Revista Educación en Ingeniería
ISSN 1900-8260
Enero a Junio de 2012, Vol. 7, N°. 13, pp. 32-43 • © 2012 ACOFI • http://www.educacioneningenieria.org
Recibido: 05/01/2012 • Aprobado: 17/05/2012 • Versión final: 25/05/2012

DESAFÍOS DEL CURSO DE INGENIERÍA DE *SOFTWARE*

CHALLENGES FOR THE SOFTWARE ENGINEERING COURSE

Gabriela Salazar Bermúdez
Universidad de Costa Rica, San José (Costa Rica) • gabriela.salazar@ecci.ucr.ac.cr

Resumen

Este artículo describe la experiencia de enseñar el curso Ingeniería de *Software* a estudiantes de pregrado en la Escuela de Computación de la Universidad de Costa Rica. Durante estos últimos tres años se han estado introduciendo cambios en la metodología de desarrollo con el fin de solucionar problemas en la curva de aprendizaje y en el proceso de pruebas. El introducir prácticas de *Scrum* y programación extrema, combinadas con el Proceso Unificado Racional (RUP) que es el que tradicionalmente se ha utilizado, ha logrado mejorar la curva de aprendizaje y obtener productos de mejor calidad. El artículo expone los desafíos que se han venido presentando, la forma como se están enfrentando y los beneficios obtenidos al aplicar la combinación de dichas metodologías. Los puntos descritos en este artículo pueden interesar a profesores que desean formar ingenieros de *software*.

Palabras clave: ingeniería del *software*, metodologías pesadas y ágiles, aseguramiento de la calidad de software

Abstract

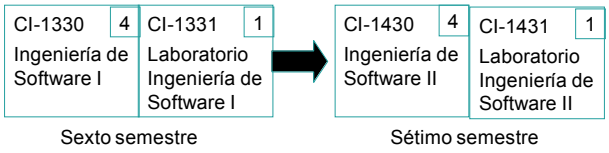
This article describes the experience of teaching the Software Engineering course to undergraduate students in the Escuela de Computación of the Universidad de Costa Rica. Over the past three years, changes in the methodology have been introduced with the purpose of solving problems in learning curve and weaknesses in testing process. By introducing Scrum practices and extreme programming combined with Rationale Unified Process (RUP), which is normally used, it has been able to improve the learning curve and to obtain products of better quality. The article presents the challenges that have been arisen, the way they have been managed and the benefits obtained by applying those methodologies. The items described in this article may be of interest for teachers who want to train future software engineers.

Keywords: software engineering, heavy and agile methodologies, quality assurance software

Introducción

En el programa de bachillerato de computación e informática en la Universidad de Costa Rica, la ingeniería de *software* se imparte a través de los cursos Ingeniería de *Software* I e Ingeniería de Software II, con sus respectivos laboratorios.

Figura 1. Secuencia de cursos de ingeniería de software según el programa de estudios



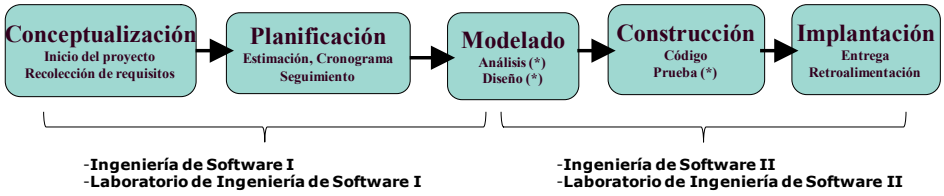
En la figura 1 se muestra la secuencia de los cursos, los cuales son semestrales, uno es requisito del otro y cada uno tiene una duración aproximada de 16 semanas lectivas. Específicamente Ingeniería de *Software* I y II son cursos teóricos en donde se les enseña metodologías, técnicas y herramientas de ingeniería de *software*. Se imparten en 4 horas lectivas semanales y por eso tienen un valor de 4 créditos cada uno. Los cursos de laboratorio se ofrecen en 2 horas lectivas semanales, valen 1 crédito cada uno y los estudiantes aprenden a aplicar el conocimiento teórico en un proyecto que inician en Ingeniería de *Software* I y terminan en Ingeniería de *Software* II.

Estos cursos se complementan con cursos electivos como: formulación y administración de proyectos de *software*, verificación y validación de *software*, diseño de interfaz humano computadora, arquitectura de sistemas de *software* y disponibilidad de *software*, entre otros.

Durante estos últimos tres años se han venido evaluando nuevos procesos de desarrollo de *software* debido principalmente a las siguientes deficiencias:

Curva de aprendizaje lenta: la metodología de desarrollo que se utilizaba era exclusivamente el Proceso Unificado Racional (RUP, por las siglas en inglés *Rational Unified Process*). Para adaptar este proceso al proyecto se dividían las fases genéricas entre los dos semestres del curso, de manera que durante el primer semestre los estudiantes iniciaban el proyecto realizando las fases de comunicación, planificación y modelado a nivel conceptual. Durante el siguiente semestre modelaban el diseño detallado y continuaban con las fases de construcción y despliegue.

Figura 2. Marco de trabajo del proceso de software (Pressman, 2010)



En la figura 2 se muestra la distribución de las actividades en los dos semestres. La distribución de las actividades del proceso en los dos semestres que dura el curso, y el hecho de que haya aproximadamente 8 semanas de vacaciones entre los dos semestres, hace que la curva de aprendizaje quede afectada negativamente. Continuar la implementación con base en lo trabajado en el primer semestre, conduce a que los estudiantes olviden el proceso de desarrollo y olviden la funcionalidad del *software*: se les hace difícil recordar de dónde vienen y hacia dónde van y, como consecuencia, deben repetir trabajo lo que, a su vez, produce desgaste y desmotivación.

Proceso de pruebas débil: no se planificaban las pruebas, ni se utilizaban herramientas de *software* que permitieran automatizar este proceso. Las pruebas se realizaban en el segundo semestre con base en los requerimientos definidos en el primero.

Adicionalmente, han surgido nuevas tendencias de desarrollo de *software* como las metodologías ágiles, que cada vez son más utilizadas por la industria y de alguna manera obligan a los formadores de los futuros profesionales a conocerlas y tomar lo mejor de ellas. Durante estos últimos tres años se han estado incorporando dichas prácticas que combinadas con el RUP, han logrado corregir las deficiencias anteriores.

Copyright © 2012 Asociación Colombiana de Facultades de Ingeniería

Modelos de proceso de software

De acuerdo con Pressman (2010), el proceso de desarrollo del *software* se define como un marco de trabajo para la tarea que se requiere en la construcción de *software* de alta calidad. Ese marco de trabajo se compone de un pequeño número de actividades genéricas: comunicación, planificación, modelado, construcción y despliegue, aplicables a todos los proyectos, sin importar el tamaño o complejidad. Existen dos tipos de modelos:

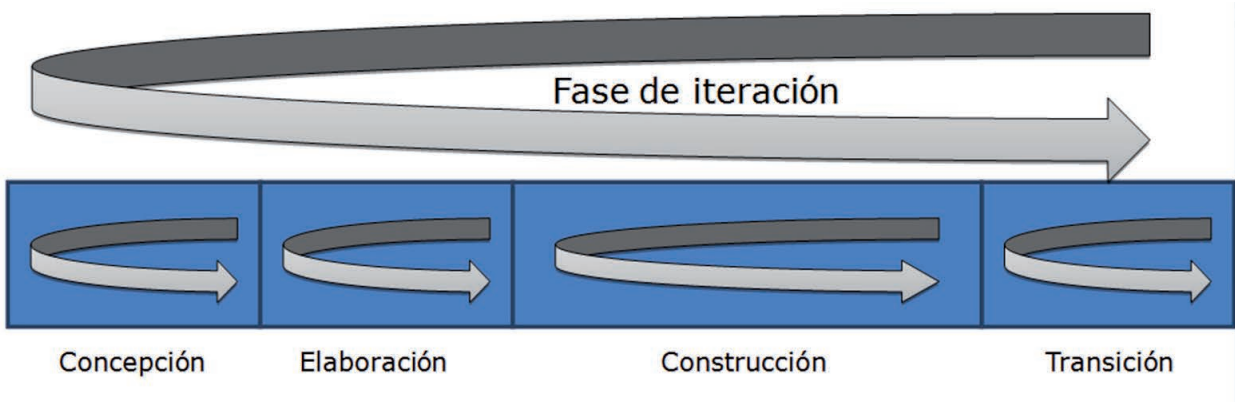
- *Prescriptivos:* llamados "metodologías pesadas" que enfatizan la definición, la identificación y la aplicación detallada de actividades y tareas del proceso. Las salidas de una etapa se usan de base para planear la siguiente. Dentro de los modelos prescriptivos existen diferentes tipos de entre ellos: cascada, proceso unificado, procesos incrementales, procesos evolutivos, procesos especializados, entre otros.
- Á*giles:* siguen un conjunto de principios que conducen a un enfoque más informal del proceso. Son ágiles porque acentúan la maniobrabilidad y la adaptabilidad del proceso y tienen como filosofía satisfacer al cliente, hacer entregas tempranas de *software* por incrementos, trabajar con equipos pequeños muy motivados, utilizar métodos de trabajo informales, implementar un mínimo de productos de trabajo de ingeniería de *software* y lograr simplicidad en el desarrollo.

A continuación se explican brevemente algunas metodologías combinadas que son las que se incluyen en la propuesta del curso: proceso unificado racional, programación extrema y *Scrum*.

Proceso Unificado Racional

El RUP es el resultado de varios años de investigación y uso práctico en el que se han unificado técnicas de desarrollo a través del UML (UML, por sus siglas en inglés, *Unified Modeling Language*).

Figura 3. Fases en el RUP (Sommerville, 2011)



El RUP es un modelo que identifica cuatro fases discretas en el proceso de software, tal como se muestra en la figura 3. De acuerdo con Sommerville (2011) las fases son:

- **Concepción o inicio**: la meta es establecer un caso empresarial para el sistema. Se deben identificar todas las entidades externas que interactuarán con el sistema y se define el alcance. Con base en esto, se evalúa la viabilidad del sistema hacia la empresa.
- **Elaboración**: se desarrolla la comprensión del dominio del problema, se establece un marco conceptual arquitectónico para el sistema, se diseña el plan y se identifican los riesgos. Al completar esta fase se obtiene un modelo de requerimientos para el sistema que podrían ser: una serie de casos de uso de UML, una descripción arquitectónica y un plan de desarrollo del *software*.
- **Construcción**: esta fase incluye el diseño, la programación y las pruebas. Partes del sistema se desarrollan en paralelo y se integran durante esta fase. Al completar esta fase debe obtenerse un sistema de *software* funcionando y la documentación lista para entregarse al usuario.

Copyright © 2012 Asociación Colombiana de Facultades de Ingeniería

- **Transición**: la fase final se interesa por el cambio del sistema desde la comunidad de usuarios y el funcionamiento en un ambiente real.

El ciclo de vida del RUP se caracteriza por lo siguiente:

- **Dirigido por casos de uso**: los casos de uso (CU) reflejan lo que los futuros usuarios necesitan y desean, lo cual se capta cuando se modela el negocio y se representa a través de los requerimientos. Los CU guían el proceso de desarrollo debido a que los modelos que se obtienen como resultado de los diferentes flujos de trabajo, representan su realización.
- **Centrado en la arquitectura**: la arquitectura muestra la visión común del sistema completo en la que el equipo de proyecto y los usuarios deben estar de acuerdo, por lo que describe los elementos del modelo que son más importantes para su construcción, los cimientos del sistema que son necesarios como base para comprenderlo, desarrollarlo y producirlo económicamente. RUP se desarrolla mediante iteraciones, comenzando por los CU relevantes desde el punto de vista de la arquitectura.
- **Iterativo e incremental**: RUP propone que cada fase se desarrolle en iteraciones. Una iteración involucra actividades de todos los flujos de trabajo, aunque se desarrollan fundamentalmente algunos más que otros. Por ejemplo, una iteración de elaboración centra su atención en el análisis y diseño, aunque también refina los requerimientos y obtiene un producto con un determinado nivel que irá creciendo incrementalmente en cada iteración.

Programación extrema

Villena (2008) señala que: "*extreme programming* (XP) es la propuesta ágil más importante en la actualidad, recoge desde la industria diversas prácticas reconocidas por su aporte en el éxito de los proyectos, y propone llevarlas al extremo". Algunas buenas prácticas ágiles utilizadas por la industria descritas por Villena (2008) y Sommerville (2011) se presentan a continuación:

- Metáfora: XP propone que cada aplicación tenga una integridad conceptual basada en una metáfora simple que facilita el entendimiento del sistema a desarrollar.
- Propiedad colectiva de código: en donde todos los desarrolladores son responsables y dueños de todo el código del sistema, de manera que comparten el conocimiento y evitan dependencias excesivas en un desarrollador específico. Esto se refuerza con la práctica ágil de "programación en parejas".
- Diseño simple: se realiza un diseño suficiente para cubrir los requerimientos actuales. De esta manera se eliminan "áreas grises" con código que nunca se usa y se ahorra tiempo de los desarrolladores al no desperdiciar esfuerzos en funcionalidades que podrían ser nunca utilizadas. Además, ninguna funcionalidad debe estar implementada más de una vez y si hay duplicidad, se debe factorizar para conservar el código simple, de bajo costo de mantenimiento y reutilizable.
- Refabricación: es una manera disciplinada de "limpiar" el código después de escrito (modifica/simplifica el diseño interno sin alterar el comportamiento externo).
- Planificación de *sprints*: consiste en un juego colaborativo en donde clientes y desarrolladores definen el alcance de cada iteración del desarrollo. Los desarrolladores estiman el tiempo de cada iteración y el cliente asigna prioridades. Cada incremento es probado y aceptado por el cliente. El resto de los requerimientos es re-examinado considerando posibles cambios en las funcionalidades y/o en las prioridades, y se define la siguiente iteración.
- Integración continua: consiste en ensamblar periódicamente y de forma automática los módulos del sistema. Las asignaciones de código se dividen en pequeñas tareas y cuando cada tarea se termina, se integra al código base colectivo.
- Cliente en sitio: permite que el cliente esté siempre disponible para formar parte del equipo, es accesible a los desarrolladores con el fin de aclarar y validar los requerimientos.
- Desarrollo guiado por pruebas: cada segmento del código se construye a partir de una prueba que define su comportamiento correcto. Se le da importancia a los siguientes tipos de pruebas:
  - Pruebas unitarias: se diseñan casos de prueba de caja blanca automatizados antes de que se produzca el código. Antes de que un programador integre su código al código base debe haber pasado el 100% de sus casos de prueba.
  - Pruebas de aceptación: XP promueve el uso de casos de prueba de aceptación escritas por el

Copyright © 2012 Asociación Colombiana de Facultades de Ingeniería

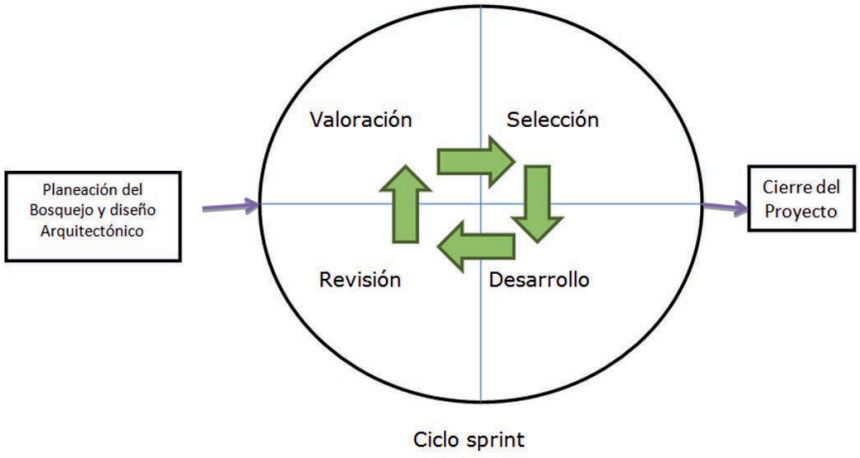
cliente para controlar la completitud del proyecto. Cuando un caso de prueba de aceptación ha pasado exitosamente, se considera que la funcionalidad especificada ha sido implementada apropiadamente. La completitud del proyecto está basada en el porcentaje de casos de pruebas de aceptación que han sido aprobadas.

- Programación en parejas: siempre hay por lo menos dos personas en un computador, colaborando en el mismo diseño, código o pruebas. Cada uno comprueba el trabajo del otro.
- Código estándar: el equipo norma un conjunto de definiciones y reglas que se definen luego de un largo proceso de análisis y diseño consensuado. Lo que se busca con esta práctica en el contexto de XP es que los desarrolladores usen convenciones comunes al escribir el código, de tal forma que se facilite el entendimiento compartido de éste.
- Entregables pequeños: implica ir entregando incrementos de valor al cliente a través de pequeños entregables lo antes posible. Al principio se desarrolla el conjunto mínimo de funcionalidad útil que ofrece valor al negocio. Las entregas son frecuentes (casi diarios) y van agregando funcionalidad en forma incremental.
- Ritmo sustentable: indica que el equipo debe realizar sus tareas dentro de la jornada normal de trabajo, dejando el esfuerzo de horas extras relegado sólo a situaciones extraordinarias, todo esto con el fin de mantener el equipo descansado y con capacidad máxima de producción.

Scrum

Aunque el enfoque de *Scrum* es un método ágil, su enfoque está en la administración iterativa del desarrollo, y no en enfoques técnicos específicos para la ingeniería de *software* ágil.

Figura 4. El proceso de Scrum (Sommerville, 2011)



La figura 4 muestra el proceso de administración de *Scrum*. Sommerville (2011) señala que considerando que este proceso no prescribe el uso de prácticas de programación, puede usarse con enfoques ágiles más técnicos como XP para ofrecer al proyecto un marco administrativo.

Scrum presenta tres fases: la primera es la planeación en donde se establecen los objetivos generales del proyecto y el diseño de la arquitectura de *software*. A esto le siguen una serie de ciclos *sprint*, en donde cada ciclo desarrolla un incremento del sistema. Finalmente, la fase de cierre concluye el proyecto, completa la documentación requerida como la ayuda y los manuales del usuario y valora las lecciones aprendidas en el proyecto (Sommerville, 2011).

La característica innovadora de *Scrum* es su fase central compuesta por los ciclos *sprint*. Un ciclo *sprint*

Copyright © 2012 Asociación Colombiana de Facultades de Ingeniería

es una unidad de planeación en la que se valora el trabajo a realizar, se seleccionan las particularidades a desarrollar y se implementa el software. Las características claves de este proceso son las siguientes (Sommerville, 2011):

- Los *sprints* tienen una longitud fija, por lo general de 2 a 4 semanas.
- El punto de partida para la planeación es la cartera del producto, que es la lista de trabajos por realizar. Durante la valoración se revisa y se asignan prioridades y riesgos. El cliente interviene en este proceso y al inicio de cada *sprint* puede introducir nuevos requerimientos o tareas.
- La fase de selección incluye a todo el equipo para seleccionar las características y la funcionalidad a desarrollar durante el *sprint*.
- Diariamente se realizan reuniones breves con todos los miembros del equipo, con el fin de revisar el progreso y, si es necesario, volver a asignar prioridades al trabajo. Durante esta etapa el equipo se separa del cliente y todas las comunicaciones se realizan a través del maestro *Scrum para* proteger al equipo de distracciones externas. El trabajo técnico depende del problema a resolver y del equipo; algunas prácticas de XP se utilizan cuando se consideran necesarias.
- Al final del *sprint*, el trabajo realizado se revisa y se presenta a los participantes. Luego comienza el siguiente ciclo.
- El maestro *Scrum* es un facilitador que ordena las reuniones diarias, rastrea el atraso del trabajo, registra las decisiones, mide el progreso del atraso y se comunica con los clientes y administradores fuera del equipo.

Metodología

En esta sección se describe la forma como se combinaron y aplicaron las metodologías ágiles y pesadas en el curso.

Aplicación de la combinación de Scrum y RUP

En la metodología propuesta se aplica el enfoque de administración iterativa de *Scrum* durante la planificación del proyecto y conforme se va implementando la aplicación este enfoque facilita la gestión de cada uno de los *sprint*s. Los equipos de trabajo son compuestos por cuatro miembros en donde uno de ellos asume el papel de maestro *Scrum. Este estudiante en su rol de líder o maestro Scrum s*e encarga de administrar el proyecto y de canalizar la coordinación con el profesor.

Al inicio del proyecto el profesor le entrega a los estudiantes una definición básica de los requerimientos que les sirve para conceptualizar la aplicación, estimar su tamaño y esfuerzo, y obtener un plan global que incluye un cronograma a nivel macro. El obtener una visión global de la aplicación les permite estimar el esfuerzo de los requerimientos y agruparlos por ciclos *sprints* de acuerdo con la complejidad y a las necesidades del usuario.

El profesor en su rol de cliente participa en la priorización de requerimientos y en la introducción de mejoras al inicio de cada *sprint*. La distribución de los *sprints* entre los dos semestres del curso la realiza el profesor en conjunto con los estudiantes.

Figura 5. Planificación del proyecto y definición de los sprints

En estas fases iniciales se realizan tareas adicionales de RUP como: la definición de la arquitectura y el modelado de requerimientos en formato simple (solo se describe el camino feliz) siguiendo las recomendaciones de Larman (1999).

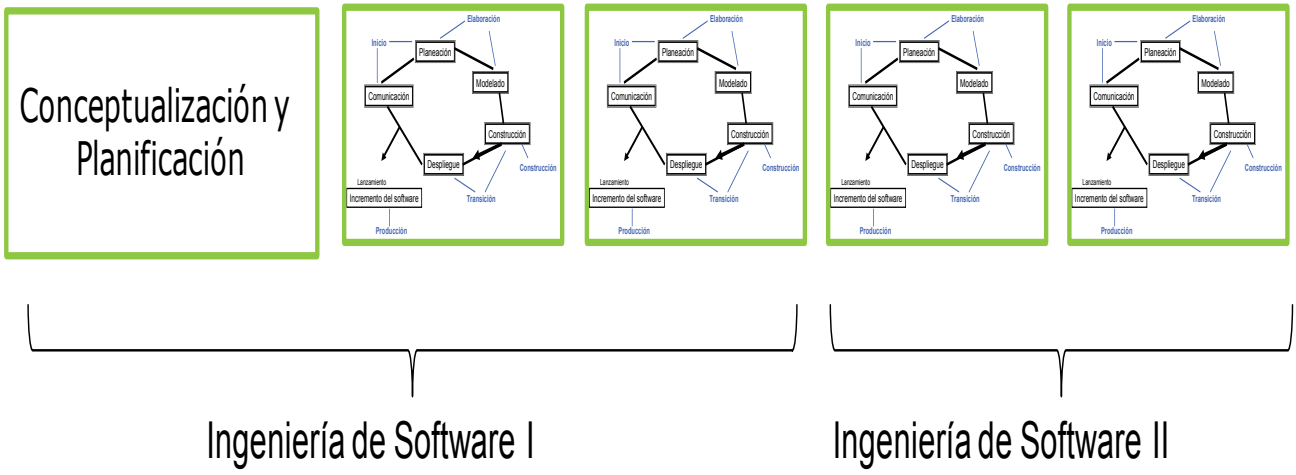
En la figura 5 se muestra cómo se definen los sprints a desarrollar, producto de las actividades de conceptualización y planificación.

Uso de RUP y XP

En la figura 6 se muestra la distribución de los ciclos *sprint*s a través de los dos semestres que dura el curso y la aplicación del RUP en cada ciclo. La cantidad de *sprint*s que se desarrollan en cada semestre, dependen de la complejidad de los requerimientos que se están implementando. Cada *sprint* puede tener uno o dos módulos pero se intenta que la duración de cada *sprint* sea de cuatro semanas.

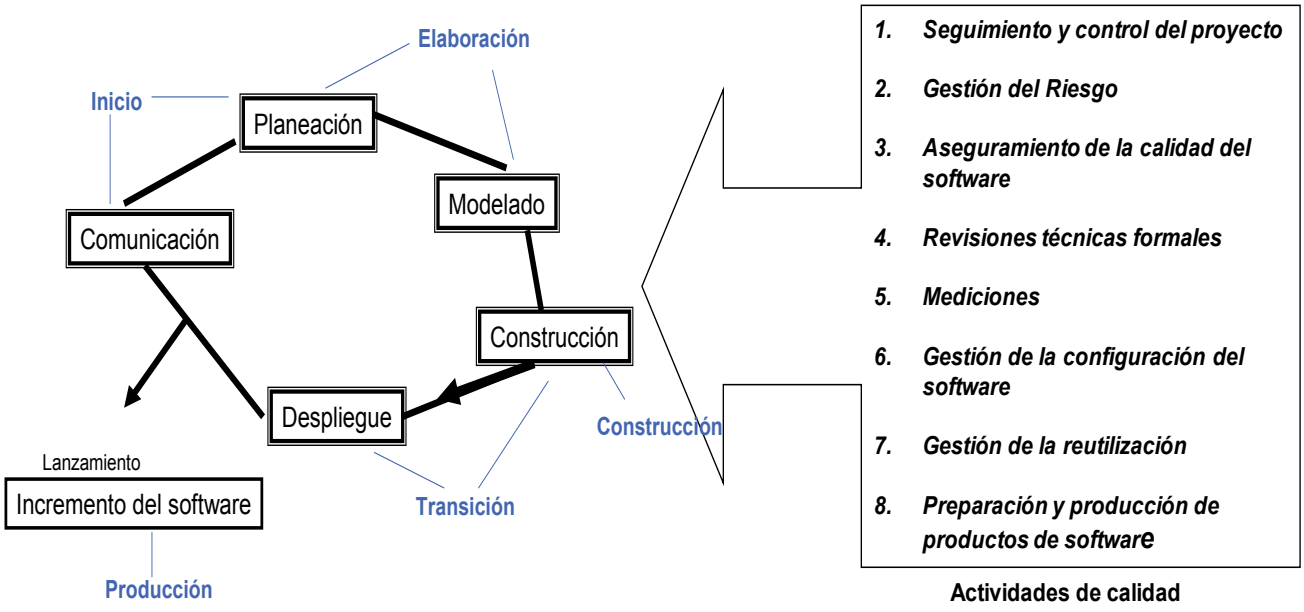
Copyright © 2012 Asociación Colombiana de Facultades de Ingeniería

Figura 6. Aplicación de RUP en cada *sprint*



Durante el desarrollo de cada *sprint* se modelan los requerimientos con base en la técnica de casos de uso de UML y se actualiza el plan de proyecto. En la construcción de cada *sprint* se combina RUP con ciertas prácticas de metodologías ágiles tomadas específicamente de XP descritas en la tabla 1.

Figura 7. Verificación y validación de la calidad en cada *sprint*



En la figura 7 se muestra cómo las actividades de verificación y validación de la calidad del *software* son realizadas durante el desarrollo de cada *sprint*. Cada una de las fases del ciclo de vida (planificación, análisis, diseño, pruebas) se realizan utilizando plantillas provistas por el profesor y elaboradas según IEEE (2003), *Software Engineering Institute* (2006) y *Project Management Institute* (2005), entre otras.

Copyright © 2012 Asociación Colombiana de Facultades de Ingeniería

Tabla 1. Prácticas de XP que combinadas con RUP se utilizan en el curso

Práctica ágil	Aplicación de la práctica en el curso
Metáfora	Los estudiantes asocian los términos del sistema con posibles objetos de la vida real durante las fases de conceptualización, análisis de los requerimientos y diseño.
Propiedad colectiva de código	Una vez ingresado el código fuente todos los miembros de un mismo equipo tienen acceso para agregar o cambiar código sin solicitar permiso a través de herramientas automatizadas de control de versiones. Muchos cambios se dan producto de la programación en parejas.
Diseño simple y refactorización	Los estudiantes realizan un diseño simple durante las fases de conceptualización y planificación de manera que les permita planificar el proyecto a nivel general y determinar el número de *sprints*. Posteriormente cuando trabajan en cada iteración, mantienen la funcionalidad y mejoran el código del prototipo inicial de interfaces y las estructuras.
Planificación de *sprints*	Se utiliza la metodología Scrum con las adaptaciones expuestas en la sección 2.3.
Código estándar	Desde el inicio del desarrollo los estudiantes definen convenciones de programación para codificar y documentar el código. Por ejemplo definen convenciones para los nombres de las clases, los nombre de los métodos, etc..
Integración continua	Las asignaciones de código las dividen en pequeñas tareas y cuando se termina la tarea la integra al código colectivo. El uso de herramientas automatizadas para el control de versiones y las convenciones de programación facilitan esta tarea.
Cliente en sitio	El programa del curso requiere 4 horas teóricas por semana y 2 de laboratorio con un total de 64 horas teóricas y 32 horas prácticas al semestre. Sin embargo, como parte de esta experiencia, de estas 96 horas, un 70% la trabajan en el laboratorio. Esto permite simular un ambiente real de trabajo en donde se facilita una fuerte interacción entre el profesor asumiendo el rol de cliente y los estudiantes en su rol de desarrolladores.
Desarrollo guiado por pruebas	Durante la fase de análisis para cada iteración diseñan casos de prueba para cada uno de los requerimientos. Posteriormente diseñan y ejecutan pruebas unitarias de caja blanca. Una vez que tienen el componente listo ejecutan pruebas funcionales verificando el cumplimiento de los casos de pruebas. En caso de que el código requiera mejoras aplican pruebas de regresión, utilizando herramientas de software que permiten automatizar el proceso. Adicionalmente, se llevan a cabo revisiones técnicas entre los diferentes equipos de trabajo y se registran los defectos encontrados clasificados de acuerdo al tipo.
Programación en parejas	A los equipos se les permite trabajar en parejas, pero esta práctica no siempre se logra porque las horas de laboratorio no son suficientes, lo que los obliga a trabajar horas extras en forma individual en otros horarios.
Entregables pequeños	Cuando se planifican las *sprints* se intenta que cada una tenga una duración entre tres y cuatro semanas, si se excede se analiza y se traslada alguna funcionalidad para la siguiente.
Ritmo sustentable	Esta práctica es difícil de aplicar debido a que generalmente los estudiantes trabajan horas extras. Esta característica se puede lograr disminuyendo el tamaño de la aplicación. Si la aplicación es muy grande los estudiantes se ven obligados a invertir horas de trabajo adicionales, que no les permite mantener un ritmo sustentable y tanto trabajo extra no aporta ningún beneficio.

Copyright © 2012 Asociación Colombiana de Facultades de Ingeniería

Resultados

Durante el año 2010, a los estudiantes se les asignó como proyecto del curso, el desarrollo de un sistema web en una arquitectura 4 capas que permitiera administrar los requerimientos de un proyecto de *software*. Dentro de las funcionalidades solicitadas estaban las siguientes:

a. Administrar la información básica de los **recursos humanos** que tienen acceso al sistema (administrador, cliente, desarrollador).
b. Administrar la **seguridad** de la aplicación restringiendo el acceso a la información de acuerdo al rol del usuario.
c. Administrar la información básica de un **proyecto,** permitiendo además, crear para cada proyecto el equipo de trabajo (desarrolladores y usuario).
d. Administrar los **requerimientos funcionales** para un determinado proyecto.
e. Administrar los **requerimientos no funcionales** para un determinado proyecto.
f. Para cada requerimiento funcional y no funcional administrar las **dependencias** y los **conflictos** asociados con otros requerimientos del mismo proyecto.
g. Administrar los **cambios** realizados al requerimiento (quién, cuándo, qué y por qué) y a partir de una línea base controlar las **versiones** para facilitar el manejo de su historial sin borrar las versiones anteriores.
h. Administrar el **material de apoyo** (casos de uso, casos de pruebas, entre otros) asociados a los requerimientos de un determinado proyecto, de manera que permita el acceso a los archivos electrónicos correspondientes.
i. Administrar el proceso de verificación y validación de la **calidad** de los entregables del proyecto.
j. Algunas **consultas** como por ejemplo: árbol jerárquico que muestre todos los proyectos con sus correspondientes requerimientos ordenados por prioridad y luego por estado.

Durante la planificación inicial las funcionalidades se distribuyeron de la siguiente manera:

Primer semestre:
- *Sprint* 1 Módulo1**:** recursos humanos integrado con el módulo de seguridad (funcionalidades a y b).
- *Sprint* 1 Módulo 2: proyecto incluyendo la funcionalidad de conformación de los equipos de desarrollo (funcionalidad c).

Segundo semestre:
- *Sprint* 2 Módulo 1: requerimientos funcionales y no funcionales incluyendo la administración de dependencias y conflictos entre requerimientos (funcionalidades d, e y f).
- *Sprint* 3 Módulo 1: se mejoró la administración de requerimientos y se agregó la funcionalidad de control de versiones y gestión de su historial (funcionalidad g). Adicionalmente, se agregó una consulta representada por una estructura jerárquica que permitiera consultar todos los proyectos con su correspondientes requerimientos funcionales y no funcionales (funcionalidad j).
- *Sprint* 4 Módulo 1: se mejoró la documentación de toda la aplicación y se ajustaron las funcionalidades del sprint 1 implementado durante el primer semestre.

Algunas funcionalidades como la h y la i no se implementaron y la j quedó incompleta.

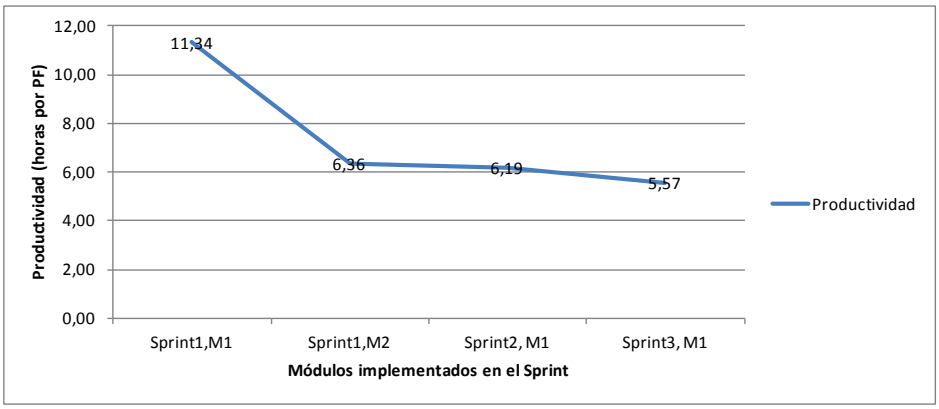
En la tabla 2 se muestran los resultados de uno de los equipos de trabajo del caso de estudio. En la primera columna se identifica el módulo y el *sprint*, en la segunda columna la duración real en horas que invirtieron para implementar el módulo, en la tercera el tamaño del módulo en Puntos de Función (PF) y en la cuarta, la productividad obtenida al desarrollarlo. El cálculo de los PF lo realizan con base en Garmus (2001). Solamente en el *sprint* 1 implementaron dos módulos, en el resto se implementó uno. El *sprint* cuatro no fue icluido en PF porque solo se realizó trabajo administrativo.

Tabla 2. Productividad a través del desarrollo de los módulos.

Descripción	Horas	PF	Productividad (horas por PF)
Sprint 1 Módulo 1 (*Sprint*1,M1)	249,5	22	11,34
Sprint 1 Módulo 2 (*Sprint*1,M2)	159	25	6,36
Sprint 2 Módulo 1 (*Sprint*2,M1)	167	27	6,19
Sprint 3 Módulo 1 (*Sprint*3,M1)	434,75	78	5,57
Sprint 4 Módulo 1 (*Sprint*4,M1)	169,5		
Total	1179,7	152	7,76

Copyright © 2012 Asociación Colombiana de Facultades de Ingeniería

Figura 8. Comportamiento de la productividad a través de los *sprints*



En la figura 8 se grafica la productividad mostrando una mejora importante de un *sprint* a otro. Se puede observar que conforme los estudiantes fueron avanzando en el desarrollo de la aplicación invirtieron menos horas por PF.

Discusión

Los objetivos planteados en esta investigación respecto a la necesidad de mejorar la curva de aprendizaje, fortalecer el proceso de pruebas y estimular el uso de metodologías de alcance mundial que les permita a los estudiantes enfrentar el mercado laboral se lograron de manera satisfactoria.

Repetir la misma metodología y herramientas en cada *sprint* mejoró la productividad de desarrollo y esto revela una mejoría de la curva de aprendizaje. En la figura 8 se puede observar que conforme fueron avanzando en el desarrollo de los *sprint*s, las horas requeridas de esfuerzo para desarrollar cada PF fue disminuyendo, por lo tanto la productividad mejoró.

Introducir prácticas de XP en el curso fortaleció el proceso de pruebas, lo que asegura un producto de mayor calidad y evita el re-trabajo de los productos de *software* implementados, lo cual también mejora la curva de aprendizaje. En la metodología anterior, las pruebas se realizaban durante el segundo semestre, después de haber realizado el diseño detallado y la codificación de los requerimientos de toda la aplicación. De la misma forma, no se aplicaban prácticas fundamentales de un proceso de pruebas como: la planificación, el diseño y la ejecución automatizada. Actualmente, desde el inicio de cada *sprint* van introduciendo las pruebas a través de todo el proceso de desarrollo.

En años anteriores las metodologías livianas se mencionaban en forma teórica, en este momento los estudiantes pueden experimentar y asimilar esas nuevas tendencias que cada vez toman más auge en la industria.

Conclusiones

Con la metodología propuesta se pueden lograr los siguientes beneficios:

Desde el punto de vista didáctico el enfoque de desarrollo a través de *sprint*s favorece la enseñanza y el aprendizaje de la ingeniería de *software* al tener que aplicar en forma repetitiva las mismas metodologías, técnicas y herramientas de *software*.

La corta duración de los *sprints* (4 a 6 semanas) obliga a que el tamaño de la funcionalidad implementada sea relativamente pequeña, lo que provee las siguientes ventajas: facilita la comprensión de los requerimientos, torna más sencilla la interacción con el usuario y se vuelve más eficiente el manejo de los cambios en los requerimientos.

La combinación de *Scrum*, RUP y XP logra sacar provecho en forma equilibrada de los beneficios de las metodologías livianas y de las pesadas. El *Scrum* apoya en las tareas de planificación y de administración de los *sprint*s. El uso de RUP durante el desarrollo asegura que en cada *sprint* se siguen prácticas de calidad en cada una de las fases de desarrollo. Finalmente la aplicación de XP mejora la construcción de los componentes, garantizando el cumplimiento de los requerimientos y la satisfacción del usuario.

Copyright © 2012 Asociación Colombiana de Facultades de Ingeniería

Con base en el principio de metodologías de ágiles que sobrepone un software funcional sobre la documentación exhaustiva, se logró aligerar el proceso de desarrollo al flexibilizar las exigencias de los entregables que los estudiantes desarrollan. Se realizó una revisión de todos los estándares de calidad utilizados en el curso y se elaboraron plantillas de trabajo con lo esencial de cada tarea, eliminando redundancias entre una plantilla y otra.

La nueva metodología logró reducir la cantidad de tiempo gastado en re-trabajo. Esto debido a que muchos defectos se producían en el primer curso pero se detectaban y corregían en el segundo. Por otro lado, la documentación exhaustiva y corrección de documentos aumentaba el tiempo de trabajo y de re-trabajo. Estos dos problemas producían desgaste emocional y físico en los estudiantes, lo cual evidentemente perjudicaba la productividad del equipo.

Conocer y experimentar la aplicación de las metodologías ágiles, que cada vez están tomando más auge en la industria, facilita a los estudiantes el proceso de inserción al mundo laboral.

Es difícil comparar los datos de productividad obtenidos en esta investigación con los de años anteriores, porque antes se recogían datos en cada una de las fases de desarrollo realizadas durante los dos semestres y se calculaba la productividad de toda la aplicación al finalizar el curso. Actualmente, se calcula después del cierre de cada *sprint*. Sin embargo. los datos mostrados en la sección 3 demuestran mejora de la productividad entre un Sprint y otro.

La planificación inicial y la organización del proyecto a través de *sprints* facilita la delimitación del alcance y permite que los estudiantes entreguen productos funcionales que pueden ser evaluados en su completitud.

Adicionalmente se puede concluir que no es conveniente que aprendan y apliquen metodologías livianas como "único" método de enseñanza en el curso de Ingeniería de Software, porque generalmente estos modelos de desarrollo promueven un proceso informal, con escasa documentación y con un mínimo de productos de trabajo de ingeniería de software que atentan con la calidad del software y pueden contradecir algunas prácticas recomendadas por IEEE (2003), *Software Engineering Institute* (2006) y *Project Management Institute* (2005), entre otras.

Referencias

Garmus, D., & Herron, D. (2001). *Function point analysis. Measurement practices for successful software projects*. Addisson Wesley.

IEEE (2003). *IEEE Standards collection: software engineering*. IEEE Inc.

Larman, C. (1999). *UML Y PATRONES. Introducción al análisis y diseño orientado a objetos*. (2da. Ed.). México: Prentice Hall Hispanoamerica, S. A.

Pressman, R. (2010). *Ingeniería de software: un enfoque práctico*. (7ma. Ed.). México, D. F.: McGraw-Hill Interamericana.

Project Management Institute. (2005). *Guide to the project management body of knowledge* (PMBOK Guide). (3era. Ed.).

Software Engineering Institute. (2006). CMMI for Development (CMMI-DEV), Version 1.2 Technical report CMU/ SEI-2006-TR-008. Pittsburg, PA: Software Engineering Institute, Carnegie Melon University.

Sommerville, I. (2011). *Ingeniería de software*. (9na Ed.). México: Addison Wesley.

Villena, A. (2008). *Un modelo empírico de enseñanza de las metodologías ágiles: el caso del curso CC62V- Taller de metodologías ágiles de desarrollo de software*. Proyecto de graduación para optar por el grado de Magister en Ciencias Mención Computación, Departamento de Ciencias de la Computación, Universidad de Chile. Recuperado el 25 de Julio de 2011 de http://es.scribd.com/doc/39816155/Un-modelo-empirico-de-ensenanza-de-las-metodologias-agiles

Copyright © 2012 Asociación Colombiana de Facultades de Ingeniería

Sobre la autora

Gabriela Salazar Bermúdez
Licenciada en Computación e Informática, Magister Scientiae en Computación e Informática. Docente en la Universidad de Costa Rica, Escuela de Ciencias de la Computación e Informática, Universidad de Costa rica, San Pedro de Montes de Oca, San José, Costa Rica.
gabriela.salazar@ecci.ucr.ac.cr

Los puntos de vista expresados en este artículo no reflejan necesariamente la opinión de la Asociación Colombiana de Facultades de Ingeniería.

Copyright © 2012 Asociación Colombiana de Facultades de Ingeniería