Universidades
ISSN: 0041-8935
udual1@servidor.unam.mx
Unión de Universidades de América Latina y el Caribe
Organismo Internacional

Cervantes Ojeda, J.; Gómez Fuentes, María del Carmen
Taxonomía de los modelos y metodologías de desarrollo de software más utilizados
Universidades, núm. 52, enero-marzo, 2012, pp. 37-47
Unión de Universidades de América Latina y el Caribe
Distrito Federal, Organismo Internacional

Universidades
UDUAL, México, n. 52, enero-marzo 2012, pp. 37-47. ISSN 0041-8935

Taxonomía de los modelos y metodologías de desarrollo de software más utilizados

J. Cervantes Ojeda
María del Carmen Gómez Fuentes
Doctores en ciencias de la computación en la Universidad Autónoma Metropolitana, Unidad Cuajimalpa, México.
Correo-e: jcervantes@correo.cua.uam.mx

Resumen

A través de una recopilación y análisis de los principales modelos de desarrollo de software existentes, proponemos una taxonomía que los integra con el fin de facilitar la elección de un modelo apropiado para cada proyecto en particular. Una buena elección de modelo (correctamente aplicado) ahorra tiempo y mejora la calidad de los sistemas que se producen. Sin embargo, la amplia variedad de modelos y metodologías en el mundo del desarrollo de software, dificulta esta elección. La taxonomía propuesta presenta un panorama general de los modelos y metodologías más aceptados y los agrupa en categorías. Discutimos las características más representativas de cada una de estas categorías.

Palabras clave

Ingeniería de software, taxonomía de modelos de desarrollo de software, ciclo de vida del software.

Abstract

Based on a survey and analysis of the main existing Software Development models, this paper describes a proposal of a new taxonomy that integrates these models in order to facilitate the task of choosing a suitable model for each particular project. A good choice of a model (if used in the right way) saves time and improves the quality of the produced systems. Though, the wide variety of models and methodologies in the Software development world makes this choice difficult. The proposed taxonomy presents a general view of the most accepted models and methodologies and groups them in categories. We discuss the most representative properties of each category.

Key words

Software engineering, taxonomy of software development models, software life cycle.

1 Introducción

Citando a Platón, "el comienzo es la parte más difícil del trabajo". Podríamos agregar que también es la parte más importante, al menos en proyectos de desarrollo de software. Podemos afirmar que el éxito de los proyectos de software depende en gran medida de que tengan un buen inicio. Uno de los factores clave para un buen inicio es que la elección del modelo de desarrollo se adecúe a las características y circunstancias del proyecto. La elección y aplicación correcta de un modelo de desarrollo de software permite ahorrar tiempo y mejorar la calidad de los sistemas que se producen. Sin embargo, la amplia variedad de modelos y metodologías en el mundo del desarrollo de software, hace que no sea sencillo elegir el modelo más apropiado para un proyecto específico, sobre todo cuando la definición de estos modelos y metodologías se encuentra dispersa en varios libros, artículos y sitios de internet. Proponemos una taxonomía que condensa toda esta información y que brinda un panorama general de los modelos existentes con sus ventajas y desventajas. Hasta donde tenemos noticia, no se ha hecho un trabajo similar a éste desde 1994 (Blum, 1994). El surgimiento de importantes modelos y metodologías desde entonces a la fecha amerita una nueva clasificación que incluya los más importantes en la actualidad.

Se han hecho estudios acerca de los fracasos en los proyectos de software, por ejemplo (Mangione, 2003) y (McManus & Wood, 2004). Actualmente siguen existiendo proyectos de software que fracasan en los que se involucran miles o millones de dólares para su realización. En la mayoría de los casos el fracaso se debe a que el tiempo utilizado para el desarrollo del proyecto hace que éste se convierta en ***no viable***. El fracaso de proyectos de software algunas veces ha implicado la pérdida de muchísimo dinero o incluso la pérdida de vidas humanas por entregar productos defectuosos (Pfleeger & Atlee, 2006: 37-38; Weitzenfeld, 2004:3-13). El tiempo de desarrollo de un producto de software se extiende mucho cuando no se tiene bien claro lo que se va a hacer. La fase de requerimientos es una parte esencial que no puede dejar de ser atendida con el debido cuidado y esfuerzo. Además de la mala especificación de requerimientos, otra de las importantes causas del fracaso de los proyectos de software es la mala elección de un modelo de desarrollo para los mismos. Pensamos que el presente trabajo contribuye a que esta elección se haga con una mejor perspectiva de las características de los modelos existentes que, a nuestro juicio, son los más importantes.

En la sección 2 de este trabajo, definimos los conceptos de ***proceso*** y ***modelo*** de desarrollo de software así como su relación con el concepto de ***ciclo de vida***. Se describen los antecedentes de modelos abstractos. En la sección 3 proponemos una taxonomía que clasifica los modelos de desarrollo de software. Además, en la sección 4 mencionamos las ventajas y desventajas de las categorías de modelos en esta taxonomía.

2 Antecedentes

2.1 Relación entre *proceso, modelo* y *ciclo de vida* del software

Un ***proceso de desarrollo de software*** es el conjunto estructurado de las actividades requeridas para realizar un sistema de software. Estas actividades son: especificación de requerimientos, diseño, codificación, validación (pruebas) y mantenimiento. Al proceso de desarrollo de software también se le conoce como ***ciclo de vida*** del software porque describe la vida de un producto de software; primero nace con la especificación de los requerimientos, luego se lleva a cabo su implantación, que consiste en su diseño, codificación y pruebas, posteriormente el producto se entrega y sigue viviendo durante su utilización y mantenimiento. En este ciclo se establece una comunicación interactiva entre cliente y

desarrollador en la que el primero solicita servicios y el segundo propone soluciones. El ciclo de vida del sistema de software termina cuando éste se deja de utilizar. Por otra parte, un ***modelo de desarrollo de software*** es una representación abstracta de este proceso (Sommerville, 2005:60). Un modelo de desarrollo de SW determina el orden en el que se llevan a cabo las actividades del proceso de desarrollo de SW, es decir, es el procedimiento que se sigue durante el proceso. Al modelo de desarrollo también se le llama ***paradigma del proceso***.

2.2 Antecedentes de modelos abstractos

Hay una gran variedad de paradigmas o modelos de desarrollo de software. Los libros más conocidos de ingeniería de software (Braude, 2003:21-33; McConnell, 1997: 148-167; Pfleger, 2002: 54-66; Pressman, 2002: 20-30; Sommerville, 2005: 60-69; Weitzenfeld, 2004: 50-64) explican sólo los que consideran más importantes y el problema en ellos es que las opiniones acerca de cuál es la lista de modelos que debe considerarse son diversas. Sommerville (Sommerville, 2005: 60-69), clasifica todos los procesos de desarrollo de software en tres modelos o paradigmas generales que no son descripciones definitivas de los procesos del software, sino, más bien, son abstracciones de los modelos que se pueden utilizar para desarrollar software, y son los siguientes.

a) **Modelos en cascada**. Las actividades fundamentales del proceso de desarrollo de software se llevan a cabo como fases separadas y consecutivas. Estas actividades son: especificación (análisis y definición de requerimientos), implantación (diseño, codificación, validación) y mantenimiento. Los modelos en cascada constan básicamente de 5 fases que son:

Análisis y definición de requerimientos. Se trabaja con los clientes y los usuarios finales del sistema para determinar el dominio de aplicación y los servicios que debe proporcionar el sistema así como sus restricciones. Con esta información se produce el documento de "Especificación de Requerimientos del Sistema".

Diseño del sistema y del software. Durante el proceso de diseño del sistema se distinguen cuales son los requerimientos de software y cuales los de hardware. Después se establece una arquitectura completa del sistema. Durante el diseño del software se identifican los subsistemas que componen el sistema y se describe cómo funciona cada uno y las relaciones entre éstos.

Implementación y validación de unidades. Consiste en codificar y probar los diferentes subsistemas por separado. La prueba de unidades implica verificar que cada una cumpla su especificación (proveniente del diseño).

Integración y validación del sistema. Una vez que se probó que funciona individualmente cada una de las unidades, éstas se integran para formar un sistema completo que debe cumplir con todos los requerimientos del software. Cuando las pruebas del sistema completo son exitosas, éste se entrega al cliente.

Funcionamiento y mantenimiento. El sistema se instala y se pone en funcionamiento práctico. El mantenimiento implica corregir errores no descubiertos en las etapas anteriores del ciclo de vida y mejorar la implantación de las unidades del sistema para darle mayor robustez (y no nuevas funcionalidades).

b) **Modelos de desarrollo evolutivo**. Los modelos evolutivos son iterativos. Se caracterizan por la forma en que permiten a los ingenieros de software desarrollar versiones cada vez más completas del sistema. Los modelos evolutivos iteran sobre las actividades de especificación, desarrollo y validación. Un sistema inicial se desarrolla a partir de los requerimientos prioritarios o los que están mejor definidos. Esta primera versión se refina en una nueva iteración con las peticiones del cliente para producir un sistema que satisfaga sus necesidades. Sommerville define dos tipos de desarrollo evolutivo:

b.1) ***Desarrollo exploratorio***. Se le presenta al cliente el desarrollo de la parte de los requerimientos que se entendió bien para recibir sus comentarios y así refinar el sistema hasta que se logra desarrollar el sistema adecuado.

b.2) ***Prototipos desechables.*** Para descubrir o terminar de comprender los requerimientos del cliente se construye un prototipo con funcionalidad simulada y, si éste no es lo que el cliente espera, se construye otro prototipo (posiblemente desde cero) con una definición mejorada de los requerimientos para el sistema. El diseño del prototipo va evolucionando según se vayan entendiendo los requerimientos, aunque la funcionalidad siga siendo simulada. Cuando se aclaran los requerimientos se completa la funcionalidad según el último prototipo.

c) **Modelos de componentes reutilizables**. Se basa en la existencia de un número significativo de componentes reutilizables. El reuso de los componentes tiene como finalidad usar de nuevo ideas, arquitecturas, diseños o código de una aplicación para construir otras. El proceso de desarrollo del sistema se enfoca en integrar estos componentes en el sistema, en lugar de desarrollarlos desde cero.

Según Sommerville (2005:64), en la mayoría de los proyectos existe algo de reutilización de software. Por lo general, esto sucede informalmente cuando las personas que trabajan en el proyecto conocen diseños de código similares al requerido. Los buscan, los modifican según lo creen necesario y los incorporan en el sistema. Las etapas de especificación de requerimientos y de validación son comparables con los otros procesos, sin embargo, las etapas intermedias en el proceso orientado a la reutilización son diferentes. Estas etapas son:

Análisis de componentes. Consiste en encontrar componentes que sirvan para desarrollar la Especificación de Requerimientos. En general, los componentes que se utilizan sólo proporcionan parte de la funcionalidad requerida por lo que se necesita modificarlos.

Modificación de requerimientos. Con la información que se tiene de los componentes ya identificados, se analizan los requerimientos. Si es posible, se modifican los requerimientos para que concuerden con los componentes disponibles. Si las modificaciones no son posibles entonces se lleva a cabo nuevamente el análisis de componentes para buscar soluciones alternativas.

Diseño del sistema con reutilización. Se diseña o se reutiliza un marco de trabajo para el nuevo sistema teniendo en cuenta los componentes que se reutilizan y los componentes que serán completamente nuevos.

Desarrollo e integración. El software que no se tiene disponible y que no se puede adquirir externamente se desarrolla integrando los componentes reutilizables disponibles. En este modelo, la integración de los sistemas es parte del desarrollo más que una actividad separada.

Los tres paradigmas o modelos de procesos genéricos: ***cascada, evolutivo*** y ***componentes reutilizables***, se utilizan ampliamente en la práctica actual de la ingeniería del software. No se excluyen mutuamente y a menudo se utilizan juntos, especialmente para el desarrollo de sistemas grandes (Sommerville, 2005: 61). El problema es que en la literatura no se hace una clasificación explícita que ubique cada modelo o metodología dentro de alguna de estas clases abstractas. En la siguiente sección hacemos esta clasificación para los modelos más representativos.

3 Taxonomía propuesta de modelos de desarrollo

A continuación (ver la ***figura 1***) proponemos la clasificación de los modelos y metodologías concretos más citados en la literatura en cinco clases abstractas: Cascada, Evolutivos, Minimización de Desarrollos, Híbridos y Ágiles. Los dividimos en ***Modelos Tradicionales*** (también llamados ***pesados***), que son los que promueven la disciplina por medio de la planificación y la comunicación escrita, y los ***Metodologías Ágiles***, que dan prioridad a la interacción entre los individuos y a la comunicación con el cliente.

Figura 1: Clasificación de modelos concretos en clases abstractas.

Modelo Abstracto		Modelos Concretos
Tradicionales o Pesados	En Cascada	Pura Con fases solapadas Con subproyectos Con reducción de riesgos
	Evolutivos	Espiral Entrega por etapas o incremental Entrega evolutiva o iterativo Diseño por planificación Cascada en V
	Minimización de Desarrollos	Componentes Reutilizables Diseño por herramientas
	Híbridos	Proceso Unificado Racional Otros
Metodologías Ágiles		Programación extrema SCRUM Desarrollo dirigido por pruebas Desarrollo dirigido por Características Agile, Lean, Crystal, …, etc.

3.1 En Cascada

Al modelo abstracto ***En Cascada*** pertenecen los siguientes: cascada puro, cascada con fases solapadas, cascada con subproyectos y cascada con reducción de riesgos (Braude, 2003:24-26; McConnell, 1997: 148-159; Pfleger, 2002: 55-57; Pressman, 2002: 23-27; Sommerville, 2005: 62-63; Weitzenfeld, 2004: 50-51). Todos estos modelos se caracterizan por una secuenciación serial de las siguientes actividades: análisis y definición de requerimientos, diseño, codificación, validación y mantenimiento. Además, en todos ellos se produce una documentación completa del sistema. El modelo en cascada con fases solapadas permite hacer actividades de la siguiente fase en paralelo a las últimas actividades de la fase anterior sin romper la secuenciación de las fases. El modelo de cascada con subproyectos, aunque divide el proyecto en subproyectos más pequeños (a partir de que se ha completado el diseño global) que se pueden desarrollar en paralelo e integrarlos todos al final, conserva el carácter secuencial de las actividades. En el modelo de cascada con reducción de riesgos se controla el riesgo en la fase de requerimientos con una espiral que los identifica y mitiga y prevé la posibilidad de retroceder en la secuencia de actividades pero las mantiene en el mismo orden que una cascada pura.

3.2 Evolutivos

A la clase abstracta de modelos ***Evolutivos*** pertenecen: espiral, entrega por etapas o incremental, entrega evolutiva o iterativo, diseño por planificación y cascada en V (Braude, 2003:26-28; McConnell, 1997:153-165; Pfleger, 2002: 57-67; Pressman, 2002: 23-27; Sommerville, 2005: 63-64 y 66-69; Weitzenfeld, 2004: 51-54). Los modelos evolutivos tienen la particularidad de visitar las diferentes

etapas de desarrollo varias veces según sea necesario pero en un orden específico, es decir, no se prevén retrocesos en la secuencia. Al modelo de entrega por etapas Pfleeger y Atlee le llaman "implementación incremental" porque se entrega el software en partes pequeñas, pero utilizables, llamadas incrementos. Al modelo de entrega evolutiva Pfleeger y Atlee le llaman "iterativo", en el que se entrega el esqueleto de un sistema completo desde el principio, y luego se va "rellenando" la funcionalidad de cada subsistema con cada versión nueva.

3.3 Minimización de desarrollos

En la clase de modelos de ***Minimización de desarrollos*** podemos encontrar: componentes reutilizables y desarrollo por herramientas (Braude, 2003:21-22; McConnell, 1997: 165-167; Pressman, 2002: 22, 28; Sommerville, 2005: 60-69***;*** Weitzenfeld, 2004: 50-64). Con estos modelos se saca ventaja de elementos desarrollados previamente y se caracterizan por aumentar la importancia (respecto a otros modelos) de esta reutilización y disminuir la importancia del cumplimiento estricto de los requerimientos con la idea de acelerar el proceso de desarrollo. Se le ofrece al cliente primero lo que es fácilmente entregable y, solamente en caso de ser necesario, se propone un desarrollo nuevo. En el diseño por herramientas, por ejemplo, se trata de hacer uso de herramientas que están ya disponibles y, a partir de un conjunto de funcionalidades soportadas por éstas, ofrecer al cliente la mayor parte posible de las funcionalidades que requiera. En el desarrollo por componentes reutilizables se parte de un sistema previamente desarrollado y, a partir de algunas de sus definiciones de requerimientos, de partes de diseños y de grupos de guiones de prueba o de datos, se desarrolla el nuevo sistema con la idea de reducir el tiempo desarrollo y costos.

3.4 Híbridos

IBM Rational (IBM, Rational Unified Process) propone el desarrollo de software basado en las mejores prácticas recopiladas de un conjunto grande de proyectos exitosos. Es una metodología que llama al proceso de desarrollo de software: ***Rational Unified Process*** (RUP) que en español es Proceso Unificado Racional. El RUP es un modelo de proceso ***Híbrido*** ya que reúne elementos de modelos de procesos genéricos (Sommerville, 2005:76). Además propone buenas prácticas para la especificación y el diseño. El proceso unificado se describe desde tres perspectivas:

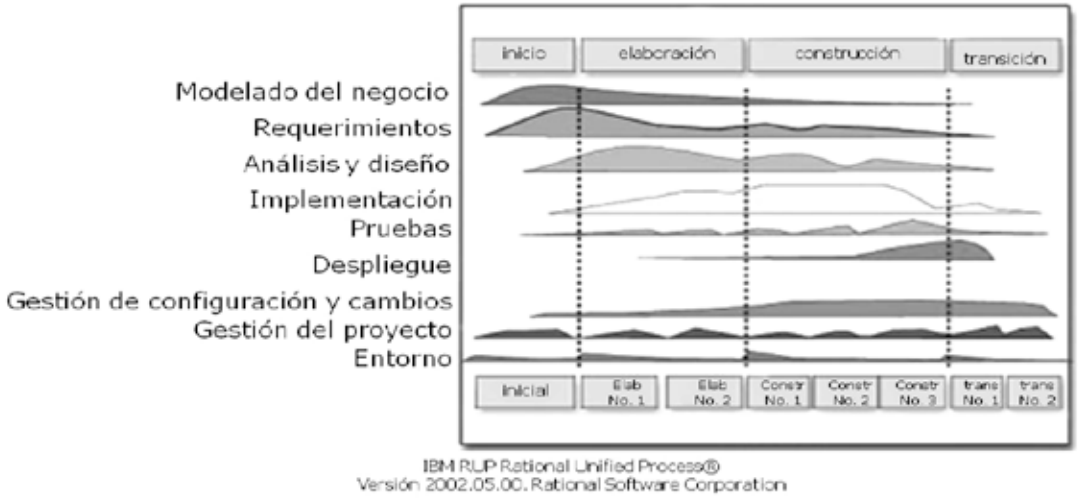
1.- ***Una perspectiva dinámica***.- Muestra las fases (también llamadas etapas) del modelo sobre el tiempo, éstas son: inicio, elaboración, construcción y transición.
2.- ***Una perspectiva estática***.- Muestra las actividades que tienen lugar durante el proceso de desarrollo, se denominan ***flujos de trabajo***, éstos son: modelado del negocio, requerimientos, análisis y diseño, implementación, pruebas, despliegue, gestión de configuración y cambios, gestión del proyecto y entorno.
3.- ***Una perspectiva práctica***.- Sugiere buenas prácticas a utilizar durante el proceso.

En cuanto a la ***perspectiva práctica***, se recomiendan seis buenas prácticas aconsejables en el desarrollo de sistemas: ***Desarrollar el software de forma iterativa*** (entregando primero los requerimientos más importantes), ***Gestionar los requerimientos*** (analizando el impacto de los cambios en el sistema antes de aceptarlos y documentando los cambios aceptados), ***Utilizar arquitecturas basadas en componentes*** (en la mayor medida posible), ***modelar el software visiblemente*** (con modelos gráficos como UML), ***verificar la calidad del software***, ***controlar los cambios del software*** y ***gestionar los cambios del software*** usando herramientas de gestión de configuraciones.

El Proceso Unificado no es un proceso apropiado para todos los tipos de desarrollo, sin embargo representa una nueva generación de procesos genéricos (Sommerville, 2005: 78). Las innovaciones más importantes son la separación de fases y los flujos de trabajo, y el reconocimiento de que la utilización del software en un entorno de usuario es parte del proceso. Las fases (o etapas) son dinámicas y tienen objetivos. Los flujos

de trabajo son estáticos y son actividades técnicas que no están asociadas con fases únicas sino que pueden utilizarse durante el desarrollo para alcanzar los objetivos de cada fase.

Figura 2 Combinación de las fases (o etapas) con los flujos de trabajo en el Proceso Unificado



En la ***figura 2*** se muestra una visión global de cómo se combinan las fases dinámicas del proceso unificado, con los flujos de trabajo estáticos. Además se ilustra que en cada fase puede haber varios incrementos.

Los requerimientos se trabajan desde la fase de inicio, y muy especialmente durante la fase de elaboración, pues el objetivo es tener claro por lo menos un 80% del sistema que se requiere construir (Wiegers, 2006: 31).

Las características del Proceso Unificado según (Ambler, 2005) son:

- **Visto a lo largo de todo el proyecto, es *serial*** en el tiempo: comienza con la etapa de inicio, luego la etapa de elaboración, después la etapa de construcción y al final la etapa de transición.
- **Visto en cada etapa es *iterativo***: la etapa puede estar compuesta de varias entregas. Hay entregas parciales del producto, las funcionalidades se van incluyendo de manera incremental.
- **Se apoya en buenas prácticas probadas en innumerables proyectos exitosos para una gran variedad de dominios.**
- **Al final de cada una de las etapas del Proceso Unificado se debe entregar un producto importante (hito): Al final del *inicio***: se entregan los objetivos y definición del alcance del proyecto. Al final de la ***elaboración***: se entrega la arquitectura del sistema. En cada iteración de la ***construcción***: se entrega un producto con la función anterior más el incremento correspondiente a la nueva iteración, de tal forma que al final de la ***construcción*** se obtiene la versión inicial del sistema con capacidad operacional, es decir, con toda la funcionalidad requerida. Al final de la ***transición***: se entrega el producto completamente funcional.

Aunque el RUP fue concebido inicialmente para sistemas orientados a objetos, en algunos casos vale la pena aplicarlo a situaciones no orientadas a objetos y obtener de todas formas algunas de las ventajas del RUP como son:

- Hacer frente a los riesgos de cambios en los requerimientos,
- Disminuir el riesgo financiero al hacer entregas parciales de software funcional que puede probarse y ser evaluado por el cliente.
- Se puede adaptar para administrar el proceso con los niveles de flexibilidad y rigor necesarios para cada situación en particular.

3.5 Las metodologías ágiles

Hasta ahora hemos hablado de las llamadas ***metodologías tradicionales,*** las cuales se basan en la disciplina y el orden, sin embargo, "por más que en los últimos años han evolucionado diversas metodologías para asegurar un mejor control del proceso, los clientes quedan frecuentemente insatisfechos con el resultado" (Aguilar, 2002:1). Si bien, la mala administración es la principal causa detectada en los fracasos en los proyectos de software, algunos también atribuyen el fracaso a la metodología empleada para su desarrollo. Las ***metodologías ágiles*** surgen como otra alternativa de desarrollo para contrarrestar estos fracasos. Las prácticas que recomiendan son algunas veces opuestas a lo recomendado por las ***metodologías tradicionales***. La metodologías ágiles se basan en un "desarrollo iterativo e incremental en muy breves ciclos y un diseño inicial simple" (Araújo, 2007:2). Según estudios recientes, las metodologías ágiles tienen una gran aceptación en la industria del software (West & Grant, 2010), sin embargo, según sus fundadores, éstas sólo son aplicables cuando se dan las siguientes condiciones (Fowler, 2000):

- Proyectos pequeños y equipos con menos de 100 personas.
- Requerimientos cambiantes.
- Equipo de desarrollo competente.
- Cliente dispuesto a participar con el equipo.

En febrero de 2001 se emitió el "Manifiesto para el desarrollo ágil del software" (Manifesto, 2001) en el cual se estipulan las características que debe tener un desarrollo ágil. Las metodologías ágiles valoran más a los individuos y las interacciones entre éstos que a los procesos y a las herramientas. Se fomenta más la comunicación cara a cara que la documentación, de tal manera que el tiempo se emplea en producir software que funciona en lugar de usarlo para producir documentación. Se le da más énfasis a la colaboración con el cliente en los aspectos claves del desarrollo que a la negociación del contrato y se concentran en la respuesta a los cambios en lugar de elaborar un plan y seguirlo ya que, según esta filosofía, es imposible anticipar todos los requerimientos desde el inicio del desarrollo (Pfleeger & Atlee, 2006:59).

West & Grant (West & Grant, 2010) realizaron un estudio sobre los métodos y metodologías más utilizados en la industria del software durante el 2008 y el 2009. Según los resultados, en el 2008 algunas metodologías ágiles ocuparon el primer lugar, éstas fueron: SCRUM (Pazderski, 2010) programación extrema, llamada en inglés "XP: eXtreme Programming" (Aguilar, 2002), Desarrollo Dirigido por Pruebas, llamado en inglés "TDD: Test-Driven development" (Araújo, 2007), Delgado o Menudo, conocido como "Lean" (Poppendieck & Poppendieck, 2003), Desarrollo Dirigido por Características, llamado en inglés "FDD: Feature Driven Development" (Anderson, 2004) y Modelado Ágil (Ambler, 2008). En segundo lugar se utilizaron modelos iterativos y en tercer lugar el modelo en cascada. Otras metodologías ágiles y el Proceso Unificado tuvieron una participación muy pequeña. En el 2009, los resultados fueron muy similares, el 35% de 1,298 encuestados utilizaron metodologías ágiles (destacaron las mismas que en el 2008), el 21% utilizó métodos iterativos y el Proceso Unificado y el 13.4% utilizó el cascada. Es interesante mencionar que el 30.6% de los encuestados no utilizaron ninguna metodología formal.

4 Elección del modelo de desarrollo de software

"La elección de un modelo de ciclo de vida erróneo puede dar lugar a la omisión de tareas y a una secuenciación inapropiada de las mismas, lo cual va en contra de la planificación y eficiencia del proyecto. La elección de un modelo apropiado tiene el efecto contrario, asegurando que todo el esfuerzo se utiliza eficientemente" (McConnell, 1997: 501).

Puesto que la necesidad de un proyecto se genera a partir del surgimiento de requerimientos, la naturaleza de éstos es uno de los aspectos importantes para la elección del modelo de desarrollo. Los requerimientos están

estrechamente relacionados con la elección del modelo para desarrollar el proyecto.

En los ***modelos tipo cascada***, los requerimientos tienen que estar bien definidos desde el inicio del proyecto y la probabilidad de que cambien debe ser mínima. Cabe mencionar que esto aplica, tanto al desarrollo de sistemas nuevos, como al desarrollo de modificaciones sobre un sistema existente. Pfleeger & Atlee (Pfleeger & Atlee: 2006: 145) recomiendan además el uso de un modelo en cascada cuando los requerimientos están fuertemente acoplados o cuando son complejos, es decir, cuando no es sencillo separar los requerimientos para desarrollarlos uno por uno, ya que se corre el riesgo de que el desarrollo de unos no sea compatible con la de otros. Otro caso en el que el modelo en cascada es viable, es cuando muchas personas participan en el proyecto, ya sea porque el proyecto es grande (en cuyo caso ya se ha justificado el uso de un modelo tipo cascada) o porque se requiere de la colaboración de especialistas. En estos casos es mucho más sencillo el uso de modelos tipo cascada ya que resulta muy importante tener procedimientos de control estrictos y una comunicación formal y disciplinada entre los participantes. Una recomendación importante (McConnell, 1997) es que no se aumente el número de participantes cuando los tiempos de entrega son cortos ya que solamente se logra que la organización se complique debido a lo complejo de la comunicación entre las personas. Si el tiempo de entrega es muy importante entonces no se recomienda el uso de modelos tipo cascada. Solamente cuando la calidad del producto sea prioritaria sobre el tiempo de entrega es bueno adoptar uno de estos modelos (Pfleeger & Atlee, 2006: 145).

En suma, los modelos tipo cascada son recomendables para: requerimientos bien definidos y que no cambian; requerimientos fuertemente acoplados o complejos; proyectos donde interviene una gran cantidad de personas y, cuando es más importante entregar un sistema funcionando correctamente que cumplir con una fecha de entrega preestablecida. Un estudio realizado por Califa (2000) confirma que para sistemas grandes, los desarrolladores prefieren el modelo en cascada, mientras que para sistemas pequeños los procesos evolutivos son más aceptados otros.

En los ***modelos evolutivos***, los requerimientos se trabajan al inicio de cada iteración para aumentarlos, corregirlos o redefinirlos. En estos modelos se complica mantener actualizada y correcta la documentación. Además, en sistemas muy grandes, cada nueva adición puede implicar que el código se corrompa debido a una mala administración de los cambios. En este tipo de procesos, la especificación de requerimientos se desarrolla junto con el software, esto puede crear conflictos en las organizaciones en las que la especificación de requerimientos es parte del contrato (Sommerville, 2005:66). Los procesos evolutivos permiten mostrar al cliente una versión parcial preliminar que permita obtener retroalimentación y evite problemas con la integración de un código muy grande. Para que este modelo sea útil se debe poder comenzar con algunos requerimientos prioritarios y dejar para los ciclos posteriores los demás requerimientos, además hay que contar con la participación del usuario, quien debe dedicar tiempo a evaluar y retroalimentar las entregas parciales (Braude, 2007:27). Los modelos evolutivos como el espiral o el incremental tienen grandes ventajas: los clientes pueden comenzar a utilizar un sistema que tiene los requerimientos prioritarios para ponerlo a prueba y reportar sus fallas. De esta manera aumenta la probabilidad de entregar un software que opere satisfactoriamente. Además se facilita la recolección de métricas acerca del proceso en cada iteración. Sin embargo (Sommerville, 2005: 67) recomienda que el código no rebase las 20,000 líneas en cada incremento y a veces puede ser difícil adaptar los requerimientos del cliente al tamaño apropiado de un incremento. Braude (Braude, 2006: 27) señala un punto importante: "con el propósito de optimizar la productividad en equipo, con frecuencia es necesario comenzar una nueva iteración antes de que la anterior haya terminado", esto no sólo dificulta la coordinación de la documentación, sino que dificulta la coordinación de los cambios en los diferentes módulos del sistema cuando un requerimiento tiene impacto en varios de estos módulos.

En los ***modelos de minimización de desarrollos***, las funcionalidades de los componentes o módulos deben ser similares a las nuevas funcionalidades que se requieren, de tal forma que el esfuerzo en modificar los componentes base no sea tan alto que el proceso se entorpezca en lugar de agilizarse. Se considera una buena práctica en la ingeniería de software hacer diseños modulares ya que éstos tienen el potencial de la reutilización de algunas de sus partes (Braude, 2006: 21-22), sin embargo, esto no es fácil. Cuando se adopta este modelo, normalmente se negocia con el cliente para hacer modificaciones a sus requerimientos con la finalidad de que éstos se adapten a los componentes base. Es muy importante cuidar que estas modificaciones no produzcan un sistema que no cumpla con las necesidades reales de los usuarios. La calidad de un sistema basado en reutilización de componentes dependerá mucho de la robustez de éstos y el mantenimiento del sistema estará limitado por la facilidad de acceso que se pueda tener para modificarlos.

El ***Proceso Unificado Racional*** (RUP) es un modelo híbrido que pretende sacar las ventajas de los modelos cascada, evolutivos y las de los de componentes reutilizables. Como se mencionó en la sección 2, visto a lo largo de todo el proyecto, es decir, desde una perspectiva dinámica, el RUP es ***serial*** en el tiempo, tal y como el modelo en cascada. La parte evolutivo/iterativa en la que se descompone cada una de las etapas, reduce riesgos y es hasta cierto punto flexible en los cambios de requerimientos. La reutilización de componentes que se fomenta con este modelo permite reducir costos y tiempo de desarrollo. El uso del Lenguaje de Modelado Unificado: UML (Booch et al., 1999) asociado al RUP, facilita el análisis y el diseño de los componentes del sistema. Sus procedimientos de control de calidad y control de cambios contribuyen a la producción de un software satisfactorio. Sin embargo, el RUP es un modelo complicado, se requiere de una alta capacitación del administrador del proyecto para llevarlo a buen término. Además, los miembros del equipo de desarrollo también deben tener una alta capacitación en el uso de este complejo modelo, probablemente este sea el motivo por el cual, según estudios recientes (West & Grant, 2010), el RUP no es uno de los modelos más utilizados.

Las ***metodologías ágiles*** están pensadas para afrontar el problema de los requerimientos inciertos o cambiantes, las entregas iniciales tienen el objetivo de desarrollar los requerimientos esenciales del cliente. Con el uso de estas primeras versiones se comprende mejor el problema a solucionar y emergen nuevos requerimientos que son cubiertos en entregas posteriores. Además, la entrega continua de nuevas versiones permite hacer frente a los cambios de última hora. Cada entrega contiene requerimientos determinados al momento. El riesgo de éste tipo de metodologías es la carencia del documento de Especificación de Requerimientos. Por ejemplo, en la programación extrema, y en el Desarrollo Dirigido por Pruebas la documentación de los requerimientos se sustituye por "casos de prueba" que el sistema debe pasar cuando se implantan ciertos requerimientos. Esta escasa documentación dificulta hacer cambios al sistema cuando ya se borraron, agregaron o modificaron requerimientos anteriores sobre los que influyen estos nuevos cambios. Además, si la documentación de los requerimientos en los casos de prueba es pobre, posiblemente surgirán malos entendidos en su implantación o modificación (Pfleeger & Atlee, 2006: 145).

El modelo de desarrollo que se adopte depende de cada proyecto ya que cada uno tiene necesidades diferentes. Hay que tomar en cuenta la capacidad y experiencia del personal en el tipo de proyecto a desarrollar para elegir algún método específico. La cantidad de personal con el que se cuenta también puede llegar a ser decisivo. No es necesario limitar la elección a un solo modelo de desarrollo, pues en algunos casos es mejor combinar varios modelos (Braude, 2006:33, Sommerville, 2005: 61). Los estudios de West & Grant (2010) informan que un alto porcentaje de las empresas de software decide mezclar modelos tradicionales con metodologías ágiles. Por otra parte, hay que tomar en cuenta que, independientemente del modelo que se elija, descuidar la calidad del proyecto en sus fases

iniciales implica un desperdicio de esfuerzo al corregir los errores al final, justo cuando es más caro (McConnell, 1997:13). Esto no significa que el inicio sea *lo* importante pero sí que es lo más importante y decisivo para que un proyecto tenga éxito.

5 Conclusiones

Para aumentar las probabilidades de éxito de los proyectos de software es necesario hacer un esfuerzo adicional en su inicio. Consideramos que la elección de un modelo de desarrollo adecuado es un aspecto clave para iniciar un proyecto de software correctamente, ya que un modelo que no se adapte al proyecto entorpece su desarrollo. La nueva taxonomía propuesta en este trabajo aclara muchas dudas surgidas de la investigación de la amplia variedad de modelos de desarrollo de software. Las principales ventajas y desventajas de cada una de las categorías en esta taxonomía proporcionan una perspectiva general que facilita la elección de él o los modelos convenientes para cada proyecto en particular. La utilidad de este tipo de trabajos es evidente y resulta necesario que se sigan haciendo periódicamente para lograr un mayor entendimiento entre teóricos y practicantes de la ingeniería de software.

Referencias

Ambler, Scott (2005). "A Manager's Introduction to The Rational Unified Process (RUP)", http://www.ambysoft.com/downloads/managersIntroToRUP.pdf [22 de Julio de 2011].

_____ (2008). *The Object Primer,* 3rd Edition: Agile Model Driven Development with UML 2, U.S.A.: Cambridge University Press.

Anderson, David (2004). Feature-Driven Development: towards a TOC, Lean and Six Sigma solution for software engineering, Theory of Constraints, International Certification Organization, Microsoft.

Aguilar Sierra Alejandro (2002). "Introducción a la programación extrema", en *Revista Digital Universitaria*, v.(3), n.4, México: UNAM.

Araújo Alejandro (2007). *Test Driven Development: Fortalezas y Debilidades*. Montevideo-Uruguay: Instituto de Computación, Fac. de Ingeniería. UDELAR.

Booch Grady, Rumbaugh Jim, & Jacobson Ivar (1999). *The Unified Modeling Language User Guide*. Addison-Wesley.

Braude, Eric (2007). *Ingeniería de software, una perspectiva orientada a objetos*, México: Alfaomega.

Blum, Bruce (1994). "A taxonomy of software development methods", en *Communications of the ACM,* v.37 Issue 11, U.S.A.

Fowler, Martin (2000). "The new methodology". http://www.martinfowler.com/articles/newMethodology.html. Traducción al español. http://www.programacionextrema.org/articulos/newMethodology [21 de Julio de 2011].

IBM, Rational Unified Process. "Best Practices for Software Development Teams", en *Rational Software white paper TP026B*, n.11/01.

Jackson Michael (1998). *Software Requirements & Specifications, a lexicon of practice, principles and prejudices*. Essex: ACM press/Addison-Wesley.

Khalifa Mohamed, Verner June M. (2000). "Drivers for Software Development Method Usage", en *IEEE Transactions on Engineering Management*, v. 47, n.3, pp. 360-369.

Mangione Carmine (2003). "Software Project Failure: The Reasons, The Costs", http://www.cioupdate.com/reports/article.php/1563701/Software-Project-Failure-The-Reasons-The-Costs.htm [21 de Julio de 2011].

"Manifesto for Agile Software Development" (2001), http://www.agilemanifesto.org [21 de Julio de 2011].

McConnell, S. (1997). *Desarrollo y gestión de proyectos informáticos*, España: McGraw Hill y Microsoft Press.

McManus John & Wood-Harper Trevor. "A study in project failure". The chartered institute for IT, http://www.bcs.org/server.php?show=ConWebDoc.19584 [21 de Julio de 2011].

Norris & Rigby (1994). *Ingeniería de Software explicada*, México: Megabyte-Noriega editores.

Pazderski P. (2010). *Agile through SCRUM,* Software Process Consultant Inc. 26 May 2010 CQAA Lunch & Learn.

Pfleeger, Shari, Atlee Joanne (2002). I*ngeniería de software, Teoría y práctica*, Buenos Aires: Pearson Education, Buenos Aires.

_____ (2006). *Software Engineering, Theory and practice*. Third edition, New Jersey U.S.A.: Pearson Prentice Hall.

Poppendieck, Mary and Poppendieck Tom (2003). *Lean Software Development, an Agile Toolkit*, by Addison-Wesley Professional.

Pressman, Roger (2002). *Ingeniería del Software: Un enfoque práctico*, 5ª edición, España: McGraw Hill.

Sommerville, Ian (2005). *Ingeniería del Software*, 7ª Ed., Madrid: Pearson Addison Wesley.

Weitzenfeld, Alfredo (2004). *Ingeniería de Software Orientada a Objetos con UML, Java e Internet*. México: Editorial Thomson.

West, Dave and Grant Tom (2010). "Agile Development: Mainstream Adoption Has Changed Agility. Trends in Real-World Adoption Of Agile Methods. Application Development & Program Management Professional, January", Forrester Research Inc., http://www.mendeley.com/research/agile-development-mainstream-adoption-changed-agility/ (27 de Julio de 2011).

Wiegers, Karl (2006). *More about software requirements. U.S.A.*: Microsoft Press.