La enseñanza tradicional en Ingeniería de Software se suele realizar con clases expositivas y proyectos prácticos. Sin embargo, las estrategias tradicionales de enseñanza, en ocasiones, no suscitan la motivación requerida por los estudiantes, puesto que la seriedad de los temas genera verdaderos mitos al respecto. En este libro, se aborda la Ingeniería de Software con una propuesta humorística, que busca desmitificar estos temas y complementar, de esta forma, su enseñanza.

Los autores

Carlos Mario Zapata es PhD en Ingeniería de la Universidad Nacional de Colombia. Actualmente se desempeña como profesor Asociado en la Escuela de Sistemas de la Universidad Nacional de Colombia, Sede Medellín

Yris Olaya es PhD en *Mineral Economics* de *Colorado School of Mines*. Actualmente es profesora Asistente en la Escuela de Sistemas de la Universidad Nacional de Colombia, Sede Medellín

Ingeniería de Software para Analistas – C. M. Zapata – Yris Olaya

Ingeniería de Software

Carlos Mario Zapata

Yris Olaya

Prólogo de Oscar Pastor

Ingeniería de Software

Carlos M. Zapata J.
Yris Olaya M.

Ingeniería de Software
Para Analistas

CARLOS MARIO ZAPATA J.
YRIS OLAYA M.

ISBN: 978-958-44-2274-3

INGENIERÍA DE SOFTWARE PARA ANALISTAS

Carlos Mario Zapata J.
Yris Olaya M.

Editor-Autor: Carlos Mario Zapata J.

DERECHOS RESERVADOS
Queda prohibida la reproducción o transmisión
total o parcial del texto de la presente obra
bajo cualesquiera formas, electrónica o mecánica,
incluyendo fotocopiado, almacenamiento en un
sistema de recuperación de información, o
grabado sin el consentimiento previo y por escrito
del editor.

Todas las imágenes, excepto la de la página 27,
fueron creadas por Nito
y están protegidas por derechos de autor.

Datos para Catalogación Bibliográfica:
Zapata, Carlos Mario y Olaya, Yris
Ingeniería de Software para Analistas

ISBN: 978-958-44-2274-3

Esta obra se terminó de imprimir en Noviembre de 2007
en los talleres de LITONUEVE, Medellín.

Impreso en Colombia
Printed in Colombia

DEDICATORIA

A Vicky, Sebas y Pipe,
y el gran cuento de la vida real que representan,
porque son los que motivan la realización de todos mis sueños.

Carlos M.

A Babas y Horacio.

Yris

AGRADECIMIENTOS

Una obra de esta naturaleza no puede nacer sin la credibilidad de muchas personas. Sin su aliento constante, no hubiera sido posible llevar a cabo esta empresa. Muchas gracias a María Teresa Berdugo, Gloria Giraldo y Fernando Arango, quienes apoyaron esta iniciativa de principio a fin, escuchándonos los cuentos que teníamos por contar y retroalimentando todo lo que hacíamos. Gracias también a Oscar Pastor, por sus palabras de aliento y por ser el primer lector de nuestro trabajo. También, muchísimas gracias a las generaciones de estudiantes que nos han permitido desarrollar nuestras ideas; esperamos que sea para ellos una obra de consulta permanente.

Deseamos expresar también nuestro sentido de gratitud a la Escuela de Sistemas de la Facultad de Minas, Universidad Nacional de Colombia sede Medellín, y a la DIME (Dirección de Investigación de la Sede Medellín), por el apoyo que hemos recibido en los diferentes proyectos de Investigación que han nutrido nuestras ideas y nos han dado la oportunidad de ensayarlas.

PRÓLOGO

Son muchas las obras que, en las últimas décadas, han sido escritas con el fin de orientar adecuadamente a los estudiantes de Ingeniería del Software. En un contexto tan competitivo, es lógico pensar que parezca imposible poder aportar una obra original, con una perspectiva moderna, productiva y exitosa. En mi opinión, este libro que están a punto de leer tiene la gran virtud de haber conseguido satisfacer ese desafío de una manera ejemplar, y ese es justamente su gran valor: desde la explotación de un principio tan juicioso y universal como es el sentido del humor, se plantea el aprendizaje de los principios básicos de la Ingeniería del Software, desde una perspectiva que conjuga eficazmente rigor y frescura en la exposición, formalismo y simplicidad, precisión en los conceptos y simpatía en su exposición. Todo ello conforma un trabajo que es, a la vez, interesante y original, y que aporta a la docencia de la Ingeniería de Software un conjunto de valores ciertamente relevantes. Tengo la certeza de que los estudiantes aprenderán con su lectura muchos de los hábitos que deben de acompañar el ejercicio de una Ingeniería del Software profesional, bien fundada y rigurosa en sus conceptos básicos y en sus técnicas.

Como decía antes, me gustaría destacar el acertado uso presente en esta obra del tan necesario sentido del humor. Una dosis de humor facilita la comprensión de principios que deben de estar presentes en la Ingeniería del Software moderna. La original exposición de problemas y soluciones, hace que el lector se sienta parte del escenario retratado. Independientemente del rol jugado—ya se trate del "interesado", del "analista-diseñador" o del "desarrollador"—uno se siente retratado de una u otra forma en los problemas descritos, y siente, en consecuencia, próxima y necesaria la solución presentada. ¿Qué más se le puede pedir a un libro orientado a la docencia de una Ingeniería del Software profesional y adecuada a los desafíos que se le exige resolver?

En definitiva, lean y juzguen por ustedes mismos. Les vaticino el disfrute de la lectura de una obra que les va a resultar entretenida y edificante, y con la que van a poder tanto aprender—los estudiantes—como repasar—los que ya tengan conocimientos en la materias—los principios básicos de la Ingeniería del Software con una sonrisa. Tremendo desafío, inteligente y eficientemente resuelto por los autores.

Oscar Pastor López
Octubre de 2007

TABLA DE CONTENIDO

PARTE I

Personajes (En orden de aparición)

El Software
El Interesado
El Analista-Diseñador
El Desarrollador

CAPÍTULO 1: EL SOFTWARE

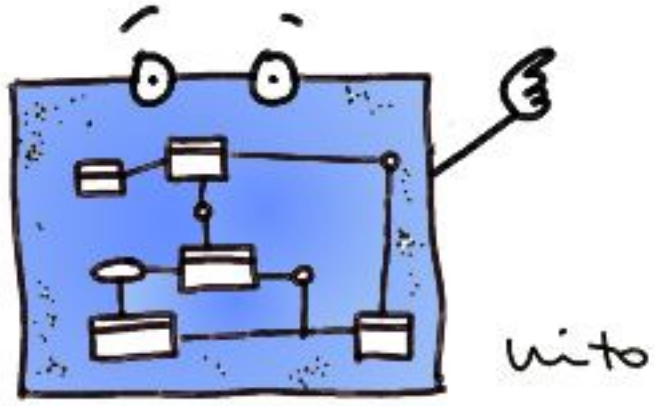
¿Quién no tiene que ver con aplicaciones de software hoy en día? Basta con que el lector se levante temprano en la mañana y se dirija rápidamente a realizar la primera obligación del día: leer el correo electrónico. O tal vez sólo desee pagar algunas cuentas o leer su periódico favorito. A partir de ese momento comienza un conjunto de interacciones con las llamadas "Aplicaciones de Software".

El asunto no se detiene allí. Suponga que el lector es un sufrido estudiante de una universidad cualquiera y lo han citado ese día para realizar el registro en línea de sus asignaturas; en el momento del registro está interactuando con uno de los tipos más comunes de software: las aplicaciones de bases de datos. Así por el estilo, casi cualquier actividad que realiza cada persona en el mundo está ligada con las aplicaciones de software.

¿Cómo saber que se está interactuando con una aplicación de software?

No todas las aplicaciones de software están condenadas al fracaso. Sin embargo, las fallas en este tipo de aplicaciones son muy notorias. Mientras el software que sincroniza los semáforos de la ciudad funcione, nadie lo notará porque las luces alternarán entre rojo y verde; si, por el contrario, este software falla, los semáforos se quedan parpadeando en amarillo y el tráfico se convierte en un caos: cualquiera puede notar el problema.

Lo que busca la ingeniería de Software es evitar los problemas asociados con el software de mala calidad. Desarrollar software de mala calidad es, por lo general, fácil, pero, si se aplica un método organizado al proceso de desarrollo, es posible evitar las malas prácticas y crear aplicaciones y sistemas de software confiables y que no fallen.



Hola. Soy el software; seguramente me conocen como "sistema", "aplicación" o "programa". Acompáñenme a aprender cómo diseñar software de forma metódica y eficiente.

Con este libro, se pretende que el lector aprenda los fundamentos de la Ingeniería de Software y los aplique para mitigar los efectos adversos que tienen las malas prácticas en el desarrollo de software.

CAPÍTULO 2: EL INTERESADO

¡Se perdió la nómina!

El usuario final es la persona que más se queja del funcionamiento de la aplicación: “este programa se bloquea mucho” o “yo pedí otra cosa, pero lo que me instalaron fue este programa”.

El interesado (*stakeholder*, en inglés) tiene algún interés en la elaboración de la aplicación de software, ya sea en el funcionamiento (como usuario final), en la información (como administrador de la base de datos), en el costo y el tiempo de entrega (como cliente) o simplemente para evitar que con la aplicación se evadan los impuestos (como Gobierno).

Cuando una aplicación de software falla, es probable que en su proceso de desarrollo se haya olvidado considerar la opinión de alguno de los interesados,

o que se haya tergiversado lo que tenía que decir en relación con el software, o simplemente, que no se haya entendido porqué necesitaba la aplicación.

Por lo general, el interesado no tiene la más remota idea de lo que necesita para solucionar sus problemas. Ésta es sólo la punta del *iceberg,* pues es necesario que el grupo que está desarrollando el software comprenda el entorno del interesado, su vocabulario, sus necesidades y expectativas en relación con la aplicación que está desarrollando. La información que entrega el interesado suele ser poco específica, pues éste, por lo general, tiene poco tiempo para hablar ("ya te atiendo" en el idioma del interesado puede representar una larga espera a la entrada de una oficina) y confía en que, con lo poco que diga en relación con sus problemas, necesidades y expectativas, el software haga cualquier cosa que está pensando.

El interesado es, en esencia, un idealista que suele pensar: "No es posible que agregarle un simple botoncito a esa pantalla cueste ese dineral!!!!". En su mundo de sueños nada es imposible y todo es gratis. Eso sumado a que necesita todo para mañana ("lo que yo quiero es que el software me haga esto, lo otro y lo de más allá… ¿Cierto que mañana podemos hacer una cita para ver algunas pantallitas de lo que me están haciendo?"). Por esas características especiales, el interesado es la fuente número uno de rechazos al momento de las entregas parciales o totales de software. Las cosas nunca son como las estaba esperando, y el culpable siempre es el grupo de desarrollo, nunca la escasa información que brinda. De este personaje, dependerán en gran medida los ingresos de una empresa desarrolladora de software, razón por la cual es necesario que se hagan todos los esfuerzos necesarios para capturar la mayor cantidad de información de él, con el fin de no tener tropiezos durante el desarrollo de cualquier aplicación de software.

CAPÍTULO 3: EL ANALISTA

El Analista X se enfrenta a un problema.

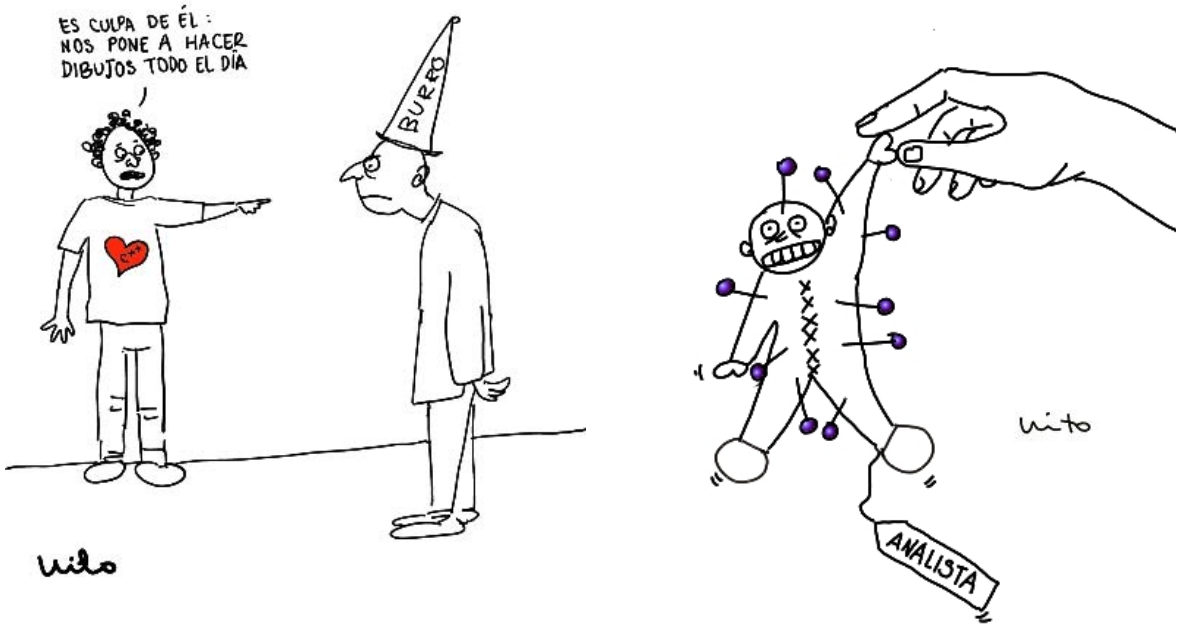
La aplicación de un proceso disciplinado para el desarrollo de software necesita un protagonista: el Analista. Este sufrido cargo surgió en los años setenta cuando un comité de la OTAN (Gibbs, 1994) decidió que las aplicaciones de software deberían dejar de ser productos informales de las ideas de un grupo reducido de "genios" (véase el capítulo 4: el desarrollador) y pasar a convertirse en verdaderas obras de Ingeniería, con planos que detallaran su funcionamiento interno.

Los analistas son el puente entre los interesados y el software. Ellos deben traducir las necesidades y expectativas de los interesados en diagramas y representaciones lógicas que después los desarrolladores puedan traducir, mediante lenguajes de programación, en aplicaciones de software.

Para los analistas, interactuar con los interesados durante el proceso de desarrollo del software es una necesidad imperiosa, aunque ello no implica que sea una labor fácil. Y es una labor difícil porque al analista le toca hacer de consejero, psicólogo, médico, sacerdote o padre de familia, para escudriñar en la mente del interesado y descubrir lo que finalmente servirá como base para el desarrollo de la aplicación. Los interesados son personas volubles, cambian de parecer día a día y van cambiando sus necesidades y expectativas

a medida que va pasando el tiempo. Con estas características, y las demás descritas en el capítulo 2, tienen que lidiar los analistas para poder llegar a obtener un conjunto de diagramas, que le sirvan de insumo a los desarrolladores para su trabajo.

Es necesario, en este punto, hacer una aclaración. Existe en el grupo de desarrollo de software un cargo adicional: el de diseñador, el cual se encarga de acercar los resultados que entrega el analista a los lenguajes de programación, para que luego el desarrollador produzca el resultado final. Dado que su función conceptual es bastante similar, en este libro se utilizará el término "analista" para referirse a ambos cargos (al fin y al cabo, los insultos que reciben de los demás actores son bastante similares). El karma del analista-diseñador es la responsabilidad de los retrasos de los proyectos, el incremento en los costos y todas las malas energías que rodean los proyectos de desarrollo de software.



Parece increíble que, con este oscuro panorama, aún existan personas con alma de mártir que se adentren en el mundo de los modelos para procurar que el software se realice de manera disciplinada y metodológica. Este libro es un homenaje a tales personas, futuros Ingenieros de Sistemas e Informática, con alma de Ingenieros de Software, que confían en alcanzar la mezcla perfecta de ingeniería y arte, que aumente la calidad de las aplicaciones de software.

CAPÍTULO 4: EL DESARROLLADOR

¡Pero no me dibujen con afro!

Este personaje solitario e introvertido, generalmente con una inteligencia que bordea la genialidad, tiene la difícil tarea de convertir las elucubraciones febriles de los interesados, que los analistas traducen en diagramas incomprensibles, en aplicaciones de software con la calidad suficiente como para que el Cliente, el mayor de los interesados, diga al menos "Bueno, se parece un poco a lo que pedimos, aunque necesita unos pequeños ajustes…".

Es casi un milagro que la aplicación tenga al menos un parecido lejano con las ideas del interesado o con los diagramas del analista. Esto es porque, aún estando en la misma oficina que uno, los desarrolladores prefieren comunicarse por medio del *chat* y con un lenguaje repleto de *class, void, public, string, new, http, cat* y otras palabras que sólo tienen sentido en el

oscuro mundo de la programación y que los simples mortales (como el interesado) no suelen entender.

Antaño, el desarrollo de software era el producto de extrañas conversaciones entre interesados y desarrolladores, en las cuales los interesados procuraban expresarle a los desarrolladores qué era lo que querían en términos de pantallas e informes. En esa época dorada de la programación, los desarrolladores tenían atribuciones casi infinitas en términos de las aplicaciones de software. Sus palabras eran incontrovertibles y los interesados preferían cambiar sus organizaciones antes que desafiar la ira de los desarrolladores, quienes eran los únicos capaces de adentrarse en la maraña de signos extraños que representaba el código de los programas, para realizar algún cambio solicitado, tímidamente, por los interesados. Para colmo, las aplicaciones de software eran completamente dependientes del desarrollador que las hubiera creado; en su ausencia, la única opción viable para el software que se quisiera actualizar era la extinción.

En la actualidad, las cosas han cambiado. Los interesados se han dado cuenta de su importancia en el ciclo de vida del software, como promotores de las ideas y poseedores del capital para su desarrollo. Por otro lado, el desarrollador ya no es el amo y señor del proceso puesto que, debido a la capacidad de mantenimiento y adaptación que requiere el software, el código se debe elaborar actualmente de forma tal que no sólo quien lo haya desarrollado pueda comprenderlo. Este libro constituye una introducción a la terminología de este mundo difícil del desarrollo de software, pasando por el modelado, el ciclo de vida del software y otros términos que conforman la “Ingeniería de Software para analistas”.

PARTE II
CONCEPTOS

CAPÍTULO 5: ALGUNOS CONCEPTOS FUNDAMENTALES

SECCIÓN 5.1: MODELO

El concepto “Modelo” se ha ligado, históricamente, con la ingeniería. Hay muchos tipos de modelos: matemáticos, físicos, gráficos, etc. Los modelos, en ingeniería, son una representación de la realidad, construidos con el fin de estudiar las características de un determinado fenómeno. Esta representación de la realidad se debe concentrar sólo en aquello que es realmente importante para el análisis del fenómeno; por ello, contiene unos pocos elementos de la realidad compleja. Así, si se trata de un modelo matemático, la realidad se expresará mediante ecuaciones, variables y operadores. Algo similar ocurre con el modelo físico.

En Ingeniería de Software, los modelos pueden ser formales y semiformales. Los modelos formales son expresiones de tipo matemático y lógico, y cuyo significado en ocasiones se reserva a las personas con amplio conocimiento en esas áreas. Los modelos semiformales suelen ser conjuntos de diagramas que

tienen una sintaxis definida (que se puede expresar también en forma de expresiones textuales) y cuyas características las pueden o no comprender los interesados.

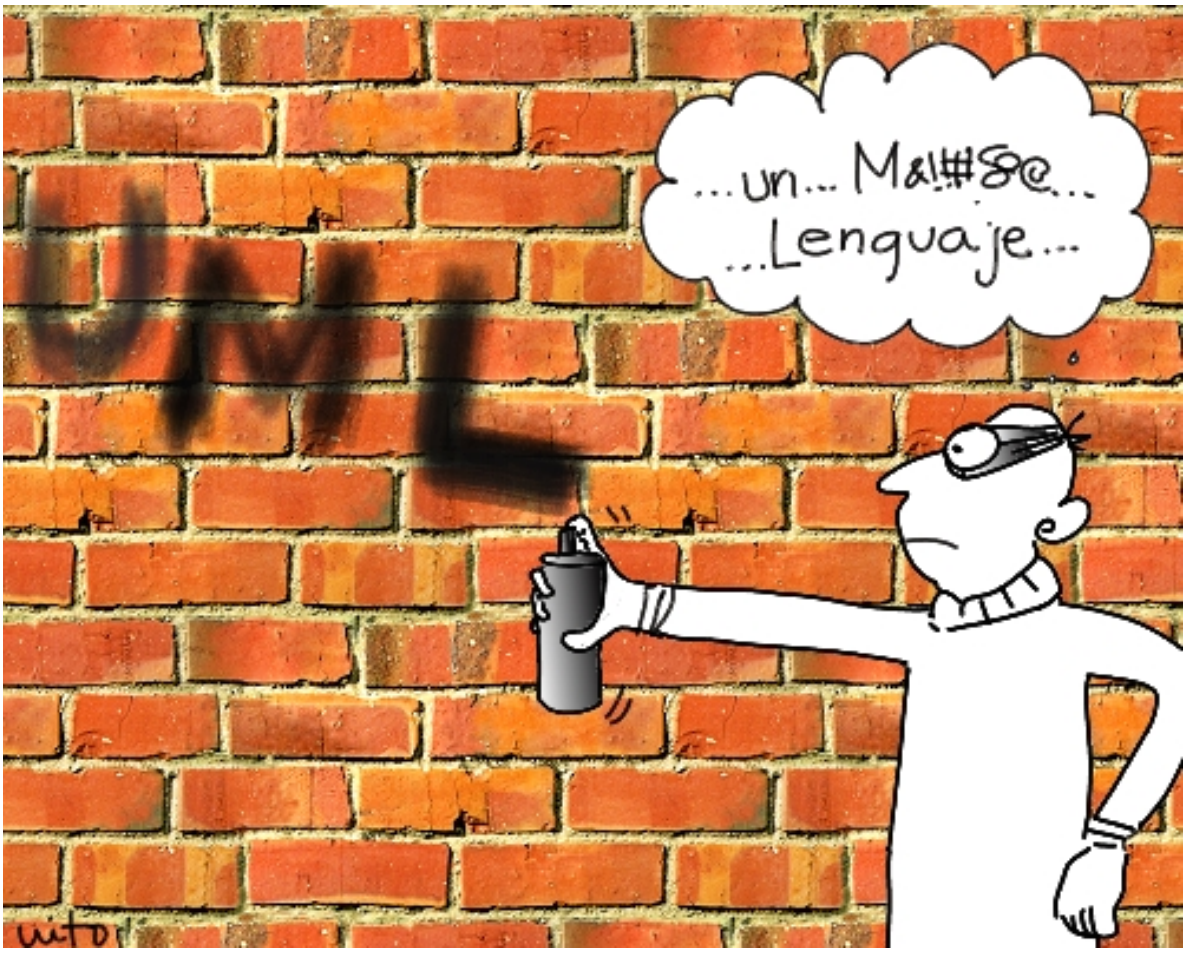
Los modelos de Ingeniería de Software son los "planos" de la aplicación de software que se está desarrollando. Debido a una larga tradición de desarrollo de software sin la utilización de diagramas, los detractores de los desarrollos disciplinados y metodológicos de software cuestionan continuamente los modelos, pues aducen que son una carga pesada a lo largo del proceso de desarrollo. La creencia de que los modelos únicamente consumen tiempo del proyecto y no contribuyen a obtener una aplicación de calidad es muy generalizada, en especial en las pequeñas empresas de desarrollo. Esta posición está muy lejos de la realidad. Si se ejecutan adecuadamente las etapas de modelado, se puede adquirir una idea muy cercana del funcionamiento futuro de la aplicación de software; paulatinamente, se puede acercar la realidad del interesado a la aplicación de software que deberá elaborar el desarrollador.

El analista está encargado de la elaboración de los modelos.
¡Esto no es un juego de niños!

El analista es, comúnmente, el responsable de la elaboración de los modelos, en un proceso caótico y lleno de dificultades que se describirá con mayor amplitud en las secciones 6.2, 6.3 y 6.4. Por ahora, basta con decir que el analista debe conocer en profundidad la organización para la cual se está elaborando la aplicación de software, a tal punto que, en algún momento del proceso, se convertirá en la persona que tiene más conocimiento del área y la problemática para la cual se está realizando el desarrollo.

Por lo general, se presentan confusiones entre los términos "modelo" y "diagrama". Un modelo es un conjunto de diagramas que describe una porción de un dominio específico. Un diagrama es una vista parcial de un modelo, es decir, un subconjunto de las características específicas del modelo; los diagramas generalmente se elaboran en un lenguaje de modelado (ver sección 5.2).

SECCIÓN 5.2: LENGUAJE DE MODELADO



Los diagramas, que constituyen los modelos, se deben expresar en una sintaxis especial, que está dada por los lenguajes de modelado. Desde los inicios de la Ingeniería de Software, han existido varios lenguajes de modelado, que han definido los diferentes tipos de diagramas que se pueden realizar. Por ejemplo, la definición que hizo Peter Chen del diagrama entidad-relación (Chen, 1976) se puede considerar un lenguaje de modelado especial para el manejo de las bases de datos. Otro ejemplo de lenguaje de modelado, un poco más completo porque se refiere a varios diagramas, es la propuesta de orientación a objetivos organizacionales de la especificación denominada KAOS (Lamsweerde *et al.*, 1993).

El lenguaje de modelado más conocido es el UML, sigla de *Unified Modeling Language* o Lenguaje Unificado de Modelado. Este lenguaje se originó a finales del milenio pasado, cuando los denominados "tres amigos" (Grady Booch, Ivar Jacobson y Jim Rumbaugh; Booch *et al.*, 1999) unieron sus trabajos previos para conformar el estándar más aceptado en la actualidad para la elaboración de diagramas.

UML ha evolucionado desde su creación. La versión actual, 2.0, es una verdadera pesadilla para los analistas novatos por el carácter semiformal que posee y por la gran cantidad de diagramas que define (13 en total), algunos de los cuales tienen muy poca utilidad hasta ahora en el desarrollo de software, porque los dominan únicamente un puñado de analistas en el mundo. Los diagramas más conocidos (clases, casos de uso, secuencias y máquina de estados) siguen siendo los más trabajados en el ciclo de vida del software y, prácticamente, constituyen la totalidad de documentación que se elabora en la mayoría de las empresas de desarrollo de software, por lo menos en Colombia.

A pesar de que el documento que soporta la especificación de UML[1] es bastante completo, aún se pueden encontrar muchos vacíos en su elaboración. Por la forma en que está escrita la superestructura, se requiere un conocimiento muy profundo en modelado para comprender las especificaciones generales que conducen a los diferentes diagramas. La mayoría de los lectores preferirá usar el lenguaje y no tener que ver con la especificación. Sin embargo, tampoco hay un consenso general en el uso del lenguaje.

[1] Este documento se denomina "Superestructura de UML" y está disponible, de manera gratuita, en la página web del *Object Management Group*, encargado de la liberación de nuevas versiones del lenguaje

Basta con hacer un ejercicio de modelado en UML, partiendo de un enunciado simple en lenguaje natural y luego solicitando la realización de los diagramas más conocidos de UML, para descubrir que el estándar no lo es tanto: como resultado de este ejercicio, se obtienen tantos diagramas diferentes como participantes accedan a realizarlos. Este resultado, que puede parecer inquietante a la luz de la Ingeniería de Software, se origina también en la existencia de múltiples soluciones de software a un mismo enunciado (por lo general se podrían considerar infinitas soluciones a un mismo problema), pero ello no deja de develar uno de los mayores problemas de esta disciplina: el carácter "blando" de sus soluciones.

En el caso de ciencias exactas como la Física y las Matemáticas, los resultados a partir de un mismo enunciado son predecibles y repetibles; en Ingeniería de Software eso no suele ser posible. Lo que olvidan los detractores es que la Ingeniería de Software se basa en la interpretación y traducción y no en un conjunto de leyes y axiomas. A este respecto, los autores se limitan a opinar que la formalización y el carácter repetible de la ingeniería de Software llegarán con el tiempo.

SECCIÓN 5.3: ESTÁNDAR DE CALIDAD

Una buena aplicación de software debería ser el resultado de un proceso impecable de realización. Las empresas que no siguen estándares de calidad son más comunes de lo que se piensa, y eso muchas veces puede marcar enormes diferencias en los costos de desarrollo de una aplicación. No obstante, el costo directo no debería ser el único parámetro para comparar dos aplicaciones porque, cuando no se considera el costo de mantenimiento, (véase la sección 6.7 para mayor información al respecto) la aplicación que se

está adquiriendo o contratando puede parecer mucho más cara de lo que realmente es.

¿Qué es un estándar de calidad? Un estándar de calidad es un conjunto de lineamientos o normas diseñadas para que las aplicaciones de software se puedan catalogar como “buenas” al final del proceso.

Ahora, es posible que algunas aplicaciones alcancen ciertos aspectos de calidad sin aplicar los estándares necesarios para ello. Es posible, por ejemplo, que los creadores de cualquiera de los portales de búsqueda más famosos de la actualidad, encerrados en su garaje, no hayan aplicado sistemáticamente los estándares de calidad y nadie duda lo exitoso que fue el resultado de su invención.

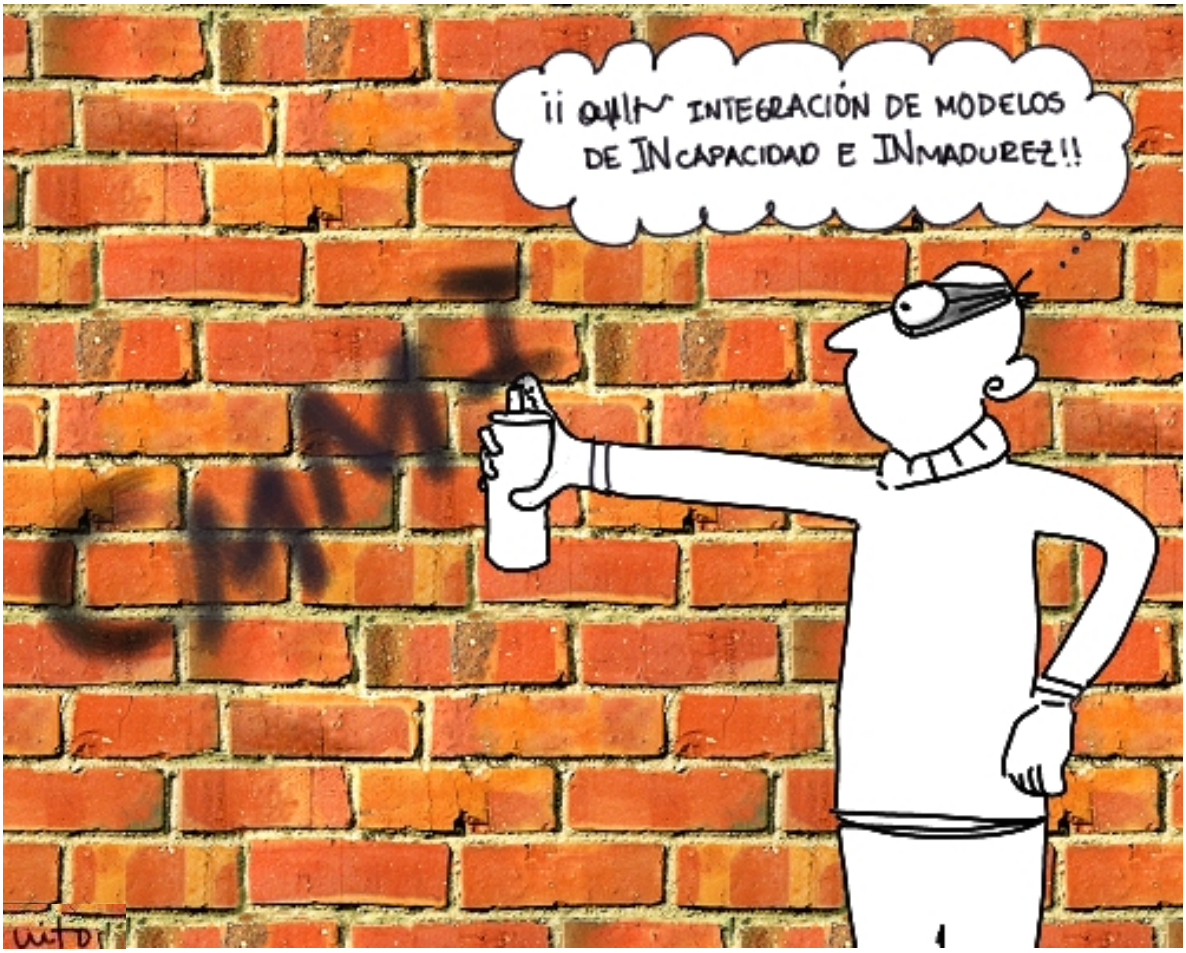
Como este tipo de resultados es la excepción y no la regla, es preferible aplicar los estándares de calidad y buscar que las aplicaciones sean “buenas”, sin depender de golpes de suerte.

Por sí solos, los estándares de calidad no garantizan que la aplicación de software, al final, sea considerada como “buena”. En la disciplina de la Ingeniería de Software, existe gran cantidad de riesgos e imponderables que pueden echar por tierra las aspiraciones de tener una “buena” aplicación, aún con el uso adecuado de los estándares de calidad. No queda otra cosa que hacer, sino contribuirle a la suerte con un buen proceso de desarrollo; de otra manera, lo que sí estará garantizado es el fracaso de la aplicación.

Existen numerosos estándares de calidad en el mundo. Quizá el lector conozca la familia de estándares ISO 9000, un conjunto de normas internacionales para garantizar la calidad del proceso de desarrollo, la cual generó un *boom* internacional por la certificación de los procesos a finales del milenio pasado y principios del presente milenio. Estas normas, se aplicaban a empresas de diferentes tipos de productos y servicios, pero no eran especialmente adaptadas para empresas de software. En este tipo de empresas, la aplicación de los estándares ISO se restringía a algunas de las exigencias respecto al proceso mismo de desarrollo.

La ISO desarrolló también algunos conjuntos de normas de calidad que, como la norma ISO 9126 (ISO, 2001), definieron la terminología para describir la calidad en un producto de software. De estas normas, salieron términos impronunciables y poco castizos como *usabilidad, portabilidad,*

mantenibilidad, recuperabilidad, comprensibilidad y otras "ilidades" tan confusas como sus nombres. Existen también otros estándares igualmente importantes, como el de calidad del software en uso, denominado ISO 14598-1 (ISO, 1999) y la calidad en uso y *usabilidad* (ISO 9241-11), en las cuales se definen los aspectos fundamentales en la interacción son aplicaciones de software.



En la actualidad, el estándar de calidad que revoluciona la ingeniería de software se denomina CMMI, por sus iniciales en inglés: *Capability Maturity Model Integration* o Integración de Modelos de Capacidad y Madurez. Este estándar no se enfoca en el proceso o en el producto, como los anteriormente descritos, sino en la capacidad y madurez de la empresa que desarrolla software.

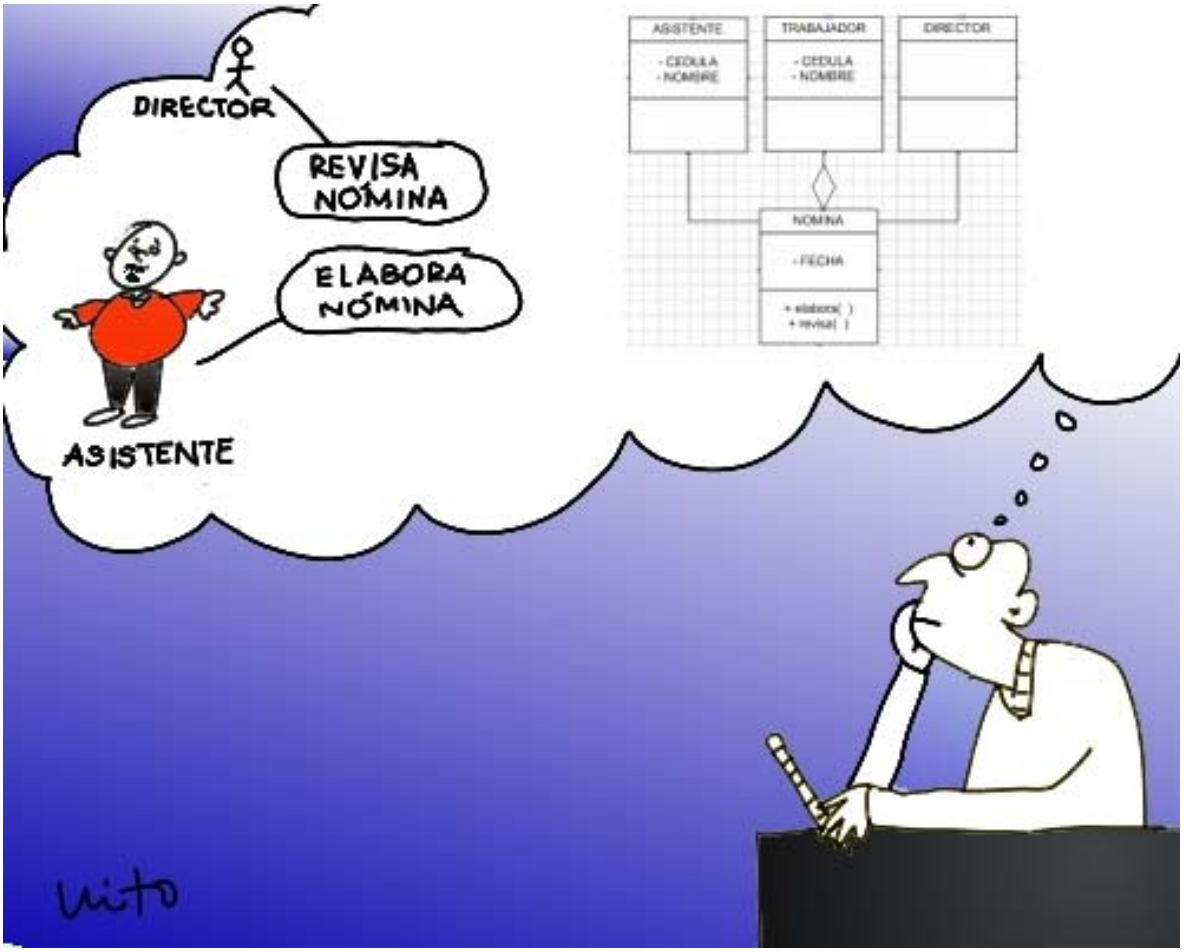
En CMMI se presenta un conjunto de las denominadas "buenas prácticas" que debe seguir una compañía desarrolladora de software para alcanzar alguno de los seis niveles de capacidad (incompleto, ejecutado, gestionado, definido,

gestionado cuantitativamente u optimizado) o madurez (inicial, gestionado, definido, gestionado cuantitativamente u optimizado). Se procura, en este caso, evaluar la organización, con el fin de garantizar que las aplicaciones de software que generan tengan la calidad debida.

En cualquiera de los casos, la aplicación de los estándares de software permite alcanzar una certificación, que no es otra cosa que una licencia para cobrar más caros los productos pues se supone que una empresa certificada tiene las cualidades necesarias para entregar productos y servicios de calidad.

Desde el punto de vista del interesado, las certificaciones ayudan a tener más tranquilidad en relación con la compañía con la cual se ha contratado el desarrollo de una aplicación o a la cual se le ha comprado un software genérico.

SECCIÓN 5.4: CONSISTENCIA



La *Consistencia* se puede entender como la uniformidad en el manejo de la información (conceptos o acciones), cuando ésta se incluya en diferentes diagramas. A este tipo de consistencia se le denomina *Intermodelo*[2]).

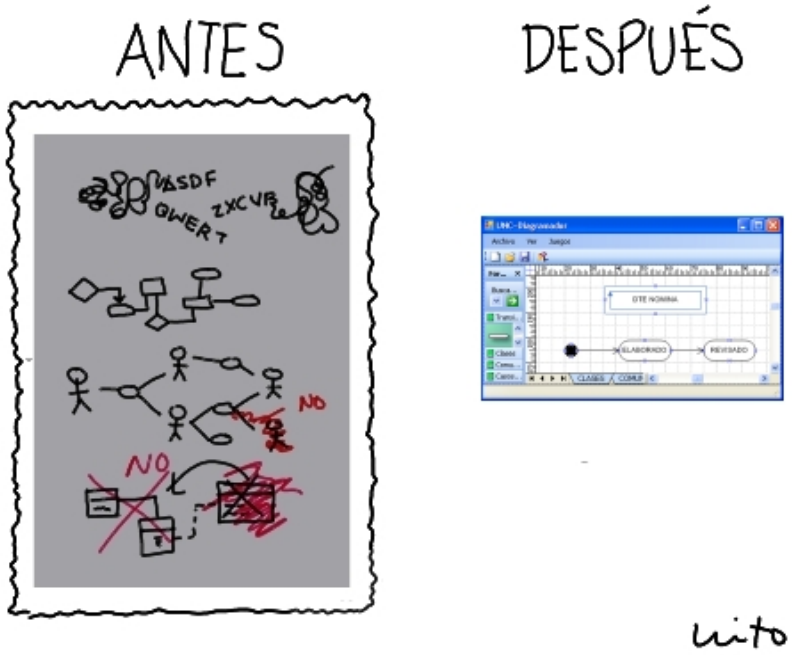
El *Object Management Group*, en la superestructura de UML, reconoce la consistencia intermodelo de forma parcial y no la especifica de manera tan cuidadosa como lo hace con la consistencia interna que deben poseer los diagramas, la cual se denomina consistencia *Intramodelo*. La consistencia intramodelo, se refiere al cumplimiento de normas de utilización de los diferentes elementos de un diagrama. Por ejemplo, no sería posible que existiesen dos palabras "nombre" dentro del segundo cajón de la imagen (reservado para lo que se conoce como "atributos") de la clase "trabajador"; igualmente, no deberían coexistir dos clases con el mismo nombre.

Los dos tipos de consistencia mencionados (intra e intermodelo) constituyen uno de los dolores de cabeza más intensos que puede sufrir un analista en el proceso de elaboración de los modelos. Esto ocurre porque la mayoría de las herramientas que se utilizan para la elaboración de modelos, llamadas herramientas CASE, *Computer-Aided Software Engineering* o Ingeniería de Software asistida por computador, no contemplan el manejo automático de estos tipos de consistencia. El pobre analista, además de tener que interpretar el discurso del interesado para la elaboración de los diagramas, también tiene que velar por la consistencia intra e intermodelo.

SECCIÓN 5.5: REFINAMIENTO

Si alguna vez el lector se topa en la calle con un analista, podrá reconocerlo por sus ojeras, que son producto de la falta de sueño y de las preocupaciones. Una de estas preocupaciones surge durante el ciclo de vida del software cuando, además de conservar la consistencia, debe ir creando sucesivas versiones de los diferentes diagramas y de otros nuevos, de manera que, poco a poco, el confuso discurso del interesado se acerque al excesivamente reglado y preciso código elaborado por el desarrollador. Este proceso sucesivo de versionado se conoce como *Refinamiento*.

[2] sería más adecuado llamarla consistencia *Interdiagramas.* Ver la sección 5.1 para aclarar la diferencia entre términos "modelo" y "diagrama"



El Refinamiento se debe ejecutar con cuidado porque, en cada una de las versiones de los diagramas, se puede agregar, suprimir o modificar la información. La decisión de cuál acción realizar para manejar adecuadamente la información corre a cargo del analista, quien debe conocer las características necesarias en cada una de las versiones para poder tomar una decisión lo más acertada posible. De allí, la preocupación del analista: él debe garantizar que los diagramas en cada una de las versiones continúen representando aquello para lo cual fueron diseñados. Además, también debe garantizar la *consistencia en el refinamiento,* es decir, que a medida que se agreguen, supriman o modifiquen detalles en los diagramas, cada uno de ellos debe guardar una relación estrecha con sus versiones iniciales y con el discurso inicial del interesado.

SECCIÓN 5.6: ENTREGABLE

Si la consistencia y el refinamiento son fuente constante de preocupación y angustia para los analistas, esas mismas sensaciones se reproducen en el interesado cuando le entregan la aplicación de software y los manuales correspondientes.

La expresión que tiene el interesado en la imagen se repite constantemente en el ciclo de vida del software, puesto que los analistas requieren permanente validación de su trabajo y, para ello, la mayoría de los métodos de desarrollo de software ha dispuesto un conjunto de entregas periódicas que deben ser revisadas por ambas partes (interesados y grupo de desarrollo de software). Esos documentos se denominan *Entregables* y suelen contener la información de los diagramas que se están realizando y otros artefactos (entendiendo por "artefacto" cualquier elemento que permita aclarar o complementar la información de un diagrama, por ejemplo tablas, gráficos, ecuaciones, etc.).

Los entregables contienen información técnica, mediante la cual se trata de expresar el dominio de la aplicación. Sin embargo, su principal dificultad, desde el punto de vista de los interesados, es precisamente el lenguaje en el

que están escritos. Por lo general, los interesados comprenden poco de los diagramas y otros artefactos del desarrollo de software. Sin embargo, esa información técnica es de gran interés, sobre todo para el crecimiento futuro de las aplicaciones, el cual se puede estudiar perfectamente desde los entregables existentes.

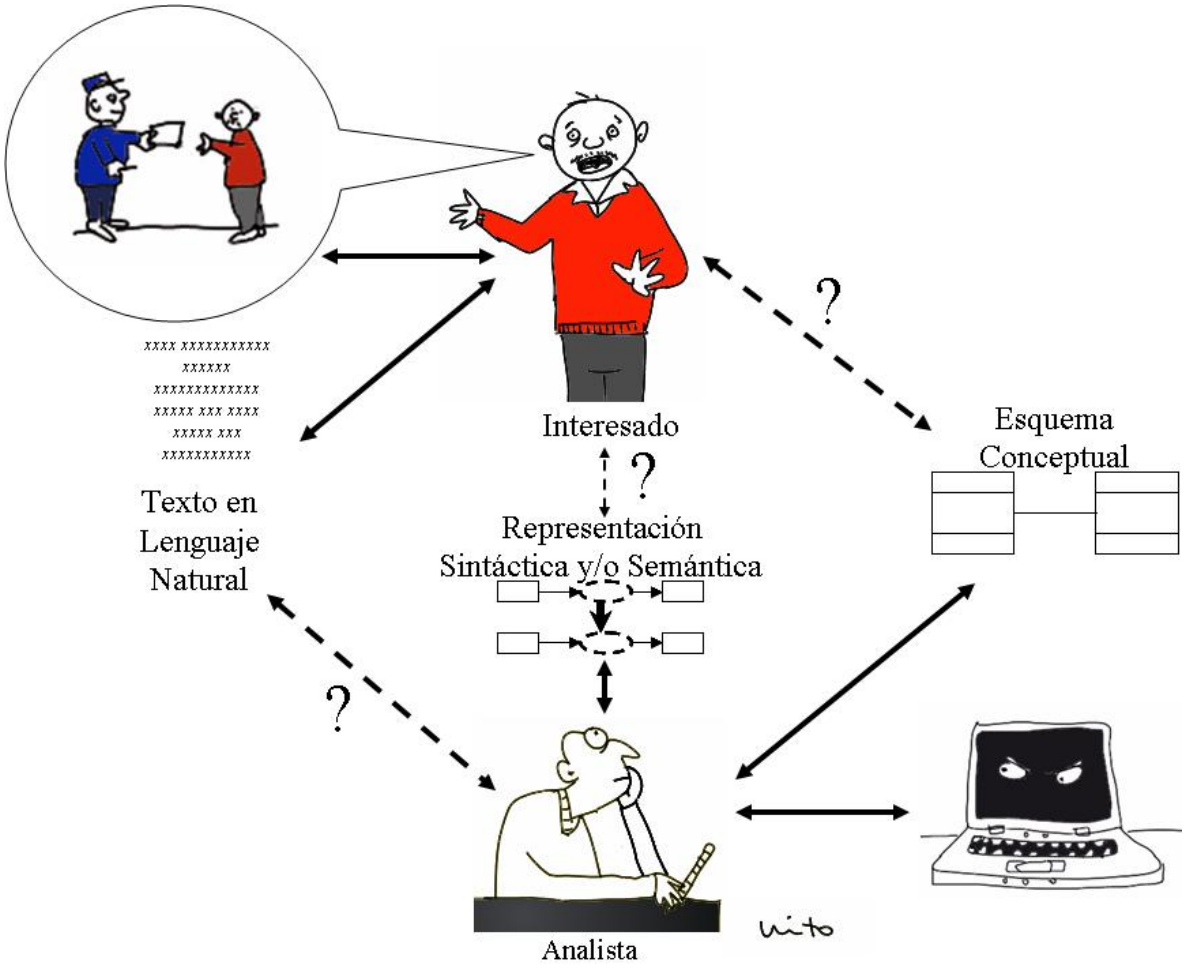
En ausencia de los entregables, el mantenimiento de una aplicación se debe emprender desde la revisión cuidadosa del código fuente, el cual no siempre es completamente claro para otros desarrolladores diferentes de quien lo programó.

Los entregables suministran un esquema coherente del desarrollo de software porque permiten seguir la representación de un elemento a lo largo de los diagramas que se elaboran en las diferentes fases del desarrollo. Esto se conoce como *trazabilidad*.

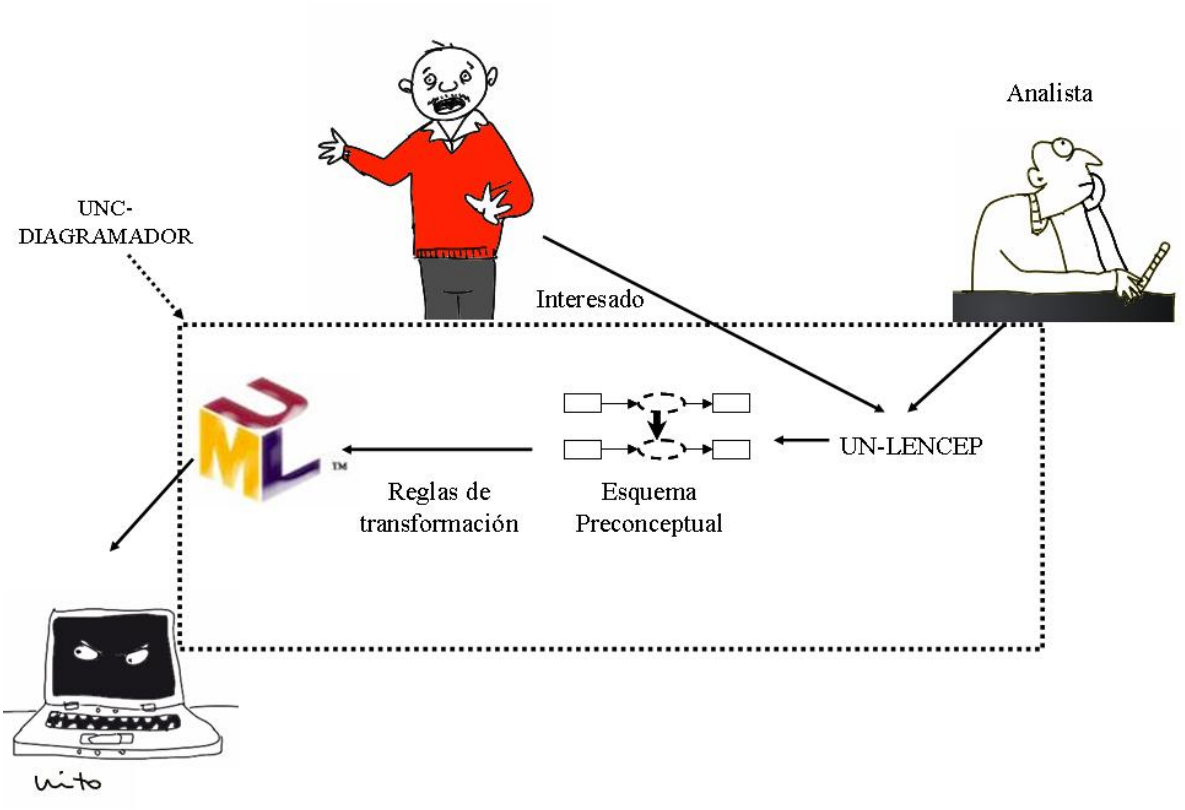
SECCIÓN 5.7: ESQUEMAS PRECONCEPTUALES

Cuando es necesario elaborar una aplicación de software, el analista y el interesado comienzan una serie de reuniones para tratar de precisar la información perteneciente al mundo del interesado (que se suele conocer como *dominio*). En el caso del interesado de la imagen siguiente, su “mundo” se refiere a los carteros y su trabajo. En las reuniones que se realizan (que, por otra parte, se pueden complementar con documentos de la organización a la que se le está desarrollando el software, y, en general, con otras muchas técnicas para capturar los requisitos), se procura determinar un conjunto de términos, características, relaciones y restricciones, pertenecientes al dominio del interesado, para luego convertirlo en un texto en lenguaje natural. Posteriormente, el analista toma ese texto y lo interpreta en su cabeza, para generar (también internamente, en su cabeza) una representación sintáctica y/o semántica de ese discurso.

Finalmente, el analista, como persona bien inteligente que es, elabora el modelo (véase la sección 5.1), empleando el lenguaje de modelado (véase la sección 5.2), que está conformado por varios *esquemas conceptuales*. De esta manera, el analista acerca el dominio del interesado al mundo del computador.



El cuento sería color de rosa si el proceso descrito fuera siempre exitoso. Sin embargo, no hay tal, y este proceso suele ser bastante caótico. Y lo es porque el interesado y el analista son personas bien diferentes (y, si el lector no lo cree así, por favor repase los capítulos 2 y 3 de este libro) y tienen diferencias apreciables en su formación y expectativas. Mientras el interesado es un idealista al que poco le interesan los computadores y espera que todo se pueda resolver de la manera más adecuada con esos "aparatos mágicos", el analista tiene los pies bien puestos en la tierra y conoce bien las limitaciones de los computadores, pero desconoce (al menos por lo general) los términos del dominio del interesado. Esas diferencias generan ciertos vacíos en la información (que en la imagen se representan con líneas discontinuas y símbolos de interrogación), que impiden que los esquemas conceptuales que se obtienen tengan la calidad debida y, peor aún, que se puedan validar por parte de los interesados.

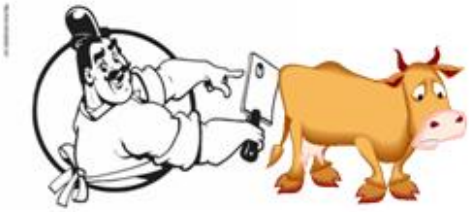


El grupo de investigación en Ingeniería de Software de la Escuela de Sistemas de la Facultad de Minas, perteneciente a la Universidad Nacional de Colombia, sede Medellín, abordó este problema como una de sus líneas de investigación, y propuso la solución que se muestra en la imagen. De esta manera, se desarrolló un lenguaje en el que el analista y el interesado se pueden comunicar, denominado UN-LENCEP, y que es un subconjunto del lenguaje natural para representar los elementos mencionados del dominio del interesado. A partir de un texto en UN-LENCEP se genera el denominado *Esquema Preconceptual*, que es una expresión gráfica de ese texto y, aplicando unas reglas de transformación, se obtienen los esquemas conceptuales de UML (Zapata *et al.*, 2006). De esta manera, se puede acercar el mundo del interesado al desarrollo de software y a las herramientas CASE convencionales.

Por ejemplo, el dominio del interesado puede ser la granja lechera que se muestra en la imagen siguiente, en la cual aparece una vaca (de muy triste aspecto, por lo que se describe seguidamente), cuyo nombre es Clarita y se identifica con el número 150, un ordeñador, cuya función es recolectar la leche que produce Clarita, y un matarife, que se encarga de sacrificar a

cualquiera de las vacas de la granja cuando la cantidad de leche que producen es inferior a 30 litros.

No. 350 - Clarita



<30 litros

Esta es una descripción básica del mundo que el analista debe interpretar para generar los esquemas conceptuales requeridos. Sin embargo, para lograr que este discurso se procese para generar los esquemas preconceptuales, se requiere que el analista y el interesado lo transformen en un texto en UN-LENCEP, que viene a ser algo como lo siguiente:

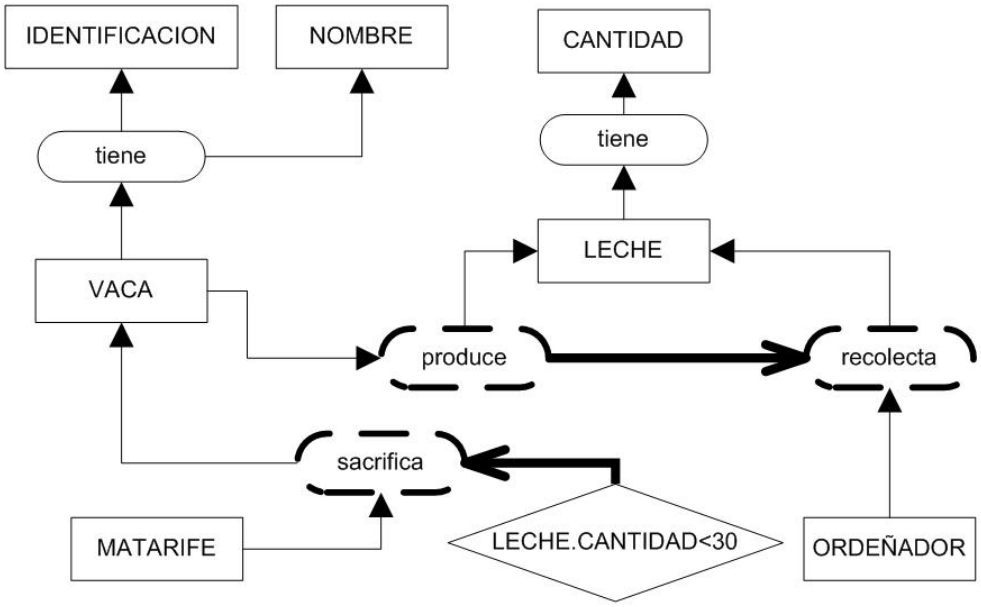
Una vaca posee una identificación
Una vaca posee un nombre
Cuando la vaca produce leche, el ordeñador recolecta la leche
La leche posee una cantidad
Si leche.cantidad<30, entonces el matarife sacrifica la vaca

Aunque, a primera vista, este texto parece lenguaje natural, en verdad presenta un conjunto de restricciones que se deben generar por parte del analista, con la participación del interesado. Por ejemplo, no es muy natural decir frases como

“la leche posee una cantidad”, porque no es una frase que se pueda originar de manera espontánea de un interesado. Con este tipo de frases, se pretende acercar la descripción del interesado (que, por lo general, se realiza en lenguaje natural, completamente plagado de las imprecisiones de este tipo de lenguaje y además con las grandes dudas que aún posee el interesado sobre su área y los problemas que motivan la aparición del software) al modelo que puede representar esa descripción (y el cual se suele expresar en términos de los lenguajes de modelado). Al representar el dominio del interesado en términos del lenguaje controlado UN-LENCEP se ganan dos cosas: precisión en los conceptos y relaciones pertenecientes al lenguaje natural y validación indirecta, por parte del interesado, de los futuros diagramas que harán parte del modelo.

En este caso, con la solución propuesta por la Escuela de Sistemas, el analista no necesita realizar el proceso de interpretación del discurso del interesado para generar los esquemas conceptuales, porque, siempre y cuando el discurso se traduzca al texto en UN-LENCEP, este proceso lo hará automáticamente una herramienta, denominada UNC-DIAGRAMADOR que contiene todo el entorno que se definió (el UN-LENCEP, los esquemas preconceptuales y los esquemas conceptuales de UML). De esta manera, al ingresar el discurso en UN-LENCEP, se obtiene el esquema preconceptual que se muestra en la imagen siguiente, y que representa, de manera aproximada, el discurso que se describió para la granja lechera. El UNC-DIAGRAMADOR genera, adicionalmente, tres diagramas de UML: el diagrama de clases, el de comunicación y el de máquina de estados. Esos diagramas, correspondientes al esquema preconceptual anotado, también se muestran en la imagen siguiente.

Una pregunta que, por lo general, surge en los foros y publicaciones internacionales en los cuales se suele presentar este trabajo (y otros trabajos que también hablan de la automatización de las labores en Ingeniería de Software), tiene que ver con la posibilidad de desaparición de algunos de los actores del proceso de desarrollo de software con la presencia, cada vez creciente, de herramientas de diferente índole que buscan reemplazar las actividades humanas por series de comandos e instrucciones que, de manera automática, realizan los mismos procesos. Un proceso similar ocurrió en la industria con el advenimiento de la automatización de procesos, y que generó una nueva manera de repensar el papel de la mano de obra en las industrias contemporáneas.



Esquema Preconceptual

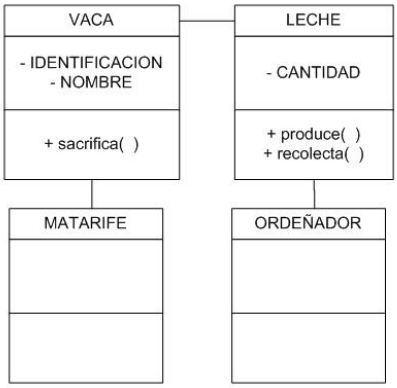


Diagrama de clases

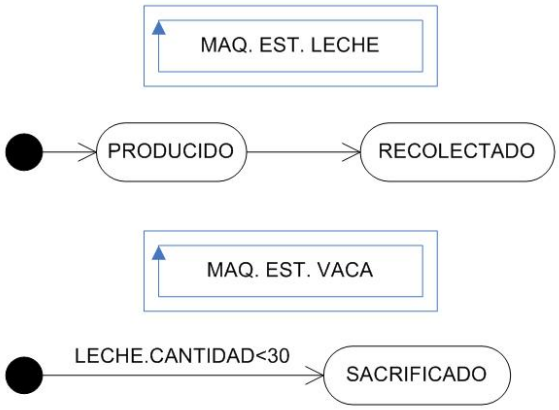


Diagrama de Máquina de estados

Diagrama de comunicación

A este respecto, solo cabe aclarar que la automatización de los procesos en Ingeniería de Software, al igual que la automatización de la industria en su momento, sólo pretende liberar las cargas de los actores involucrados en el proceso, para que se puedan dedicar a realizar su función de manera mucho más profesional cada vez. Por ejemplo, en ausencia de herramientas como el UNC-DIAGRAMADOR, los analistas deben invertir gran cantidad de tiempo en elaborar los diagramas que conciben, utilizando para ello las herramientas CASE convencionales, para que tengan las características de consistencia (véase la sección 5.4) y refinamiento (véase la sección 5.5) que requiere la aplicación de software para tener la calidad que necesita como producto.

Finalmente, es bueno recordar que las herramientas CASE vienen generando, desde mediados de los años setenta del siglo pasado, una ola de automatización que es difícil de detener, y que marcará la evolución en la elaboración de software en los años venideros. Ante este tipo de fenómenos, lo mejor que pueden hacer las generaciones futuras de Ingenieros de Sistemas e Informática, y los practicantes de profesiones afines, es tener la mente abierta a estos cambios que se vienen sucediendo, para comprender que este nuevo milenio trae nuevos retos que harán que la función de los participantes en el desarrollo de software (sean interesados, analistas, diseñadores o programadores) cambie radicalmente y se especialice para generar cada vez mejores aplicaciones de software.

CAPÍTULO 6: EL CICLO DE VIDA DEL SOFTWARE

SECCIÓN 6.1: INTRODUCCIÓN

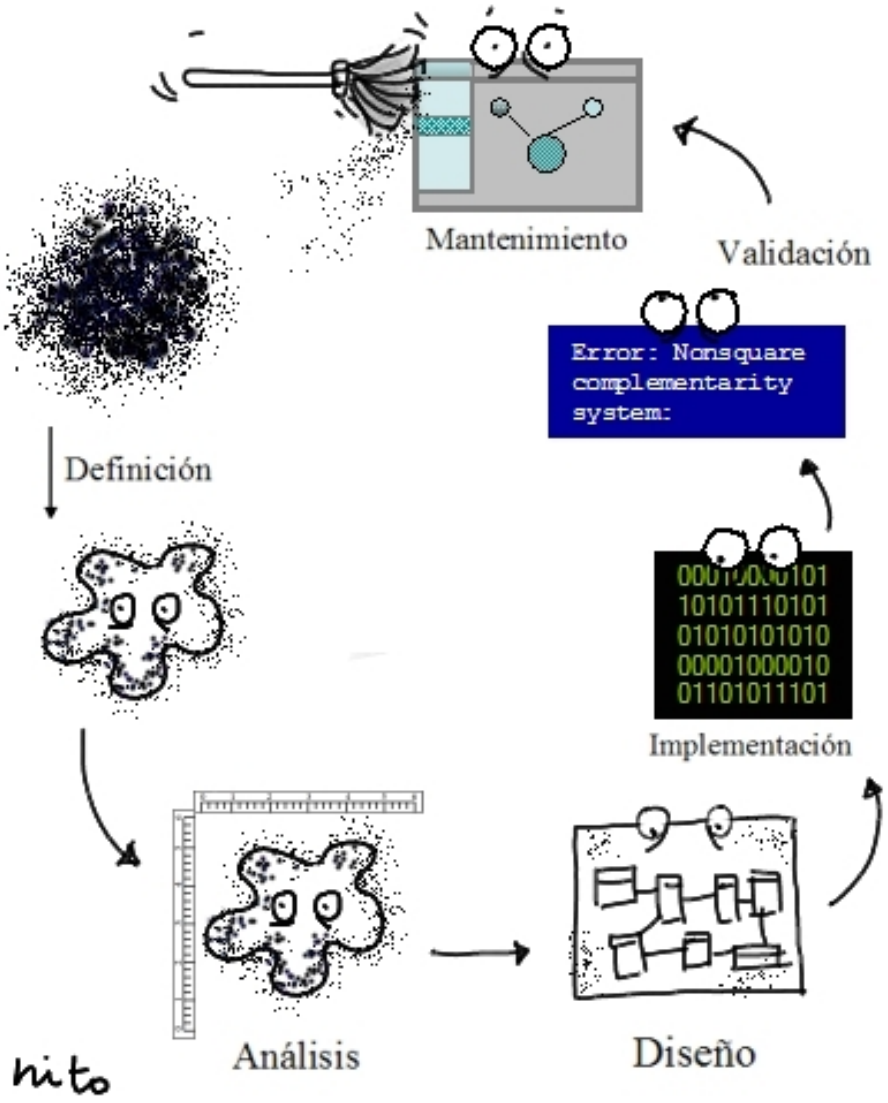
Los interesados suelen creer que elaborar una aplicación es sencillo y que el software viene listo para armar. Durante mucho tiempo, los desarrolladores han compartido esta creencia con los interesados y es por esto que algunos desarrolladores venden aplicaciones genéricas (COTS por las iniciales en inglés *Commercial off-the-shelf* o aplicaciones en inventario de tipo comercial).

El problema de estas aplicaciones comerciales, aún de las más estandarizadas del mercado, es que requieren algún grado de personalización para adaptarlas a las necesidades de las diferentes organizaciones. En síntesis: no hay software prefabricado.

Algo que los interesados no perciben es que, como señalaba Frederick P. Brooks Jr., uno de los más connotados profesores en el área de Ingeniería de Software, el desarrollo de software es un proceso que tiene un tiempo de maduración o, en sus propias palabras, “la gestación de un niño se tarda nueve meses, sin importar cuántas mujeres se asignen a esa tarea”. No es posible sacar una aplicación de software “de la manga” como si fuera el truco de un mago, y el tiempo de maduración requerido engendra un proceso por etapas que se conoce como el c*iclo de vida del software*, cuya descripción pormenorizada se abordará en las secciones siguientes de este capítulo.

El ciclo de vida del software se realiza para lograr que una aplicación de software realice las funciones requeridas para resolver los problemas de la organización. Aún las aplicaciones más sencillas y que han sido largamente estudiadas, tales las aplicaciones de nómina, requieren que el ciclo de vida del software se lleve a efecto de manera estricta. Esto porque aún en el mismo dominio las condiciones son diferentes para cada organización. Los objetivos de la organización, el funcionamiento de la misma y/o los problemas que motivan el desarrollo de una aplicación de software pueden generar aplicaciones distintas para organizaciones distintas.

Eso no quiere decir que las aplicaciones tipo COTS no tengan futuro, sino que, en caso de que se vaya a desarrollar una aplicación de ese tipo, es necesario que el ciclo de vida del software se realice de manera meticulosa, buscando conocer las características de múltiples organizaciones con problemáticas similares, con el fin de determinar cuáles serían las posibles variaciones en la aplicación y poderlas personalizar cuando el software ya esté listo y se vaya a adaptar a una determinada organización.



El ciclo de vida del software, que se esquematiza en la imagen, establece el proceso de transformación de una idea hasta convertirse paulatinamente en una aplicación de software completamente definida y delimitada. Ese proceso (largo y tedioso, en opinión de los detractores de la Ingeniería de Software) lo

deben conducir diferentes personas (analistas y desarrolladores) en las diferentes fases, contando siempre con la información provista por los interesados, pero teniendo en mente el objetivo común que debe marcar la funcionalidad futura del software. En las secciones siguientes, se describe de manera más detallada cada una de las etapas o fases del ciclo de vida del software, tomando en consideración las características del proceso en cada caso.

SECCIÓN 6.2: DEFINICIÓN

A primera vista, puede parecer que el interesado de la imagen se encuentra en una sesión de psicoanálisis, tratando de develar alguna profunda frustración de su niñez. En realidad, esta imagen se filtró en el libro, porque, efectivamente, el interesado está en una sesión de terapia con su psicoanalista. Sin embargo, el proceso en la fase de definición es similar a éste. En la etapa de definición de la aplicación de software, se realiza una serie de entrevistas entre analistas e interesados para tratar de determinar el *contexto* en el cual se desenvolverá

dicha aplicación. Esas entrevistas, corresponden al inicio de un proceso que se extiende entre las fases de definición y análisis (en inglés *elicitation*, en español, *educción*) y para las cuales se han definido numerosos protocolos diferentes, con múltiples técnicas, que prometen resultados fascinantes.

En las entrevistas, surgen diferentes elementos que dan una idea general de todo aquello que rodeará el software futuro. También, surge una gran cantidad de información que nada tiene que ver con el desarrollo de software; información personal de los interesados, de sus hijos o de sus problemas, de sus sentimientos en relación con la organización o de la manera como los directivos están conduciendo las riendas de la misma. El analista debe asumir una posición de observador, registrando la mayor cantidad de información posible, porque en ese discurso se combinan cosas triviales e intrascendentales con funciones, responsabilidades, objetivos y problemas del dominio que se intenta representar. Entre más información logre recabar el analista, mucho más completo puede ser el posterior proceso de análisis de esa información.

La fase de definición, se puede también complementar con información proveniente de la documentación de la organización. Desde los formatos en los cuales los interesados registran la información hasta los documentos informales que permiten la comunicación entre las distintas dependencias, todos estos elementos deberían ser objeto de escrutinio por parte del analista, con el fin de precisar la manera como los interesados realizan sus funciones dentro de la organización. A partir de este tipo de documentos, se pueden recopilar grandes cantidades de elementos que luego van a poblar las bases de datos, una vez la aplicación de software haya entrado en funcionamiento. Además, son de especial utilidad los archivos en herramientas de oficina, como procesadores de texto y hojas de cálculo, en los cuales los interesados acomodan los procedimientos de la organización a sus necesidades específicas, realizando una especie de sistema de información “informal”, que puede suministrar pistas concretas en relación con el funcionamiento futuro de una solución para el dominio particular de los interesados.

No es de extrañar que la información tenga la forma de la figura porque, como recordará el lector del capítulo 2, el interesado puede no tener ideas respecto de la apariencia de la aplicación que solucionará sus problemas. Es más, en muchos casos el interesado no es consciente de todos los posibles problemas de su organización y, por lo tanto, será difícil que pueda encontrar una forma más clara de expresar sus ideas, con el fin de facilitarle el proceso al analista.

El analista podrá encontrar dichos problemas asumiendo una actitud inquisitiva, como la que poseen los niños de manera natural, pero en la cual se puede tratar de establecer la información que sea necesaria para complementar el análisis, la segunda de las fases del desarrollo de software.

En la fase de definición, se procura representar, mediante diagramas muy simples (y preferiblemente con la posibilidad de que los interesados los puedan entender), toda la información inicial que se haya podido recopilar. Esos diagramas sufrirán un proceso de refinamiento continuo, incluso hasta finalizar la implementación de la aplicación de software. Es, en este punto del proceso, que se intensifican las dificultades del analista, puesto que se requiere que los interesados validen la información que se encuentra en los diagramas.

A causa de las dificultades de validación de los diagramas, ha surgido una tendencia a automatizar el proceso por completo, pero esa es otra historia…

SECCIÓN 6.3: ANÁLISIS

Por lo general, los interesados entregan información compleja durante el proceso de educción. A partir de esta información, en la fase de análisis, los diagramas iniciales se traducen en un modelo (un conjunto de diagramas) que representa de manera consistente varios aspectos de la futura aplicación: los objetivos de la organización y su relación los requisitos y expectativas de los interesados respecto a la aplicación de software, así como los problemas del área de la organización que requiere el software y las funciones que desempeñan los diferentes actores que están ligados con la organización. Con base en esta información, el analista debe complementar el modelo con los diagramas de UML (véase la sección 5.2 para una ampliación del término) que representan la estructura del mundo, la dinámica que le imprimen los actores

que participan en el proceso y el comportamiento de los diferentes objetos del mundo.

Cabe aclarar, que ésta debería ser una fase supremamente creativa para los analistas, porque deben proponer un conjunto de soluciones que varían desde lo que el interesado desea (prácticamente un sistema que lea el pensamiento y, sin oprimir ninguna tecla, entregue los reportes adecuados para tomar las decisiones correctas) y lo que realmente puede pagar. Existen infinitas soluciones posibles entre estos extremos de solución, y, por ello, se requiere que el analista, de manera asertiva, proponga un conjunto adecuado de ellas, con las ventajas y desventajas de cada una, que permitan al interesado tomar la decisión más adecuada para la selección de la aplicación de software que se va a construir.

Una vez se ha definido cuál de las diferentes opciones de aplicación se va a elaborar, el analista debe proceder, como parte final de esta fase, a expresar la solución seleccionada de manera semiformal, por ejemplo, con una versión más refinada del diagrama de clases de UML, y/o formal, empleando lógica de predicados de primer orden o el lenguaje de restricción de objetos[3], de manera que esta representación sirva como insumo principal en la siguiente fase del ciclo de vida del software.

Al igual que en la fase de definición, en la fase de análisis persisten problemas como la falta de claridad de los interesados acerca de la aplicación que se pretende realizar. Estos problemas, se agravan por la carencia de un mecanismo para validar los diagramas que construye el analista. Una buena fase de definición sería la solución a estos problemas, pero, en la realidad, la información que se debe extraer del interesado va creciendo a medida que se va avanzando en la elaboración del modelo (de ahí que la frase más común del interesado en estas etapas sea "¿No te había mencionado eso? ¡Qué curioso!"). Sin embargo, y pese a la existencia de estos problemas, la fase de análisis suele terminarse e inicia la fase de diseño.

SECCIÓN 6.4: DISEÑO

El diseño trata de llevar la solución, creada y especificada durante las fases de definición y análisis, a un lenguaje cada vez más cercano al computador. En

[3] OCL, que es el lenguaje estándar propuesto para complementar los diagramas en UML.

esta fase, al menos en teoría, los problemas de comunicación con el interesado deberían haber sido resueltos, porque los diagramas que hacen parte del diseño, deben contener la información necesaria para iniciar la elaboración del código.

Si el diseño aún presenta problemas, el desarrollador tendrá la tentación de complementar la información faltante con sus propias suposiciones en relación con los diagramas que hacen parte del diseño, generando uno de los grandes problemas del desarrollo de software: la entrega de una implementación que no corresponde a lo que necesita el interesado. Por ello, toda la información que tiene que ver con fórmulas de cálculo, modos de proceder, permisos de acceso a la información, y otras características semejantes de la aplicación, debe haber sido previamente discutida con el interesado y se deben haber alcanzado los acuerdos necesarios para definir de forma clara cómo incorporar tales características en el diseño.

La diferencia entre los diagramas que se emplean en el análisis y aquellos que hacen parte del diseño es supremamente sutil, puesto que, en muchas ocasiones, se trata de los mismos diagramas, pero con la adición de detalles complementarios acerca de la plataforma de implementación, el lenguaje de programación, etc., empleando el proceso denominado refinamiento (véase la sección 5.5 para mayor claridad en relación con el término). Otros diagramas y artefactos son propios de la fase de diseño, en particular aquellos que contienen información sobre la manera en que el software futuro se empleará, la cantidad de usuarios que simultáneamente usarán la aplicación, la cantidad de datos que se almacenarán, etc.

Algunos de los temas que se han discutido, por ejemplo seguridad en el acceso a la información, cantidad de usuarios simultáneos o eficiencia en el procesamiento de los datos, hacen parte de una categoría especial que cobra vigencia en la fase de diseño. Esa categoría se denomina *requisitos no funcionales* y se suele asociar con características de calidad propias de la aplicación de software como tal. Los requisitos no funcionales se llaman así para distinguirlos de los *funcionales*, que están más ligados con la forma en que opera la organización que requiere la aplicación de software.

Casi con absoluta seguridad, el computador portátil de la imagen no podrá tomar de manera autónoma una decisión tan radical como la que está pensando, pero, con el diseño, el analista debe garantizar que se hará el mayor

esfuerzo para prevenir que la aplicación no presentará un comportamiento negativo cuando entre en funcionamiento en el futuro.

SECCIÓN 6.5: IMPLEMENTACIÓN

Durante esta fase, el desarrollador transforma los diferentes diagramas en lo que será la aplicación definitiva, programada en un determinado lenguaje de programación, como *Java®, C++®, php®, jsp®* o alguno de los muchos otros que existen en el mercado. Las decisiones, sobre el tipo de lenguaje o sobre el manejador de bases de datos que se piensa usar, se debieron tomar antes, durante la fase de diseño, con base en la infraestructura disponible por la organización, la arquitectura requerida por el sistema, las condiciones económicas de la compañía, etc.

Cuando las aplicaciones de software se desarrollaban sin que mediara un enfoque metodológico como el que se plantea en este capítulo, la implementación era la única fase que se ejecutaba, dado que era el desarrollador quien, con base en algunas instrucciones del interesado, pasaba directamente a usar los lenguajes de programación para crear la aplicación de software.

Un enfoque similar lo rescataron recientemente los denominados *métodos ágiles de desarrollo de software*, creados en oposición a los denominados *métodos monumentales*[4]. Uno de los principales métodos ágiles, llamado XP (*Extreme Programming* o programación extrema), utiliza como grupo de desarrollo un conjunto de pequeñas células conformadas por interesados y desarrolladores, que trabajan de manera conjunta para elaborar versiones sucesivas de prototipos que van evolucionando hasta obtener el software definitivo.

En este tipo de enfoques, sin embargo, el desarrollador, quien finalmente lidera la construcción de la aplicación, debe poseer una amplia experiencia en desarrollo de software, pues se trata de sustituir la inmensa documentación requerida por los métodos monumentales por un producto documentado únicamente en el código fuente, pero que recoge adecuadamente las necesidades y expectativas de los interesados, dado que ellos participan de manera activa en el proceso de desarrollo. En ausencia de desarrolladores experimentados, a través de los métodos ágiles, difícilmente se puede producir software de calidad.

En la sección 5.4 se hablaba someramente de las herramientas CASE, un conjunto de herramientas que permiten la creación y edición de los diferentes diagramas que se emplean en el ciclo de vida del software. Durante años, los promotores de las herramientas CASE han prometido que el desarrollo de software se podría automatizar completamente desde los diagramas. Sin embargo, hasta ahora esa promesa no ha sido completamente cumplida por la mayoría de las herramientas CASE del mercado, puesto que se han limitado a generar parcialmente el código necesario para la aplicación y a emplear un pequeño subconjunto de diagramas para ello (generalmente el diagrama de clases y el de secuencias de UML).

Únicamente algunos trabajos realizados por grupos de investigación y ciertas herramientas CASE también comerciales, aunque poco difundidas, por ejemplo *ONME®*, las iniciales de *OlivaNova Model Execution*, una herramienta también surgida de grupos de investigación españoles, presentan adelantos en la elaboración de aplicaciones de software a partir de modelos.

[4] En inglés *plan-driven methods* o métodos dirigidos por planes, dado que recorren pormenorizadamente el ciclo de desarrollo del software.

Hasta ahora, la implementación ha sido una actividad más artística y creativa que ingenieril, pero por lo menos el enfoque que se viene dando a esta actividad desde el punto de vista de la Ingeniería de Software ha permitido que se vaya formalizando, lo cual posibilitará en el futuro una transición más suave desde los diagramas hacia las aplicaciones de software concluidas.

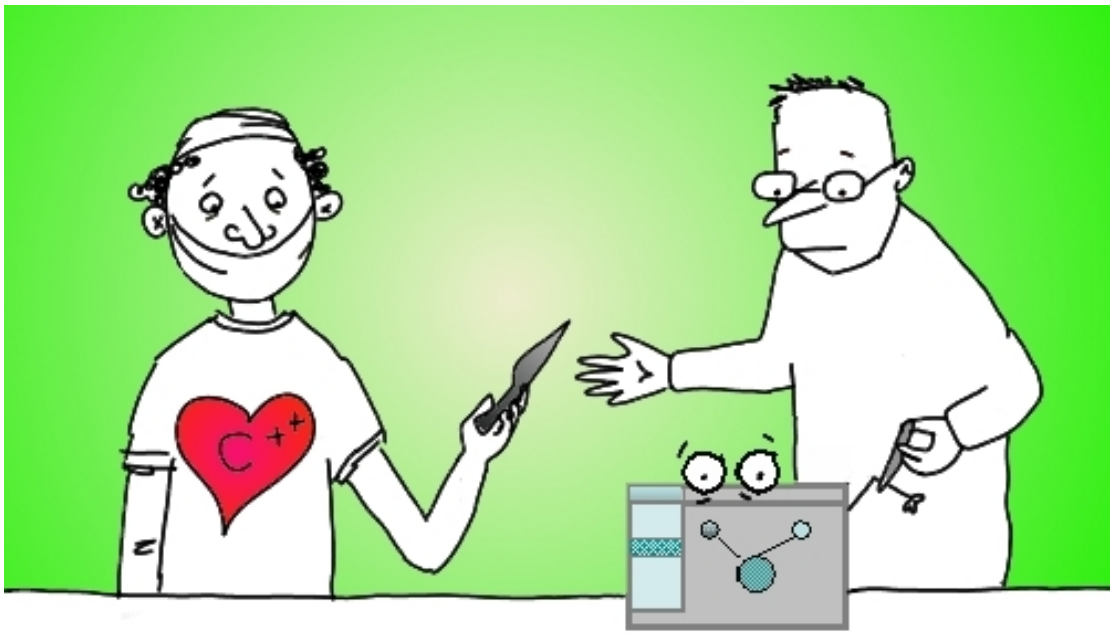
SECCIÓN 6.6: VALIDACIÓN

El desarrollo de software sigue siendo una actividad netamente humana, en la cual la participación de los computadores se ha limitado a suministrar herramientas de ayuda en la creación y edición de diagramas y código. Sin embargo, el contenido de esos diagramas y del código de la aplicación siguen siendo responsabilidad de los analistas y los desarrolladores, respectivamente. Por ser una actividad humana, se pueden cometer (y de hecho, continuamente se cometen) muchos errores, que hacen que la aplicación pueda fallar en alguna funcionalidad aislada (en el caso más favorable) o incluso de manera catastrófica (en el caso más desfavorable). En la fase de validación, se procura

identificar esos errores o *bugs*, en el argot de los desarrolladores, con miras a su corrección.

Con la depuración (*debugging*), se busca que el funcionamiento de la aplicación de software sea el adecuado a las necesidades de los interesados, los cuales no deben ver perturbada su actividad por efecto de las fallas que se pueden encontrar en el software.

Al proceso de detección de errores que se realiza en esta fase del ciclo de vida del software se le denomina *testing* (una de esas palabrejas cuya traducción es un dolor de cabeza para cualquier estudiante de esta área del conocimiento), que se puede asociar con la realización de pruebas sobre la aplicación o la documentación para identificar los principales errores que puedan tener.



wilo

En la realidad es un poco difícil emplear un bisturí o un escalpelo para "cortar" los errores en el software, como en la imagen, pero aún así, se pueden hacer pruebas al software. Las principales son las pruebas de *caja negra*, en las cuales se examina la funcionalidad de la aplicación, es decir, los resultados que arroja cuando se le ingresan ciertos valores, y las pruebas de *caja blanca*,

que son revisiones exhaustivas del código o la documentación en busca de errores de tipo semántico o pragmático en la aplicación y/o la documentación.

Es imposible que una aplicación de software esté exenta de errores. Incluso, algunos errores se pueden presentar después de la entrega definitiva del software, en particular si están en partes de la aplicación que se usan sólo en ciertas condiciones.

La importancia de la validación radica en que procura garantizar la calidad de la aplicación de software, realizando las pruebas a la cuales se refiere la fase de validación de manera intensiva y cuidadosa. Los estándares de calidad que se mencionaron en la sección 5.3 sólo se pueden alcanzar cuando el equipo de desarrollo (incluyendo los interesados) se encuentra comprometido completamente con el ciclo de vida del software y con la calidad final de la aplicación, de forma que se trabaje conjuntamente en la identificación y corrección de la mayoría de los errores que se puedan presentar en ella.

SECCIÓN 6.7: MANTENIMIENTO

Cuando el grupo de desarrollo (después de intensas súplicas y desmanes) consigue que la aplicación de software sea recibida para su uso en la organización de los interesados, se produce lo que se denomina *entrada a producción* de la aplicación, un proceso caracterizado por el uso intensivo de la aplicación por parte de ciertos interesados, que se denominan *usuarios finales*. De esta manera, la aplicación inicia su larga (o en ocasiones corta) vida útil, para comenzar a resolver los problemas de los interesados que motivaron su elaboración.

Es de esperarse que, si la aplicación se realizó con el mayor cuidado durante las fases previas del ciclo de vida del software, la fase de mantenimiento sea relativamente tranquila para el grupo de desarrollo, y es éste el objetivo último de la Ingeniería de Software.

Sin embargo, no siempre el software defectuoso es la causa de los altos costos de mantenimiento que puede tener una aplicación. Una vez el software se entrega, la organización para la cual fue construido puede cambiar, haciendo que los requisitos que se determinaron durante las fases de definición y análisis cambien con la organización misma. Ello puede ocurrir, porque las fases iniciales del ciclo de vida del software pueden tener una diferencia en tiempo considerable con las fases finales del ciclo, especialmente con la

validación y el mantenimiento. En tal caso, es importante hacerle notar a los interesados que las deficiencias aparentes son realmente nuevas características del software que no habían sido consideradas en principio por el hecho de que la organización no había evolucionado.

Lo que sí es cierto es que, cuando el software no se desarrolla de manera disciplinada y metodológica, los interesados pueden sufrir consecuencias desastrosas, o incluso sentirse tan engañados con la aplicación como se encuentra el interesado de la imagen con el software de *Pirasoft*. El mantenimiento, cuando el software se realizaba directamente, sin mediar

documentos que representaran su evolución dentro del ciclo de vida, solía ser la fase más costosa de todo el desarrollo, y, en ocasiones, era incluso objeto de contratos adicionales que ofrecían las mismas empresas desarrolladoras para subsanar las deficiencias que se habían presentado en el proceso inicial.

SECCIÓN 6.8: EPÍLOGO

El interesado de la imagen con toda seguridad no hizo una buena selección del proveedor de su aplicación de software, porque, para elegir la compañía más adecuada para esa labor, debería conocer, así fuera tangencialmente, el ciclo

de vida del software y los diferentes conceptos que se han presentado en este libro.

El proceso de desarrollo se debe realizar cuidadosamente, con el fin de que las necesidades y expectativas de los diferentes interesados se traduzcan efectivamente en la aplicación de software que logre resolver sus problemas.

La importancia del software en la sociedad moderna es innegable y, por ello, el uso de métodos de desarrollo debería ser imperativo, con el fin de garantizar que las diferentes aplicaciones se desarrollen adecuadamente, con la calidad necesaria y con las posibilidades plenas de mantenimiento y adaptación a futuros requisitos de la organización.

Este libro se dirige inicialmente a analistas, pero tiene los elementos necesarios para que cualquier persona involucrada en un proceso de desarrollo de software se entere de los conceptos fundamentales de ese proceso, desde las funciones de los diferentes actores que participan en él, continuando con algunos de los conceptos que se suelen emplear en ese dominio (como los conceptos de modelo, lenguaje de modelado, consistencia, refinamiento, estándares de calidad y esquemas preconceptuales, que se mostraron en el capítulo 5) y finalizando con el ciclo de vida del software, que se discute en el presente capítulo.

Aunque podría parecer que este libro tiende más hacia métodos monumentales de desarrollo, porque el concepto de ciclo de vida del software es menos evidente en los métodos ágiles, lo que se pretende es impulsar el empleo de la Ingeniería de Software como un recurso para combatir el desarrollo de software a la manera antigua de elaboración de aplicaciones en la cual, como se ha discutido previamente, el programador era el único responsable de la elaboración de la aplicación después de charlas someras con los interesados y que algunos autores del área han identificado como una de las principales fuentes de la denominada *crisis del software*.

PARTE III
LA PARTE SERIA

CAPÍTULO 7: ¿POR QUÉ ESTE LIBRO?

SECCIÓN 7.1: PRESENTACIÓN DEL PROBLEMA

La Ingeniería de Software es una profesión relativamente reciente. Gibbs (1994) reporta cómo, hacia finales de la década de los sesenta del siglo pasado, un comité especial de la OTAN propuso la creación de una ingeniería que ayudara a superar los efectos adversos que se estaban produciendo en el desarrollo de software, tales como desbordamientos en los presupuestos de los proyectos y plazos que reiteradamente se extendían, haciendo casi imposible la predicción del valor final y la fecha de entrega de tales proyectos.

Desde sus inicios, la Ingeniería de Software se ha enseñado con estrategias de tipo tradicional, que combinan clases expositivas con pequeños proyectos de "juguete", que buscan desarrollar en los estudiantes las habilidades requeridas para el futuro desempeño profesional (Baker *et al.*, 2005).

Esas habilidades se reconocen, prácticamente, desde los trabajos iniciales, en el currículo de la Ingeniería de Software, a tal punto que Fairley (1978) definió los principales lineamientos curriculares de lo que debían ser las áreas de interés de esta profesión. Este trabajo, lo retomaron posteriormente otros investigadores para conformar las áreas temáticas fundamentales que deberían ser parte de la enseñanza de la Ingeniería de Software (Bagert *et al.*, 1999; Budgen y Tomayko, 2005; Kitchenham *et al.*, 2005; Cowling, 2005).

En síntesis, en estos trabajos se reconocen varios tipos de habilidades y conocimientos, tales como el desarrollo matemático y la capacidad de razonamiento característicos de los ingenieros, pero incluyendo también áreas diversas como el modelado y una serie de características pertenecientes a otras áreas, profesiones y disciplinas, tales como la gestión de proyectos, el trabajo en equipo, las comunicaciones interpersonales y la gerencia de negocios.

De los conocimientos y habilidades mencionados, por medio de la enseñanza tradicional se ha procurado enseñar el manejo de algunos de ellos, que se suelen relacionar con la inteligencia lingüística y la lógico-matemática, en el marco de las inteligencias múltiples enunciadas por Gardner (1983). Sin embargo, otros tipos de inteligencia de esa misma teoría, tales como la inteligencia interpersonal y la intrapersonal, requieren otras estrategias que les permitan a los alumnos vivenciar los conocimientos o, por lo menos, que al

docente le permitan transmitirlos de manera más adecuada, dada la complejidad de su ejercicio en la profesión.

Las diferentes estrategias que se suelen emplear para la enseñanza de la ingeniería fueron recopiladas por Wankat y Oreovicz (1993). Si bien las clases expositivas y los proyectos prácticos se encontraban entre estas estrategias, estos autores y algunos otros (Rugarcia *et al.*, 2000) reconocieron la necesidad de complementar las estrategias tradicionales con otras estrategias que buscaran afianzar el aprendizaje significativo de los alumnos, especialmente en habilidades y conocimientos que no podrían ser cubiertos por ese tipo de enseñanza. Además, desde la formulación de la teoría de competencias (White, 1959), se ha reconocido que, en la enseñanza, la motivación es un factor preponderante para lograr el aprendizaje, y, según lo establecen Soloway *et al.* (1994), con métodos tradicionales a veces es difícil lograr la motivación de los estudiantes.

SECCIÓN 7.2: ANTECEDENTES

Algunas otras profesiones y áreas del conocimiento sufren parte de los problemas que se esbozan en la sección anterior. En efecto, en el ejercicio del derecho, por ejemplo, se requieren habilidades de argumentación que no son fácilmente adquiribles mediante clases expositivas, sino más bien a través del ejercicio mismo de la profesión. En medicina, las habilidades para realizar los cortes adecuados en un paciente sin que por ello se arriesgue su vida sólo se pueden obtener practicando dichos cortes de manera repetida. En negociación, nuevamente la capacidad de argumentación y las habilidades para conducir de manera certera el rumbo de un negocio se adquieren, como en el derecho, con el ejercicio de la práctica profesional.

Estas áreas se nutren de las experiencias en métodos alternativos y complementarios a la enseñanza tradicional. Algunos de los trabajos que se presentan para solucionar estos inconvenientes son:

- Estudio de casos. es una técnica particularmente útil para profesiones como la administración, el derecho y la medicina, y que impulsó, inicialmente, la Escuela de Negocios de la Universidad de Harvard (Barnes, Christensen y Hansen, 1994). Con el estudio de casos se pretende la revisión de una situación hipotética o real, preparada para la clase, y en la cual se incentiva en los estudiantes la discusión de esa situación, con el fin de establecer una forma de solución a la problemática planteada.

- Juegos y simulaciones. En disciplinas como la administración y la negociación se emplean estas estrategias para simular situaciones similares a las que tendrían que enfrentar sus futuros practicantes, pero sin poner en riesgo empresas o personas reales. Senge (1994) reporta el uso de dos de tales estrategias: *el juego de la cerveza* y los *micromundos*. La primera experiencia, se emplea para enseñar a las diferentes generaciones de estudiantes y profesionales de administración los riesgos inherentes al manejo de inventarios sin generar riesgos en empresas reales. La segunda experiencia, se usa como una simulación de la realidad (de manera análoga a como se usan los simuladores de vuelo para enseñar aviación) en ciertas disciplinas y profesiones para generar prácticas de ciertos aspectos de las mismas, tales como compra y venta de acciones en bolsa o negociación de algún asunto. Christopher y Smith (1991) presentan *el juego de la negociación*, un conjunto de estrategias lúdicas que permiten simular procesos de negociación y aprender de ellos.

- Caricaturas como parte del proceso de enseñanza-aprendizaje. Diversas disciplinas y profesiones han empleado las caricaturas impresas y animadas como una estrategia didáctica para afianzar los conocimientos requeridos por sus practicantes. Algunas de ellas se listan seguidamente:

  - Mains (1945) propone la enseñanza de gramática mediante el uso de caricaturas.

  - Hess y Kaplan (1975) presentan una revisión del uso hasta ese entonces de las caricaturas políticas como medio educativo. Este mismo uso fue acogido por el proyecto *Opper* de la Universidad del Estado de Ohio (Opper, 2007).

  - Lochrie (1992) propone el uso de caricaturas para generar pensamiento crítico e incrementar la participación de los estudiantes de secundaria.

  - Scanlan y Feinberg (2000) promueven el uso de la serie animada *los Simpsons* para estudiar comportamientos desde el punto de vista de la sociología.

  - Perales y Vilchez (2002) proponen la enseñanza de la física en estudiantes de secundaria por medio de caricaturas animadas como

Pokemon o *los Simpsons*, enfocando los estudiantes en situaciones que contradigan las leyes físicas en esos programas.

- Hyun (2006) reporta la creación de caricaturas, al interior de una asignatura, como una manera de enseñar temas específicos de manejo de suelos. Las caricaturas se publicaban en un periódico de granjeros y aparecían semanalmente.

- Una propuesta reciente incluye caricaturas conceptuales (concept cartoons, 2007) para la enseñanza de las ciencias, las matemáticas y los idiomas a estudiantes de primaria y secundaria.

En Ingeniería de Software, en particular, los trabajos en estrategias complementarias a la enseñanza tradicional sólo se ocupan de los dos primeros tópicos mencionados, de la siguiente manera:

- El estudio de casos se utiliza para determinar la capacidad de uso de las aplicaciones (Carroll y Rosson, 2005), que es una aplicación de la Interacción Humano Computador (HCI por sus siglas en inglés) y en otra de sus ramas denominada diseño centrado en el usuario (HCD, por sus siglas en inglés; Seffah, 2003). También se ha empleado en cursos de Ingeniería de Software a nivel general (Busenberg y Tam, 1979).

- En lo relativo a los juegos, se han reportado muchísimas aplicaciones, especialmente en el uso de juegos de computador o video para afianzar ciertos conocimientos o para realizar prácticas en ciertas áreas como:

  - Patrones de diseño, empleando el denominado juego de la vida (Wick, 2004 y 2005); patrones de diseño y polimorfismo empleando el juego de los conjuntos (Hansen, 2004).

  - Fases de la Ingeniería de Software (Claypool y Claypool, 2005) y procesos de Ingeniería de Software empleando SimSE (Navarro y Van der Hoek, 2004).

  - Algoritmos, gráficos, inteligencia artificial y procesamiento paralelo empleando Reversi (Valentine, 2005).

  - Lenguajes de programación y programación orientada a objetos (Sanders, 2005; Kafai, 1996).

- Juegos no tecnológicos, como *problemas y programadores* (Baker *et al*., 2005) y *el juego de los requisitos* (Zapata y Awad, 2005).

No se encontraron trabajos específicamente en Ingeniería de Software que empleen caricaturas animadas o impresas como una estrategia de enseñanza-aprendizaje correspondiente a esta profesión.

SECCIÓN 7.3: JUSTIFICACIÓN DE LA PROPUESTA

En este libro, se optó por la utilización de caricaturas como un medio de acercamiento a los estudiantes de Ingeniería de Software, puesto que esta estrategia se practica con éxito en otras disciplinas, aunque en públicos más diversos (por lo general estudiantes de primaria y secundaria). Se busca explotar la afinidad que puedan tener los estudiantes de esta profesión con las caricaturas, para propiciar un espacio de discusión que permita afianzar los conocimientos impartidos durante los diferentes cursos de Ingeniería de Software. Igualmente, se pretende generar un espacio más ameno en el que se hable de la Ingeniería de Software, pero no con la seriedad que lo hacen los libros formales sobre el tema, sino en un ambiente que pueda propiciar el diálogo con los estudiantes en espacios diferentes a las clases expositivas.

Yang (2003), propone un conjunto de ventajas del uso de caricaturas como estrategia de enseñanza: motivación, impacto visual, permanencia, intermediación y popularidad. Seguidamente, se realiza un análisis de esas ventajas a la luz de la propuesta que se ha impulsado en este libro para la enseñanza de la Ingeniería de Software.

- Motivación: Mantener la atención de los estudiantes en temas “serios” de Ingeniería de Software es cada vez más difícil. Por el contrario, los estudiantes no tienen inconveniente para centrar su atención durante horas en caricaturas impresas o incluso en caricaturas animadas. El éxito de juegos como *Yu-gi-oh!*® o series animadas como *Pokemon* o *Dragon Ball Z*, aún en jóvenes universitarios es una prueba de ello. En estos casos, la motivación juega un papel preponderante, dado que se despierta el interés por la continuación de las historias que se narran en dichas series. Igualmente, las versiones impresas de esas series son igualmente exitosas, lo que genera toda una cultura de las caricaturas entre los jóvenes.

- Impacto Visual y Permanencia: Las caricaturas son de alta recordación, en tanto que la memoria se debe ejercitar mucho más cuando se trata de conceptos textuales. Se trata, entonces, de generar un impacto visual en los estudiantes de forma tal que se puedan asociar los textos con las caricaturas a las que sirven de complemento en cada uno de los capítulos y secciones de este libro.

- Intermediación: En Ingeniería de Software, el modelado es una de las estrategias fundamentales para lograr la solución de los problemas. Una de las bases del modelado es la simplificación de los problemas complejos empleando la estrategia "divide y vencerás", con el fin de descomponer tales problemas en subproblemas cuya solución sea más sencilla. Con el uso de las caricaturas se logra un efecto similar: la realidad compleja se transforma en ideas más sencillas y fáciles de entender, que facilitan la aprehensión de los conceptos por parte de los estudiantes.

- Popularidad: En las caricaturas animadas o impresas se recurre a la popularidad de los personajes para "grabar" en la memoria de los televidentes o los lectores las diferentes situaciones en que se ven involucrados. Esta estrategia es poco usada en este libro (salvo la alusión que se hizo a Donald Trump en la sección 6.1). Sin embargo, se espera que la lectura del libro pueda impactar la memoria de los lectores, de forma tal que los personajes que acá se han propuesto (analista, desarrollador, software, interesado e incluso el "grafitero" y el vampiro) impacten la memoria del lector y puedan facilitar el proceso de recordación de los diferentes conceptos que se plasman en el libro.

SECCIÓN 7.4: CONCLUSIONES Y TRABAJO FUTURO

Este libro, constituye un acercamiento a una estrategia de aprendizaje que se practica con éxito en otras áreas (como la física, las matemáticas o la historia) y dirigida a otros públicos (estudiantes de primaria y secundaria), pero que, en este caso, se aplica específicamente a los estudiantes universitarios de Ingeniería de Software. Esta forma de empleo de las caricaturas para tratar de afianzar los conceptos que se imparten en el estudio de esta profesión, es novedosa, en tanto no se conocen experiencias previas que impulsen esta forma de enseñanza para este público en particular y con el uso de los conceptos de la Ingeniería de Software. Se señala, adicionalmente, que no se pretende la sustitución de las estrategias convencionales de enseñanza de esta profesión, es decir las clases expositivas y los proyectos prácticos, sino, por el

contrario, lo que se pretende es complementar dichas estrategias con otras que contribuyan a facilitar el aprendizaje significativo de los estudiantes en estos temas.

Se espera que este libro se pueda usar en los cursos regulares de Ingeniería de Software en las diferentes universidades, como una manera didáctica y divertida de "romper el hielo" generado por temas tan serios como los que se abordan, para emplear las ventajas que tradicionalmente se han asociado con este tipo de enseñanza en otras áreas y disciplinas, tales como la motivación, el impacto visual, la permanencia, la intermediación y la popularidad. Estas ventajas, en conjunción con la guía apropiada por parte de los docentes, pueden conducir a la reducción de la complejidad de los temas de la Ingeniería de Software y, consecuentemente, a un acercamiento mayor a los futuros practicantes de esta profesión.

Queda mucho trabajo por hacer en esta área, especialmente en la difusión de este tipo de estrategias y la evaluación de los resultados concretos que se puedan generar, mediante experimentos que comprueben las bondades de esta estrategia de enseñanza como complemento a las estrategias tradicionales con las cuales se ha pretendido formar a los futuros ingenieros de software. Se espera, igualmente, poder aplicar esta estrategia en otras ramas de la tecnología afines con la Ingeniería de Software.

Este trabajo hace parte de la estrategia que viene desarrollando el grupo de Investigación en Ingeniería de Software de la Escuela de Sistemas de la Facultad de Minas, Universidad Nacional de Colombia, sede Medellín, como un intento de hacer más accesibles los diferentes temas de Ingeniería de Software a un público cada vez más joven, como es el grupo de estudiantes que día a día está ingresando en las facultades de ingeniería de las diferentes universidades. Como parte de este proceso, se están desarrollando algunas otras estrategias, como el uso de juegos no tecnológicos en el aula de clase para el afianzamiento de los diferentes conceptos de la ingeniería de software y la creación de un juego de roles en red que ayude a los estudiantes a comprender el intrincado mundo de las compañías de desarrollo de software. En esta iniciativa falta aún muchísimo trabajo, pero paulatinamente se vienen dando pasos en esta dirección, para lograr que la enseñanza de estos temas se desmitifique y se acerque cada vez más a los estudiantes, que son, finalmente, la razón de ser de todas estas iniciativas.

AGRADECIMIENTO

Este libro hace parte del proyecto de investigación “Definición de un esquema preconceptual para la obtención automática de esquemas conceptuales de UML”, financiado por la DIME. Este tipo de fomento permite que la creación intelectual de los investigadores colombianos se vea plasmada en materializaciones concretas como este trabajo. A esa entidad, nuestra gratitud eterna.

REFERENCIAS

Bagert, D., Hilburn, Th., Hislop, G., Lutz, M., McCracken, M. y Mengel, S. (1999). Guidelines for Software Engineering Education Version 1.0. *Technical Report CMU/SEI-99-TR-032 ESC-TR-99-002,* Software Engineering Institute, Carnegie Mellon University.

Baker, A., Navarro, E. y van der Hoek, A. (2005). An experimental card game for teaching software engineering processes. En: *The Journal of Systems and Software,* Vol. 75, 3–16.

Barnes L. B., Christensen, C. R. y Hansen, A. B. (1994). *Teaching and the case method: text, cases, and readings.* Harvard Business School Press, Boston.

Booch, G., Rumbaugh, J. y Jacobson, I. (1999). *The Unified Modeling Language User Guide*. Addison-Wesley, Reading.

Budgen, D. y Tomayko, J. (2005). The SEI Curriculum Modules and their influence: Norm Gibbs' legacy to Software Engineering Education. En: *Journal of Systems and Software*, Vol. 75, 55–62.

Busenberg, S. y Tam, W. (1979). An Academic Program Providing Realistic Training in Software Engineering. En: *Communications of the ACM*, Volumen 22, No. 6, 341–345.

Carroll, J. y Rosson, M. B. (2005). A Case Library for Teaching Usability Engineering: Design Rationale, Development, and Classroom Experience. En: *ACM Journal on Educational Resources in Computing*, Vol. 5, No. 1, 1–22.

Chen, P. P. (1976). The entity-relationship model—toward a unified view of data. En: *ACM Transactions on Database Systems (TODS)*, Vol. 1 No. 1, 9–36.

Claypool, K. y Claypool, M. (2005). Teaching software engineering through game design. En: *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*, Caparica, 123–127.

Concept Cartoons. (2007). Disponible en el sitio http://www.conceptcartoons.com/index_flash.html [última consulta Septiembre 12 de 2007].

Cowling, A. J. (2005). The role of modeling in the software engineering curriculum. En: *The Journal of Systems and Software,* No. 75, 41–53.

Christopher, E. y Smith, L. (1991). *Negotiation Training through Gaming: strategies, tactics and manoeuvres*. Kogan, London.

Fairley, R. (1978). Educational Issues in Software Engineering. En: *Proceedings of the 1978 annual conference ACM/CSC-ER*, Washington, D.C., 58–62.

Gardner, H. (1983). *Frames of mind: The theory of multiple intelligences*. Basic Books, New York.

Gibbs, W. (1994). Software's Chronic Crisis. En: *Scientific American*, Septiembre 1994, 72–81.

Hansen, S. (2004). The Game of Set®–An Ideal Example for Introducing Polymorphism and Design Patterns. En: *Proceedings of the 35th SIGCSE technical symposium on Computer science education*, Norfolk, 110–114.

Hess, S. y Kaplan, M. (1975). *The Ungentlemanly Art: A History of American Political Cartoons*. Macmillan, New York.

Hyun, H. (2006). Cartoon as a teaching aid for soil sciences. En: *18th world congress of Soil Science*, Philadelphia.

ISO/IEC (1999). Information technology—Software product evaluation—Part 1: General overview. En: *TECHNICAL REPORT ISO/IEC 14598-1:1999(E)*.

ISO/IEC (2001). Software engineering—Product quality—Part 1:Quality model. En: *International Standar ISO/IEC 9126-1:2001(E)*.

Kafai, Y. (1996). Software by Kids for Kids. En: *Communications of the ACM*, Vol. 39, No. 4, 38–39.

Kitchenham, B., Budgen, D., Brereton, P. y Woodall, Ph. (2003). An investigation of software engineering curricula. En: *The Journal of Systems and Software*, Vol. 74, 325–335.

Lamsweerde, A., Dardenne, A. y Fichas, S. (1993). Goal-directed Requirements Acquisition. En: *Science of Computer Programming*, Vol. 20. pp. 3-50.

Lochrie, K. (1992). Using Cartoons as an Effective Learning & Teaching Strategy. En: *Research in Education*, No. 51.

Mains, E. (1945). The Cartoon and the Teaching of Grammar. En: *The English Journal*, Vol. 34, No. 9, 506–507.

Navarro, E. y Van der Hoek, A. (2004). *SimSE: an educational simulation game for teaching the Software engineering process*. En: Proceedings of

the 9th annual SIGCSE conference on Innovation and technology in computer science education, Leeds, 233–233.

Opper, F. (2007). *The Opper Project: using editorial cartoons to teach history*. Disponible en el sitio: http://hti.osu.edu/opper/index.cfm [última consulta Septiembre 12 de 2007].

Perales F. y Vilchez, J. (2002). Teaching physics by means of cartoons: a qualitative study in secondary education. En: Physics Education, Vol. 37, 400–406.

Rugarcia, A., Felder, R., Woods, D. y Stice, J. (2000). The Future of Engineering Education I: The vision for a new century. En: *Chemical Engineering Education*, Vol. 34, No. 1, 16–25.

Sanders, D. (2005). Software tools to support an objects-first curriculum. En: *Journal of Computing Sciences in Colleges*, Vol. 20, No. 4, 37–38.

Scanlan, S. y Feinberg, S. (2000). The Cartoon Society: Using The Simpsons to Teach and Learn Sociology. En: *Teaching Sociology*, Vol. 28, No. 2.

Senge, P. (1994). *The Fifth Discipline: The Art and Practice of the Learning Organization*. Currency Doubleday, New York.

Seffah, A. (2003). Learning the ropes: Human-Centered Design Skills and Patterns for Software Engineers' Education. En: *Interactions*, Vol. 10, No. 5, 36–45.

Soloway, E., Guzdial, M. y Hay, K. (1994). Learner-Centered Design: The Challenge For HCI In The 21st Century. En: *Interactions*, Vol. 1, No. 2, 36–48.

Valentine, D. (2005). Playing around in the CS curriculum: reversi as a teaching tool. En: *Journal of Computing Sciences in Colleges*, Vol. 20, No. 5, 214–222.

Wankat, P.C. y Oreovicz, F.S. (1993). *Teaching Engineering*, McGraw-Hill, Nueva York.

White, R. W. (1959). Motivation reconsidered: the concept of competence. En: *Psychological review*, Vol. 66, 297–333.

Wick, M. (2004). Using the game of life to introduce freshman students to the power and elegance of design patterns. En: *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, Vancouver, 103–105.

Wick, M. (2005). Teaching Design Patterns in CS1: a Closed Laboratory Sequence based on the Game of Life. En: *Proceedings of the 36th SIGCSE technical symposium on Computer science education*, St. Louis, 487–491.

Yang, G. (2003). *Comics in Education.* Disponible en el sitio http://www.humblecomics.com/comicsedu/index.html [última consulta Septiembre 12 de 2007].

Zapata, C. M. y Awad, G. (2005). El Juego de los Requisitos: Enseñanza de la Gestión de Proyectos de Software. En: *Memorias del XIII Congreso Iberoamericano de Educación Superior en Computación CIESC2005*, 33–43.

Zapata, C. M., Gelbukh, A. y Arango, F. Pre-conceptual Schemas: a Conceptual-Graph-like Knowledge Representation for Requirements Elicitation. En: *Lecture Notes in Computer Sciences*, Vol. 4293, 2006, pp. 17–27.