programar le hace ver el mundo diferente. Lo que antes parecía ordinario pasa a ser algo fascinante. La programación cambió su forma de ver el mundo.	Programación orientada a objetos, variables, bases de datos, eficiencia, diagramas.	Bases de datos en línea y acceso a ella desde un programa externo a la red. Verificación de requisitos de instalación mediante *software*.

Fuente: elaboración propia.

Bibliografía

Anderson, J., Farrel, R. y Sauers, R. (1984). Learning to program in lisp. *Cognitive Science, 8*: 87-129.

Beck, K., Beedle, M., Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Greening, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R., Mellor, S., Schwaber, K., Sutherland, J., Thomas, D. (2014). *Manifesto for ggile software development.* Disponible en http://agilemanifesto.org/. Consultado el 1 de junio de 2014.

Chamillard, A. y Braun, K. (2002). *The software engineering capstone: Structure and tradeoffs.* New York.

Claypool, K. y Claypool, M.(2005). *Teaching software engineering through game design.* Portugal.

Damian, D., Hadwin, A. y Al-Ani, B.(2006). *Instructional design and assessment strategues for teaching global software development: A framework.* New York.

Gallivan, M. (2004). Examining IT proffesional's adaptation to technological change: The influence of gender and personal attributes. *ACM SIGMIS Database, 35*(3), 28-49.

Glass, R.(2002). Sorting out software complexity: Underlying complexity escalates exponentially. Some little-known research findings. *Communications of the ACM, 45*(11).

Hilburn, T.(2000). *Software Quality: A Curriculum Postscript?* New York.

Kumaran, A. (2010). *Why software engineering is complex?* Disponible en: http://java.dzone.com/articles/why-*software*-engineering. Consultado el 1 de mayo de 2014.

Pressman, R. (2009). *Software engineering: A practitioner's approach* (6th ed). McGrawHill.

Riemenschneider, C., Hardgrave, B. y Davis, F. (2002). Explaining *software* developer acceptance of methodologies. A comparison of five theoretical models. *IEEE transactions on software engineering, 28*(12).

Robins, A., Rountree, J. y Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer Science Education, 13*(2): 137-172.

Tecnológicos, I. (2009). *Plan de estudios de la materia. Programación orientada a objetos-enfoque por competencias.*