UNIVERSIDAD NACIONAL DE LA PLATA
FACULTAD DE INFORMÁTICA

Metodologías ágiles y desarrollo basado en conocimiento

Trabajo final integrador
presentado para obtener el grado de
Especialista en Ingeniería de Software

Junio de 2012

Autora: Lic. Loraine Gimson
Director: Mg. Gustavo Daniel Gil
Co-Director: Dr. Gustavo Rossi

A mi abuelo

AGRADECIMIENTOS

Este trabajo sólo fue posible gracias al aporte, directo o indirecto de muchas personas. A "todos" ellos quiero agradecer, especialmente …

- A Dios por permitirme culminar esta etapa.
- A mi Director por su tiempo dedicado, constante guía y aliento para seguir adelante.
- A mi co-Director por apoyar este trabajo y creer que era posible.
- A mi mamá, Graciela, por su apoyo sobre todo en la finalización de este trabajo.
- A mi esposo, Severo, por su paciencia constante.
- A mi hijo, Severito, por llenarme la vida de alegría.
- A mi hija que pronto conoceré…

INTRODUCCIÓN

El presente trabajo es el trabajo integrador para obtener la Especialidad en Ingeniería de Software de la Universidad Nacional de La Plata. Tiene como objetivo armar un marco teórico de las metodologías ágiles y el desarrollo basado en conocimiento. El desarrollo contiene un resumen de las características principales de las metodologías ágiles más conocidas, buscando a su vez que las mismas tengan diferentes enfoques, algunas más prescriptivas, otras más abiertas. Para realizar el trabajo se realizó una investigación bibliográfica tratando de abarcar los diferentes aspectos de estas metodologías y buscando las opiniones más actualizadas sobre las mismas.

Motivación

En este trabajo de investigación se combinan dos conceptos novedosos, fundamentalmente para nuestra provincia, respecto a la forma de encarar un desarrollo de software, tanto en el ámbito público como privado: las metodologías ágiles y el desarrollo basado en conocimiento (o también llamadas bases de datos del conocimiento). Si bien en la actualidad es más frecuente escuchar hablar de metodologías ágiles, no es común encontrar en la ciudad de Salta una empresa pública o privada que aplique concretamente alguna de ellas. En esta ciudad recién se está comenzando a tratar de incorporar algunas de las prácticas que estas metodologías proponen, y capacitar al personal en estas metodologías (mayormente en SCRUM). Además existen varias empresas públicas y privadas que están trabajando con bases de datos del conocimiento sin una metodología de desarrollo bien definida, tratando de definir un proceso de desarrollo poco burocrático que podría verse enriquecido de incorporar un marco de trabajo como el que proponen las metodologías ágiles.

Por todo lo antes expuesto, creemos que es importante poder investigar diferentes propuestas de las metodologías ágiles y profundizar en el conocimiento de las bases de datos del conocimiento. La intención es posteriormente desarrollar la tesis de la maestría de Ingeniería de Software en esta dirección, tratando de analizar el uso de bases del conocimiento en la ciudad de Salta, por parte de empresas locales salteñas públicas y privadas, para poder armar un mapa de la situación actual de cómo se está trabajando y ver si es posible sugerir una metodología ágil que acompañe este tipo de desarrollos, tratando de delinear una metodología ágil para el desarrollo basado en conocimiento. La virtud más importante de lo que se pudiera obtener es la posibilidad de combinar dos metodologías novedosas y que a simple vista no son combinables.

Objetivo General

Este trabajo busca realizar una investigación bibliográfica tendiente a exponer los fundamentos de diferentes metodologías ágiles propuestas para el desarrollo de sistemas y también realizar paralelamente una recopilación bibliográfica sobre el desarrollo basado en conocimiento.

Metodologías ágiles

Hace casi dos décadas que se comenzó a buscar una alternativa a las metodologías formales o tradicionales que estaban sobrecargadas de técnicas y herramientas y que se consideraban excesivamente “pesadas” y rígidas por su carácter normativo y fuerte dependencia de planificaciones detalladas previas al desarrollo

Las metodologías ágiles conllevan una filosofía de desarrollo de software liviana, debido a que hace uso de modelos ágiles. Se considera que un modelo es ágil o liviano cuando se emplea para su construcción una herramienta o técnica sencilla, que apunta a desarrollar un modelo aceptablemente bueno y suficiente en lugar de un modelo perfecto y complejo.

Existen actualmente una serie de metodologías que responden a las características de las metodologías ágiles y cada vez están teniendo más adeptos. Aunque los creadores e impulsores de las metodologías ágiles más populares han suscrito el manifiesto ágil y coinciden con sus postulados y principios, cada metodología ágil tiene características propias y hace hincapié en algunos aspectos más específicos.

Actualmente se pretende poder desarrollar software en el menor tiempo posible y con el menor costo. Para tratar de reducir el tiempo de programación, la solución no está relacionada tanto en mejorar más todavía los lenguajes de programación sino en la programación en sí. En los desarrollos de sistemas tradicionales se desarrolla y se realiza el mantenimiento con programación manual. Si se "describe" en vez de "programar", se pueden maximizar las descripciones declarativas y minimizar las especificaciones procedurales, haciendo desarrollo basado en conocimiento y no en programación. Esta pretensión constituye un cambio esencial de paradigma e implica un choque cultural.

La Base del conocimiento inicialmente tiene asociado un conjunto de mecanismos de inferencia y contiene reglas generales que son independientes de cualquier aplicación particular. Al describir la realidad del usuario objeto se almacenan las descripciones en el Modelo Externo. El sistema, automáticamente, captura todo el conocimiento contenido en el Modelo Externo y lo sistematiza, agregándolo también a la Base del conocimiento. Adicionalmente, sobre el conocimiento anterior, el sistema infiere lógicamente un conjunto de resultados que ayudan a mejorar la eficiencia de las inferencias posteriores. En este tipo de desarrollo el foco está en ocuparse únicamente del Modelo Externo (el “qué”) y abstenerse de tratar la Base del Conocimiento, que lo contiene y lo mantiene, (y que forma parte del “cómo”).

Contenidos

El trabajo se divide en dos partes principales, una Primera Parte donde se plantean las metodologías ágiles y una Segunda Parte donde se describe el Desarrollo basado en Conocimiento. Dentro de la primera parte se presenta un marco teórico de las Metodologías ágiles en general, su razón de ser, el manifiesto ágil para posteriormente adentrarse en diferentes metodologías ágiles. La segunda parte contiene un marco teórico de este tipo de desarrollo y luego se describe la herramienta actual que brinda este tipo de desarrollo y una futura herramienta que está siendo desarrollada. El trabajo finaliza con unas conclusiones breves de lo aprendido e investigado y se plantean líneas futuras de investigación.

INDICE

1. METODOLOGÍAS ÁGILES 1
Inicios de las Metodologías ágiles 1
El manifiesto ágil 2
Definición de Metodologías Ágiles 4
Ciclo de las metodologías ágiles [MendesCalo2008] 5
Lo nuevo de las Metodologías Ágiles [Abrahamsson2002] 5
Metodologías a describir 6
2. PROGRAMACIÓN EXTREMA (XP) 10
Introducción 10
¿Qué es XP?: Características generales 10
Valores y Principios de XP 10
Instrumentos usados en XP 11
Las Historias de Usuario 11
Roles XP 12
El Proceso de desarrollo en XP 13
Prácticas XP 15
Conclusiones 18
3. SCRUM 19
Introducción 19
Principales elementos 19
Roles 20
Roles comprometidos con el Proyecto 20
Roles involucrados en el proceso 20
El proceso Scrum 20
Planificación pre-sprint 21
Sprint 22
Reunión diaria (Scrum Diario) 23
Reunión Post Sprint – Revisión del Sprint 24
Finalizando el sprint 25
Comenzando el próximo Sprint 25
Conclusiones 25

4. KANBAN 27
Introducción 27
Lean 27
Kanban 27
Objetivos 28
Beneficios 28
¿Qué es Kanban? 28
Sistema Kanban 29
Flujo de valor 29
Límites 29
Arrastrar vs Empujar 30
Entrega rápida 30
Tablero 30
Indicadores 32
Costos 32
Funcionamiento 32
Políticas y clases 33
Coordinación y sincronización 33
Control 34
Conclusiones 34
5. OPENUP 35
¿Qué es Open UP? 35
Visión general de OpenUP 36
Principios OpenUP 37
Roles 38
Disciplinas OpenUP - Tareas 39
Tareas - Tasks 40
Artefactos - Artifacts 40
Ciclo de vida de la iteración 42
Micro_incrementos 43
Ciclo de vida del desarrollo de software mediante OpenUP 44
Fase de Inicio 46
Fase de Elaboración 46

Fase de construcción....46
Fase de transición....46
Como comenzar....47
Conclusiones....47
6. CRYSTAL CLEAR....48
Introducción....48
¿Cuál es la base para Crystal?....48
La Familia Crystal....50
El código genético de Crystal....50
Crystal Clear y la familia Crystal....51
Roles en Crystal Clear....52
Los productos de trabajo....52
Proceso Crystal Clear....53
Tipos de Procesos Involucrados....53
El ciclo del Proyecto....55
El Ciclo de Iteración....55
El episodio de distribución....56
Reflexión sobre el Proceso....57
Conclusiones....57
7. TDD (Test Driven Development)....58
Procesos.....58
Principios de TDD:....60
Roles y responsabilidades.....60
Prácticas.....60
Refactoring.....60
Integración continúa.....61
Herramientas....61
Conclusiones....61
8. DSDM (Dynamic Systems Development Method)....62
Introducción....62
Contexto....63
DSDM Atern [DSDM]....63
Método de Desarrollo de Sistema Dinámico (DSDM)....64

Variables de un proyecto [DSDM] 64
Los principios de DSDM 65
Roles. 66
Proceso Atern [CaineSitio] 69
Entregables 70
Conclusiones 71
9. CONCLUSIONES METODOLOGIAS AGILES 72
10. BASE DE CONOCIMIENTO 73
Nuevo Paradigma: Describir en vez de Programar [Gonda2007] 73
Metodologías tradicionales de desarrollo y problemas asociados [Artech2008] 73
¿Paradigma orientado al conocimiento? [Gonda2003] 74
Nuevo Paradigma: desarrollo basado en conocimiento [Gonda2010] 75
Modelo Externo y Base de Conocimiento [Gonda2007] 77
Genexus 79
Orígenes de Genexus [Gonda2007] 81
Tratamiento Automático del Conocimiento por Genexus [Gonda2010] 81
Ciclo de desarrollo incremental Genexus [Artech208] 82
Tendencias de Genexus 86
¿Por qué elegir este nuevo paradigma? 87
Proyecto Altagracia 87
Motivación del proyecto 87
Altagracia 87
Conclusiones 88
11. CONCLUSIONES GENERALES 89
12. BIBLIOGRAFÍA 90
Metodologías ágiles 90
Referencias Bibliográficas 94
Base de datos del conocimiento 96

1. METODOLOGÍAS ÁGILES

"Desde hace unos pocos años ha habido un interés creciente en las metodologías ágiles (léase "livianas"). Caracterizadas alternativamente como antídoto a la burocracia, han suscitado interés en el panorama del software." Martin Fowler [Fowler]

Inicios de las Metodologías ágiles

Tal como lo menciona Martin Fowler en su artículo *La nueva metodología*, podría decirse que el desarrollo de software es una actividad caótica, frecuentemente caracterizada por la frase "codifica y corrige". El software se escribe con un mínimo plan subyacente, y el diseño del sistema se arma con muchas decisiones a corto plazo. Esto realmente funciona muy bien si el sistema es pequeño, pero conforme el sistema crece llega a ser cada vez más difícil agregar nuevos aspectos al mismo. Además los errores llegan a ser cada vez más frecuentes y más difíciles de corregir.

Se ha sobrevivido con este estilo de desarrollo por mucho tiempo, pero también surgió una alternativa desde hace muchos años: emplear una Metodología de Desarrollo Ingenieril. Las metodologías imponen un proceso disciplinado sobre el desarrollo de software con el fin de hacerlo más predecible y eficiente. Lo hacen desarrollando un proceso detallado con un fuerte énfasis en planificar inspirado por otras disciplinas de la ingeniería. Dichos procesos definen artefactos de desarrollo, documentación a producir, herramientas y notaciones a ser utilizadas, y actividades a realizar y su orden de ejecución, entre otras definiciones. Como resultado, los procesos generan gran cantidad de documentación con el objetivo de facilitar compartir el conocimiento entre los integrantes del equipo de trabajo. Si bien existen varios procesos de desarrollo – Proceso Unificado [Jacobson], Proceso V [IABG], etc., la mayoría de estos procesos se derivan del Modelo de Cascada propuesto por Boehm [Bohem1976]. Las metodologías ingenieriles, denominadas *tradicionales*, han estado presentes durante mucho tiempo y han demostrado ser efectivos, particularmente en lo que respecta a la administración de recursos a utilizar y a la planificación de los tiempos de desarrollo, en proyectos de gran tamaño con requerimientos estables. La crítica más frecuente a estas metodologías es que son burocráticas. Fowler afirma que, hay tanto que hacer para seguir la metodología que el ritmo entero del desarrollo se retarda.

Sin embargo, debido a entornos comerciales más competitivos, conducidos por la rapidez para producir y entregar servicios [Ridderstrale2000], el enfoque propuesto por estas metodologías tradicionales o burocráticas no resulta el más adecuado. Usualmente, estos nuevos entornos se caracterizan por el desarrollo de proyectos donde los requerimientos del sistema son desconocidos, inestables o muy cambiantes, los tiempos de desarrollo se deben reducir drásticamente, y al mismo tiempo, se espera la producción de un producto de alta calidad, que a su vez garantice mínimo riesgo ante la necesidad de introducir cambios a los requerimientos.

Como una reacción a estas metodologías tradicionales, "pesadas", complejas, muy estructuradas y estrictas y sobrecargadas de técnicas y herramientas, ha surgido un nuevo grupo de metodologías desde los años 90. Durante algún tiempo se conocían como las metodologías ligeras, pero el término aceptado y definido por la Agile Alliance es *metodologías ágiles*. Estas metodologías están especialmente indicadas para productos cuya definición detallada es difícil de obtener desde el comienzo, o que si se definiera, tendría menor valor que si el producto se construye con una retro-alimentación continua durante el proceso de desarrollo. Manteniendo prácticas esenciales de las metodologías tradicionales, las metodologías ágiles se centran en otras dimensiones del proyecto, como por ejemplo: intentan trabajar con un mínimo de documentación necesaria, reemplazándola por la comunicación directa y cara a cara entre todos los integrantes del equipo; la colaboración activa de los usuarios durante todas las etapas del proceso de desarrollo; el desarrollo incremental del software con iteraciones muy cortas y que entregan una solución a medida; y la reducción drástica de los tiempos de desarrollo pero a su vez manteniendo una alta calidad del producto. Buscan un justo medio entre ningún proceso y demasiado proceso, proporcionando simplemente suficiente proceso para que, como dice Fowler, el esfuerzo valga la pena.

Por todo lo mencionado puede verse fácilmente que este nuevo concepto dentro de la ingeniería de software, denominado modelado ágil de sistemas o metodologías ágiles, se trata

de una filosofía de desarrollo liviana, debido a que hace uso de modelos ágiles. Logró adeptos, inició una corriente filosófica dentro del concierto de metodologías para el desarrollo de sistemas de software, conformó un corpus documental de sus principios y de sus prácticas y obtuvo éxitos resonantes en proyectos de envergadura. [Giro] [Ferrer] [Canós]

En el año 2001, miembros prominentes de la comunidad se reunieron en Snowbird, Utah, y adoptaron el nombre de "metodologías ágiles", definiendo además el manifiesto ágil. Poco después, algunas de estas personas formaron la Agile Alliance, una organización sin fines de lucro que promueve el desarrollo ágil de aplicaciones. Muchos métodos ágiles fueron creados antes del 2000. Entre los más notables se encuentran: Scrum (1986), Crystal Clear (cristal transparente), programación extrema o XP (1996), desarrollo de software adaptativo, Dynamic Systems Development Method (Método de desarrollo de sistemas dinámicos) (1995).

El manifiesto ágil

El Manifiesto Ágil es el documento que define los principios rectores de las metodologías ágiles de desarrollo de software. Este importante documento fue creado por un grupo de académicos y expertos de la industria del software reunido en Snowbird, Utha, Estados Unidos en febrero de 2001, a través de una iniciativa de Kent Beck. Este grupo estaba conformado por 17 representantes de procesos de desarrollo livianos. Las 17 personas fueron Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Hihsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Stephen J. Mellor, Ken Schwaber, Jeff Sutherland, Dave “Pragmatic” Thomas. Otras personas fueron invitadas pero estas 17 asistieron.

La razón de la reunión fue ver si había algo en común entre las diferentes metodologías livianas de ese momento: Adaptative Software Development, XP, Scrum, Crystal, FDD, DADM y “Pragmatic Programming”. Tal como lo aclara Cockburn [Cockburn2007], ninguno estaba interesado en combinar las prácticas para crear una Metodología Liviana Unificada (Unified Light Methodology – ULM). El objetivo de la reunión era discutir los valores y principios que facilitarían desarrollar software más rápidamente y respondiendo a los cambios que surjan a lo largo del proyecto. La idea era ofrecer una alternativa a los procesos de desarrollo tradicionales, caracterizados por su rigidez y dirigidos por la documentación que se genera en cada etapa.

En esta reunión se decidió adoptar el término *Ágil* y como resultado se obtuvo un documento conocido como el Manifiesto Ágil [ManifiestoAgil]. El Manifiesto Ágil incluye cuatro postulados y una serie de principios asociados. Tal como hace observar Alistair CockBurn [Cockburn2007], el propio manifiesto comienza diciendo que se están sacando a la luz mejores formas de desarrollar software, no las están inventando. Además, que el resultado se obtuvo de la experiencia directa y la reflexión sobre la experiencia y que se reconoce valor a las herramientas, procesos, documentación, contratos y planes, si bien ponen mayor énfasis en otros valores. Por esto es que las metodologías ágiles cambian significativamente algunos de los énfasis de las metodologías tradicionales lo que puede apreciarse en los postulados del manifiesto ágil.

Sus postulados son:

1) *Valorar al individuo y a las interacciones del equipo de desarrollo por encima del proceso y las herramientas*. Este postulado enuncia que las personas son componentes primordiales en cualquier desarrollo [Cockburn1999]. Tres premisas sustentan este postulado:

a) los integrantes del equipo son el factor principal de éxito;

b) es más importante construir el equipo de trabajo que construir el entorno; y

c) es mejor crear el equipo y que éste configure el entorno en base a sus necesidades. [MendesCalo2010]

Se presta atención a las personas en el equipo como opuesto a los roles en un diagrama de proceso, donde las personas son reemplazables. Y por otro lado se presta atención a la

interacción de los individuos. Es preferible un proceso no documentado con buenas interacciones que un proceso bien documentado con interacciones hostiles [Cockburn2007]

2) *Valorar el desarrollo de software que funcione por sobre una documentación exhaustiva*. El postulado se basa en la premisa que los documentos no pueden sustituir ni ofrecer el valor agregado que se logra con la comunicación directa entre las personas a través de la interacción con los prototipos. Se debe reducir al mínimo indispensable el uso de documentación que genera trabajo y que no aporta un valor directo al producto [MendesCalo2010].

El sistema funcionando es lo único que muestra lo que el equipo *ha desarrollado*. Correr código es honestidad implacable. La documentación la utiliza el equipo para reflexionar, como pistas de lo que se debería realizar. El acto completo de reunir requerimientos, diseñar, codificar, probar el software revela información del equipo, el proceso, y la naturaleza del problema a resolver. Esto, junto con la posibilidad de correr el resultado final, provee la única medida confiable de la velocidad del equipo, y un vistazo de lo que el equipo deberían estar desarrollando. Los documentos pueden ser útiles pero deben usarse en la “medida justa” o “a penas suficiente” [Cockburn2007]

3) *Valorar la colaboración con el cliente por sobre la negociación contractual*. En el desarrollo ágil el cliente se integra y colabora con el equipo de trabajo. Se asume que el contrato en sí, no aporta valor al producto, sino que es sólo un formalismo que establece líneas de responsabilidad entre las partes. [MendesCalo2010]

Es muy importante la relación entre la persona que quiere el software y los que la construyen. La *Colaboración* se refiere a amigabilidad, toma de decisiones conjuntas, comunicación rápida, y conexiones de interacción entre individuos. Aunque a veces los contratos son útiles, la colaboración fortalece el desarrollo tanto cuando hay un contrato como cuando no existe el contrato. [Cockburn2007]

4) *Valorar la respuesta al cambio por sobre el seguimiento de un plan*. La evolución rápida y continua deben ser factores inherentes al proceso de desarrollo. Se debe valorar la capacidad de respuesta ante los cambios por sobre la capacidad de seguimiento y aseguramiento de planes pre-establecidos. [MendesCalo2010]

El valor final se refiere a la rapidez para ajustarse a los cambios del proyecto. Construir un plan es útil y cada metodología ágil contiene actividades de planificación específicas. Pero también tienen mecanismos para manejar cambios de prioridades.

Scrum, DSDM, Adaptative Software Development convocan a un desarrollo en períodos fijos (timebox) con repriorización posterior (no durante) a cada periodo fijo (timebox) – XP permite repiorización durante el timebox. Los periodos son de 2 a 4 semanas. El trabajar de esta forma garantiza que el equipo tenga el tiempo y tranquilidad mental para desarrollar *software que funcione*. Al trabajar con iteraciones cortas, los sponsors del proyecto pueden cambiar las prioridades para que reflejen sus necesidades. [Cockburn2007]

Los postulados descriptos anteriormente, inspiraron los doce principios del Manifiesto Ágil. Estos principios son las características que diferencian un proceso ágil de uno tradicional. Dos de ellos hacen referencias a generalidades, luego hay seis que tienen que ver directamente con el proceso de desarrollo de software a seguir y cuatro que están más directamente relacionados con el equipo de desarrollo, en cuanto metas a seguir y organización del mismo.

1) la prioridad es satisfacer al cliente mediante entregas de software tempranas y continuas;
2) los cambios en los requerimientos son aceptados;
3) software que funcione se entrega frecuentemente, con el menor intervalo posible entre entregas;
4) el cliente y los desarrolladores deben trabajar juntos a lo largo del proyecto;
5) el proyecto se construye en base a individuos motivados;

6) el dialogo cara a cara es el método más eficiente y efectivo para comunicar información dentro del equipo;
7) el software que funcione es la medida principal del progreso;
8) los procesos ágiles promueven el desarrollo sostenido;
9) la atención continua a la excelencia técnica y al buen diseño mejora la agilidad;
10) la simplicidad es esencial;
11) las mejores arquitecturas, requerimientos y diseños surgen de equipos auto-organizados, y
12) el equipo reflexiona en cómo ser más efectivos, y ajusta su comportamiento en consecuencia.

Aunque los creadores e impulsores de las metodologías ágiles más populares han suscrito el manifiesto ágil y coinciden con los principios enunciados anteriormente, cada metodología tiene características propias y hace hincapié en algunos aspectos más específicos.

En el manifiesto no se encuentra la necesidad de diferentes formas de trabajar en diferentes situaciones, pero Jim Highsmith y Alistar Cockburn [Cockburn2007] aconsejan tener esta idea en mente. Ser ágil es diferente, por ejemplo, para un proyecto de 100 personas que para uno de 10 personas. El punto es que las metodologías deben adaptarse o afinarse para el proyecto que se tiene en frente. Esta idea no se capturó en el manifiesto.

Algunos de los autores recomiendan metodologías ágiles para situaciones muy fluctuantes pero por ejemplo para Cockburn, en su propia experiencia aún sirvió esta metodología en proyectos supuestamente estables.

Definición de Metodologías Ágiles

De acuerdo con lo que menciona Damon B. Poole, en su libro "*Do it yourself agile",* dar una definición concisa de Metodología Ágil no es nada fácil, probablemente porque Ágil es realmente un paraguas de una variedad amplia de metodologías y porque Ágil está definido oficialmente como los 4 valores en el Manifiesto Ágil

- Individuos e interacciones sobre procesos y herramientas
- Software funcionando sobre documentación extensiva
- Colaboración con el cliente sobre negociación contractual
- Respuesta ante el cambio sobre seguir un plan

Dado que hay tantas prácticas asociadas al desarrollo ágil, una forma más simple de definir Ágil es hacerlo en término de beneficios. Poole define el desarrollo ágil como aquel que, en comparación con el desarrollo tradicional, provee beneficios de mayor flexibilidad, Retorno de Inversión más alto, realización más rápida del Retorno de Inversión, más alta calidad, mayor visibilidad, y paz sostenible. [Poole]

Poole hace notar que se habla de Metodologías Ágiles como un solo paquete: si se adhiere a los principios del manifiesto ágil, se obtienen los beneficios. Pero, hay otra forma de mirarlas y es pensar que las metodologías ágiles introducen un conjunto nuevo y completo de prácticas al conjunto de herramientas de desarrollo. Estas prácticas incluyen: product backlog, programación de a pares, cliente en el lugar, integración continua, refactorización, desarrollo dirigido por pruebas (TDD) y muchas otras. Mientras que todas estas prácticas han estado asociadas con las Metodologías Ágiles o fueron creadas como resultado de las Metodologías Ágiles, su aplicación y beneficios resultantes pueden aplicarse completamente independientes de cualquier metodología específica. [Poole]

Por otro lado, según [Giro] [Ferrer] [Canós], se considera que un modelo es ágil o liviano cuando se emplea para su construcción una herramienta o técnica sencilla, que apunta a

desarrollar un modelo aceptablemente bueno y suficiente en lugar de un modelo perfecto y complejo. Un modelo es suficientemente bueno cuando cumple con los objetivos para los que fue creado, esto es, logra principalmente el propósito de la comunicación; es entendible por la audiencia para la que fue concebido; no es perfecto y puede presentar algunos errores e inconsistencias no esenciales; posee un grado de detalle adecuado; suma valor al proyecto; y es suficientemente simple de construir [Ambler].

Ciclo de las metodologías ágiles [MendesCalo2008]

Si bien el ciclo de desarrollo que aplican las Metodologías Ágiles es iterativo e incremental, tal como se referencia en varios trabajos relacionados, el factor humano es fundamental para el éxito del proyecto [Cockburn1999] [McConnell2003] [Glass2002]. Este modelo permite entregar el software en partes pequeñas y utilizables, conocidas como incrementos. Estas metodologías, adhiriendo a lo expuesto por Mendes Calo, aportan nuevos métodos de trabajo que requieren una cantidad de procesos, que no se desgastan con cuestiones administrativas – tales como planificación, control, documentación - ni tampoco defienden la postura extremista de la total falta de proceso. Debido a que se tiene conciencia que los cambios indefectiblemente se producirán, el objetivo es reducir el costo de rehacer parte del producto por los cambios introducidos. Se intenta guiar el desarrollo hacia un objetivo que puede no permanecer constante en el tiempo a medida que aumenta el conocimiento de la aplicación a ser construida. Cada iteración se puede considerar como un mini-proyecto en el que las actividades de análisis de requerimiento, diseño, implementación y testing son llevadas a cabo con el fin de producir un subconjunto del sistema final. El proceso se repite varias veces produciendo un nuevo incremento en cada ciclo. Dicho proceso concluye cuando se haya elaborado el producto completo. Si bien todas las metodologías ágiles adoptan este ciclo, cada una presenta sus propias características. Si bien la mayoría de las metodologías ágiles satisfacen los postulados y principios del Manifiesto Ágil, no todas lo hacen de la misma manera. Más aún, el proceso de seleccionar la metodología que mejor se adapta a un problema en particular, se torna dificultoso.

Lo nuevo de las Metodologías Ágiles [Abrahamsson2002]

Según Highsmith y Cockburn [Highsmith2001], "lo que es nuevo sobre los métodos ágiles no son las prácticas que usan, sino el reconocimiento de personas como los conductores primarios al éxito del proyecto, junto con un foco intenso en la efectividad y mantenibilidad. Esto produce una nueva combinación de valores y principios que definen una vista *ágil* del mundo". Bohem en [Bohem2002] ilustra el espectro de diferentes métodos de planificación como se muestra en la **Figura 1.1**., donde los hackers están ubicados en un extremo y el enfoque contractual pormenorizado con límites de acero en el extremo opuesto.

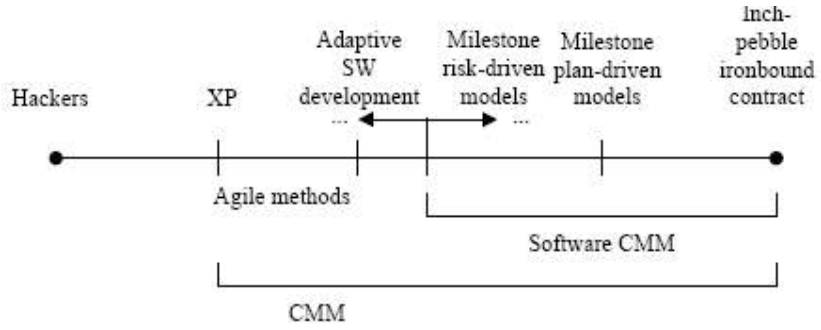


Figura 1.1. El espectro de la planificación [Bohem2002]

Hawrysh y Ruprecht, [Hawrysh2002], declaran que una sola metodología no puede funcionar para todo un espectro de proyectos diferentes, pero en su lugar el administrador de proyectos debe identificar la naturaleza específica del proyecto en mano y luego seleccionar la mejor metodología de desarrollo aplicable. Para remarcar más ese punto, según [McCourley2001] hay una necesidad tanto de métodos ágiles como de métodos orientados a procesos, así

como no hay un modelo de desarrollo de software que le vaya bien a todos los propósitos imaginables. Esta opinión es compartida por varios expertos en el campo [Glass2000].

Cockburn, [Cockburn2002], define el corazón de los métodos de desarrollo de software ágil como el uso de reglas livianas pero suficientes sobre el comportamiento de un proyecto y el uso de reglas orientadas a las personas y a la comunicación. El proceso ágil es tanto liviano como suficiente; liviano significando mantenerse maniobrable y suficiente como un asunto relacionado a mantenerse en el juego.

Según Abrahamsson Pekka, Salo Outi, Ronkainen Jussi & Juhani Warsta [Abrahamsson2002], estudios han demostrado que las metodologías de desarrollo de software tradicionales dirigidas por planes no se usan en la práctica. Se ha argumentado que las metodologías tradicionales son demasiado mecánicas para usarse en detalle. Como resultado, los desarrolladores de software industriales se han vuelto escépticos sobre "nuevas" soluciones que son difíciles de comprender y por lo tanto permanecen sin usar. Los métodos de desarrollo de software ágil, que "oficialmente" comenzaron con la publicación del manifiesto ágil, hacen un intento por hacer un cambio en el campo de la ingeniería de software. Los métodos ágiles claman por colocar más énfasis en las personas, la interacción, el software funcionando, la colaboración del cliente, y el cambio más que en los procesos, las herramientas, los contratos y los planes.

El pensar de manera ágil es una vista centrada en las personas para desarrollar software [Abrahamsson2002], Las estrategias centradas en las personas son una fuente importante de ventaja competitiva, porque, a diferencia de la tecnología, los costos o las nuevas tecnologías, estas estrategias humanas son difíciles de imitar ([Pfeffer1998];[Miller2001]). Eso sin embargo no es una nueva creación. Una edición de verano de 1990 de American Programmer (Ed Yourdon's Software Journal, Vol3. No 7-8) estuvo dedicada exclusivamente a "Peopleware". El editor señala que "todos saben que la mejor forma de mejorar la productividad y calidad software es enfocarse en las personas". Por lo tanto, la idea que traen las metodologías ágiles no son nuevas, ni tampoco los agilistas se adjudican esto. Ellos, sin embargo, creen que los métodos ágiles de desarrollo de software proveen una forma novedosa de aproximarse a los problemas de ingeniería de software, así como también sostienen que los métodos no son de ninguna forma exhaustivos o capaces de resolver todos los problemas

Metodologías a describir

El desarrollo ágil requiere innovación y mantenerse receptivo, está basado en generar y compartir conocimiento entre el grupo de desarrollo y con el cliente. Los desarrolladores de software ágil hacen uso de las fortalezas de clientes, usuarios y desarrolladores, encontrando solo un proceso suficiente para balancear calidad y agilidad [Cockburn2007]

Existe gran variedad de metodologías ágiles. Pudiendo complementarse unas con otras dado que el enfoque en cada una puede ser diferente. Por ejemplo XP se centra en la programación y Scrum en la administración. Pero muchas organizaciones están utilizando estas metodologías, como por ejemplo: Google, Canon, NEC, Seros, Fuji, Oracle, Toyota, Honda, Nokia, Yahoo!, Microsoft, HP, 3M, Sun, Epson [Programación-extrema]

La última encuesta del "Estado del Desarrollo Ágil" realizada por VersionOne fue desarrollada entre el 22 de julio y el 1 de noviembre de 2011. Esta encuesta fue respondida por 6042 empresas. Más de la mitad de los encuestados dijo que personalmente había seguido las prácticas ágiles desde hace 2 años, y una tercera ha llevado la metodología ágil con ellos a otra empresa. Casi dos tercios de los encuestados dijo que hasta la mitad de los proyectos de su empresa se realizaron utilizando ágil, y que su empresa ha adoptado las prácticas ágiles a través de 3 o más equipos.

A continuación se muestran algunos de los datos extraídos de las encuestas [VersionOne] y posteriormente se presentan algunos gráficos asociados a estas encuestas:

- **Razones para adoptar una metodología ágil**: Acelerar el tiempo de comercialización es de la razón número uno. Las tres más elegidas fueron: acelerar el tiempo de comercialización, aumentar productividad y hacer más fácil la administración de cambios en las prioridades.

- **Beneficios obtenidos de una implementación ágil**: Los tres beneficios más votados fueron: capacidad de administrar cambios en las prioridades, mejorar la visibilidad del proyecto, aumentar la productividad.
- **Metodologías ágiles utilizadas**: se destaca Scrum y sus variantes que conforman más de dos tercios y se registra un aumento del uso de Kanban respecto a encuestas anteriores.
- **Técnicas ágiles empleadas**: Las más destacadas son Reunión Diaria de Pie (Daily Sandup), planificación de la iteración y pruebas de unidad. Hubo un aumento en el uso de principios Kanban respecto de la encuesta anterior.
- **Implementaciones ágiles futuras**: el 59% planea implementarlas, un 33% no está seguro y solo el 8% restante no piensa utilizarlo

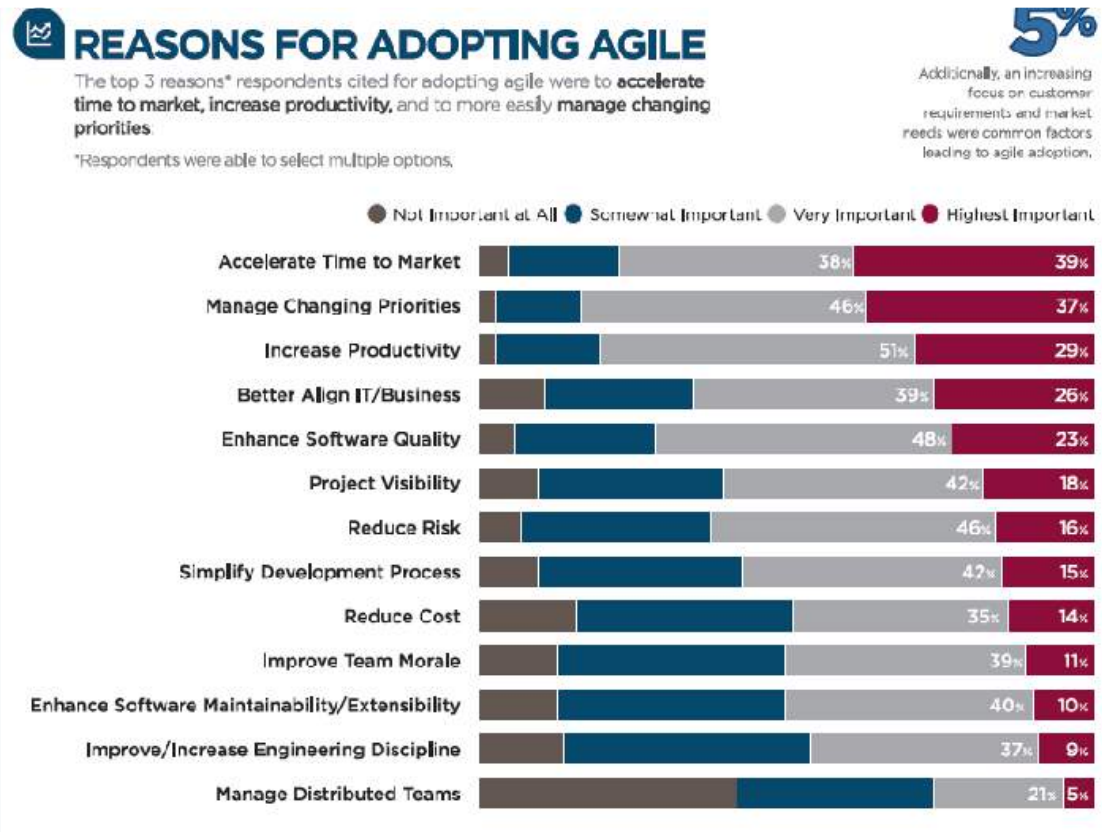


Figura 1.2. Razones para adoptar una metodología ágil [VisionOne]

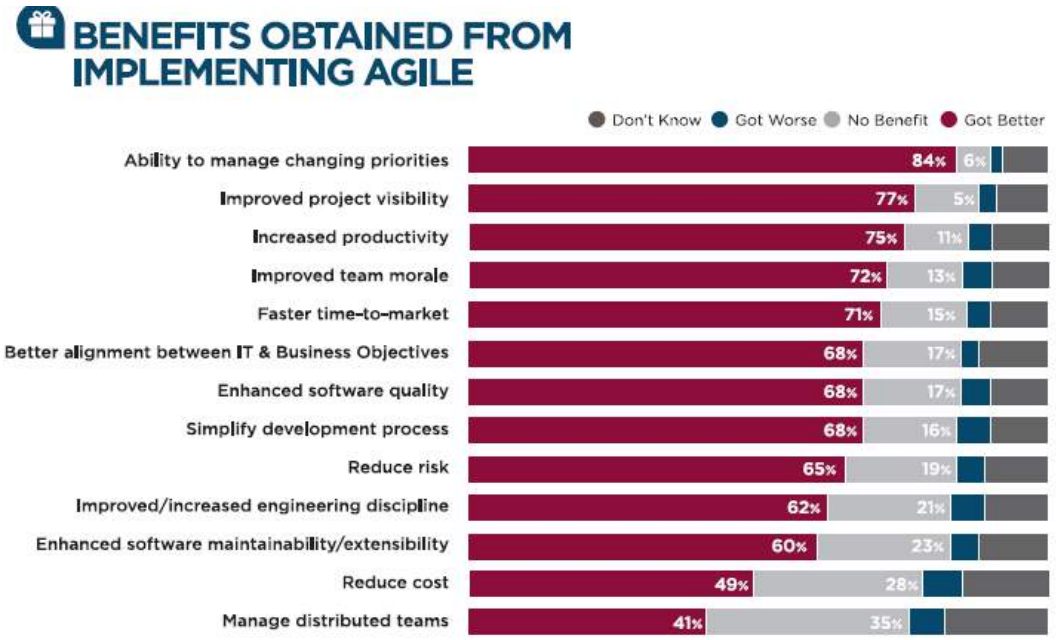


Figura 1.3. Beneficios obtenidos por implementar una metodología ágil [VisionOne]

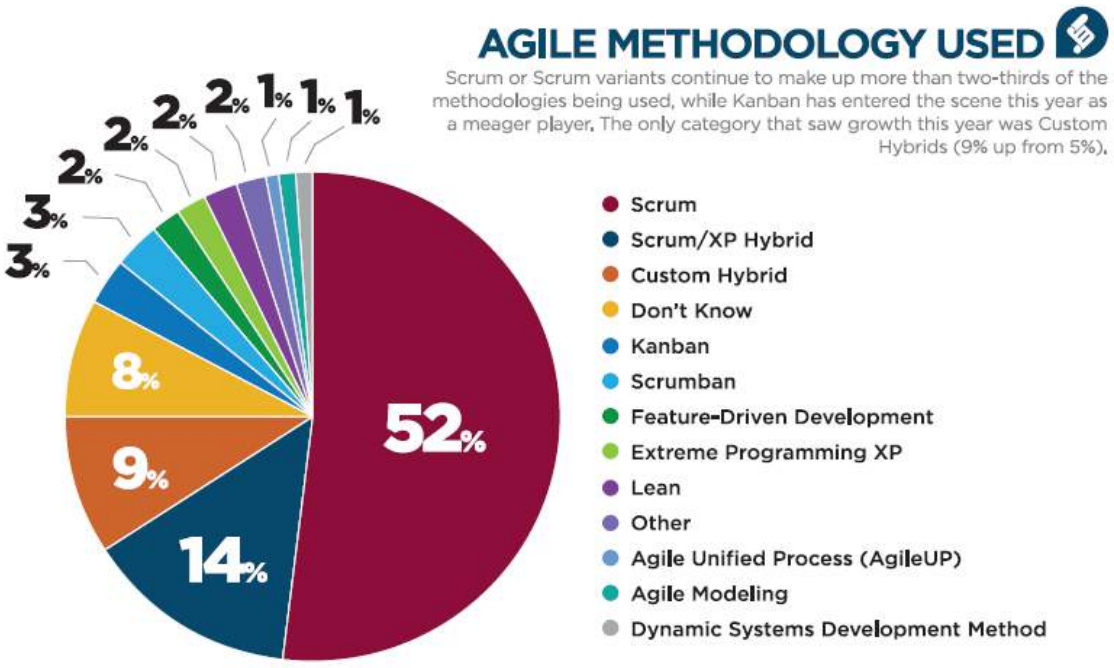


Figura 1.4. Metodologías ágiles utilizadas [VisionOne]

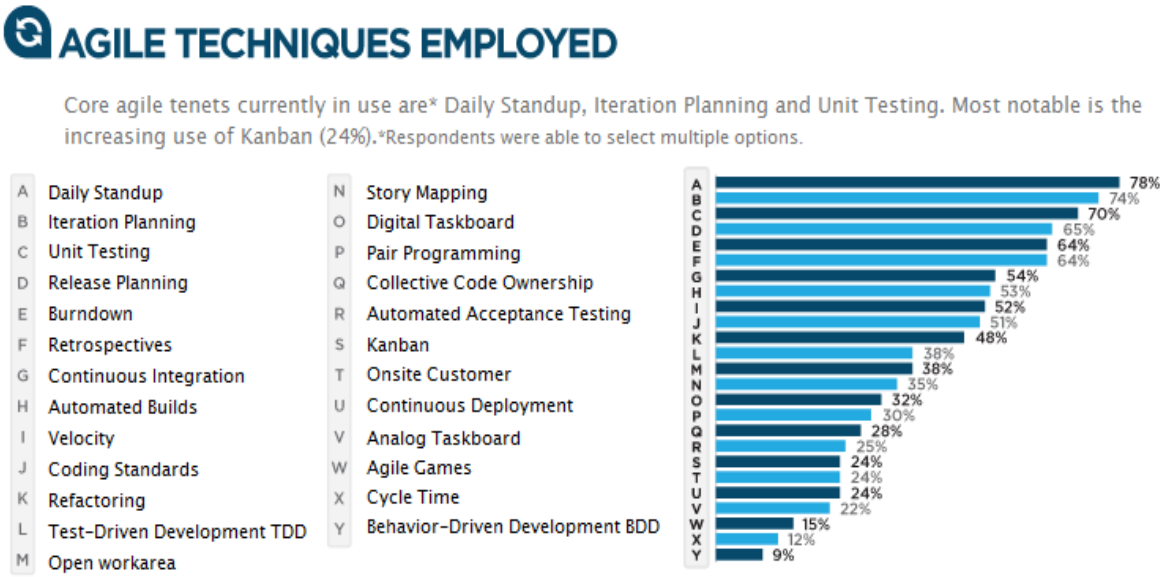


Figura 1.5. Técnicas ágiles empleadas [VisionOne]

En Argentina, un estudio realizado por M. Alvarez, F. Escobar, A. Nardin, E. Ricci Aparicio, G. Bioul, denominado Análisis de la implementación de prácticas ágiles en Argentina publicado en AgilSpain en diciembre de 2011 afirma que en Argentina de un grupo heterogéneo de empresas consultadas, el 85% usa Scrum aunque no en forma completa sino adaptándolo las técnicas que consideran más apropiadas. Algunas usan otras prácticas ágiles, XP y TDD, sin especificar porcentajes. [AgileSpain]

Las metodologías a describir en este trabajo son XP, Scrum, Kankan, OpenUP, Cristal, TDD y DSDM. Las prácticas de ingeniería de XP evolucionaron junto con Scrum y los dos principales procesos ágiles de desarrollo trabajan bien juntos. Scrum y XP son las prácticas ágiles más utilizadas en todo el mundo y sus creadores son co-autores del Manifiesto Ágil [Sutherland2011]. Debido a esto es que se detallarán un poco más que el resto de las metodologías aquí mencionadas. Kankan está surgiendo con gran adhesión, por lo que es indispensable considerarla. Open UP presenta una metodología más cercana a lo que sería una metodología tradicional como es el Proceso Unificado. Crystal Clear, por el contrario de Open UP, va al extremo opuesto dejando la mayor libertad a la hora de definir un proceso de desarrollo. A TDD algunos la consideran más bien una técnica dentro de otra metodología, como por ejemplo dentro de XP, pero su planteo es sumamente provechoso a la hora de encarar un proceso de desarrollo ágil. DSDM fue la primera metodología ágil y sigue siendo utilizada mayormente en Europa. La intención final es tener un panorama general de diferentes propuestas ágiles actuales.

2. PROGRAMACIÓN EXTREMA (XP)

Introducción

La expresión Programación eXtrema es la traducción del inglés de "Extreme Programming" (XP) acuñada por Kent Beck quien ahora se considera como una de las principales figuras de este modelo de programación, junto a Ron Jeffries y Ward Cunningham quienes fueron participes de la conformación y divulgación de una metodología mucho más arriesgada, versátil y flexible para el desarrollo de software. [Fowler_b]. Esta propuesta metodológica fue denominada extrema porque lleva a límites extremos algunos elementos y actividades comunes de la forma tradicional de programar, de ahí proviene su nombre.

Si bien la programación extrema surgió en la década de los 90's con el desarrollo de un proyecto de elaboración de software, XP hizo explosión en la escena del desarrollo de software en 2000-2001. Varias de las otras metodologías ágiles estuvieron prácticamente en la oscuridad antes de que XP apareciera en escena. Según [Highsmith], XP ayudó a llevar a la luz a toda la categoría "Ágil".

XP deriva de buenas prácticas que han estado alrededor por largo tiempo. Son prácticas simples, que podrían considerarse ingenuas o extrañas al principio, fácilmente adoptadas luego, apoyadas unas en otras, con reducción de actividades improductivas. El propio Kent afirma que "*ninguna de las ideas en XP son nuevas. Muchas son tan viejas como la programación*". Aunque, adhiriendo al comentario de Jim Highsmith, *"Mientras las prácticas que usa XP no son nuevas, las bases conceptuales y como se mezclan entre sí son nuevas y mejoran grandemente estas "viejas" prácticas*".

¿Qué es XP?: Características generales

XP es una metodología que sigue la filosofía de las metodologías ágiles, cuyo objetivo es conseguir la máxima satisfacción del cliente en forma rápida y eficiente ante los cambios de requisitos. XP [Beck] es una metodología ágil centrada en potenciar las relaciones interpersonales como clave para el éxito en desarrollo de software, promoviendo el trabajo en equipo, preocupándose por el aprendizaje de los desarrolladores, y propiciando un buen clima de trabajo. XP se basa en realimentación continua entre el cliente y el equipo de desarrollo, comunicación fluida entre todos los participantes, simplicidad en las soluciones implementadas y coraje para enfrentar los cambios. Propone realizar diseños simples, códigos simples y proporcionar rápida respuesta de lo requerido y lograr un cliente contento.

Se sustituye la documentación escrita por la comunicación directa entre clientes y desarrolladores o entre los propios desarrolladores. Propone un desarrollo iterativo a través de cuatro pasos, planificación, diseño, codificación y prueba. En cada iteración se añaden nuevas funcionalidades al software. Es especialmente adecuada para proyectos con requisitos imprecisos y muy cambiantes, y donde existe un alto riesgo técnico.

Se basa en una serie de prácticas y principios que se han ido gestando a lo largo de toda la historia de la ingeniería del software, son de sentido común pero llevadas al extremo. Usadas conjuntamente proporcionan una nueva metodología de desarrollo software que se puede englobar dentro de las metodologías ágiles o ligeras. Kent Beck describe la filosofía de XP en [Beck] sin cubrir los detalles técnicos y de implantación de las prácticas. Posteriormente, otras publicaciones de experiencias se han encargado de dicha tarea.

"*La programación extrema se basa en la simplicidad, la comunicación y el reciclado continuo de código, para algunos no es más que aplicar una pura lógica"* [Calero]

Valores y Principios de XP

El proceso de desarrollo en XP está fundamentado en una serie de valores y principios que lo guían. Los valores representan aquellos aspectos que los autores de XP han considerado como fundamentales para garantizar el éxito de un proyecto de desarrollo de software. Un valor es una descripción de cómo debe enfocarse el desarrollo de software.

Los partidarios de la programación extrema dicen que estos cuatro valores son los necesarios para conseguir diseños y códigos simples, métodos eficientes de desarrollo software y clientes contentos [Deister 2006] [Tldp 2006]. Estos valores le brindan consistencia y solidez al equipo de trabajo [Calero]. Los valores deben ser intrínsecos al equipo de desarrollo. Los cuatro valores de XP son:

- **Comunicación** que prevalece en todas las prácticas de XP. La comunicación cara a cara es la mejor forma de comunicación, entre los desarrolladores y el cliente. Es un método muy ágil. También apoya agilidad con la extensión del conocimiento tácito dentro del equipo del desarrollo, evitando la necesidad de mantener la documentación escrita.
- **Simplicidad** o sencillez que ayuda a que los desarrolladores de software encuentren soluciones más simples a problemas, según el cliente lo estipula. Los desarrolladores también crean características en el diseño que pudieran ayudar a resolver problemas en un futuro.
- **Realimentación** continua o feedback del cliente que permite a los desarrolladores llevar y dirigir el proyecto en una dirección correcta hacia donde el cliente decida. Apunta a la respuesta rápida, constante e iterativa que se le ofrece al cliente.
- **Coraje** que requiere que los desarrolladores vayan a la par con el cambio, ya que el cambio es inevitable, pero el estar preparado con una metodología ayuda a ese cambio.

De los cuatro valores, quizás el que llame más la atención es el de coraje. Detrás de este valor se encuentra el lema "si funciona, mejóralo", que va en contra de la práctica habitual de no tocar algo que funciona, por si acaso. Aunque también es cierto que al tener las pruebas unitarias, no se pide a los desarrolladores una heroicidad, sino sólo coraje. Otro punto de vista del coraje, es el del administrador del proyecto quien debe estar lo suficientemente seguro de sí mismo para abandonar un buen trato de control, ya que no tendrá más la autoridad para determinar qué quiere y cuándo lo quiere.

Los principios suponen un puente entre los valores, algo intrínseco al equipo de desarrollo y las prácticas, que se verán posteriormente, y que están más ligadas a las técnicas que se han de seguir. Los principios fundamentales se apoyan en los cuatro valores previamente definidos y también son cuatro: realimentación veloz, modificaciones incrementales, trabajo de calidad y asunción de simplicidad.

Instrumentos usados en XP

Dentro de los instrumentos usados en XP resaltan las historias de usuarios como el instrumento principal de la metodología. Además, se utilizan otros artefactos como las tareas de ingeniería o programación y las tarjetas CRC (Tarjetas de Clases, Responsabilidades y Colaboración).

Las Historias de Usuario

Las historias de usuario son la técnica utilizada en XP para especificar los requisitos del software. Representan una breve descripción del comportamiento del sistema. Se trata de tarjetas de papel en las cuales el cliente describe brevemente las características que el sistema debe poseer, sean requisitos funcionales o no funcionales. Se emplea terminología del cliente sin lenguaje técnico, se realiza una historia de usuario por cada característica principal del sistema. Se emplean para hacer estimaciones de tiempo y para el plan de lanzamientos. [Jefrries_Sitio]

El tratamiento de las historias de usuario es muy dinámico y flexible, en cualquier momento las historias de usuario pueden romperse, reemplazarse por otras más específicas o generales, añadirse nuevas o ser modificadas. Cada historia de usuario es lo suficientemente comprensible y delimitada para que los programadores puedan implementarla en unas semanas. [Letelier_b] Para efectos de planificación, las historias pueden ser de una a tres semanas de tiempo de programación (para no superar el tamaño de una iteración).

Las historias de usuario tienen tres aspectos: tarjeta (se almacena suficiente información para identificar y detallar la historia), conversación (cliente y programadores discuten la historia para ampliar los detalles) y pruebas de aceptación (permite confirmar que la historia ha sido implementada correctamente). Las historias de usuario se descomponen en tareas de programación y son asignadas a los programadores para ser implementadas durante una iteración.

Respecto de la información contenida en la historia de usuario propiamente dicha, existen varias plantillas sugeridas pero no existe un consenso al respecto. Beck en su libro [Beck] presenta un ejemplo de ficha (*customer story* y *task card*) en la cual pueden reconocerse los siguientes contenidos: fecha, tipo de actividad (nueva, corrección, mejora), prueba funcional, número de historia, prioridad técnica y del cliente, referencia a otra historia previa, riesgo, estimación técnica, descripción, notas y una lista de seguimiento con la fecha, estado cosas por terminar y comentarios.

Roles XP

En este apartado se describen los roles de acuerdo con la propuesta original de Beck, si bien en otras fuentes de información aparecen algunas variaciones y extensiones de roles en XP. Los roles originales fueron: Programador, Cliente, Encargado de Pruebas, Encargado de Seguimiento, Entrenador, Consultor y Gestor.

- **Programador:** es el encargado de escribir las pruebas unitarias y producir el código del sistema. Es responsable de las decisiones técnicas y de construir el sistema. Debe existir una comunicación y coordinación adecuada entre los programadores y otros miembros del equipo.

- **Cliente:** escribe las historias de usuario y las pruebas funcionales para validar su implementación. Además, asigna la prioridad a las historias de usuario y decide cuáles se implementan en cada iteración buscando aportar mayor valor al negocio. El cliente es sólo uno dentro del proyecto pero puede corresponder a un interlocutor que está representando a varias personas que se verán afectadas por el sistema.

- **Encargado de pruebas (Tester):** ayuda al cliente a escribir las pruebas funcionales. Ejecuta las pruebas regularmente, difunde los resultados en el equipo y es responsable de las herramientas de soporte para pruebas.

- **Encargado de seguimiento (Tracker):** proporciona realimentación al equipo en el proceso XP. Su responsabilidad es verificar el grado de acierto entre las estimaciones realizadas y el tiempo real dedicado, comunicando los resultados para mejorar futuras estimaciones. También realiza el seguimiento del progreso de cada iteración y evalúa si los objetivos son alcanzables con las restricciones de tiempo y recursos presentes. Determina cuándo es necesario realizar algún cambio para lograr los objetivos de cada iteración.

- **Entrenador (Coach):** es responsable del proceso global. Es el líder del equipo y quien toma las decisiones importantes. Es necesario que conozca a fondo el proceso XP para proveer guías a los miembros del equipo de forma que se apliquen las prácticas XP y se siga el proceso correctamente y tiende a estar en segundo plano cuando el equipo madura.

- **Consultor:** es un miembro externo del equipo con un conocimiento específico en algún tema necesario para el proyecto. Guía al equipo para resolver un problema específico.

- **Gestor (Big boss):** es el vínculo entre clientes y programadores, ayuda a que el equipo trabaje efectivamente creando las condiciones adecuadas. Su labor esencial es de coordinación.

El Proceso de desarrollo en XP

El desarrollo es la pieza clave de todo el proceso XP. Todas las tareas tienen como objetivo realizar el desarrollo a la máxima velocidad, sin interrupciones y siempre en la dirección correcta. A grandes rasgos, el ciclo de desarrollo se puede simplificar en los siguientes pasos [Letelier_b]:

1. El cliente define el valor de negocio a implementar.

2. El programador estima el esfuerzo necesario para su implementación.

3. El cliente selecciona qué construir, de acuerdo con sus prioridades y las restricciones de tiempo.

4. El programador construye ese valor de negocio.

5. Vuelve al paso 1.

En todas las iteraciones de este ciclo tanto el cliente como el programador aprenden. No se debe presionar al programador a realizar más trabajo que el estimado, ya que se perderá calidad en el software o no se cumplirán los plazos. De la misma forma el cliente tiene la obligación de manejar el ámbito de entrega del producto, para asegurarse que el sistema tenga el mayor valor de negocio posible con cada iteración.

El ciclo de vida ideal de XP consiste de seis fases [Beck]: Exploración, Planificación de la Entrega (Release), Iteraciones, Producción, Mantenimiento y Muerte del Proyecto:

1- **Exploración**: en esta fase, el cliente plantea a grandes rasgos las historias de usuario que son de interés para la primera entrega del producto. Al mismo tiempo el equipo de desarrollo se familiariza con las herramientas, tecnologías y prácticas que se utilizarán en el proyecto ya que se prueba la tecnología y se exploran las posibilidades de la arquitectura del sistema construyendo un prototipo. Esta fase toma pocas semanas a pocos meses, dependiendo del tamaño y familiaridad que tengan los programadores con la tecnología.

2- **Planificación de la Entrega**: en esta fase el cliente establece la prioridad de cada historia de usuario, y correspondientemente, los programadores realizan una estimación del esfuerzo necesario de cada una de ellas. Se toman acuerdos sobre el contenido de la primera entrega y se determina un cronograma en conjunto con el cliente. Una entrega debería obtenerse en no más de tres meses. Esta fase dura unos pocos días.

   Las estimaciones de esfuerzo asociado a la implementación de las historias la establecen los programadores utilizando como medida el punto. Un punto, equivale a una semana ideal de programación. Las historias generalmente valen de 1 a 3 puntos. Por otra parte, el equipo de desarrollo mantiene un registro de la "velocidad" de desarrollo, establecida en puntos por iteración, basándose principalmente en la suma de puntos correspondientes a las historias de usuario que fueron terminadas en la última iteración.

3- **Iteraciones**: esta fase incluye varias iteraciones sobre el sistema antes de ser entregado. El plan de entrega está compuesto por iteraciones de no más de tres semanas. En la primera iteración se puede intentar establecer una arquitectura del sistema que pueda ser utilizada durante el resto del proyecto. Esto se logra escogiendo las historias que fuercen la creación de esta arquitectura, sin embargo, esto no siempre es posible ya que *es el cliente quien decide* qué historias se implementarán en cada iteración. Al final de la última iteración el sistema estará listo para entrar en producción.

   Los elementos que deben tomarse en cuenta durante la elaboración del plan de la iteración son historias de usuario no abordadas, velocidad del proyecto, pruebas de aceptación no superadas en la iteración anterior y tareas no terminadas en la iteración anterior. Todo el trabajo de la iteración es expresado en tareas de programación, cada una de ellas es asignada a un programador como responsable, pero llevadas a cabo por parejas de programadores.

   Será tarea del cliente retroalimentar al equipo de desarrolladores después de cada iteración con los problemas con los que se ha encontrado, mostrando sus prioridades, expresando sus sensaciones. El cliente puede cambiar de opinión sobre la marcha y a cambio debe encontrarse siempre disponible para resolver dudas del equipo de desarrollo y para detallar los requisitos especificados cuando sea necesario. Es el cliente el que tiene que escribir lo que quiere, no se permite que alguien del equipo de desarrolladores lo escriba por él.

   Antes de codificar se debe hacer un diseño. Como XP es una metodología simple, el diseño debe ser simple. Aunque en general el diseño es realizado por los propios desarrolladores en ocasiones se reúnen aquellos con más experiencia o incluso se involucra al cliente para diseñar las partes más complejas. En estas reuniones se suelen emplean las tarjetas CRC cuyo objetivo es facilitar la comunicación y documentar los resultados.

4- **Producción**: la fase de producción requiere de pruebas adicionales y revisiones de rendimiento antes de que el sistema sea trasladado al entorno del cliente. Al mismo tiempo, se deben tomar decisiones sobre la inclusión de nuevas características a la versión actual, debido a cambios durante esta fase.

   Es posible que se rebaje el tiempo que toma cada iteración, de tres a una semana. Las ideas que han sido pospuestas y las sugerencias son documentadas para su posterior implementación (por ejemplo, durante la fase de mantenimiento).

5- **Mantenimiento**: mientras la primera versión se encuentra en producción, el proyecto XP debe mantener el sistema en funcionamiento al mismo tiempo que desarrolla nuevas iteraciones. Para realizar esto se requiere de tareas de soporte para el cliente. De esta forma, la velocidad de desarrollo puede bajar después de la puesta del sistema en producción. La fase de mantenimiento puede requerir nuevo personal dentro del equipo y cambios en su estructura.

6- **Muerte del Proyecto**: es cuando el cliente no tiene más historias para ser incluidas en el sistema. Esto indica que se satisfacen todas las necesidades del cliente en aspectos como rendimiento y fiabilidad del sistema. Se genera la documentación final del sistema y no se realizan más cambios en la arquitectura. La muerte del proyecto también ocurre cuando el sistema no genera los beneficios esperados por el cliente o cuando no hay presupuesto para mantenerlo.

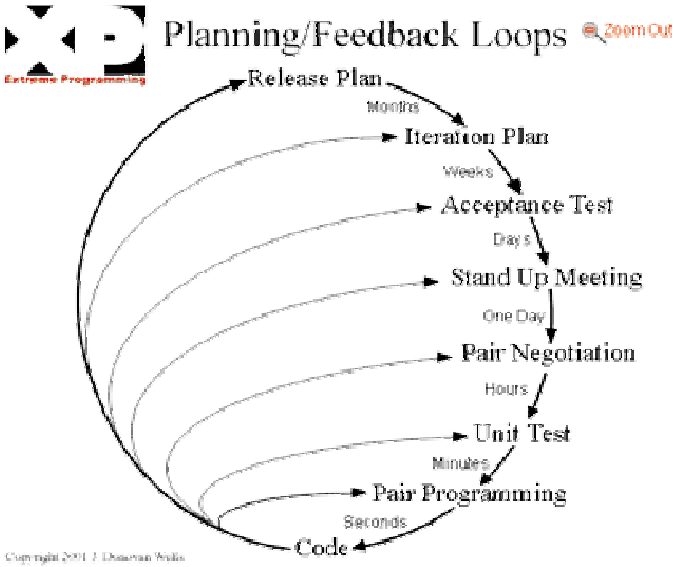


Figura 2.1. Proceso XP – Ciclos de planificación y retroalimentación [XP_sitio]

Prácticas XP

La principal suposición que se realiza en XP es la posibilidad de disminuir la mítica curva exponencial del costo del cambio a lo largo del proyecto, lo suficiente para que el diseño evolutivo funcione. XP apuesta por un crecimiento lento del costo del cambio y con un comportamiento asintótico. Esto se consigue gracias a las tecnologías disponibles para ayudar en el desarrollo de software y a la aplicación disciplinada de ciertas prácticas [Letelier_b].

XP especifica ciertas prácticas concretas de programación que deben llevarse a cabo al implementar este modelo. Las mismas tienen su origen en prácticas bien conocidas en la ingeniería del software, tal como se mencionó previamente. Las prácticas de XP facilitan el desarrollo de una aplicación y permiten a los desarrolladores obtener software de calidad. Las doce prácticas que caracterizan a la metodología son:

- **Juego de la Planificación:** es un espacio frecuente de comunicación entre el cliente y los programadores. El juego de la planificación define el límite o el punto de interacción entre clientes y desarrolladores, especificando las responsabilidades de ambas partes. El equipo técnico realiza una estimación del esfuerzo requerido para la implementación de las historias de usuario y los clientes deciden sobre el ámbito y tiempo de las entregas y de cada iteración. La duración de cada iteración se suele estimar en 2-3 semanas de trabajo.

- **Entregas pequeñas**: la idea es producir rápidamente versiones del sistema que sean operativas, aunque obviamente no cuenten con toda la funcionalidad pretendida para el sistema pero si que constituyan un resultado de valor para el negocio. Kent Beck enuncia en su libro *Extreme Programming Explained: Embrace Change* [Beck] *"cada entrega debe ser tan pequeña como sea posible, conteniendo los requerimientos con mayor valor de negocio".* Las entregas pequeñas proveen un sentido de logro o éxito, que a menudo falta en proyectos largos, y retroalimentación frecuente y relevante. Una entrega no debería tardar más de 3 meses.

- **Metáfora:** en XP no se enfatiza la definición temprana de una arquitectura estable para el sistema. Dicha arquitectura se asume evolutiva y los posibles inconvenientes que se generarían por no contar con ella explícitamente en el comienzo del proyecto se solventan con la existencia de una metáfora. El sistema es definido mediante una metáfora o un conjunto de metáforas compartidas por el cliente y el equipo de desarrollo. Una metáfora es una historia compartida que describe cómo debería funcionar el sistema. La práctica de la

metáfora consiste en formar un conjunto de nombres que actúen como vocabulario para hablar sobre el dominio del problema. Este conjunto de nombres ayuda a la nomenclatura de clases y métodos del sistema [Letelier_b]. El principal objetivo de la metáfora es mejorar la comunicación entre todos los integrantes del equipo al crear una visión global y común del sistema que se pretende desarrollar.

- **Diseño simple:** El diseño simple tiene dos partes: (1) diseñar para la funcionalidad que se definió, no para potenciales funcionalidades futuras y (2) crear el mejor diseño y mas simple que puede proveer esa funcionalidad. Se debe diseñar la solución más simple que pueda funcionar y ser implementada en un momento determinado del proyecto. La complejidad innecesaria y el código extra debe ser removido inmediatamente. Según Kent Beck, en cualquier momento el diseño adecuado para el software es aquel que supera con éxito todas las pruebas, no tiene lógica duplicada y tiene el menor número posible de clases y métodos.

- **Pruebas:** XP usa dos tipos de pruebas: de unidad y funcionales (o de aceptación). La producción de código está dirigida por las pruebas unitarias. Las pruebas unitarias son establecidas antes de escribir el código y son ejecutadas constantemente ante cada modificación del sistema. Por lo tanto, la práctica de pruebas de unidad involucra desarrollar las pruebas para la característica deseada antes de escribir el código. Una vez que se escribe el código, éste está sujeto a las suites de prueba con retroalimentación inmediata. Las pruebas unitarias se llevan a cabo por los desarrolladores cada vez que se añade código, y además sobre cada historia por separado antes de ser integrado al software. Los clientes escriben las pruebas funcionales para cada historia de usuario que deba validarse y son las que él mismo ejecuta para validar el software, asistidos por el tester (Ver **Figura 2.2**). En este contexto de desarrollo evolutivo y de énfasis en pruebas constantes, es crucial la automatización para apoyar esta actividad [Letelier_b].

Caso de Prueba de Aceptación	
Código:	Historia de Usuario (Nro. y Nombre):
Nombre:	
Descripción:	
Condiciones de Ejecución:	
Entrada / Pasos de ejecución:	
Resultado Esperado:	
Evaluación de la Prueba:	

Figura 2.2. Modelo propuesto para una prueba de aceptación [Canós]

- **Refactorización (Refactoring)**: la refactorización es una actividad constante de re-estructuración del código con el objetivo de remover duplicación de código, mejorar su legibilidad, simplificarlo y hacerlo más flexible para facilitar los posteriores cambios. La

refactorización mejora la estructura interna del código sin alterar su comportamiento externo [Letelier_b]. Tiene su justificación principal en que el código no sólo tiene que funcionar, también debe ser simple. Esto hace que, a la larga, refactorizar ahorre mucho tiempo y suponga un incremento de calidad. Por cierto, tal es el énfasis que se pone en la refactorización que, inclusive se deben refactorizar las pruebas unitarias. Según Highsmith, la refactorización distingue a XP de los demás enfoques – el continuo rediseño de software para mejorar su respuesta al cambio. Debería pensarse a XP como un rediseño continuo. En tiempos de cambios rápidos y constantes se necesita prestar más atención en refactorización.

- **Programación en parejas:** *"Programación de a pares es un diálogo entre dos personas tratando de programar simultáneamente y entender cómo programar mejor"* [Beck]**.** Toda la producción de código debe realizarse con trabajo en parejas de programadores frente a una computadora, con un sólo *mouse* y un sólo teclado. Cada miembro de la pareja juega su papel: uno codifica en la computadora y piensa la mejor manera de hacerlo, el otro piensa más estratégicamente. Tener dos personas sentadas frente a una misma terminal, una ingresando código o casos de prueba y otra revisando y pensando, crea un intercambio dinámico y continuo.

- **Propiedad colectiva del código:** Brinda a cualquier programador en el proyecto el "permiso" para cambiar cualquier parte del código en cualquier momento. Esta práctica motiva a todos a contribuir con nuevas ideas en todos los segmentos del sistema, evitando a la vez que algún programador sea imprescindible para realizar cambios en alguna porción de código. Por otro lado, si bien nadie conoce cada parte igual de bien, todos conocen algo sobre cada parte, esto ayuda a preparar para la sustitución no traumática de cada miembro del equipo.

- **Integración continua**: cada pieza de código se integra en el sistema una vez que esta lista. Así, el sistema puede llegar a ser integrado y construido varias veces en un mismo día. El código se debe integrar como mínimo una vez al día, y realizar las pruebas sobre la totalidad del sistema. Una pareja de programadores se encargará de integrar todo el código en una máquina y realizar todas las pruebas hasta que éstas funcionen al 100%. para que el nuevo código sea incorporado definitivamente. La integración continua a menudo reduce la fragmentación de los esfuerzos de los desarrolladores por falta de comunicación sobre lo que puede ser reutilizado o compartido. El equipo de desarrollo está más preparado para modificar el código cuando sea necesario, debido a la confianza en la identificación y corrección de los errores de integración [Letelier_b].

- **40 horas por semana**: se debe trabajar un máximo de 40 horas por semana. Las horas extras son síntoma de serios problemas en el proyecto. Si esto ocurre, probablemente está ocurriendo un problema que debe corregirse. El trabajo extra desmotiva al equipo. Los proyectos que requieren trabajo extra para intentar cumplir con los plazos suelen al final ser entregados con retraso. En lugar de esto se puede realizar el juego de la planificación para cambiar el ámbito del proyecto o la fecha de entrega.

- **Cliente in-situ**: el cliente tiene que estar presente y disponible todo el tiempo para el equipo. Gran parte del éxito del proyecto XP se debe a que es el cliente quien conduce constantemente el trabajo hacia lo que aportará mayor valor de negocio y los programadores pueden resolver de manera inmediata cualquier duda asociada. La comunicación oral es más efectiva que la escrita, ya que esta última toma mucho tiempo en generarse y puede tener más riesgo de ser mal interpretada. En [Jeffries2001] Jeffries indica que se debe pagar un precio por perder la oportunidad de un cliente con alta disponibilidad.

- **Estándares de programación:** XP enfatiza la comunicación de los programadores a través del código, con lo cual es indispensable que se sigan ciertos estándares de programación (del equipo, de la organización u otros estándares reconocidos para los lenguajes de programación utilizados). Los estándares de programación mantienen el código legible para los miembros del equipo, facilitando los cambios. Son un apoyo a la metáfora ya que consiste en una correcta elección de los nombres que se escojan durante

el proyecto para las clases, métodos, variable, procesos, etc. Nombres bien puestos implican claridad, reusabilidad y simplicidad.

El mayor beneficio de las prácticas se consigue con su aplicación conjunta y equilibrada puesto que se apoyan unas en otras. Las prácticas de XP son realmente un sistema de prácticas diseñadas para interactuar, hacer contrapeso y reforzar cada una, por eso a su vez elegir algunas para usar y descartar otras puede ser delicado. [Highsmith]

Conclusiones

De todas las metodologías ágiles, ésta es la que ha recibido más atención [Fowler2003]. Los defensores de XP son cuidadosos en expresar cuándo consideran que es apropiado XP y cuándo no lo es. Defensores como Kent Beck y Ron Jeffries pueden imaginar que XP tiene una aplicabilidad más amplia, generalmente son cautelosos sobre sus afirmaciones. Por ejemplo ambos son claros de la aplicabilidad de XP en equipos pequeños (menos de 10 personas) en un mismo lugar, una situación donde ellos han tenido experiencia. Ellos no tratan de convencer a las personas que las prácticas funcionarán en equipos de 200. [Higsmith]

Sin embargo, como se resalta en [Letelier_b], hay que tener presente una serie de inconvenientes y restricciones para su aplicación, tales como: están dirigidas a equipos pequeños o medianos (Beck sugiere que el tamaño de los equipos se limite de 3 a 20 miembros como máximo, otros dicen no más de 10), el entorno físico debe ser un ambiente que permita la comunicación y colaboración entre todos los miembros del equipo durante todo el tiempo, cualquier resistencia del cliente o del equipo de desarrollo hacia las prácticas y principios puede llevar el proceso al fracaso (el clima de trabajo, la colaboración y la relación contractual son claves), el uso de tecnologías que no tengan un ciclo rápido de realimentación o que no soporten fácilmente el cambio, etc.

Aunque en la actualidad ya existen libros asociados a cada una de las metodologías ágiles existentes y también abundante información en internet, es XP la metodología que resalta por contar con la mayor cantidad de información disponible y es con diferencia la más popular.

3. SCRUM

Introducción

Scrum no es una metodología es un camino" Ken Schwaber, Co, Nov 2005. [Higsmith]

El concepto de Scrum tiene su origen a principios de los 90. Está basado en un estudio de gestión de equipos de 1986 desarrollado por Hirotaka Takeuchi e Ikujiro Nonaka llamado The New New Product Developement Game. Era un estudio sobre los nuevos procesos de desarrollo utilizados en productos exitosos en Japón y los Estados Unidos (cámaras de fotos de Canon, fotocopiadoras de Xerox, automóviles de Honda, ordenadores de HP y otros). En este estudio se comparaba la forma de trabajo de equipos de desarrollo, altamente productivos y multidisciplinares, con la colaboración entre los jugadores de Rugby y su formación de Scrum. [Sutherland2011].

Scrum es un marco de referencia para desarrollo ágil de productos software. Esta metodología, si bien ha sido aplicada principalmente a proyectos de software, un número de proyectos que no están relacionados con el software han sido administrados con Scrum – los principios son aplicables a cualquier proyecto [Highsmith]. Actualmente SCRUM es uno de los métodos ágiles que está creciendo más y ya en 2011 lo usa el 75% de equipos ágiles alrededor del mundo. [SutherlandSitio] [Sutherland2011]. La Scrum Alliance [ScrumAlliance.org] es la organización sin ánimo de lucro que se encarga de difundir Scrum.

Está siendo utilizado por grandes y pequeñas compañías, incluyendo Yahoo, Microsoft, Google, Lockheed Martin, Motorola, SAP, Cisco, GE Medical, Capital One y la Reserva Federal de los Estados Unidos [Deemer]. Muchos equipos que están usando Scrum reportan mejoras importantes, y en algunos casos transformaciones completas, tanto en productividad como en la moral. Scrum es simple, poderoso y cimentado en el sentido común. [Highsmith]

"El corazón del enfoque Scrum es la creencia que la mayoría de los desarrollos de sistemas tienen las bases filosóficas erróneas ". Ken Schwaber afirma que el desarrollo de sistemas no es un "proceso definido" como asumen las metodologías rigurosas, sino un "proceso empírico". [Higsmith]. Un proceso empírico requiere un estilo completamente diferente de administración. Los procesos empíricos, a diferencia de los rigurosos, no pueden repetirse consistentemente, requieren por lo tanto monitoreo y adaptación constante. "*Los desarrolladores y administradores de proyectos son forzados a vivir una mentira – deben pretender que pueden planificar, predecir y entregar*". Eso dice Ken Schwaber.

Scrum comienza con la premisa que el mundo es complicado y por lo tanto "*no se puede predecir o planear definitivamente lo que se entregará, cuando se entregará y que calidad y costo tendrá"* dice Ken Schwaber. No se basa en seguir un plan sino en la adaptación continua a las circunstancias de la evolución de proyecto. Mientras XP tiene un sabor definido a programación (programación de a pares, estándares de codificación, refactorización), Scrum tienen un énfasis en la administración de proyecto.

Principales elementos

Estos son los principales elementos de Scrum, si bien se utilizan otros que se irán describiendo más adelante.

- **Product Backlog:** El Product Backlog es una lista priorizada de funcionalidades técnicas y de negocio. Estas funcionalidades son requisitos a muy alto nivel de lo que debe hacer la aplicación, donde se listan características, funciones, tecnología, mejoras, bugs, etc. que serán aplicadas al producto. El Product Backlog es el punto de inicio.
- **Sprint Backlog:** Lista de tareas de un sprint. El Sprint Backlog identifica y define el trabajo a ser alcanzado por el equipo de desarrollo durante un Sprint. A un nivel el Sprint Backlog identifica las características mientras que a otro nivel, identifica las tareas requeridas para implementar esas características.

- **Incremento:** Parte de un sistema desarrollado en un Sprint. Este producto desarrollado es potencialmente entregable al final de cada Sprint, implica que todo está completamente terminado en cada Sprint; y se podría realmente empaquetar o desplegar inmediatamente después de la Revisión del Sprint con mínimas tareas, si bien a veces se necesitan ciertos trabajos de acabado tales como pruebas o documentación.

Roles

En Scrum se definen varios roles, estos están divididos en dos grupos. Por un lado figuran los que están comprometidos con el proyecto y el proceso Scrum. Por otro lado, aquellos que en realidad no forman parte del proceso Scrum, pero que deben tenerse en cuenta y cuya participación es importante. [CaraballoMaestre]

Roles comprometidos con el Proyecto

- **Product Owner**: Representa la voz del cliente. Escribe *historias de usuario*, las prioriza, y las coloca en el *Product Backlog*.
- **Scrum Master**: Protege al equipo de distracciones y de otros elementos externos y lo mantiene enfocado. Elimina obstáculos que alejen al grupo de la consecución de objetivos del sprint. No es el líder del grupo, ya que el grupo se autogestiona. Dirige los scrums diarios. Realiza el seguimiento del avance.
- **Equipo**: Conformado por no más de 8 personas (si hay más se organizan varios equipos que trabajan sobre el mismo product backlog). Tiene la responsabilidad de entregar el producto. Son autónomos y auto-organizados. Deben entregar un conjunto de ítems del Backlog al final del Sprint.

Roles involucrados en el proceso

- **Usuarios**: Son los destinatarios finales del producto.
- **Stakeholders o Interesados (Clientes, Proveedores)**: Se refiere a la gente que hace posible el proyecto y para quienes el proyecto producirá el beneficio acordado que lo justifica.
- **Managers**: Son aquellos que establecen el ambiente para el desarrollo del producto.

El proceso Scrum

Scrum define un marco de administración de proyecto donde las actividades de desarrollo – recolección de requerimientos, diseño, programación – tienen lugar. El período de desarrollo es de una iteración de 30 días llamada Sprint, si bien podría trabajarse con sprints de menor duración. El marco de trabajo de Scrum tiene tres componentes o cuatro según como se los mire.

- pre-sprint: planificación del sprint
- sprint: ciclo de trabajo
- post-sprint: revisión y retrospectiva del sprint (pueden considerarse como dos componentes diferentes posteriores al sprint)

El punto focal es el sprint, donde se desarrolla software que funcione. (Ver **Figura 3.1**.)

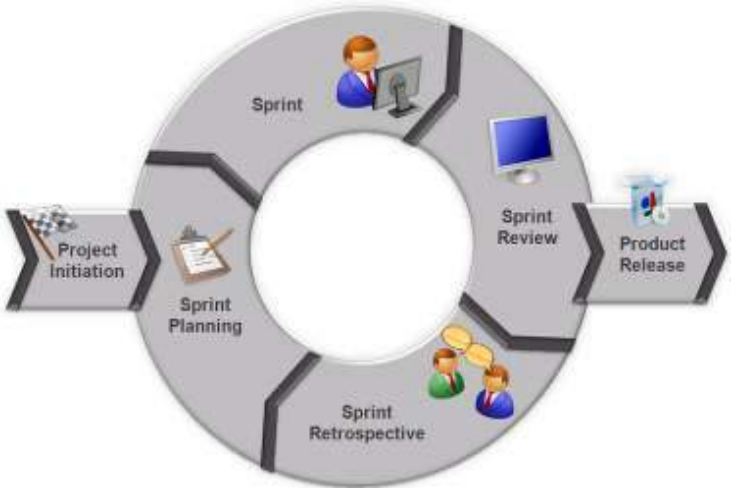


Figura.3.1. Ciclo de vida de un proyecto Scrum [Parasuraman]

Planificación pre-sprint

Antes de iniciar esta actividad el Product Owner ya tiene que haber creado y priorizado el Product Backlog., posiblemente en una reunión previa para no extender más la duración de esta reunión. En esta reunión están presentes el equipo de desarrollo, el administrador y el Product Owner. Aquí, el Product Owner define las características a incorporar en el siguiente Sprint y el equipo define las tareas necesarias para entregar esas características. El equipo determina la lista de tareas y estima cuanto puede cumplirse en el Sprint. El equipo y el Product Owner iteran en el proceso hasta que las características planeadas están acordes con los recursos disponibles para el Sprint, quedando definido entonces el Sprint Backlog.

Una pieza final del proceso de planificación es desarrollar un Objetivo del Sprint, un propósito de negocio para el Sprint. El objetivo del Sprint puede alcanzarse aún cuando algunas características o tareas fueron demoradas o dejadas de lado. "*La razón de un Objetivo de Sprint es dar al equipo cierta flexibilidad respecto a la funcionalidad"* tal como Ken Schwaber y Mike Beedle lo enuncian. (Agile Software Development with Scrum – Schwaber & Beedle - 2002). El objetivo de un Sprint recuerda constantemente al equipo cuales son las tareas detalladas que deben alcanzarse. Sin este objetivo, el equipo puede enfocarse demasiado en las tareas y perder la razón por la que se están realizando. Además, mantener el objetivo en mente anima al equipo a adaptarse a condiciones que pudieran surgir durante el transcurso del Sprint.

Figura.3.2. Pre–sprint o Planificación del sprint [PalacioBañeres]

Podría decirse que la Planificación del sprint consiste de dos sub-reuniones donde se logra

- Determinar las funcionalidades y el objetivo
- Desglosarlas en tareas y estimar el esfuerzo

Sprint

Es un ciclo de producción dentro de un desarrollo iterativo e incremental, la duración estándar es de 30 días (si bien algunos aducen que el sprint puede variar de una semana a un mes). Durante un Sprint los miembros del equipo eligen tareas a realizar, cada uno se esfuerza para alcanzar el Objetivo del Sprint y todos participan de la reunión diaria de Scrum. No hay planes elaborados durante un Sprint – se espera que el equipo use sus talentos para entregar resultados.

Algo que hace de Scrum un poderoso enfoque es que reduce la fragmentación de tiempo y el cambio constante de las prioridades al "bloquear" las características durante el Sprint. Excepto en situaciones extremas, las prioridades no se cambian durante el sprint por nadie, ni siquiera por el Product Owner, ni el administrador. Al final de Sprint, el Product Owner podría elegir descartar todas las características desarrolladas o reorientar el proyecto, pero durante un sprint las prioridades se mantienen constantes. Esta es la parte de "control" del "caos controlado". Esto es crítico, porque en un entorno de constante cambio debe haber un cierto punto de estabilidad. Es necesaria cierta estabilidad para que el equipo pueda sentir que pueden lograr algo. Si aparece una circunstancia externa que cambie significativamente las prioridades, y signifique por lo tanto que el equipo estaría perdiendo el tiempo si continúa trabajando, el Product Owner puede terminar el Sprint, significando que el equipo deja todo el trabajo que estuvieran haciendo y comienza una reunión de planificación de un nuevo Sprint. . La cancelación de un sprint es un costo, lo que sirve para desalentar al Product Owner a realizarlo salvo circunstancias extremas.

Hay una influencia positiva poderosa que proviene de proteger al equipo de cambios de objetivos durante el sprint. Primero, el equipo trabaja sabiendo con absoluta certeza que el compromiso asumido no cambiará, lo que refuerza el foco del equipo en asegurar cumplir con ese objetivo. Segundo, ayuda a que el Product Owner realmente piense a conciencia los ítems que prioriza en el Product Backlog. Saber que el compromiso es para toda la duración el sprint hace que el Product Owner sea más diligente al decidir lo que va a solicitar desde el principio.

SCRUM BOARD: TABLERO DE TRABAJO

Durante el Sprint es conveniente organizar un Tablero de Trabajo con las Historias ordenadas por su estado, en su forma más básica, el tablero tiene 3 columna: *Tareas a realizar, Tareas en Ejecución, Tareas Finalizadas*, pero puede (y es recomendable) agregar estados intermedios como por ejemplo *análisis, codificación y pruebas.*

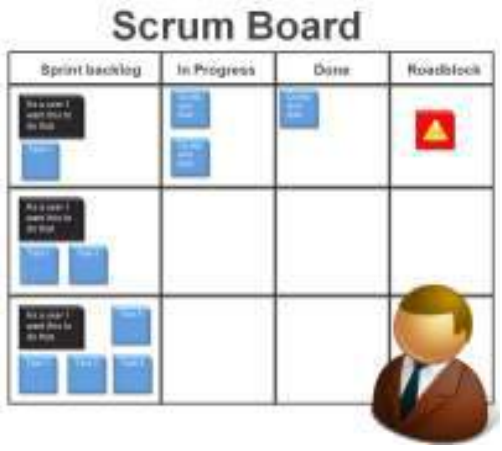


Figura 3.3: Ejemplo de ScrumBoard [Parasuraman]

Cada tarea está representada por una tarjeta que pueda moverse por el tablero (post-it o papeles con chinches). Cada tarjeta de tarea debe contener al menos un identificador (numero único), nombre de la tarea, breve descripción y responsable (si es aplicable). En algunos casos también se usan carriles horizontales para identificar el proyecto o el responsable de la tarea.

Al inicio del sprint todas las tareas están en la primera columna de la izquierda, es decir en estado de planificadas. Al finalizar el sprint todas deberían estar en la última columna es decir completadas.

Reunión diaria (Scrum Diario)

Dentro de un sprint, la reunión diaria (Daily Standup Scrum) da energías al propio Sprint. Existen principios muy viejos que enuncian "aumentar la comunicación" o "involucrar al cliente". Los administradores no pueden controlar las comunicaciones o el grado de participación del cliente pero puede influenciar y crear un entorno que fomente trabajar juntos. [Highsmith]

Dado que los procesos empíricos se definen por su incertidumbre y cambio, es importante que se ejecuten chequeos diarios. La reunión diaria de Scrum permite al equipo monitorizar el estado, enfocarse en el trabajo a realizar y detectar problemas y cuestiones. Las reuniones se realizan de pie, frente al Scrum Board y a la misma hora. En promedio debe durar 15 minutos y debe ser moderada por el Scrum Master. Todos los miembros del equipo de desarrollo deben participar. También los administradores deben asistir para tener un seguimiento del estado del proyecto pero no para participar. Las reuniones se usan para detectar ciertas cuestiones y obstáculos pero no para proponer soluciones ya que eso implicaría una duración mucho mayor. Posteriormente el Scrum Master tratará de solucionar las cuestiones que generan obstáculos en el equipo. Cada participante debe comentar qué hizo, que hará y qué obstáculos tuvo en el camino para poder realizar su trabajo. Cada miembro se debe dirigir al Equipo, no al Scrum Master ni a ningún otro miembro en concreto.

MONITOREO DEL PROGRESO

Un Sprint terminando tardíamente es un indicador que el proyecto no está marchando correctamente. Sin embargo sería muy tarde esperar que un Sprint termine para ver que no todas las tareas se completaron.

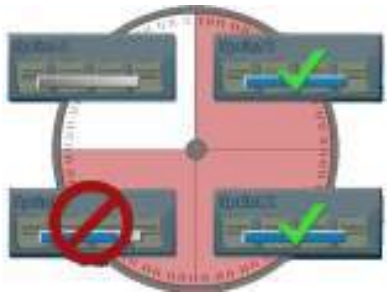


Figura 3.4 Progreso en un sprint [CockburnSitio]

Jeff Sutherland ha refinado el monitoreo de Scrum con un razonable aumento diario de actividad administrativa – un minuto por día para los desarrolladores y 10 minutos por día para del administrador de proyectos. Después de la reunión de scrum diario, cada miembro del equipo actualiza la cantidad de tiempo que le falta para completar cada tarea que asumieron realizar durante el sprint en el Sprint Backlog (ver **Figura 3.5**). Posteriormente el administrador del proyecto suma las horas restantes de trabajo que figuran en el Sprint Backlog y las refleja en una herramienta gráfica (ver **Figura 3.6**). Jeff Sutherland propone utilizar una herramienta simple pero a su vez poderosa para monitorear el progreso del proyecto: el Burndown chart del Sprint Backlog. Lo importante es que se muestre al equipo progreso hacia su objetivo, no en términos de cuanto hicieron en el pasado (un hecho irrelevante en término de progreso), sino en término de cuanto trabajo queda en el futuro –lo que separa al equipo de su objetivo.

El Burndown chart muestra en el eje horizontal los días y el trabajo restante expresado en horas en el eje vertical; al final de los 30 días, el trabajo faltante debería ser cero. Al día cero, el trabajo restante debe ser igual a la cantidad de horas estimadas para el Sprint. Cada día los desarrolladores registran los días (horas) invertidos en una tarea y su porcentaje de completitud usando alguna herramienta automática de backlog que calcula el trabajo completado y el trabajo restante. Al mirar el gráfico de Sprint Backlog, el líder de proyecto tiene

retroalimentación diaria del progreso (o falta de progreso) y cualquier estimación que resultó ser inadecuada.

				Nuevo esfuerzo estimado Lo que queda al final del día...					
Elemento de la Pila de Producto	**Tarea del Sprint**	**Voluntario**	**Esfuerzo estimado inicial**	**1**	**2**	**3**	**4**	**5**	**6**
Como comprador quiero poner un libro en el carrito de la compra	modificar base de datos	Sanjay	5	4	3	0	0	0	
	crear página web (interfaz de usuario)	Jing	3	3	3	2	0	0	
	crear página web (lógica Javascript)	Tracy & Sam	2	2	2	2	1	0	
	escribir pruebas de aceptación automáticas	Sarah	5	5	5	5	5	0	
	actualizar la página de ayuda del comprador	Sanjay & Jing	3	3	3	3	3	0	
	. . .								
Mejorar el rendimiento de procesamiento de transacciones	juntar el código DCP y completar los test del nivel de capa		5	5	5	5	5	5	
	completar la orden de máquina para pRank		3	3	8	8	8	8	
	Cambiar el DCP y el lector para usar el API http de pRank		5	5	5	5	5	5	
			...						
		Total (personas-hora)		50	49	48	44	43	34

Figura 3.5. Actualizaciones diarias de Trabajo Restante en la Pila del Sprint [DeemerEs]

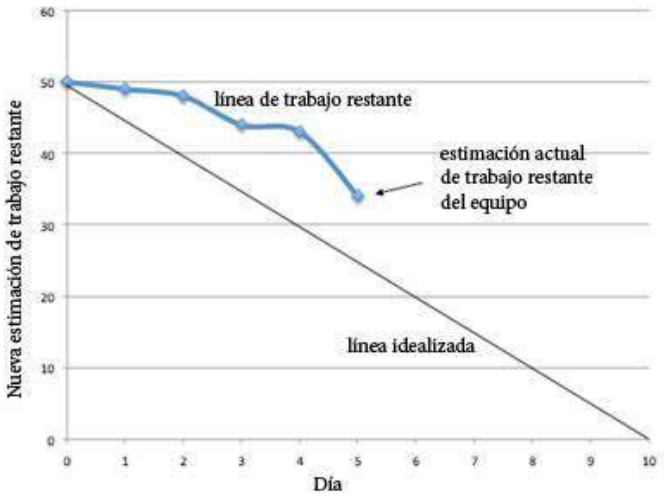


Figura 3.6. Gráfica de Trabajo Restante del Sprint [DeemerEs]

Idealmente el trabajo remanente siempre va disminuyendo, pero es normal (aunque no debe suceder a menudo) que el trabajo remanente aumente debido a tareas que consumen más horas de las planeadas.

Mike Beedle y Ken Schwaber señalan que "*esto <el monitoreo> no debe confundirse con reporte de tiempo que no es parte de Scrum Los equipos se miden por alcanzar objetivos no por cuantas horas toman para alcanzarlo" (Schwaber & Beedle 2002).* Los administradores están mirando la tendencia general en el gráfico del Sprint BackLog y no estimaciones y completitud de tareas de micro-administración.

Reunión Post Sprint – Revisión del Sprint

Al final de la iteración del Sprint, se realiza una reunión post-sprint para revisar el progreso, mostrar al cliente la demo de lo que se estuvo construyendo y revisar el proyecto desde una perspectiva técnica. Es una presentación del incremento desarrollado y cualquiera de los presentes es libre de realizar preguntas y hacer comentarios. Asisten el Product Owner, el Scrum Master, el Equipo y personas que podrían estar involucradas en el proyecto. El Equipo

es quién muestra los avances realizados en el Sprint. Aquí se obtiene el feedback del cliente para el siguiente sprint y se define la fecha para la reunión del siguiente sprint.

El análisis de Retrospectiva podría realizarse fuera de la reunión de Revisión realizando una reunión informal del equipo al terminar el Sprint donde se plantean los inconvenientes tenidos durante el desarrollo del Sprint, así como las mejoras posibles para trabajos futuros. Esta es una práctica que muchos equipos saltean, y esto es desafortunado ya que es una de las herramientas más importantes para hacer un Scrum exitoso. Es una oportunidad del equipo de discutir lo que está funcionando y lo que no y consensuar los cambios a probar.

Finalizando el sprint

Uno de los principios fundamentales de Scrum es que nunca se prolonga la duración del Sprint –termina en la fecha asignada aunque el equipo no haya terminado el trabajo comprometido. Si el equipo no completó el objetivo del Sprint, deben reconocer al final del Sprint que no cumplieron lo que se habían comprometido a alcanzar. Esto crea un ciclo de retroalimentación visible y el equipo se ve forzado a mejorar en la estimación de lo que es capaz de lograr en un Sprint y entonces entregarlo sin fallar.

Si bien la duración sugerida por los autores es de 30 días, a los equipos se les recomienda que escojan una duración para el Sprint (por ejemplo dos semanas o 30 días) y que no la cambien. Una duración consistente ayuda al equipo a saber cuánto pueden hacer, lo que les ayuda en la estimación y en la planificación de la entrega a más largo plazo. También ayuda a que el equipo adquiera un ritmo de trabajo; en Scrum a esto se le llama el “pulso” del equipo.

Comenzando el próximo Sprint

Después de la Revisión del Sprint, el Dueño de Producto puede actualizar el Product Backlog con nuevas ideas. Puede presentar todas las nuevas prioridades que hayan surgido durante el sprint e incorporarlas al Product Backlog. También puede modificar algunos ítems del Product Backlog, reordenarlos o eliminarlos. En este punto, el Product Owner y el equipo están listos para empezar otro ciclo de Sprint. No hay tiempo de descanso entre Sprints –los equipos normalmente van de la Retrospectiva del Sprint una tarde a la Planificación del próximo Sprint la mañana siguiente (o después del fin de semana).

Uno de los principios de desarrollo ágil es “ritmo sostenible”, y solo trabajando dentro de las horas delimitadas por el horario laboral a un ritmo razonable permite que los equipos continúen este ciclo indefinidamente.

Conclusiones

Esta metodología, SCRUM, se centra en la gestión del proyecto y puede aplicarse a otros proyectos que no tienen que ver con el desarrollo de software. Busca el trabajo cooperativo de equipos multidisciplinares altamente productivos. Define sprints de duración fija y determina una serie de reuniones a realizar a lo largo del proyecto. Como resultado de cada sprint se entrega un incremento. Plantea realizar un monitoreo del proceso y una adaptación constante. Actualmente es uno de los métodos ágiles que más está creciendo. En la **Figura 3.7** se puede tener una visión general de esta metodología.

En el primer Scrum se usaron todas las prácticas ingenieriles de XP y sentaron las bases de la ingeniería actual. La mayoría de los equipos con alta performance usan conjuntamente SCRUM y XP. Es difícil obtener un Scrum con velocidad extrema sin las prácticas ingenieriles de XP. Por otro lado no se puede escalar XP sin Scrum. [SutherlandSitio]

.

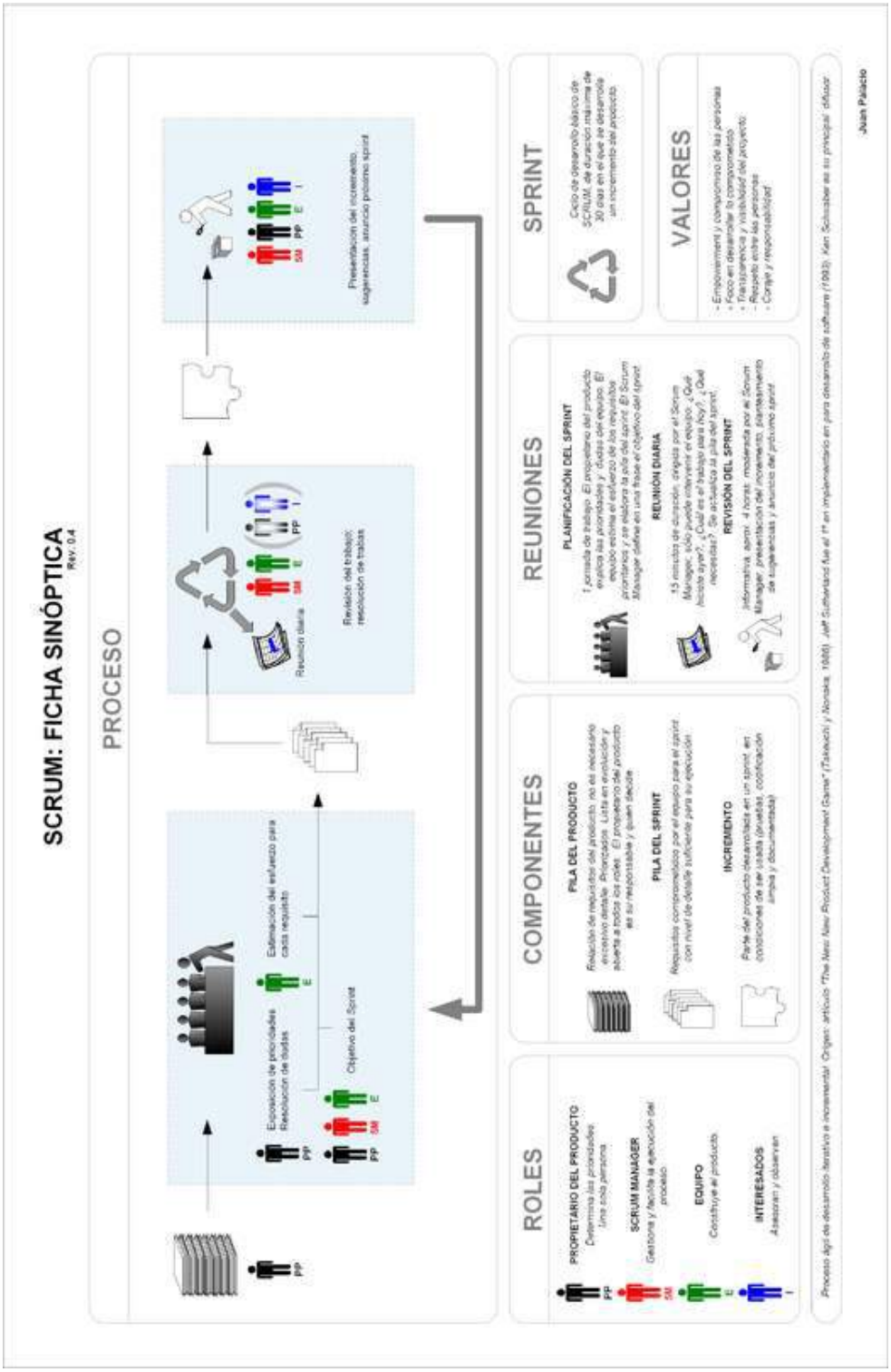


Figura 3.7: Scrum – Ficha sinóptica [PalacioBañeres]

4. KANBAN

Introducción

Kanban se considera el enfoque de Lean al desarrollo de software [crispSitio], por lo tanto antes de introducir los conceptos de Kanban propiamente dichos se presentará brevemente Lean

Lean

Lean está definido como un C*onjunto de Principios*, no es un proceso que pueda ser directamente adaptado a distintos ambientes. Lean Development fue iniciado por Bob Charette y se inspira en el éxito del proceso industrial Lean Manufacturing, bien conocido en la producción automotriz y en manufactura desde la década de 1980. Este proceso tiene como precepto la eliminación de residuos a través de la mejora constante, haciendo que el producto fluya a instancias del cliente para hacerlo lo más perfecto posible.

Las practicas Lean fueron implementadas y perfeccionadas por *Toyota Production System* hace más de 40 años y luego fueron copiadas por otras empresas automotrices con mucho éxito. También fueron llevadas a otras parte de la organización que no tienen mucho que ver con la producción, es allí donde estas prácticas debieron ser adaptadas.

La diferencia entre la producción y otras áreas es que en estas últimas no es posible combatir la variabilidad, sistematizar y estandarizar el trabajo como se hace en la producción. Sin embargo la adaptación de las practicas Lean en otras áreas fue tan exitosa o más que en la producción. [Modezki2011]

Los principios que definen un sistema Lean son los siguientes [DeSeta][Rubio]:

- **Calidad perfecta a la primera** - búsqueda de cero defectos, detección y solución de los problemas en su origen. Todo lo que se hace, debe de intentar hacerse bien, no rápido, ya que cuesta más tiempo hacer algo rápido y tener que arreglarlo después, que hacerlo bien desde el principio.
- **Minimización del desperdicio** - eliminación de todas las actividades que no son de valor añadido y la optimización del uso de los recursos escasos (capital, gente y espacio). Hacer lo justo y necesario (y hacerlo bien, como se decía antes) y no entretenerse en otras tareas secundarias o no necesarias (principio YAGNI).
- **Mejora continua** - reducción de costos, mejora de la calidad, aumento de la productividad y compartir la información. Ir mejorando continuamente los desarrollos, según los objetivos a lograr y alcanzar.
- **Procesos "*pull*"** - los productos deben ser solicitados por el cliente final, no empujados al mercado desde la fábrica hacia el cliente.
- **Flexibilidad** - producir rápidamente diferentes mezclas de gran variedad de productos, sin sacrificar la eficiencia debido a volúmenes menores de producción. Lo siguiente a realizar se decide del *backlog* pendiente, con lo que las tareas entrantes se pueden priorizar y condicionar según las necesidades puntuales.
- **Construcción y mantenimiento de una relación a largo plazo** con los proveedores tomando acuerdos para compartir el riesgo, los costos y la información.

Lean es básicamente todo lo concerniente a obtener las cosas correctas en el lugar correcto, en el momento correcto, en la cantidad correcta, minimizando el desperdicio, siendo flexible y estando abierto al cambio. [Ferreiras2010]

Las practicas Lean tienen por objetivo generar el máximo valor lo más rápidamente posible, en forma sostenida y sustentable.

"Todo lo que hacemos es mirar la línea de tiempo y reducimos el tiempo reduciendo todo aquel desperdicio que no agrega valor" Taiichi Ohno, creador de Lean y Just In Time

Kanban

El sistema Kanban utilizado en Toyota es la inspiración detrás de lo que se conoce como *Kanban para Ingeniería de Software*. Kanban implementa completamente los principios Lean. Mary y Tom Poppendieck publican el primer libro, por el año 2003, sobre la aplicación de los principios de Lean al desarrollo de software llamado *Lean Software Development* [Poppendieck2003], donde se refiere a Kanban. Aunque el contexto en que es usado no es técnicamente correcto (porque se refiere al tablero, pero no habla de los límites) el término *Tablero Kanban* se deslizó dentro del vocabulario Ágil. [Joyce2009].

En los últimos años, los sistemas Kanban se han vuelto un tema de candente de creciente interés en la comunidad ágil, gracias particularmente a la excelente introducción que Tom y Mary Poppendieck realizaron sobre los principios propuestos por Lean aplicados al desarrollo de software [Poppendieck2003]. Dentro de esta línea se pueden destacar tres personas que han estado trabajando con sistemas Kanban dentro del marco de desarrollo ágil: David Anderson, Arlo Belshee y Kenji Hiranabe. [shoreSitio]

Kanban se basa en una idea muy simple: el trabajo en curso (Work In Progress, WIP) debería limitarse, y sólo se debería empezar con algo nuevo cuando un bloque de trabajo anterior haya sido entregado o ha pasado a otra función posterior de la cadena. El Kanban (o tarjeta señalizadora) implica que se genera una señal visual para indicar que hay nuevos bloques de trabajo que pueden ser comenzados porque el trabajo en curso actual no alcanza el máximo acordado. Según David J. Anderson, Kanban parece un cambio muy pequeño pero aun así cambia todos los aspectos de una empresa. [Kniberg2010]

El sistema Kanban es la última tendencia en el desarrollo de software de Lean, enfatizando un enfoque visual para maximizar la fluidez y detectar cuellos de botella y otros tipos de aspectos [Anderson2010]. Uno de los conceptos claves de Kanban es todo el desarrollo debe optimizarse, evitando optimizaciones locales y esforzándose por alcanzar una optimización global.

Objetivos

- Balancear la demanda con la capacidad.
- Limitar el trabajo en proceso, mejorar el “flujo” del trabajo, descubrir los problemas tempranamente y lograr un ritmo sostenible.
- Controlar el trabajo (no la gente), coordinar y sincronizar, descubrir los cuellos de botella y tomar decisiones que generen valor.
- Equipos auto-organizados.
- Lograr una cultura de optimización incremental.

Beneficios

- Ajuste de cada proceso y flujo de valor a medida.
- Reglas simples que permiten optimizar el trabajo en función del valor que genera.
- Mejor manejo del riesgo.
- Tolerancia a la experimentación,
- Difusión “viral” a lo largo de la organización con mínima resistencia.
- Incremento de la colaboración dentro y fuera del equipo.
- Mejora de la calidad del trabajo.
- Ritmo de trabajo sostenido y sustentable.

¿Qué es Kanban?

Kanban es una palabra que proviene del japonés. Kan significa “visual” y ban “tarjeta” o tablero. O sea, Kanban significa “tarjeta visual” o “tarjeta indicadora”. Cada tarjeta representa un ítem de trabajo. El ítem de trabajo puede ser “escribir la documentación” o “agregar comentarios a una función”. La intención de la tarjeta Kanban es balancear la demanda con la capacidad y priorizar todo lo que mejore el valor del negocio. La limitación del número de tarjetas en el tablero (WIP) impide sobrecargar el sistema. [Modezki2011]

Figura 4.1: Tarjeta Kanban como se usa en Toyota [Modezki2011]

Sistema Kanban

Los sistemas Kanban son un enfoque para planificación del trabajo. Los proyectos ágiles típicos usan iteraciones de tiempo prefijadas o perentorias (time-boxed). Es decir, al inicio de la iteración el equipo se reúne y elige las historias del backlog que pueden entregarse al final de la misma, tal como es el caso de Scrum. Durante la iteración, desarrollan esas historias y al final de la misma la entregan.

Estos sistemas son diferentes. En vez de utilizar iteraciones de tiempo prefijado, se enfocan en el flujo continuo de trabajo. El equipo toma una historia del backlog, la desarrolla y entrega tan pronto como se finaliza. Luego toman la siguiente historia del backlog, la desarrollan y entregan. El trabajo se entrega tan pronto esté listo y el equipo solo trabaja en una historia a la vez. Este es el ideal, los enfoques varían .El principio de Kanban es que se comienza con lo que se está haciendo actualmente. Se comprende el proceso actual mediante la realización de un mapa del flujo de valor y entonces se acuerda los límites de trabajo en curso (WIP) para cada fase del proceso. A continuación se comienza a hacer fluir el trabajo a través del sistema comenzándolo ("pull") cuando se van generando las señales Kanban. [shoreSitio]

Flujo de valor

El flujo de valor de un determinado trabajo está compuesto por todos los procesos y subprocesos que van desde la planificación hasta la entrega. En el siguiente tablero los procesos aparecen en columnas

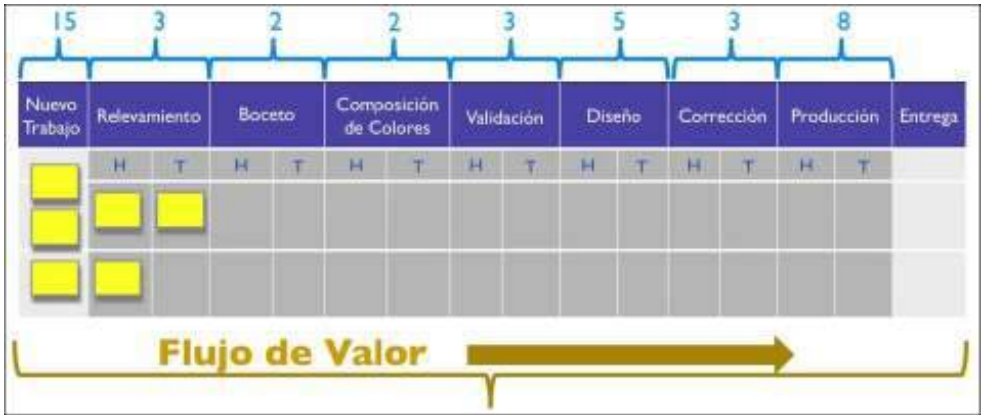


Figura 4.2: Flujo de Valor [Modezki2011]

Cada proceso tiene dos sub-columnas; H para el trabajo que se está haciendo y T para los terminados. Los números que aparecen encima de cada columna son los límites a la cantidad de ítems que se puede tener en cada proceso. [Modezki2011]

Límites

Para el sistema Kanban no solo es importante balancear demanda y capacidad sino también poner límite en cada proceso y subproceso.

Empíricamente se comprueba que cuando se limita la cantidad de trabajo a la capacidad, la calidad aumenta y eso reduce los costos, aumenta la velocidad y reduce los re trabajos por errores. [Modezki2011]. Eso es porque:

- Las cosas se hacen con más velocidad cuando menos tareas paralelas hay. Esta característica que cuando los sistemas están cargados de trabajos funciona más lentamente se demuestra prácticamente en cada trabajo que se pueda hacer.

- La multitarea genera una pérdida de foco y de tiempo al pasar de una tarea a otra. Estas pérdidas son un desperdicio.

- Al terminar más rápidamente un trabajo y pasar al siguiente proceso se tiene una retroalimentación más rápida y se descubren los problemas con mayor velocidad. Esto genera menos daños y por tanto menos costos. Los errores tardíos se acumulan y se hace más difícil descubrir dónde y por qué se generaron, incrementando los costos.

Un aspecto importante es que la cantidad asociada a Nuevo Trabajo, es decir aquello planificado para hacer también está limitado. Esto es porque muchas veces se planifica con anticipación aquello que después se tiene que cambiar, esto conlleva un costo porque es perjudicial a la empresa. Perder tiempo y esfuerzo en una planificación que no va a ser realizada o cuyas prioridades y objetivos cambiaran también es un desperdicio. [Modezki2011]

En el ámbito de la ingeniería de software es más práctico pensar el límite no como la cantidad de ítems, sino como la suma de *story points* o algún otro equivalente. [Joyce2009]

Arrastrar vs Empujar

El proceso de Relevamiento (ver Figura 4.2) tiene un límite de 3, esto es así básicamente para no generar colas que por último resultan en desperdicio.

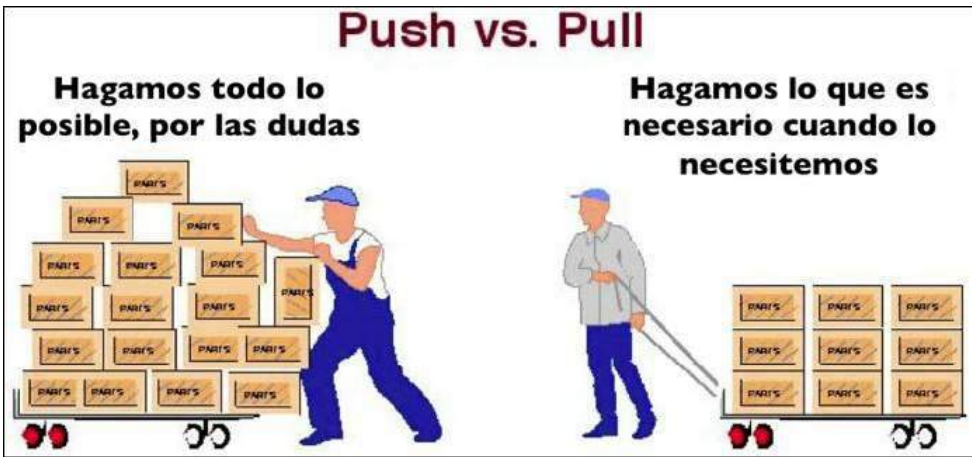


Figura 4.3: Empujar vs. Arrastrar [Modezki2011]

Kanban es un sistema de Pull (o Arrastre), no de Push (o Empuje). De esta manera produce solo el trabajo estrictamente necesario y esto es igualmente válido para la planificación.

Entrega rápida

Limitar el trabajo en curso evita los olvidos, mala información, malos entendidos, duplicación, pérdida de calidad y retrasos. El objetivo de Kanban al limitar el trabajo en curso es que cada tarea se haga lo más rápidamente posible. Los límites deben ser fijados empíricamente, con prueba y error. [Modezki2011]

Tablero

En el desarrollo del software, el sistema Kanban se puede resumir como la visualización de las tareas mediante un tablero, en el que se van moviendo entre los sectores delimitados, con el objetivo de tener siempre presente el trabajo a realizar y lo que hace cada uno. Que nadie se quede sin trabajo y que todas las tareas importantes se realicen primero. [Rubio]

El tablero Kanban, el cual debe estar visible a todo el equipo de trabajo, tiene la peculiaridad de ser un tablero continuo. Esto quiere decir que, no se rellena con tarjetas y se van desplazando hasta que toda la actividad ha quedado realizada (como pasa en Scrum), sino que a medida que se avanza, las nuevas tareas (mejoras, fallos o tareas del proyecto) se van acumulando en la sección inicial, en las reuniones periódicas con el dueño de producto (o el cliente) se priorizan las más importantes, y se encolan en las siguientes zonas. [Rubio]

El tablero Kanban debe contener todo lo necesario para que el equipo tome las decisiones más convenientes sin necesidad de supervisión.

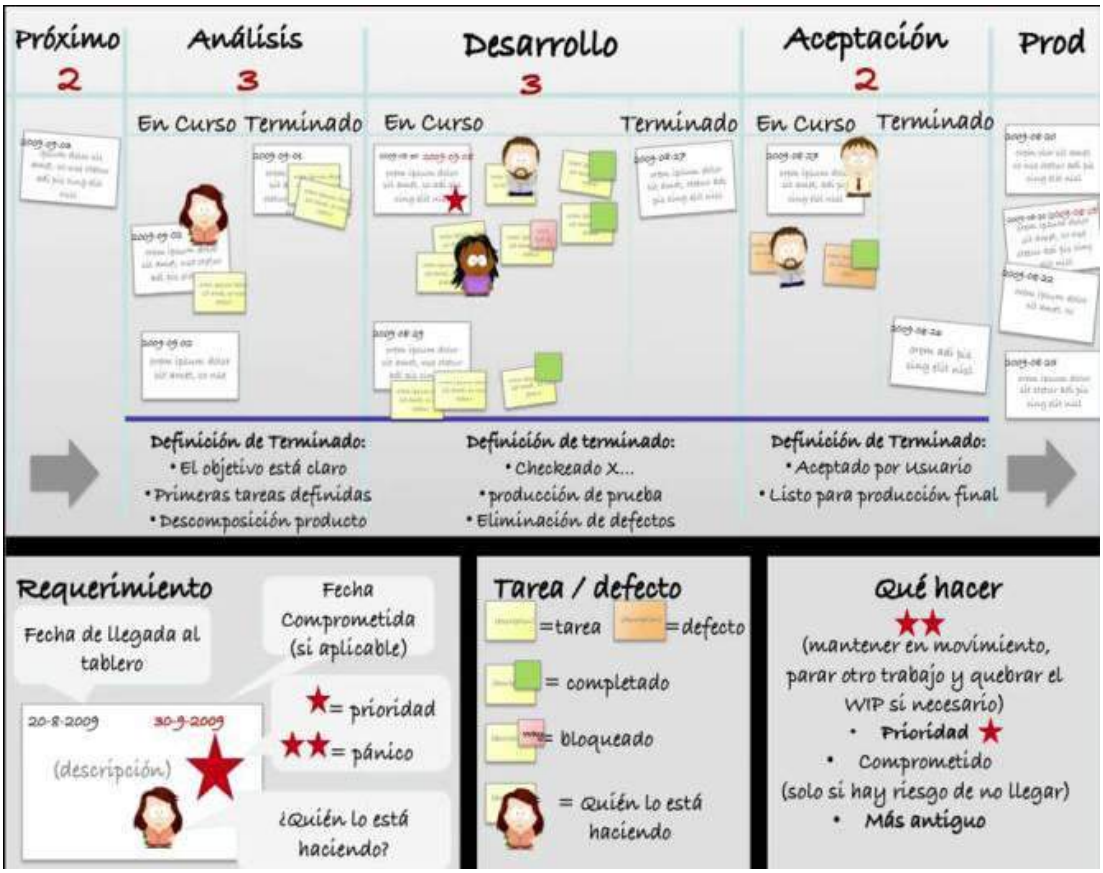


Figura 4.4: Tablero Kanban [Modezki2011]

- Las columnas representan los procesos, con los números que son los límites de trabajo.
- Las tarjetas blancas son productos en proceso que se deben hacer.
- Las tarjetas amarillas son tareas que deben realizarse para terminar un producto o subproducto.
- Los avatares (personas en dibujos) son los responsables de cada tarea. No es conveniente poner el nombre del responsable en cada tarjeta porque esta tarea va siendo trabajada por distintas personas en distintos procesos.
- Los marcadores verdes son trabajos completados.
- Los marcadores rojos son trabajos bloqueados.

Existen reglas de decisión (ver figura 4.4 - abajo a la derecha) que indican el procedimiento para decidir cómo arrastrar el trabajo. Debajo de cada proceso está su definición de "terminado", esta es una característica muy importante, ya que define la calidad con la que se termina una determinada tarea.

Cada tarjeta tiene: Fecha de ingreso, Fecha comprometida (si es aplicable), Descripción y Prioridad. Las tarjetas deben facilitar el sistema de pull (tirar, arrastrar) y alentar la toma de decisiones autónoma del equipo. [Modezki2011]. Las tarjetas de Kanban suelen tener bastante más información, ya que el método consiste en tener todo visual, para saber de forma rápida la carga total de trabajo, ya sea de los grupos, como del departamento, etc. Es importante destacar que las tarjetas deben de tener la estimación de tiempo que tiene asignada la tarea, así como se pueden anotar las fechas de entrada en cada cuadrante, para tener información, al término de la tarea, de si ha sido una buena estimación, así como obtener el rendimiento del equipo de trabajo.

El sistema Kanban, no se dedica a llevar información de un solo proyecto, sino que se pueden entremezclar tareas y proyectos. El método se basa en tener a los trabajadores con un flujo de trabajo constante, las tareas más importantes en cola para ser realizadas y un seguimiento pasivo, a modo de no tener que interrumpir al trabajador para saber qué hace en cada momento.

El sistema Kanban tiene ciertas ventajas con relación a otras metodologías, ya que permite no solo llevar el seguimiento de un proyecto de forma individual, sino también de las incidencias que se van sucediendo, así como otros proyectos paralelos que tenga que hacer el mismo equipo de desarrollo, por lo que se puede deducir es que no se lleva pista de los proyectos, sino de los equipos de trabajo.

Indicadores

Kanban provee indicadores muy claros de la salud del proyecto. El límite de trabajo es fácil de controlar, ayuda a decidir si se debe comenzar o no un nuevo trabajo. Es muy importante indicar la cantidad y fecha en que se fijó el límite. Regular los límites reporta los problemas en los procesos, mientras estos están sucediendo. Las colas acumulando ítems están seguidas por un proceso bloqueado. Las colas no permiten dobles interpretaciones.

Las consecuencias son altamente predecibles. Antes que cualquier parte colapse el sistema completo responde bajando la velocidad. Entonces se liberan recursos que deben acudir a responder el problema. [Joyce2009]

Costos

Lean identifica 7 tipos de desperdicio en la manufacturación. La metáfora de "desperdicio" no es muy útil en la Ingeniería de Software. El "desperdicio" en las actividades puede ser una forma de descubrir información, y por lo tanto no siempre es un desperdicio. El desperdicio se trata más que nada del *costo de retraso*: *maximizar el valor* y *disminuir los riesgos*.

En la ingeniería de software el costo de retraso (desperdicio) proviene de 3 tipos [Joyce2009]:

- Re-trabajo – Defectos, fallas en la demanda
- Costos de Transacción – Planeación, liberación, entrenamiento
- Costos de coordinación – Planificación, recursos

Funcionamiento

Ya en funcionamiento la limitación del trabajo en curso debería permitir lograr un ritmo de trabajo suave, constante, armónico y sostenido. Ese debe ser el foco para lograr todos los beneficios que el sistema Kanban provee.

Uno de los aspectos que se debe evitar es la formación de colas o cuellos de botella, no solo porque interrumpe el flujo de trabajo sino porque son una fuente enorme de defectos, errores y olvidos que se deben evitar. Puede producirse debido a problemas de definición de límites,

problemas de capacidad o una oportunidad de mejorar algunos procesos. En caso de producirse los cuellos de botella, lo ideal es usar los recursos libres para liberar las colas y luego mejorar los procesos. De esta manera sucede una mejora incremental del flujo de valor.

Otra situación indeseable es la hambruna, una persona tiene todas sus tareas todas terminadas pero no puede continuar porque no tiene de dónde “arrastrar”, ya que la persona encargada de realizar la tarea anterior no tiene ninguna tarea finalizada. Nuevamente puede ser problema de definición de límites, problemas de capacidad o una oportunidad de mejorar algunos procesos.

Una tercera situación se presenta si un recurso dentro de un proceso queda liberado y tiene un compañero que no finalizó su tarea. La persona liberada podría arrastrar un trabajo de la cola de tareas o bien ayudar a su compañero para que el trabajo en proceso sea terminado lo más rápido posible. La regla es que el recurso libre debe ayudar a su compañero a terminar el trabajo siempre que esto sea posible. Una vez que ambos terminen arrastran 2 trabajos de la cola. Esto es otra muestra de colaboración y coordinación autónoma, sin necesidad de consultar a un supervisor.

Por todo esto si una persona terminó un trabajo, lo primero que debe tratar de hacer es ayudar a un compañero a terminar un trabajo. En caso de no poder ayudarlo debería buscar un cuello de botella y liberarlo. Y si no es posible liberar un cuello de botella, debería arrastrar un trabajo de la cola si es posible. En caso de no ser posible, buscaría otra cosa interesante para trabajar.

Kanban muestra que la utilización máxima de los recursos no es su objetivo, sino por el contrario, si todos trabajaran al máximo no podría haber mejoras. Los únicos lugares que hacen máxima utilización de recursos son los cuellos de botella, que por definición debería ser uno solo. [Modezki2011]

Políticas y clases

Las Clases de Servicio proveen de una manera conveniente de clasificar el trabajo para ofrecer al cliente más flexibilidad mientras se optimiza el resultado económico. Se definen en función del impacto en el negocio, o sea la generación de valor. Para entender el impacto en el negocio debe entenderse como juegan en combinación distintos factores entre ellos costos fijos, demanda y riesgo.

Las entregas, implementaciones y prioridades varían entre diferentes clases. La clasificación resultará en *Niveles de servicio* específicos a cada clase. [Modezki2011]. Los trabajos deben fluir por el sistema *optimizando el riesgo*. El resultado debe ser una liberación optimizada en riesgo, o sea *máximo valor para el negocio* y *mínimo costo de retraso*. Se debe facultar a todos a tomar decisiones priorizando los alineamientos con los riesgos, o sea, decidir cualquier día, sin ninguna intervención gerencial o supervisión. [Joyce2009]

Por otro lado, las Políticas son reglas asociadas a cada Clase de Servicio, que indican qué “arrastrar” primero y qué hacer cuando debemos tomar una decisión. [Modezki2011]. Las Políticas pueden incluir: Prioridad de arrastre, Límites, Restricciones de Tiempo y Riesgo, Color de la tarjeta y Anotaciones complementarias.

Se recomienda que las Clases de Servicio no excedan más de 6. Esto busca evitar hacer el sistema complejo. [Joyce2009]

Coordinación y sincronización

Lo conveniente es organizar una reunión al comienzo de cada día. En esta reunión la gente está parada y no debe durar más de 15 minutos. El objetivo de la reunión diaria es analizar el flujo de trabajo en el tablero desde la derecha (el final) hacia la izquierda. El foco es siempre terminar el trabajo en curso.

Se analiza el flujo de trabajo, como se comporta y si se están formando colas o cuellos de botella para saber por qué esto está sucediendo y como resolverlo. Todos pueden entender que es lo que pasa visualmente. [Modezki2011]

Los proyectos Kanban no tienen problemas en escalarse hasta 40 integrantes o más. La diferencia de las reuniones Kanban es que existe un facilitador que enumera el trabajo, no las personas. Durante la reunión todos los miembros pueden modificar la pizarra, que debe mantenerse permanentemente actualizada.

Si hubiera problemas puntuales, existe una segunda reunión espontánea solo con las personas involucradas con el fin de introducir mejoras y evitar que los problemas se repitan. [Joyce2009]

Control

Kanban combina la flexibilidad de la producción artesanal con el control de flujo de una cañería.

- El trabajo en proceso es limitado
- El tiempo de ciclo es manejado
- Los procesos son altamente transparentes y repetibles
- Están dadas todas las condiciones necesarias para la mejora continua

Conclusiones

Kanban es un sistema muy simple que se basa principalmente en: Visualización del flujo de trabajo, Limitación del trabajo en proceso, Medición y Gestión del Flujo y Mejora incremental. Tal como lo mencionan los autores de esta metodología, son justamente los métodos simples los que generan los menores trastornos y generalmente los más sustentables beneficios.

Esta metodología puede verse como un sistema transparente y visual de limitación del trabajo en curso y arrastre (pull) del valor. Ayuda a entregar valor promoviendo el flujo y exponiendo los cuellos de botella, las colas, el impacto de los defectos, los costos económicos y por lo tantos el desperdicio.

Kanban, tal como uno de sus mayores exponentes David Anderson afirma, ha demostrado ser útil en equipos que realizan desarrollo Ágil de software, pero también están ganando fuerza en equipos que utilizan métodos más tradicionales. Kanban se está introduciendo como parte de iniciativas Lean para transformar la cultura de las organizaciones y fomentar la mejora continua. [Kniberg2010]

Kanban dentro de un proyecto o en un área tiene que ser adaptado a su contexto y a sus características específicas.

5. OPENUP

¿Qué es Open UP?

Open Up es un proceso unificado ágil y liviano, que aplica un enfoque iterativo e incremental dentro de un ciclo de vida estructurado y contiene un conjunto mínimo de prácticas que ayuda al equipo a ser más efectivo desarrollando software. Open Up abraza una filosofía pragmática y ágil de desarrollo, que se enfoca en la naturaleza colaborativa del desarrollo de software. Se trata de un proceso con herramientas neutrales y de baja ceremonia que puede extenderse para alcanzar una amplia variedad de tipos de proyectos. [openUP]

Es un modelo de desarrollo de software, es parte del Framework de modelo de proceso de Eclipse (Eclipse Process Framework), desarrollado por la fundación Eclipse. Mantiene las características esenciales de RUP, en el cual se incluyen las siguientes características:

- Desarrollo incremental.
- Uso de casos de uso y escenarios.
- Manejo de riesgos.
- Diseño basado en la arquitectura.

OpenUp es un marco de trabajo para procesos de desarrollo de software. Fue propuesto por un conjunto de empresas de tecnología (IBM Corp., Telelogic AB., Armstrong Process Group, Inc., Number Six Software, Inc. & Xansa plc.) lo donaron en el año 2007 a la Fundación Eclipse. La fundación lo ha publicado bajo una licencia libre y lo mantiene como método de ejemplo dentro del proyecto *Eclipse Process Framework*.

OpenUP es mínimamente suficiente, es decir, que solamente incluye contenido fundamental. No provee guías en muchos aspectos que los proyectos podrían enfrentar como por ejemplo: equipos de desarrollo de gran tamaño, situaciones contractuales, aplicaciones críticas o de salud, guías específicas sobre tecnología, etc. Sin embargo, Open UP es completo en el sentido que puede verse como un proceso entero para construir un sistema.

Su proceso puede ser personalizado y extendido para distintas necesidades, que aparecen a lo largo del ciclo de vida del desarrollo de software, dado que su modelo de desarrollo es incremental iterativo, es capaz de producir versiones, además, una de sus mayores ventajas es que puede ser acoplado para proyectos pequeños, dado que en su grafica de roles aparecen 4 personas, que pueden trabajar bien manejando esta metodología.

Dado que mantiene las bases de RUP, aún maneja procesos tan importantes como *Manejo de Riesgos*, que es una parte del desarrollo de software que no se puede descuidar, una desventaja es que se puede utilizar este modelo sin tanto formalismo y se puede caer en el desorden y perder la trazabilidad del proyecto. [OpenUp]

Pueden distinguirse tres Niveles o Capas en OpenUP, tal como se aprecia en la **Figura 5.1**: Micro-incrementos, ciclo de vida de la iteración y ciclo de vida del proyecto.

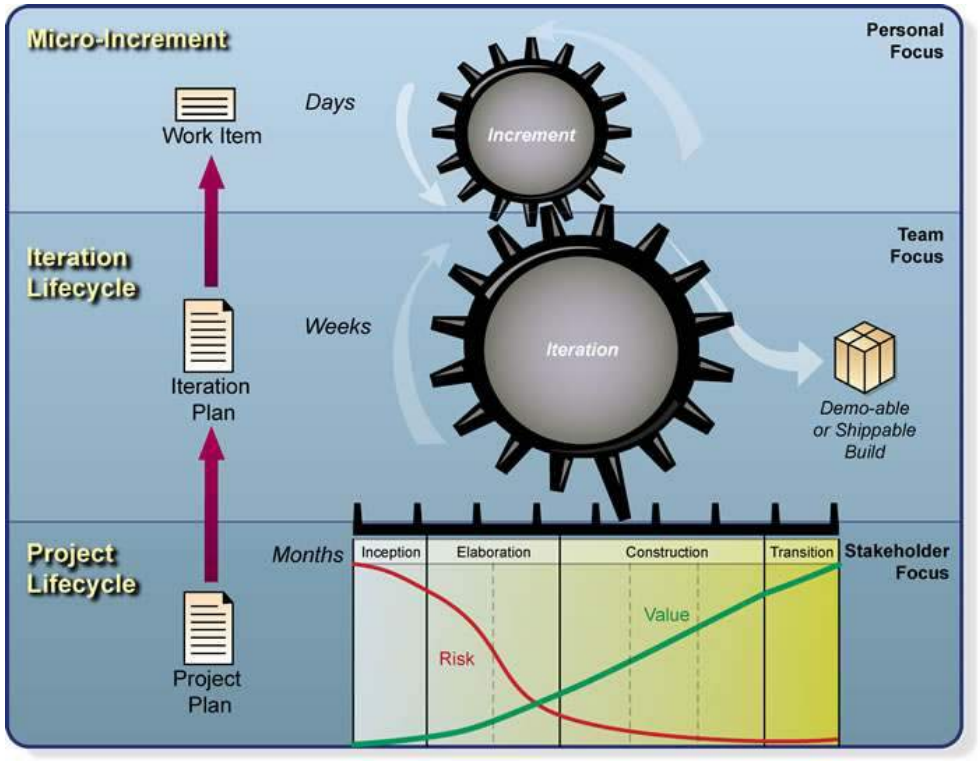


Figura 5.1: Capas de OpenUP: micro-incrementos, ciclo de vida de la iteración y del proyecto [OpenUP]

El esfuerzo personal en un proyecto OpenUP se organiza en **micro-incrementos**. Estas representan unidades cortas de trabajo que producen un ritmo estable y medible de progreso del proyecto (generalmente medido en horas o unos pocos días). Este proceso aplica colaboración intensiva a medida que el sistema se va desarrollando incrementalmente por un equipo comprometido y auto-organizado. Estos microincrementos proveen un ciclo de retroalimentación extremadamente corto que conduce a decisiones de adaptación con cada iteración.

OpenUP divide el proyecto en iteraciones: intervalos de tiempo prefijados y planeados generalmente expresados en semanas. Las iteraciones hacen que el equipo se focalice en entregar valor incremental a los stakeholders de una forma predecible. En el plan de la iteración se define lo que debe entregarse dentro de la iteración y el resultado es algo que puede mostrarse en una demo o entregarse. Los equipos OpenUP se auto-organizan según cómo lograr los objetivos de la iteración y se comprometen a entregar los resultados. Logran esto definiendo y extrayendo tareas de granularidad fina de una lista de ítems. OpenUP aplica un **ciclo de vida iterativo** que estructura cómo se aplican los microincrementos para entregar porciones de desarrollo estables y unidas del sistema que progresa incrementalmente hacia los objetivos de la iteración.

OpenUP estructura el ciclo de vida del **proyecto** en cuatro fases: Inicio, Elaboración, Construcción y Transición. Cada fase puede tener tantas iteraciones como sean necesarias. El ciclo de vida del proyecto provee visibilidad y puntos de decisión a través del proyecto a stakeholders y miembros del equipo. Esto brinda una visión efectiva y permite tomar decisiones de continuar o no en los momentos apropiados. Un plan de proyecto define el ciclo de vida y el resultado final es una aplicación entregada.

Visión general de OpenUP

OpenUP es para equipos pequeños que trabajan juntos en la misma ubicación. Los equipos necesitan involucrarse en una interacción plena cara a cara diariamente. Los miembros del equipo incluyen a los stakeholders, desarrolladores, arquitectos, administradores de proyecto y

testers. Ellos toman sus propias decisiones sobre en qué necesitan trabajar, cuáles son las prioridades y cómo alcanzar las necesidades del stakeholder de la mejor manera.

Los miembros del equipo trabajan colaborativamente. La presencia de los stakeholders como miembros del equipo es crítico para la implementación exitosa de OpenUP. Además, los miembros del equipo participan en daily stand-up meetings para comunicar los estados y situaciones. Los problemas se resuelven fuera de estas reuniones, similar a Scrum.

OpenUP se enfoca en reducir el riesgo de manera temprana en el ciclo de desarrollo. Esto requiere de reuniones regulares para revisar los riesgos y una implementación rigurosa de estrategias de mitigación.

Todo el trabajo se lista, rastrea y asigna a través de la lista de ítems de trabajo (Work Items List). Los miembros del equipo usan este simple repositorio para todas las tareas que necesitan registrarse y rastrearse. Esto incluye todas las solicitudes de cambio, bugs y solicitudes del stakeholder.

Los casos de usos se usan para elicitar y describir los requerimientos. Si bien los describen los desarrolladores, son los Stakeholders responsables de revisar y certificar que los requerimientos son correctos. Los casos de uso se desarrollan de manera colaborativa.

Los requerimientos importantes arquitectónicamente deben identificarse y estabilizarse en la fase de Elaboración de tal manera que se pueda crear una arquitectura robusta como el corazón del sistema. Un cambio en un requerimiento importante desde el punto de vista de la arquitectura que deba realizarse puede traer retrasos en el desarrollo pero el riesgo de que esto suceda se reduce significativamente en la fase de elaboración.

Las pruebas (Testing) se realizan muchas veces en cada iteración, cada vez que se incrementa la solución con el desarrollo de un requerimiento, cambio o arreglo de un bug. Estas pruebas ocurren luego que los desarrolladores han desarrollado la solución (a la que debe haberse realizado la prueba de unidad) y la integren al código base. Estas pruebas ayudan a garantizar que se está creando una solución estable y siempre está disponible a medida que progresa el desarrollo.

Principios OpenUP

OpenUP está gobernada por cuatro principios fundamentales

- Balancear las prioridades involucradas para maximizar el valor para el stakeholder

Fomentar prácticas que permitan a los participantes del proyecto y los stakeholders desarrollar una solución que maximice los beneficios de los stakeholders y sea compatible con restricciones impuestas en el proyecto (de costes, plazos, recursos, normas, etc).

- Colaborar para alinear los intereses y compartir entendimiento

Promover las prácticas que fomenten un ambiente de equipo saludable y que permitan la colaboración y desarrollo de un entendimiento compartido del proyecto.

- Enfocarse en la arquitectura tempranamente para minimizar riesgos y organizar el desarrollo

Promover prácticas que permitan al equipo enfocarse en la arquitectura para minimizar los riesgos y organizar el desarrollo

- Evolucionar para obtener constantemente retroalimentación y mejorar

Promover prácticas que permitan al equipo obtener retroalimentación temprana y continua de los stakeholders, y demostrarles el valor incremental.

Cada principio de OpenUP apoya una declaración del Manifesto Agil, como se ve en la siguiente tabla [Baudino]

Principio de OpenUP	**Declaración del Manifesto Agil**
Colaborar para alinear los intereses y compartir entendimiento	*Al individuo y a las interacciones del equipo de desarrollo por encima del proceso y las herramientas*
Balancear las prioridades involucradas para maximizar el valor para el stakeholder	*Colaboración con el cliente por sobre la negociación contractual*
Enfocarse en la arquitectura tempranamente para minimizar riesgos y organizar el desarrollo	*Software que funcione por sobre una documentación exhaustiva*
Evolucionar para obtener constantemente retroalimentación y mejorar	*Respuesta al cambio por sobre el seguimiento de un plan*.

Tabla 5.1 – Mapping between OpenUP principles and Agile Manifesto [Baudino]

Roles

OpenUp describe un conjunto mínimo de seis roles que deben cubrirse al realizar un desarrollo.

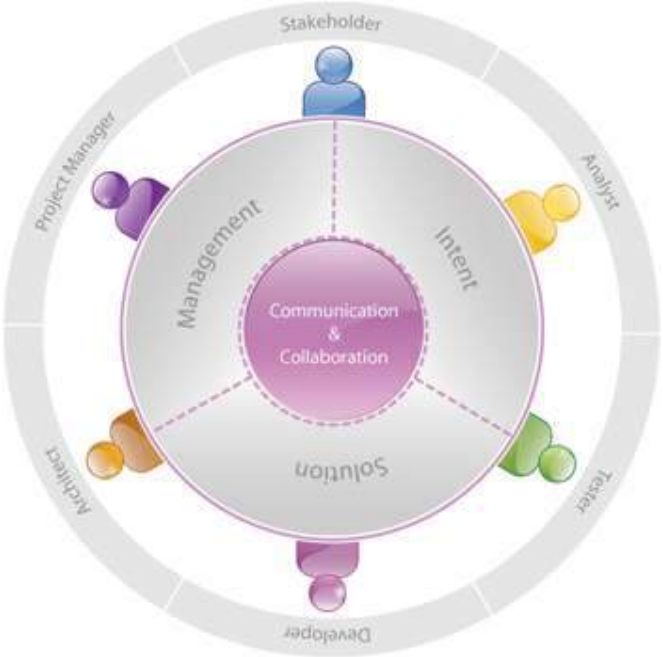


Figura 5.2: Roles principales en OpenUp y su interacción [OpenUP]

- **Administrador del proyecto (Project Manager):** lidera la planificación del proyecto, coordina las interacciones con los stakeholders, y mantiene el equipo enfocado en alcanzar los objetivos del proyecto.
- **Analista (Analist):** representa las preocupaciones de los clientes y usuarios finales mediante la recopilación de aportes de los stakeholders para entender el problema a resolver y mediante la captura y establecimiento de prioridades para los requisitos.

- **Arquitecto (Arquitect):** es responsable de definir la arquitectura de software, que incluye la toma de decisiones técnicas claves que limitan el diseño y la implementación del sistema.
- **Tester:** El Tester es el responsable de las actividades principales del esfuerzo de la prueba. Esas actividades incluyen la identificación, definición, implementación y realización de las pruebas necesarias, así como del registro de los resultados de las pruebas y el análisis de los resultados.
- **Stakeholders:** representa a grupos cuyas necesidades deben ser satisfechas por el proyecto. Es un papel que puede jugar por cualquier persona que es (o será potencialmente) afectados por el resultado del proyecto. Son aquellos que se verán afectados directamente con el resultado del proyecto.
- **Desarrollador (Developer):** es responsable de desarrollar una parte del sistema, incluyendo diseñarlo de tal forma que se ajuste a la arquitectura, posiblemente crear prototipos de la interfaz de usuario y luego implementar pruebas de unidad, e integrar los componentes que forman parte de la solución.

Disciplinas OpenUP - Tareas

Así como Proceso Unificado plantea una serie de disciplinas para ordenar los flujos de trabajo, OpenUP plantea seis disciplinas para agrupar las tareas involucradas en el desarrollo.

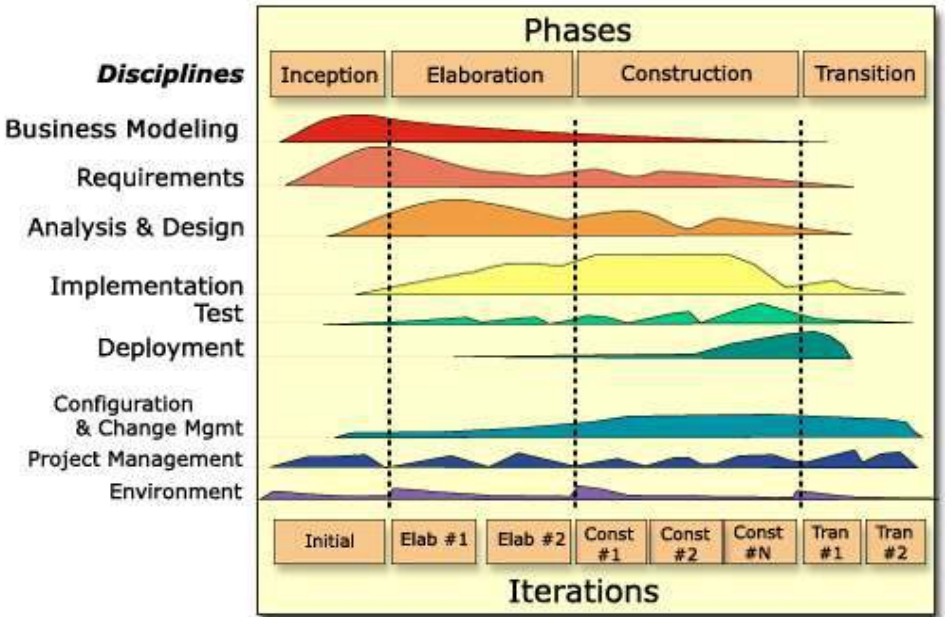


Figura 5.3: Disciplinas de PU [Baudino]

OpenUP se enfoca en las siguientes disciplinas: Requerimientos, Arquitectura, Desarrollo, Prueba, Administración de Configuración y Cambio y Administración de Proyecto. [Baudino]

- **Requierimientos (Requirements)**: explica cómo elicitar, analizar, validar y manejar los requerimientos para el sistema a desarrollar. Es importante entender la definición y alcance del problema que se intenta resolver, identificar los stakeholders y definir el problema a resolver. Durante el ciclo de vida se administran los cambios de los requerimientos.
- **Arquitectura (Arquitecture):** El propósito es lograr evolucionar a una arquitectura robusta para el sistema. Explica cómo crear una arquitectura a partir de requerimientos

arquitectónicamente (estructuralmente) importantes. Es en la Disciplina de desarrollo donde se construye la arquitectura.

- **Desarrollo (Development)**: explica cómo diseñar e implementar una solución técnica que esté conforme a la arquitectura y sustente los requerimientos
- **Prueba (Test)**: explica cómo proveer retroalimentación sobre la madurez del sistema diseñando, implementando, ejecutando y evaluando pruebas. Es iterativa e incremental. Aplica la estrategia de "prueba lo antes posible y con la mayor frecuencia posible" para replegar los riesgos tan pronto como sea posible dentro del ciclo de vida del proyecto.
- **Administración de configuración y cambio (Configuration and change management)**: explica cómo controlar los cambios de los artefactos, asegurando una evolución sincronizada del conjunto de productos (Work Products) que compone un sistema software. Esta disciplina se expande durante todo el ciclo de vida
- **Administración del Proyecto (Project Management)**: explica cómo entrenar, ayudar y apoyar al equipo, ayudándolo a manejar los riesgos y obstáculos que se encuentren al construir el software.

En OpenUP se omitieron otras disciplinas y áreas de incumbencia, como por ejemplo el modelado de negocio, entorno, administración avanzada de requerimientos y administración de la configuración. Estos aspectos se consideran innecesarios para proyectos pequeños o bien se manejan dentro de otras áreas de la organización, fuera del equipo del proyecto [Baudino]

Tareas - Tasks

Una tarea es una unidad de trabajo que se puede solicitar que lo realice un rol de trabajo. En OpenUP hay 19 tareas que los roles pueden realizar como actor principal (teniendo la responsabilidad de ejecutar esas tareas) o adicionales (ayudando y proveyendo información que se usa al ejecutar la tarea). La naturaleza colaborativa de OpenUP se ve manifiesta al tener los actores principales de las tareas interactuando con otros individuos al realizar las tareas.

Disciplina	Tarea
Arquitectura	• Refinar la arquitectura • Visualizar (Preveer) la arquitectura
Desarrollo	• Implementar las pruebas de desarrollo • Implementar la solución • Correr las pruebas de desarrollo • Integrar y crear el desarrollo • Diseñar la solución
Administración de Proyecto	• Evaluar resultados • Administar La iteración • Planear la iteración • Planear el proyecto • Solicitar cambios
Requerimientos	• Identificar y resaltar requerimientos • Detallar Escenarios de Caso de Uso • Detallar requerimientos generales del sistema • Desarrollar una visión técnica
Prueba	• Crear Casos de Prueba • Implementar pruebas • Correr pruebas

Tabla 5.2: Artefactos de OpenUP (armado con información de [OpenUP]

Artefactos - Artifacts

Un artefacto (work product) es algo producido, modificado o utilizado por una tarea. Los Roles son responsables de crear y actualizar los artefactos. Los artefactos están sujetos a control de

versión a través del ciclo de vida del proyecto. Los 17 artefactos de OpenUP se consideran esenciales y son los que debe utilizar un proyecto para capturar información relacionada con el producto y el proyecto. No hay obligación de capturar la información en artefactos formales. Se puede capturar la información informalmente en pizarras (por ejemplo para el diseño y la arquitectura), notas de reuniones (por ejemplo para la evaluación del estado), etc. Los proyectos pueden utilizar los artefactos de OpenUP o reemplazarlos con los propios.

Disciplina	**Artefacto**	**Tipo de Producto**	**Descripción General**
Arquitectura	Architecture Notebook	Especificación	Describe el contexto para el desarrollo del sw. Contiene las decisiones, razones, supuestos, explicaciones e implicancias de armar la arquitectura
Desarrollo	Diseño	Solución	Describe la realización de la funcionalidad del sistema requerida en términos de componentes y sirve como una abstracción del código fuente.
Desarrollo	Construcción	Solución	Versión operacional de un sistema o parte del sistema que muestra un subconjunto de las funcionalidades que se proveerán en el producto final.
Desarrollo	Prueba	Solución	Las instrucciones que validan que los componentes individuales del sw se comportan como se especificó.
Desarrollo	Implementación	Solución	Archivos de código sw, archivos de datos y archivos secundarios como archivos de ayuda en línea que representan las partes gruesas de un sistema.
Administración de Proyecto	Plan de iteración	Plan	Un plan con gran nivel de detalle describiendo los objetivos, asignaciones de trabajo y criterios de valuación para la iteración
Administración de Proyecto	Plan de Proyecto	Plan	Reúne toda la información requerida para administrar el proyecto. Sus partes principales están compuestas por: un plan poco detallado que contiene las fases del proyecto y los hitos
Administración de Proyecto	Lista de ítems de trabajo	Datos del proyecto	Contiene una lista de todo el trabajo calendarizado a realizarse dentro del proyecto, así como el trabajo propuesto que puede afectar al producto en este o futuros proyectos. Cada ítem de trabajo puede contener referencias a información relevante para llevar a cabo el trabajo descripto dentro del ítem de trabajo.
Administración de Proyecto	Lista de riesgo	Datos del proyecto	Lista abierta de riesgos conocidos, ordenados según importancia y asociados con acciones específicas de mitigación o contingencia.
Requerimientos	Especificación de	Especificación	Captura los requerimientos globales del sistema no capturados en escenarios o

	requerimientos secundario		casos de uso, incluyendo requerimientos de calidad y requerimientos de funcionalidad global.
Requerimientos	Visión	Concepto	Contiene la definición de la vista que tienen los stakeholders sobre el producto a desarrollar, especificado en términos de necesidades y características claves de los stakeholders. Contiene una idea general del corazón de requerimientos imaginados para el sistema
Requerimientos	Caso de Uso	Especificación	Captura la secuencia de acciones que un sistema puede realizar que produce un resultado observable de valor para los que interactúan con el sistema.
Requerimientos	Glosario	Elemento de modelo, Especificación	Define los términos importantes usados en el proyecto. Estos términos son la base para la colaboración efectiva con los stakeholders y otros miembros del equipo.
Requerimientos	Modelo de Caso de Uso	Modelo	Captura un modelo de las funcionalidades pretendidas en el sistema y su entorno, y sirve como un contrato entre el cliente y los desarrolladores.
Pruebas	Caso de Prueba	Especificación	Es la especificación de un conjunto de entradas de prueba, condiciones de ejecución y resultados esperados, identificados con el propósito de realizar la evaluación de algún aspecto particular de un escenario
Pruebas	Log de Prueba	Dato de Proyecto	Recolecta salidas sin refinar capturadas durante una única ejecución de una o más pruebas para una sola corrida del ciclo de prueba.
Pruebas	Script de Prueba	Solución	Contiene las instrucciones paso a paso que realiza una prueba, permitiendo su ejecución. Estas pueden tomar forma ya sea de instrucciones textuales documentadas que se ejecutan manualmente o bien instrucciones que una computadora puede entender y permite la ejecución automática de la prueba.

Tabla5.3: Artefactos de OpenUP (armado con información de [OpenUP])

Ciclo de vida de la iteración

El ciclo de vida de la iteración provee un conjunto de prácticas basadas en el equipo que describen cómo utilizar las iteraciones para enfocar al equipo en la entrega de valor incremental a los stakeholders de una forma predecible. Las iteraciones en OpenUP mantienen al equipo enfocado en brindar al cliente valor incremental cada pocas semanas entregando un incremento de producto completamente probado que puede ser mostrado o entregado al cliente. Esto crea un enfoque saludable de asegurarse de estar trabajando en algo que

agregue valor a los stakeholders. La toma de decisiones debe realizarse rápidamente ya que no hay tiempo para debates interminables. El desarrollo iterativo se enfoca en producir código que funcione reduciendo el riesgo de la fase de "parálisis-análisis" [OpenUP]. La demostración frecuente de código funcionando provee mecanismos de retroalimentación que permiten realizar las correcciones en curso según sean necesarias.

La planificación, estimación y rastreo de progreso en la iteración está centrada en los ítems de trabajo (work ítems). Se rea el plan de la iteración seleccionando los ítems de trabajo con mayor prioridad. Se usan técnicas ágiles de estimación para comprender cuantos ítems de trabajo pueden completarse en el tiempo prefijado de la iteración, y se filtran los work ítems para asegurar que los ítems de trabajo elegidos emitirán al equipo cumplir con los objetivos de la iteración expresados por los stakeholders. El progreso se demuestra a través de la cantidad de ítems de trabajo que se completan.

Así como el proyecto tiene un ciclo de vida, las iteraciones tienen su ciclo de vida que tiene un enfoque diferente para los equipos dependiendo si es la primera o la última semana de la iteración, ver **Figura 5.4**. Una iteración comienza con una reunión de planificación de unas pocas horas de duración. Los primeros días el enfoque está puesto en seguir planificando y definir la arquitectura, entre otras cosas, para entender las dependencias y orden lógico de los ítems de trabajo y de los impactos en la arquitectura del trabajo a realizar. La mayor parte del tiempo en una iteración se emplea para ejecutar los microincrementos. Cada microincremento debe entregar código probado al incremento así como artefactos validados. Para dar mayor disciplina, al final de cada semana se produce un incremento estable. Se emplea mayor atención en ese incremento para asegurar que no se está socavando la calidad y problemas son resueltos tempranamente de tal forma que no peligra el éxito de la iteración. La última semana o últimos días de la iteración, generalmente tienen un énfasis mayor en pulir y arreglar errores que en semanas anteriores, aunque se agregan nuevas características según sea apropiado. El objetivo es nunca socavar la calidad, y así producir un incremento de producto útil de alta calidad al final de la iteración. A iteración termina con una valoración (con los stakeholders) de lo que se construyo, y una retrospectiva para entender cómo mejorar el proceso para la siguiente iteración.

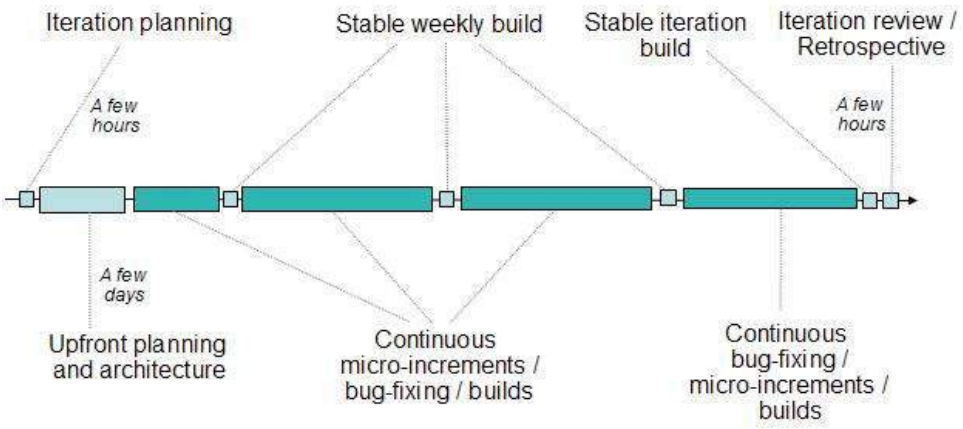


Figura 5.4: Énfasis dentro de una iteración.[OpenUp]

Dar al equipo la posibilidad de organizar su trabajo y determinar la mejor forma de alcanzar los objetivos motiva a los miembros del equipo a hacer lo mejor que pueden. También los ayuda a colaborar entre sí para asegurar que se aplican las habilidades apropiadas. La auto organización impacta en muchas áreas, incluyendo como planificar o asumir los compromisos (por el equipo, no individuos), cómo se asigna el trabajo (no se asigna a alguien sino que cada uno elige qué hacer), y cómo los miembros del equipo ven los roles en el proyecto (en primer lograr miembros del equipo, luego la función del trabajo).

Micro_incrementos

Un micro-incremento es un paso pequeño, medible hacia los objetivos de una iteración. Representa unas pocas horas hasta un día de trabajo realizado por una o unas cuantas personas colaborando para alcanzar el objetivo

El concepto de microincremento ayuda al miembro individual del equipo a partir el trabajo en pequeñas unidades que entregan algo de valor medible al equipo. Los microincrementos proveen una retroalimentación extremadamente corta que conduce a decisiones adaptativas dentro de una iteración.

Un micro-incremento debe estar bien definido y se debe poder rastrear diariamente su progreso. En un ítem de trabajo se especifican y rastrean los imicro-incrementos. Ejemplo de microincrementos:

- **Identificar Stakeholders**. Dentro de una tarea que puede llevar unas semanas com es Definir la Visión, esta actividad constituye un microincremento.
- **Desarrollar un incremento de solución**. Definir, diseñar e implementar un Caso de uso puede llevar varias semanas, se puededividir el trabajo y considerar subflujos de un caso de uso.
- **Definir el plan de la Iteración**. Podría incluir una reunión para crear el plan y realizar alguna preparación previa para la misma.

La aplicación evoluciona en microincrementos a través de la ejecución simultánea de un número de ítems de trabajo. Se logra la transparencia y posibilidad de comprender el trabajo de cada uno a través de compartir abiertamente el progreso a través de los micro-incrementos.

Ciclo de vida del desarrollo de software mediante OpenUP

Open UP es un proceso iterativo con iteraciones distribuidas en cuatro fases ya mencionadas anteriormente: Inicio, Elaboración, Construcción y Transición. Cada fase puede tener tantas iteraciones como sean necesarias (dependiendo del grado de novedad, del dominio del negocio, la tecnología usada, la complejidad de la arquitectura, el tamaño del proyecto y de otros factores). Las iteraciones pueden tener longitud variable dependiendo de las características del proyecto, si bien las iteraciones de un mes son las que se recomiendan en general.

Para ofrecer un comienzo rápido a los equipos para planear sus iteraciones, OpenUp propone desglosar el trabajo para cada iteración en un WBS (Work Breakdown Structure), y un WBS para todo el proceso del proyecto (de extremo a extremo).

En cada fase se desarrolla y entrega versiones del software estables y que funcionan. La finalización de cada iteración representa un hito menor del proyecto y contribuye a lograr el éxito del hito mayor de la fase donde se alcanzan los objetivos de la fase. Cada fase termina con un hito esperado proveyendo una revisión haciendo surgir y respondiendo un conjunto de preguntas que son comúnmente críticas para los stakeholders:

- **Inicio** (Inception). ¿Se está de acuerdo con el alcance y los objetivos? ¿Se debe continuar o no con la realización del proyecto?
- **Elaboración** (Elaboration).¿Se está de acuerdo con la arquitectura ejecutable a utilizarse para desarrollar la aplicación? ¿Es aceptable el valor entregado hasta el momento y el riesgo remanente?
- **Construcción** (Construction).¿La aplicación está suficientemente cercana a ser entregada de tal forma que se deba cambiar el foco primario del equipo a refinar, pulir y asegurar un exitoso despliegue?
- **Transición** (Transition). ¿La aplicación está lista para ser liberada?

Si la respuesta es SI a las preguntas anteriores en la fase de revisión, el proyecto continúa. En caso de ser no la respuesta, la fase es demorada (generalmente agregando una nueva iteración) hasta que se reciba una respuesta satisfactoria, o bien los stakeholders determinen que debe cancelarse el proyecto

El siguiente diagrama muestra el ciclo de vida de Open UP. Este ciclo de vida del proyecto provee a los stakeholders y miembros del equipo con visibilidad, sincronización, puntos de decisión a través del proyecto permitiendo momentos de revisión y decisiones de continuar o no continuar.

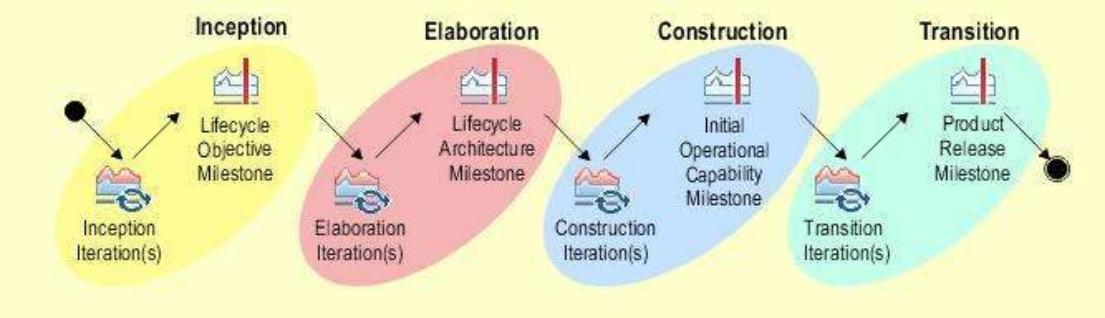


Figura 5.5. Ciclo de vida de OpenUP [OpenUP]

El ciclo de vida del proyecto provee a los stakeholders con revisión, transparencia y mecanismos de dirección para controlar las bases del proyecto, el alcance, la exposición a los riesgos, el valor provisto y otros aspectos del proceso.

Cada iteración entrega un incremento del producto, que provee a los stakeholders la oportunidad de entender el valor que ha sido entregado y que tan bien está marchando el proyecto. También le brinda al equipo de desarrollo la oportunidad de realizar cambios al proyecto para optimizar la salida.

Uno de los objetivos del ciclo de vida del proyecto es enfocarse en dos conductores claves de los stakeholders: reducción de riesgo y creación de valor. Las fases de OpenUP enfocan al equipo en la reducción del riesgo relacionado con preguntas a ser contestadas al final de la fase, mientras se rastrea la creación de valor, ver **Figura 5.6**.

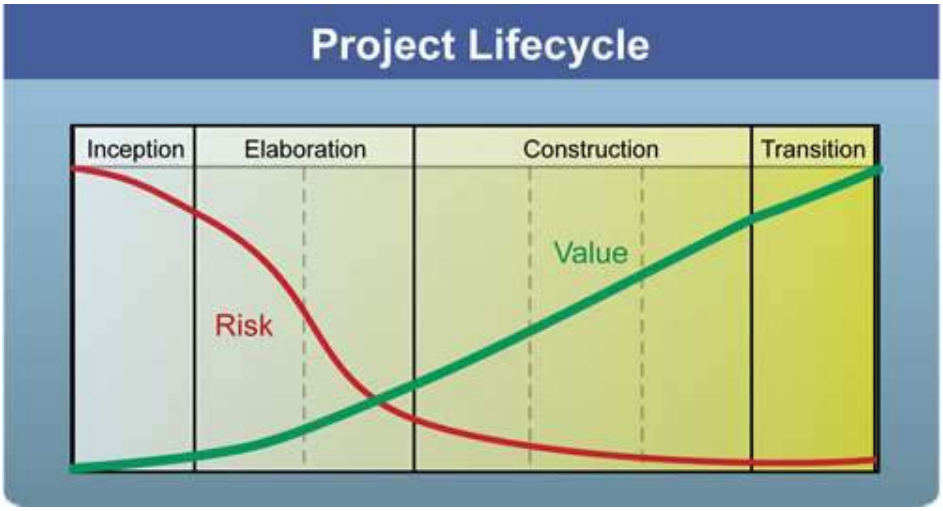


Figura 5.6: Reducción de Riesgo (curva roja) y creación de valor (curva verde) durante el ciclo de vida del proyecto.[OpenUp]

El riesgo es la posibilidad de que ocurran cosas inesperadas en el proyecto, y los riesgos se interponen en la creación de valor. El riesgo es directamente proporcional a la incertidumbre estimada, y los stakeholders generalmente quieren saber más temprano y no más tarde qué valor del proyecto se puede entregar en el tiempo estipulado. En muchos casos, se reduce el riesgo cuando se crea valor implementando y probando las características más críticas. Sin

embargo, hay situaciones donde la reducción del riesgo y la inmediata creación de valor están contrapuestas

Fase de Inicio

Como la primera de las cuatro fases, esta fase busca comprender el alcance del proyecto y sus objetivos y también obtener suficiente información para confirmar que el proyecto debe continuar o convencer que debe cancelarse. El propósito es alcanzar un acuerdo entre todos los stakeholders sobre los objetivos del proyecto.

Hay cuatro objetivos en esta fase que clarifican el alcance, objetivos del proyecto y factibilidad de la solución pretendida: [KROLL]

- **Entender lo que se va a construir.** Determinar una visión general, incluyendo alcance del sistema y sus límites. Identificar stakeholders
- **Identificar las funcionalidades claves del sistema.** Decidir qué requerimientos son más críticos.
- **Determinar por lo menos una posible solución.** Evaluar si la visión es técnicamente factible. Esto puede involucrar identificar una arquitectura candidata de alto nivel o realizar prototipos técnicos, o ambas cosas.
- **Entender** a un alto nivel la estimación de costos, calendario y riesgos asociados al proyecto.

Fase de Elaboración

Esta es la segunda fase, donde se tienen en cuenta los riesgos estructuralmente significativos. El propósito de esta fase es establecer la línea base de la arquitectura del sistema y proveer una base estable para la mayor parte del esfuerzo de desarrollo de la siguiente fase.

Hay tres objetivos que ayudan a tratar los riesgos asociados con los requerimientos, arquitectura, costos y calendario [KROLL]:

- Obtener un entendimiento más detallado de los requerimientos. Asegurarse de tener un conocimiento profundo de los requerimientos más críticos**.**
- Diseñar, implementar, validar y establecer la línea base de la arquitectura (para el esqueleto de la estructura del sistema).
- Mitigar riesgos esenciales y producir un calendario apropiado y estimación de costos.

La mayoría de las actividades durante la fase de elaboración se hace en paralelo.

Esencialmente, los principales objetivos para la elaboración están relacionados con el mejor entendimiento de los requerimientos, creando y estableciendo una línea base de la arquitectura para el sistema, y mitigando los riesgos de mayor prioridad.

Fase de construcción

La tercera fase se enfoca en diseñar, implementar y probar funciones para desarrollar un sistema completo. El propósito es completar el desarrollo del sistema tomando como base la arquitectura definida en la fase anterior.

Hay ciertos objetivos que ayudan a tener un desarrollo costo efectivo para un producto completo – una versión operativa del sistema – que puede entregarse al usuario [KROLL]:

- Desarrollar iterativamente un producto completo que esté listo para la fase de transición.
- Minimizar los costos de desarrollo y lograr cierto grado de paralelismo.

Fase de transición

La última de las fases se enfoca en desplegar el software a los usuarios y asegurarse que se alcanzaron sus expectativas sobre el software. El propósito es asegurar que el software está listo para entregarse al usuario.

Hay tres objetivos que ayudan a refinar la funcionalidad, performance, y calidad global del producto beta completado en la fase anterior [KROLL]:

- Prueba Beta para validar que se cumplen las expectativas del usuario.
- Lograr constancia por parte de los stakeholders para asegurar que el despliegue esté completo. Hay varios niveles de pruebas para la aceptación del producto.
- Mejorar el rendimiento de proyecto futuro a través de lo aprendido.

Como comenzar

El cuarto principio fundamental de OpenUP, "*Evolucionar para obtener retroalimentación continua y mejorar* ", sugiere un enfoque iterativo e incremental para adoptar OpenUP.

- Comenzar con los principios fundamentales y comprender las intenciones detrás de OpenUP.
- Dentro de procesos existentes elegir uno o dos áreas claves a mejorar.
- Comenzar usando OpenUP para mejorar esas áreas primero.
- En iteraciones ciclos de desarrollos posteriores realizar mejoras incrementales en otras áreas.
- En caso de no tener experiencia usando proceso unificado u otro proceso iterativo, usar OpenUP en un pequeño proyecto piloto.

En OpenUP puede extenderse el proceso (como por ejemplo auditorías de sistemas de seguridad críticos) o modificarse las plantillas de los productos de trabajo (work product) para adaptarse a necesidades específicas. Ya sea que se necesite mayor precisión en los work products, la organización tenga ciertos protocolos en los procesos, añadir experiencia, etc.

Conclusiones

Open UP es una metodología de desarrollo completa en el sentido que puede verse como un proceso entero para construir un sistema. Mantiene las características esenciales de RUP y realiza un manejo de Riesgos pero a su vez define cuatro principios que lo rigen y que pueden vincularse con los cuatro valores del Manifiesto ágil.

Las iteraciones son de tiempo prefijado y su proceso puede ser personalizado y extendido para distintas necesidades. Si bien se proponen una serie de artefactos esenciales, estos se pueden reemplazados con los propios del equipo de desarrollo. Cada iteración entrega un incremento del producto, que provee a los stakeholders la oportunidad de entender el valor que ha sido entregado y comprender qué tan bien está marchando el proyecto para de esta forma brindar al equipo de desarrollo la oportunidad de realizar cambios al proyecto para optimizar la salida.

Open UP es para equipos pequeños que trabajan colaborativamente en la misma ubicación. Los equipos necesitan involucrarse en una interacción plena cara a cara diariamente.

6. CRYSTAL CLEAR

Introducción

Según Alistair Cockburn, el desarrollo de software puede caracterizarse como un juego cooperativo de invención y comunicación con restricciones económicas (descripto en Agile Software Development [Cockburn2002]. La forma en que cada equipo juega cada partido tiene mucho que ver con la salida del proyecto y el software resultante. Crystal Clear ataca directamente el juego económico-cooperativo, apuntando dónde prestar atención, dónde simplificar y cómo variar las reglas.

Crystal Clear no aspira a ser la "mejor" metodología; aspira a ser "suficiente", de tal manera que el equipo lo amolde a sus necesidades y lo use. Alistair Cockburn [Cockburn2002]

Crystal Clear es una metodología que funciona con personas, descripta de la manera más simple como sigue [Cockburn2002]:

El diseñador líder y otros dos a siete desarrolladores en una gran habitación o habitaciones contiguas, con radiadores de información, como pizarrones y rotafolios en la pared, teniendo acceso a usuarios claves, distracciones mantenidas al margen, entregando y corriendo código usable y probado cada mes o dos (a lo sumo tres), reflexionando periódicamente y ajustando su propio estilo de trabajo.

¿Cuál es la base para Crystal?

La primera base para Crystal es empírica. En 1991 a Alistair Cockburn lo contrataron de IBM Consulting Group para crear una metodología para los próximos proyectos con tecnología de objetos. Comenzó a visitar y tomar notas de diferentes equipos en todo el mundo. Después de muchos años de entrevistas y trabajando directamente en proyectos, determinó que las técnicas y productos de trabajo detallados en las "metodologías" no formaban parte del corazón de propiedades de éxito más bien estaban en un segundo orden de factor de éxito. Al hacer que las personas trabajen muy juntas, comunicándose de manera bien predispuesta, entregando a menudo y obteniendo retroalimentación rápidamente de los usuarios, probablemente el equipo logrará resolver el resto por su propia cuenta, usando cualquier tecnología y técnicas que conozca. Posteriormente se dio cuenta que estas recomendaciones formaban una metodología, que contenía pocas reglas pero permitía a las personas actuar como profesionales y obtener el mejor camino al éxito por su propia cuenta

La segunda base consiste en una serie de 10 principios y de leer literatura sobre la comunicación humana y el proceso de diseño

Los principios resumidos son los siguientes:

- **Diferentes proyectos necesitan diferentes compensaciones (trade-offs)**. **de metodología.** Se debe tener en cuenta el número de personas a ser coordinadas y el grado de daño que podría causar el mal funcionamiento del sistema para considerar la metodología a adoptar.
- **Equipos más grandes necesitan mayores elementos de comunicación**. Por ejemplo, Crystal Clear está recomendado para equipos que pueden lograr comunicación osmótica.
- **Proyectos que se enfrentan a daños potenciales mayores necesitan más elementos de validación**. Por ejemplo, un grupo pequeño que trabaja en un sistema que manipula barras de boro en un reactor nuclear tendrá más cuidado en su trabajo que un sistema organizador de recetas de cocina. La diferencia está en la dimensión de verificación y validación y no en la dimensión de comunicación y coordinación.
- **Una pequeña metodología produce muchos beneficios, después de eso el peso es muy costoso**. A diferencia del pensamiento "más metodología es mejor" el autor propone "menos es generalmente mejor, mientras se cubra el resto con comunicación personal". Jim Highsmith da el consejo de comenzar con menos de lo que se piensa

necesitar y probablemente eso sea todo lo necesario; es mas fácil agregar algo después que quitar algo.

- **La formalidad, el proceso y la documentación no son sustitutos de disciplina, habilidad y entendimiento**. Este principio de Jim Highsmith (2003) articula la diferencia entre metodologías ágiles y tradicionales. El software lo construyen personas y la disciplina, la habilidad y el entendimiento son propiedades internas de una persona y no pueden remplazarse por una forma externa.
- **Comunicación interactiva, cara a cara es el canal más barato y rápido para intercambiar información**. Por ejemplo, dos o tres personas paradas frente a una pizarra, diagramando y hablando pueden tomar ventaja de saludables canales de comunicación y obtener retroalimentación casi instantánea.
- **Aumentar la retroalimentación y la comunicación reduce la necesidad de entregables intermedios**. Si el equipo desarrolla algo y lo muestra al usuario, por ejemplo cada mes, el tiempo de demora es relativamente pequeño y no es necesario hacer promesas elaboradas sino mostrar el resultado del mes y aprender directamente lo que está bien y corregir lo que es incorrecto.
- **El costo de desarrollo concurrente y serial entrecruzan por velocidad y flexibilidad**. Desarrollar de manera concurrente puede hacer más rápido el desarrollo a un costo posiblemente superior comparado con un desarrollo serial ejecutado correctamente. El problema del costo más bajo es que cualquier causa de mala interpretación implica rehacer trabajo, que es muy caro. El desarrollo concurrente permite descubrir errores de interpretación en tiempo real, pero al mismo tiempo requiere mejor comunicación entre las personas
- **La eficiencia es prescindible en actividades que no son cuello de botella**. El libro, *The Goal* (Goldratt 1992), identifica que cada proceso tiene una estación "cuello de botella", que condiciona la velocidad de toda la empresa. La familia Crystal agrega el corolario que las estaciones que no son cuello de botella pueden ayudar en ciertas formas, operando con menos eficiencia. Si las personas tienen capacidades libres pueden ayudar en otras actividades.
- **"Puntos dulces (sweet spots)" aceleran el desarrollo**. El mejor de los mundos es tener personas (1) dedicadas, (2) expertas que (3) se sientan de manera que puedan oírse, (4) usan pruebas de regresión automatizadas, (5) tienen acceso fácil a los usuarios, y (6) entregan sistemas corriendo, ya probados a esos usuarios cada mes o dos. Tal proyecto está en una posición mayor para tener éxito que otro donde falten estas características. Crystal Clear se construye sobre las cuatro últimas ya que no se puede contar con todo el personal capacitado y dedicado.

Ya que es difícil recordar los 10 principios, ellos se derivan de una idea, la del juego cooperación –económico (descripto en [Cockburn]). El manifiesto del juego cooperativo dice:

El desarrollo de software es una serie de recursos limitados, juegos cooperativos de objetivos directos de invención y comunicación. El objetivo primario de cada juego es la producción y liberación de un sistema de software; el residuo del juego es un conjunto de marcas para ayudar a los jugadores en el próximo juego. El siguiente juego es una alteración al sistema o la creación de sistemas vecinos. Cada juego por lo tanto tiene como segundo objetivo crear posiciones ventajosas para el siguiente juego. Dado que cada juego tiene recursos limitados, el objetivo principal y el secundario compiten por los recursos. [Cockburn]

La ventaja de esta expresión es que resalta un par de aspectos importantes del desarrollo de software:

- Son las *personas* y su *invención* y *comunicación* que hacen que surja el software.
- Hay dos objetivos en juego en cada instante: entregar el sistema actual y definir las bases para el siguiente juego. Cada decisión, realizadas por cualquier persona involucrada, tiene consecuencias económicas. La mayoría de las recomendaciones de esta metodología están basadas en este manifiesto del juego cooperativo.

El desarrollo de software es un juego de personas trabajando con y contra personas. Todos los motivos y emociones humanas se aplican y deben tenerse en cuenta, aún en Crystal.

La Familia Crystal

Crystal es una familia de metodologías con un código genético común, uno que enfatiza entregas frecuentes, comunicación cercana y mejora reflexiva. No hay una metodología Crystal. Hay diferentes metodologías Crystal para diferentes tipos de proyectos. Cada proyecto u organización usa el código genético para generar nuevos miembros de la familia.

El nombre "Crystal" viene de la caracterización de Alistair Cockburn de los proyectos en dos dimensiones, tamaño y criticidad, comparando con los minerales, color y dureza (ver Figura 6.1). Grandes proyectos, que requieren más coordinación y comunicación, se mapean a colores más oscuros (Clear, amarillo, naranja, rojo, y así siguiendo). Proyectos para sistemas que pueden causar más daño necesitan agregar dureza o rigidez a la metodología, así como más reglas de validación y verificación. Una metodología de cuarzo es apropiada para pocos desarrolladores creando un sistema de facturación. El mismo equipo controlando los movimientos de barras de boro en un reactor nuclear necesita una metodología de diamante, que establezca chequeos repetidos tanto en el diseño como en la implementación de sus algoritmos.

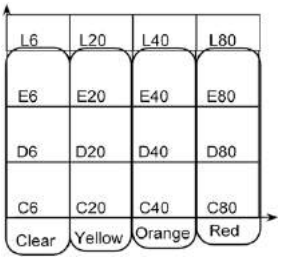


Figura 6.1: Cobertura de diferentes tipos de proyectos dentro de Crystal. [Cockburn]

El autor caracteriza a las metodologías Crystal por color, según el número de personas que son coordinadas: *Clear* es para equipos de 8 personas o menos ubicadas en un mismo lugar, *Amarillo* es para equipos de 10 - 20 personas, Naranja es para equipos de 20 - 50 personas, *Rojo* para equipos de 50 - 100 personas, y así sucesivamente, pasando por el *Marrón*, *Azul*, *Violeta*.

El código genético de Crystal

El código genético de Crystal está compuesto por:

- El modelo de juego económico-cooperativo (The economic-cooperative game model)
- Prioridades Seleccionadas
- Propiedades Seleccionadas
- Principios Seleccionados
- Técnicas de Muestreo Seleccionadas
- Ejemplos de Proyecto

El *modelo de juego económico-cooperativo,* ya explicado en páginas anteriores, conduce a las personas en un proyecto a pensar sobre su trabajo de una manera muy específica, enfocada y efectiva.

Las *prioridades* comunes a la familia Crystal son

- Seguridad en el resultado del proyecto: obtener un resultado de negocio razonable, dadas las prioridades del proyecto y las restricciones de recurso
- Eficiencia en el desarrollo es una prioridad alta porque muchos proyectos están económicamente sobre-exigidas
- Habitabilidad (*habitability*) de las convenciones (las reglas necesitan ser tales que las personas dentro del equipo puedan vivir con ellas y no las ignoren)

Las prioridades interactúan entre sí. La *habitablilidad* significa que se acepta que un conjunto de reglas dado puede no ser lo más *eficiente* posible. La Prioridad de *seguridad* significa que las reglas elegidas deben ser lo suficientemente buenas para obtener resultados adecuados. Parte de la eficiencia y habitabilidad es enfrentarse al hecho de que cada proyecto es levemente diferente y necesita su propia metodología hecha a la medida. Cada equipo adapta la metodología base según sus requerimientos, experiencia, y características tecnológicas. Esto debe hacerse rápidamente, para que no sea costoso. Para poder definir un conjunto de reglas básicas para un proyecto en particular, Alistair Cockburn construye una tabla de dos entradas, como se mostró en la **Figura 6.1.**

Crystal conduce al equipo del proyecto hacia siete *propiedades* de seguridad, las tres primeras son el corazón de Crystal. Las otras pueden agregarse para aumentar el margen de seguridad. Las propiedades son:

- Entregas Frecuentes
- Mejora reflexiva
- Comunicación cercana
- Seguridad del Personal (el primer paso en la confianza)
- Foco
- Facilidad de acceso a usuarios expertos
- Entorno técnico con pruebas automatizadas, administración de configuración e integración frecuente

Los *principios* de Crystal se describen en detalle en Agile Software Development [Cockburn2002]. Entre ellos hay unas cuantas ideas centrales:

- El nivel de detalle necesario en los documentos de requerimientos, diseño y planificación varía con las circunstancias del proyecto, específicamente la extensión del daño que podría causarse por no detectar defectos y la frecuencia de la colaboración del personal aportada al equipo.

- Podría no ser posible eliminar todos los productos intermedios y documentos legales como documentos de requerimientos, diseño y planes de proyecto; pero pueden reducirse ya que hay caminos de comunicación cortos, ricos e informales en el equipo y se entrega de manera temprana y frecuente software funcionando y probado.

- El equipo continuamente ajusta sus convenciones de trabajo para adaptarse a las personalidades particulares en el equipo, el entorno actual de trabajo local y las peculiaridades de la asignación específica.

Crystal no requiere ninguna técnica específica a utilizar, solo desarrollo incremental. Por lo tanto, el conjunto permitido de *estrategias y técnicas* es muy amplio. Sin embargo, el autor presenta un conjunto de técnicas que se incluyen como un conjunto inicial.

Crystal Clear y la familia Crystal

Cada miembro de la familia Crystal se genera al inicio del proyecto dándole forma a una metodología base según el código genético. Como la situación cambia a través del tiempo, la metodología se afina durante el transcurso del proyecto. Tanto las tareas para darle forma como para afinarla se realizan lo suficientemente rápidas de tal forma que el tiempo empleado se recupera dentro del marco del tiempo del proyecto.

Crystal Clear es una optimización de Crystal que puede aplicarse cuando el equipo consiste de 2 a 8 personas sentadas en la misma habitación u oficinas adyacentes. La propiedad de comunicación cercana se refuerza a comunicación "osmótica", significando que las personas escuchan a los otros discutiendo prioridades, estado, requerimientos, y diseño de proyecto diariamente. Esta comunicación reforzada permite al equipo trabajar más desde una comunicación tácita y notas pequeñas que de otra forma no sería posible.

Crystal Clear comparte algunas características con XP pero es generalmente menos demandante. Es una alternativa más relajada, un lugar para volver si XP no está funcionando para el grupo, o un trampolín para obtener algunas prácticas ágiles antes de saltar a XP.

Roles en Crystal Clear

Hay ocho roles definidos para Crystal Clear (sponsor ejecutivo, usuario experto, diseñador-líder, diseñador-programador, experto del negocio, coordinador, tester, writer), donde los cuatro primeros probablemente deberían ser personas diferentes. Los otros cuatro pueden ser roles adicionales asignados a personas del proyecto.

- **Sponsor Ejecutivo:** Es la persona que provee el dinero al proyecto o quien representa a esa persona. Debe mantener la visión a largo plazo en mente, balanceando las prioridades a corto plazo con aquellas de entregas posteriores o evoluciones del equipo o mantenimiento del sistema. Esta persona creará la visibilidad externa para el proyecto y proveerá al equipo con decisiones cruciales a nivel del negocio. Parte de la metodología involucra entregar al sponsor ejecutivo buena información para tomar esas decisiones.
- **Usuario Experto:** Es la persona que se supone está familiarizada con los procedimientos operacionales y el sistema en uso (si hay alguno), conociendo cuáles son modos frecuentes e infrecuentes de operación, qué atajos se necesitan, y qué información debe verse junta en la pantalla al mismo tiempo.
- **Diseñador líder:** Es la persona técnica líder, la persona que se supone tiene experiencia con el desarrollo de software, capaz de realizar el diseño mayor del sistema, que avisa cuando el equipo de proyecto está o no en cronograma, y si no lo está como volver al cronograma. El líder de diseño suele tener mayor influencia en el taller para darle forma a la metodología. Alistair usa la palabra "diseñador" para este rol para cortar el nombre "líder diseñador-programador." El diseñador líder debe diseñar y programar como los otros diseñadores-programadores.
- **Diseñador-Programador:** Al definir este rol, Alistair combina ambas palabras “diseñador” y “programador” para resaltar que cada persona programa y diseña. Ni diseñador ni programador se mantienen como nombre de un rol separado.
- **Coordinador:** La coordinación es probablemente una ocupación parcial de algún miembro del equipo. La persona que ocupa el rol de coordinador debe, como mínimo, tomar nota en la planificación del proyecto y las sesiones de estado del proyecto, así rastrear la información para enviar o presentar. El coordinador es responsable de dar, a los sponsors del proyecto, visibilidad sobre la estructura y el estado del proyecto.
- **Experto de negocio**: Es el experto sobre cómo funciona el negocio, qué estrategias o políticas son fijas, cuales pueden variar pronto, a menudo o de vez en cuando. La información que provee es diferente a la que típicamente puede proveer un usuario experto, aunque en algunos casos puede ocurrir que el usuario experto sea también el experto del negocio.
- **Tester y writer**: suelen ser asignaciones temporales que van rotando. Algunos equipos pueden contratar un writer por períodos de tiempo o tener un tester dedicado trabajando e inclusive, sentado con ellos.

Los productos de trabajo

Los productos de trabajo no son ni completamente requeridos ni tampoco completamente opcionales. Pueden usarse uno a la vez o todos juntos. Se permite una sustitución equivalente, ya que es una forma de ajustarlo y variarlo.

En una metodología pequeña como Clear, el número y la formalidad de productos de trabajo intermedios se reduce significativamente. El equipo vive de la comunicación personal, notas, pizarras o posters alrededor del cuarto, y las demos o entregables en su comunicación con el usuario.

Los productos de trabajo descriptos aquí son un conjunto por defecto para un proyecto típico de Crystal Clear porque han demostrado su valor en muchos proyectos. Pero, depende del equipo del proyecto agregar, sacar o modificar la lista basados en su situación. Cada ítem sirve

a un propósito de comunicación en el juego económico-cooperativo. Hay casi 24 productos de trabajo, dependiendo de cómo se los cuente. Los productos de trabajo aquí son livianos y están distribuidos a través de los roles, así el equipo no lo encuentra agobiante en la práctica.

No se puede definir un grupo de productos que sea el corazón. Si se saltean muchos productos de trabajo se pierde alineación y visibilidad. La dirección y apoyo pueden sufrir por ello. Por otro lado, no sería lo más apropiado insistir en realizar todos ya que los productos de trabajo específicos que necesita el equipo varían según técnicas, tecnologías, hábitos de comunicación y moda que todas son cambiantes. El desacuerdo dentro del equipo y la falta de comunicación con el mundo exterior aumenta con cada omisión. Los riesgos se acumulan, lentamente al principio, hasta que el proyecto no está en zona segura, y nunca está claro cuándo pasa de zona segura a no segura.

A continuación se presenta una tabla indicando que rol es el último responsable de cada producto. Muchos productos tienen una forma informal y otra formal de utilizarlos. Los productos de Crystal se describen en detalle en [Cockburn].

Rol- último responsable	Productos
El sponsor (patrocinador, quien financia)	La declaración de la Misión con el Trade-off de prioridades.
El equipo	La estructura y las convenciones del equipo Los resultados del trabajo de reflexión.
El coordinador, con ayuda del equipo	El Mapa del Proyecto, El Plan de Entrega, El Estado del Proyecto, La Lista de Riesgo, El plan y Estado de la Iteración La visualización del Calendario -Cronograma.
El experto del negocio y usuario experto juntos	La lista de objetivos por actor: Los Casos de Uso, El archivo de Requerimientos: El modelo del rol del usuario
El líder de diseño (diseñador líder)	La descripción de la Arquitectura.
Los diseñadores-programadores (incluyendo al líder de diseño)	Borradores de pantalla, Modelo de Dominio Común, Esquemas y notas de diseño, Código fuente, Código de Migración, Las Pruebas El sistema empaquetado.
EL tester	Reporte de errores en ese momento
El writer	El texto de la ayuda al usuario.

Tabla 6.1: Roles y productos en Crystal (basado en [Cockburn])

Proceso Crystal Clear

Tipos de Procesos Involucrados

Crystal Clear usa procesos cíclicos anidados de varias longitudes: el episodio de desarrollo, la iteración, el período de entrega y el proyecto completo. Lo que las personas realizan en cada momento depende en qué ciclo están.

Se da el nombre de *Desarrollo concurrente* cuando el trabajo se realiza en paralelo en productos de trabajo dependientes. Muchas o todas las actividades se realizan al mismo tiempo. Las personas intercambian notas continuamente para mantenerse al día con el estado cambiante de cada parte.

Se definen siete ciclos que se encuentran en la mayoría de los proyectos:

1. El proyecto (una unidad de fondos, que puede ser de cualquier duración)
2. El ciclo de entrega (una unidad de entrega, una semana a tres meses)

3. La iteración (una unidad de estimación, desarrollo y celebración, una semana a tres meses)
4. La semana de trabajo (ritmo calendario por ejemplo reuniones de los lunes)
5. El período de integración (una unidad de desarrollo, integración y prueba de sistema, 30 minutos a tres días)
6. El día de trabajo (ritmo calendario por ejemplo daily stand up meeting)
7. El *episodio* de trabajo (desarrollando y chequeando en una sección de código, tomando de unos minutos a unas horas)

Crystal Clear requiere múltiples entregas por proyecto, pero no múltiples iteraciones por entrega. Cada ciclo tiene su propia secuencia y su propio ritmo. En un día dado se realizan diferentes actividades de diferentes ciclos.

Las **Figuras 6.3** a **6.5** muestran diferentes formas de desenrollar estos ciclos. Se puede notar que episodios, días y períodos de integración pueden anidarse de diferente forma dentro de una iteración (**Figuras 6.3** y **6.4** muestran dos formas posibles).

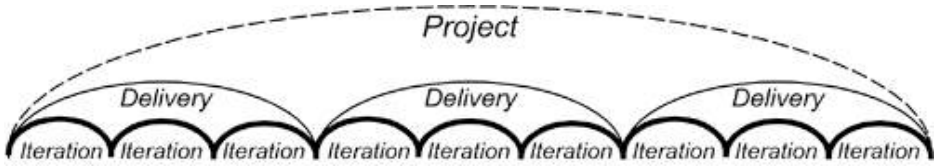


Figura 6.2: Iteración y ciclos de entrega dentro de un proyecto [Cockburn]

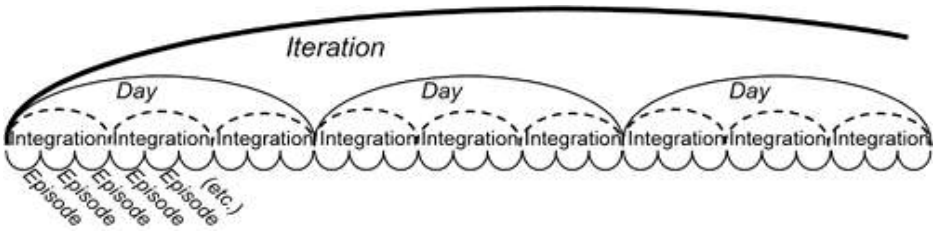


Figure 6.3: Un ciclo de iteración, integrando varias veces por día [Cockburn]

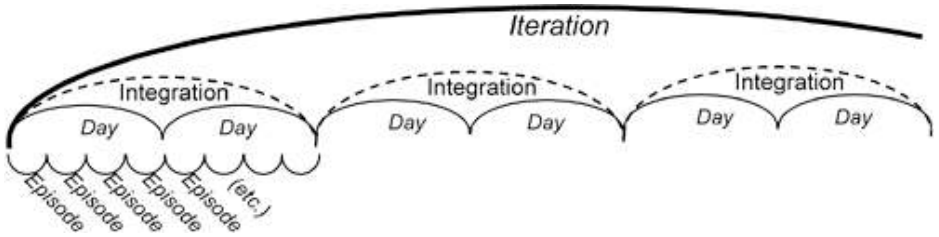


Figure 6.4: Un ciclo de iteración con varios días por integración [Cockburn].

La **Figura 6.5** muestra la expansión de cada ciclo de forma separada, listando las actividades específicas que ocurren para ese tipo de ciclo

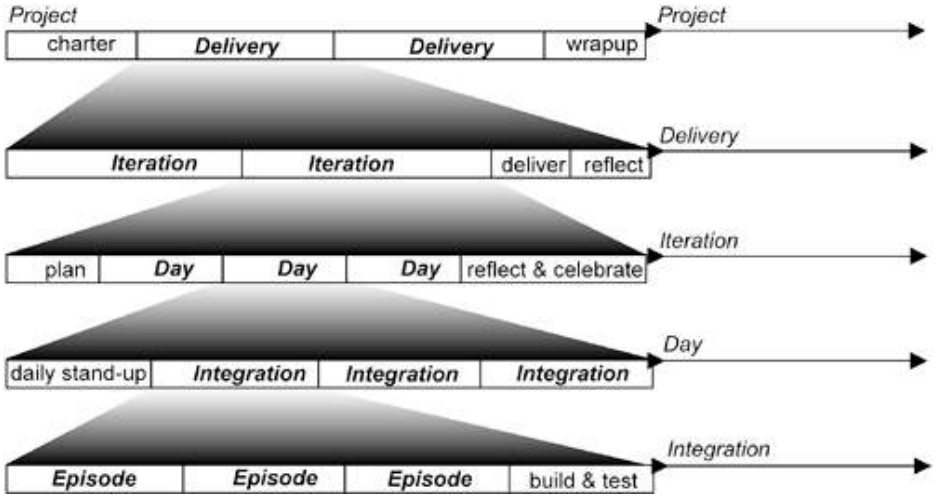


Figura 6.5: Los ciclos expandidos para mostrar actividades específicas. [Cockburn]

De los ciclos antes mencionados se ampliarán solo algunos de ellos que requieren mayor explicación.

El ciclo del Proyecto

Aunque un proyecto se crea una vez para todas las actividades, es seguido por otro proyecto, en un ciclo que se repite. Tiene tres partes:

- **Una actividad de caracterización** (*chartering*): Esta actividad toma de unos pocos días a unas semanas. Consiste de cuatro pasos: Definir el corazón del equipo, realizar la *exploración de 360°* (pudiendo resultar en la cancelación del proyecto), dar forma y afinar las convenciones de la metodología y construir el plan inicial del proyecto
- ***Dos o más* ciclos de entrega** Este ciclo tiene tres o cuatro partes: una re-calibración del plan de entregas, una serie de una o más iteraciones, cada una resultando en código integrado y probado, entrega a usuarios reales y un ritual de finalización, incluyendo reflexión tanto en el producto que se está creando como en las convenciones que se usan. Se resalta que el tener solo un ciclo de entrega es una violación a Crystal Clear.
- **Un ritual de finalización**: el empaquetado del proyecto Es por eso que después de una entrega se debe proporcionar un tiempo de relajación por las presiones del ciclo. Después de la entrega el equipo tiene dos aspectos más para reflexionar:

  1. ¿Cómo fue la distribución? ¿Qué se podría hacer para reducir los padecimientos a la hora de distribuirlo a los usuarios y entrenarlos?
  2. ¿Qué piensan los usuarios sobre el sistema? ¿Cuáles son los puntos fuertes y débiles? ¿Se puede aprender algo de lo que realmente necesitan los usuarios respecto de lo solicitado originalmente?

  La reflexión en el proceso de entrega es la misma que para cualquier otro workshop de reflexión. Se pregunta el grupo, lo que quieren mantener o hacer diferente. El punto a destacar es que se revisa el *producto*, no el proceso.

El Ciclo de Iteración

La duración y el formato de las iteraciones varían según los equipos. Podría considerarse una duración desde 1 semana a 2 meses. Una iteración tiene tres partes:

1. Planificación de la iteración
2. Actividades diarias y de ciclo de integración

3. Ritual de finalización (workshop de reflexión y celebración)

Durante el período de la iteración, el equipo estará agregando requerimientos, evaluando diseños de interfaz de usuario, entendiendo la arquitectura del sistema, agregando funcionalidad, mostrando el sistema al usuario, agregando pruebas.

En la *planificación* se definen las prioridades de la iteración. Se divide el trabajo en pequeñas partes. La duración variará si es una iteración de una semana o de dos meses. Cada día el equipo tiene reunión diaria de *check-in* or *stand-up* meeting. La esencia es mantener la reunión corta (a lo sumo 10 minutos) y permitir a cada miembro saber qué le está sucediendo a los otros miembros del equipo.

Luego, el equipo simplemente realiza sus actividades normales de desarrollo. Los episodios de *desarrollo* consisten en simplemente tomar una asignación de trabajo, desarrollarla y chequearla respecto al sistema de administración de configuración, más realizar la integración y pruebas de sistema, si se usa esa convención en el equipo. Probablemente discutan y muestren su trabajo al sponsor o usuario experto.

Si se trabaja con períodos muy cortos, el equipo suele olvidarse de llevar el sistema a usuarios reales. Se puede agregar un súper ciclo, que incluya visitas del usuario o sino crear explícitamente una agenda de Visitas del usuario. Si por el contrario se trabaja en ciclos muy largos, es recomendable tener un workshop de reflexión intermedio ya que permite descubrir si se ha detectado algo que pudiera poner en riesgo el éxito de la iteración.

Dentro del *ritual de finalización*, se realiza el workshop de reflexión. Se debe revisar cada aspecto de sus trabajos, desde sus relaciones con el sponsor y usuarios, hasta patrones de comunicación, hostilidad, la forma de recolectar requerimientos, convenciones de codificación, el entrenamiento que obtienen o no obtienen, nuevas técnicas que quisieran probar, etc. En iteraciones muy cortas es mejor realizarlos mensualmente o cada dos meses, para que se reflexione sobre un periodo mayor (Notar que esto crea el súper ciclo mencionado anteriormente). El workshop de fin de ciclo brinda un tiempo para reflexionar en lo que consideran positivo y negativo a cerca de sus hábitos de trabajo.

Después de la primera iteración o ciclo de entrega, muy a menudo los equipos ajustan sus estándares, obtienen más entrenamiento, hacen más eficientes sus flujos de trabajo, aumentan las pruebas, encuentran un “usuario amigable”, definen las convenciones de administración de configuración. Al final de ciclos subsiguientes, sus cambios tienden a ser mucho menores, a menos que estén experimentando con un proceso radicalmente nuevo.

El punto de esta reflexión periódica es atrapar errores a tiempo para poder repararlos dentro del proyecto y brindar la oportunidad a las personas de notar patrones ineficientes.

Se suelen realizar también actividades que logren descomprimir la saturación que tiene el equipo. Se pude utilizar una serie de juegos de computadora, juegos de ping pong, una discusión ligera sobre algún tópico profesional, un paseo en bicicleta, una salida a una confitería.

El episodio de distribución

Ward Cunningham acuño el término *episodio* para describir la unidad básica de trabajo de un programador en un desarrollo ágil. Durante un episodio, una persona toma una tarea de diseño pequeña, la programa hasta su finalización (idealmente con pruebas de unidad), y chequea respecto al sistema de administración de configuración. Esto puede tomar de 15 minutos a varios días, dependiendo del programador y las convenciones del proyecto. Funciona mejor mantener los episodios con una duración menor a un día.

Para Crystal Clear programar y diseñar no vale la pena discriminarlos, van juntos. La frase “tomar una tarea de diseño pequeña, programarla hasta su finalización” significa tomar una tarea pequeña que requiere diseño y programación, luego diseñar, programar, depurar y probarla hasta la finalización.

Reflexión sobre el Proceso

La vista de ciclos anidados permite la discusión de progreso y actividades de operaciones. Ambas son importantes para el equipo y forman parte del "proceso". En Crystal Clear, está permitido tener una iteración por ciclo de entrega, pero si se realiza esto, se deben tener *visitas* intermedias de usuarios reales. Si se tienen múltiples iteraciones por entrega, algunas de ellas deben incluir *visitas* de usuarios reales. Si no se realiza esto se está armando un equipo efectivo que produce el software equivocado muy eficientemente.

Conclusiones

Esta metodología se centra en las personas, la interacción, comunicación directa, habilidades, talentos con la convicción que son éstos quienes tienen el efecto mayor en el desempeño, dejando al proceso en un segundo lugar [Highsmith en *Agile Software Development Ecosystems*]. Cada equipo tiene un conjunto distinto de talentos y habilidades y por lo tanto cada equipo debe utilizar un proceso de desarrollo adaptado a él. Minimiza el proceso significativamente, una de las razones para realizar ésto es que exige que la ubicación del equipo esté en un solo lugar físico.

Es una metodología ágil con la cual el equipo no solo va haciendo evolucionar los entregables sino el proceso mismo de desarrollo. Tal vez la contribución más importante es ofrecer las *Siete Propiedades de los Proyectos Exitosos*. Alistair ha estudiado proyectos ágiles exitosos y ha identificado rasgos comunes entre ellos. Estas propiedades llevan al proyecto al éxito; en cambio su ausencia pone en riesgo el proyecto. [Cockburn]

A diferencia de XP, para el autor de esta propuesta, si todos agregan y modifican código puede quedar desprolijo y llevar tiempo sacar mucha basura. Es mejor tener dueños de clases o casos de uso. Cualquiera puede hacer un cambio a corto plazo pero debe notificar al dueño para que lo revise.

7. TDD (TEST DRIVEN DEVELOPMENT)

Test Driven Development, TDD o desarrollo orientado a las pruebas, es la técnica por antonomasia de XP, como se detalló en la sección sobre XP. Se encuadra dentro del ciclo de vida de la metodología XP, en las fases de iteración, producción y mantenimiento. Según Fowler, [FowlerSitio4], TDD es una técnica que conduce el proceso de desarrollo a través de pruebas. Pero, debido a su importancia, mucha gente ya la considera una metodología independiente de XP. Tal como lo afirma Carvajal Riola, se puede considerar a TDD una metodología ya que presenta un conjunto de prácticas y métodos de desarrollo, además de que condiciona la mentalidad de los desarrolladores guiándolos a través del desarrollo y aumentando la calidad del producto final. Puede ser aplicada independientemente a XP, en proyecto con Scrum, FDD o metodologías tradicionales. Debido a su radical planteamiento a la hora de escribir código, tal como se afirma en [Carvajal], cambia drásticamente la mentalidad de cualquier equipo de desarrollo, generalmente agilizando los resultados y aumentando la calidad del sistema.

TDD [Beck2003] [Astels], es un enfoque de desarrollo evolutivo que combina *TFD* (Test-First Development) donde se escribe una prueba justo antes de escribir el código de producción que cumpla con esa prueba y *Refactorización*. Para algunos el objetivo de TDD es la especificación y no la validación [Martin2003]. Es una manera de pensar a través de los requerimientos o diseño antes de escribir código funcional (implicando que TDD es una técnica importante para requerimientos ágiles y para diseño ágil). Para otros TDD es una técnica de programación. Ron Jeffries afirma que el objetivo de TDD es escribir código limpio que funcione. Scott Amber adhiere a la primera línea de pensamiento. [AgileData]

Procesos.

En esencia se siguen tres simples pasos que se repiten dentro de TDD:

- Escribir una prueba para la siguiente porción de funcionalidad que se pretende incorporar
- Escribir el código funcional hasta que pase la prueba
- Refactorizar tanto el código viejo como el nuevo para que esté bien estructurado

Se sigue iterando paso a paso, con una prueba por vez, construyendo la funcionalidad del sistema.

Programar la prueba primero (Test First Programming) provee dos beneficios principales. En primer lugar es una forma de tener código auto-probado ya que solo se escribe código funcional para que una prueba sea superada. El segundo beneficio es que al pensar en las pruebas primero fuerza a pensar acerca de la interfaz con el código primero. Esto hace focalizarse en la interfaz y en cómo usar una clase ayudando a separar la interfaz de la implementación [FowlerSitio4]

Según David Astels [Astels] TDD es un estilo de desarrollo donde mantienes un juego de pruebas del programador exhaustivo, ninguna parte del código pasa a producción a no ser que pase sus juegos asociados, primero se escriben las pruebas y las pruebas determinan el código que se necesita escribir para que puedan pasarse las pruebas.

Podrían simplificarse los pasos con el siguiente gráfico

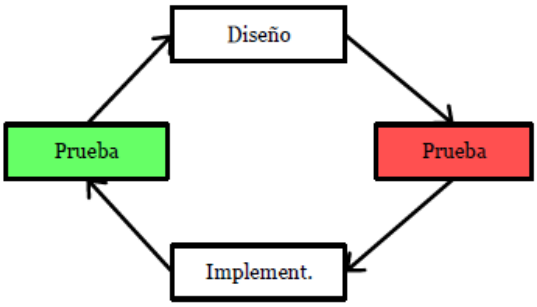


Figura 7.1: Proceso simplificado TDD [Kirrily]

Amber utiliza una simple fórmula para explicar en qué consiste TDD, y es la siguiente:

TDD = Refactoring + TFD

TDD cambia completamente el desarrollo tradicional. Al tratar de implementar una nueva funcionalidad primero se pregunta si el diseño actual es el mejor diseño posible para permitir implementar la funcionalidad. En caso afirmativo se procede con un enfoque TFD. En caso contrario se realiza una refactorización localmente para cambiar la porción de diseño afectada por la nueva característica, permitiendo agregar la característica lo más simple posible. Como resultado, siempre se mejorará la calidad del diseño, haciendo más fácil el trabajo futuro. [Ambler en AgileData]

Los pasos para TFD pueden verse en un diagrama de actividad, en la siguiente figura. El primer paso es agregar rápidamente una prueba, básicamente código suficiente para que falle. Luego se corre la prueba para asegurar que falle realmente. Luego se actualiza el código funcional para que pueda pasar las nuevas pruebas. El cuarto paso es correr las pruebas nuevamente. Si fallan se debe actualizar el código funcional y volver a probar. Una vez que las pruebas pasan, se comienza nuevamente (tal vez se deba re-factorizar cualquier duplicación del diseño, volviendo a TFD en TDD).

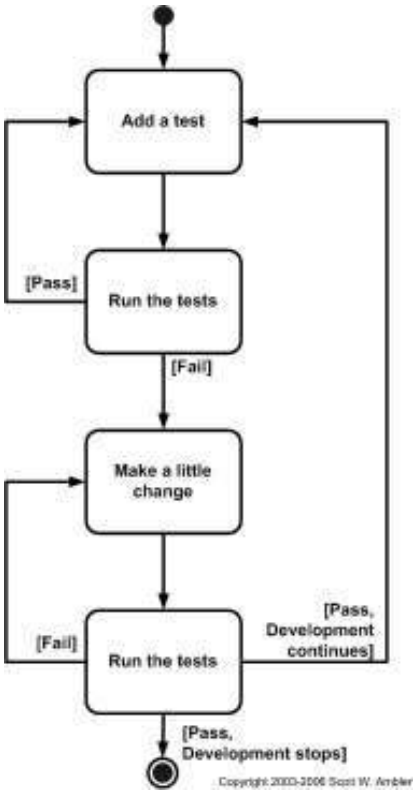


Figura 7.2: Proceso de TFD [AgileData]

Principios de TDD:

Existen cuatro principios subyacentes en TDD:

- Mantener un juego exhaustivo de pruebas del programador.

Las pruebas del programador prueban que las clases se comporten de la forma esperada, son escritas por los desarrolladores que escriben el código que será probado. Se llaman pruebas del programador porque aunque se parecen mucho a las pruebas unitarias, se escriben por diferentes motivos. Las pruebas unitarias se escriben para demostrar que el código escrito funciona, las pruebas del programador son escritas para definir qué significa que el código funcione. También se llaman así para diferenciarlas de las pruebas escritas por los usuarios.

En TDD se dispone de un gran juego de pruebas, esto es así porque no puede haber código si no existen las pruebas que los prueben. Primero se escriben las pruebas y luego el código que será probado, no hay por tanto código sin pruebas, concluyendo que las pruebas son por tanto exhaustivas.

- Todo código que pasa a producción tiene sus pruebas asociadas.

Esta característica recuerda una propiedad fundamental en TDD y es que una funcionalidad no existe hasta que existe un juego de pruebas que vaya con él. Esta propiedad brinda la oportunidad de utilizar un par de técnicas muy comunes como son el refactoring y la integración continua. Ambas técnicas solo pueden ser ejecutadas si realmente se está seguro de que el código sigue cumpliendo las características definidas en las pruebas.

- Escribir las pruebas primero.

Cuando se tiene una nueva funcionalidad, antes de implementarla se debe escribir sus pruebas. Esta es una de las características más “extremas” de TDD. La manera de proceder es escribir unas pruebas pequeñas y entonces, escribir un poco de código que las ejecute y las pase, entonces otra vez se amplían las pruebas, y se vuelve a escribir código, y así sucesivamente.

- Las pruebas determinan el código que tienes que escribir.

Se está limitando la escritura de código a tan solo la prueba que ya se tiene implementada. Solo se escribe lo necesario para pasar la prueba, esto quiere decir que se hace la cosa más sencilla para que funcione.

Roles y responsabilidades.

Los dos roles que como mínimo debe disponer todo proyecto con TDD son clientes y desarrollador

- **Cliente**: Desarrolla las historias con los desarrolladores, las ordena según la prioridad y escribe las pruebas funcionales.
- **Desarrollador:** Desarrolla las historias con el usuario, las estima, y entonces toma responsabilidad de su desarrollo utilizando TDD, lo que quiere decir que ejecuta las pruebas, implementa y refactoriza.

Prácticas.

Existen dos prácticas muy vinculadas con TDD, que forman parte de XP. Sin estas prácticas TDD no podrían realizarse o sería casi una utopía. Las prácticas son: refactoring e integración continua. Ambas fueron descriptas en la sección de XP.

Refactoring.

Esta actividad está muy relacionada con TDD ya que muchas veces se duplica código y se introduce mucha "basura" cuando se está intentando programar algo para que pase unas pruebas específicas. Se suele hacer refactorización cuando existe código duplicado, cuando se percibe que el código no está lo suficientemente claro o cuando parece que el código tiene algún problema. Luego de refactorizar se debe correr un juego de pruebas para verificar que no se introdujeron cambios al comportamiento.

Fowler, [FowlerSitio4] recalca que la refactorización es clave en TDD ya que mantiene limpio al código, sino se terminaría con una mezcla de agregaciones de fragmentos de código

Integración continúa.

Del mismo modo que con la técnica de refactoring, está íntimamente ligada a TDD, ya que cada vez que se realiza una integración un juego de pruebas exhaustivo se ejecuta y valida el código introducido. Sin estas pruebas, la integración continua no tendría sentido, ya que no garantizaría ni la cohesión, ni la validez del código integrado.

Herramientas

Algunas herramientas fundamentales en el uso de TDD, ya que sin ellas sería imposible realizar las tareas que la metodología propone, son.

- **Pruebas unitarias**: En los últimos años TDD ha popularizado el uso de la familia de herramientas xUnit, que son herramientas que permiten la ejecución de pruebas unitarias de forma automática.
- **Repositorios de código**: Es una herramienta obligatoria para el uso de la metodología TDD. Las más conocidas son CVS o Subversion y facilitan el control de versiones, acceso e integración de código.
- **Software de integración continúa**: Existen diferentes aplicaciones web como Hudson, Apache Gump, Cruise Control, etc, el cometido de las cuales es construir la aplicación a partir del código fuente, normalmente situado en un repositorio de código (subversión,cvs,...) y ejecutar las pruebas especificadas. Si las pruebas son satisfactorias, se despliega en el entorno de producción, en caso contrario no se incluyen las modificaciones. Este es un proceso que se puede ejecutar varias veces a lo largo del día y garantiza que se realiza la técnica de integración continua.
- **Herramientas de construcción automáticas**: Ayudan y mucho la utilización de herramientas como maven o Ant para la compilación y ejecución de las pruebas automáticamente.

Conclusiones

Test Driven Development es una de las metodologías con mayor acogida en el campo profesional y que continúa expandiéndose debido a sus buenos resultados. La tendencia actual es integrar TDD independientemente en cualquier metodología ya sea ágil [Schmidkonz] o tradicional [Leteiler2003] y aprovechar los beneficios de practicar una metodología que siempre permite deshacer los errores, asegurar una calidad del producto y protegerse de errores tanto malintencionados como humanos.

Dr. Hakan Erdogmus, editor jefe de IEEE Software [Hartmann], comenta que TDD a veces es entendido como un procedimiento de aseguramiento de lo calidad, provocando que algunos administradores no lo utilicen porque creen que sus equipos ya tienen otros procedimientos que garantizan la calidad. Hakan hace hincapié en que originalmente TDD fue pensado como una técnica para mejorar la productividad y que el aumento de la calidad es un efecto secundario, si bien actualmente se reconoce a TDD como una metodología ágil en si misma.

8. DSDM (DYNAMIC SYSTEMS DEVELOPMENT METHOD)

Introducción

Unos años antes de la aparición de XP, en enero de 1994, se reunieron en Londres, Reino Unido, un grupo de profesionales para discutir sobre la creación de un proceso iterativo normalizado para el desarrollo RAD. RAD son las siglas de Rapid Application Development, un método de desarrollo creado por James Martin que se caracteriza por usar un ciclo de vida iterativo, prototipos y herramientas CASE (Computer Aided Software Engineering). Se formó un consorcio sin fines de lucro e independiente. Este consorcio se dedicó a entender las mejores prácticas en el desarrollo de aplicaciones y codificándola de tal forma que fuera ampliamente enseñado e implementado. [Stapleton]. El objetivo era desarrollar y promover de manera conjunta un framework RAD independiente combinando sus mejores experiencias prácticas.

El resultado del trabajo de este consorcio, después de más de un año, fue DSDM (Dynamic Systems Development Method), una forma de desarrollar sistemas de aplicación que realmente satisfagan las necesidades del negocio. [Stapleton]. La versión 1 fue publicada en febrero de 1995. Este resultado fue un método genérico que cubre personas, proceso y herramientas y que fue formado a partir de las experiencias de las organizaciones de todo tipo de sector y tamaño. DSDM Consortium es dueña y administra el framework DSDM y solo sus miembros pueden emplearlo con fines comerciales. [SitioNavegapolis]. Habiendo empezado con 17 fundadores ahora tiene más de mil miembros y ha crecido fuera de sus raíces británicas. Siendo desarrollado por un consorcio, tiene un sabor diferente a muchos de los otros métodos ágiles. Tiene una organización de tiempo completo que lo apoya con manuales, cursos de entrenamiento, programas de certificación y demás.

DSDM contempla el ciclo de vida iterativo e incremental, involucrar continuamente al usuario y la adaptación al cambio. [Frankel]. Tal como lo expone claramente [Caine], es la única metodología ágil que cubre el ciclo de vida del proyecto entero. Incorpora disciplinas de la administración de proyecto y provee mecanismos para asegurar que los beneficios del proyecto sean claros, que sea factible la solución propuesta y que haya sólidas bases antes de comenzar con el trabajo detallado.

Se considera a DSDM como el primer método ágil [Frankel]. Estuvo representada en la firma del Manifiesto Ágil y recientemente ha tenido un revival (despertar) en su popularidad, especialmente en el Reino Unido. Esto es porque la APMG-International, los fundadores de Prince2, eligieron combinar DSDM Atern y PRINCE2 para producir un nuevo título Administración de Proyecto Ágil (Agile Project Management - APM). El gobierno del Reino Unido está buscando adoptar APM para todos sus proyectos IT. Se volvió popular en Europa, si bien actualmente tiene presencia en Inglaterra, EE.UU. Benelux, Dinamarca, Francia y Suiza; y con interés y contactos para futuras representaciones en Australia, India y China [SitioNavegapolis].

Como todo enfoque ágil, DSDM ha ganado a través de la experiencia de los años. Dane Falker, director de DSDM Consortium de Estados Unidos en 2001 afirma que si bien el acrónimo se mantiene igual, las palabras y significado han cambiado. [Highsmith]

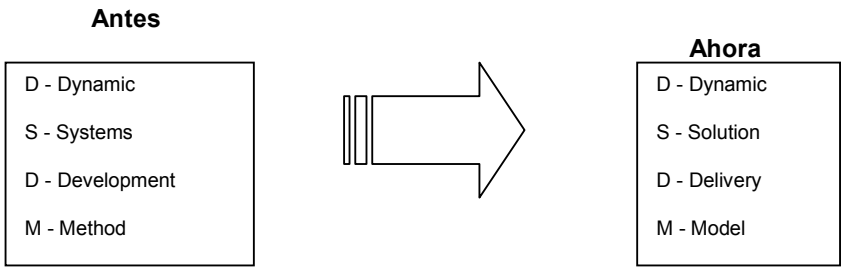


Figura 8.1. Acrónimo de DSDM. Elaborado según [Highsmith]

La primera D del acrónimo por “Dinámico” refleja la habilidad para adaptarse a los cambios. La S ahora refleja el enfoque en las “Soluciones” del negocio más que “Sistemas”. DSDM se enfoca en las soluciones al cliente y el valor de negocio. La segunda D refleja Entrega (Delivery), un concepto más amplio que “Desarrollo” e indica la importancia de los productos entregables (características o historias) más que las tareas tradicionales. Finalmente la M representa más un Modelo que un Método, reflejando una perspectiva de negocio en los proyectos. Por lo tanto podría decirse según Falker que DSDM es un Modelo de entrega de soluciones dinámicas. [Highsmith]

DSDM enfatiza el uso de Workshops facilitadores tanto en las fases de Estudio de Negocio como del Modelo Funcional para involucrar los clientes y usuarios claves en tener el proyecto iniciado apropiadamente. [Highsmith]

Contexto

El modelo ágil que más áreas comprende es DSDM, y según afirma el propio DSDM Consortium, su aplicación equivaldría a un nivel 2 de CMM, dejando por tanto fuera de su ámbito los objetivos y práctica de los niveles 3, 4 y 5. [Navegapolis2010]

La idea fundamental de DSDM se basa en que en vez de fijar las funcionalidades de un producto primero y después el tiempo y el coste, fija primero el tiempo y el coste y con esto fijado, determina las funcionalidades que se pueden implementar en el producto.

Es la más veterana de las metodologías ágiles, y la más próxima a los métodos formales; de hecho, la implantación de un modelo DSDM en una organización, la lleva a alcanzar lo que en CMM (modelo no ágil) sería un nivel 2 de madurez. Surgió en 1994 de los trabajos de Jennifer Stapleton, la actual directora del DSDM Consortium. [Navegapolis2010]

En común con los métodos ágiles, DSDM considera imprescindible una implicación y una relación estrecha con el cliente durante el desarrollo, así como la necesidad de trabajar con métodos de desarrollo incremental y entregas evolutivas.

DSDM cubre los aspectos de gestión de proyectos, desarrollo de los sistemas, soporte y mantenimiento y se autodefine como un marco de trabajo para desarrollo rápido más que como un método específico para el desarrollo de sistemas. [Navegapolis2010]

Es frecuente que DSDM se implante en combinación con XP o Prince2 (que es un modelo predictivo tal como lo es PMBOK utilizado por el gobierno del Reino Unido como el estándar de administración de proyectos para proyectos públicos) [SitioNavegapolis]

DSDM Atern [DSDM]

DSDM ha evolucionado a través de los años, desde aquella primera versión en 1995. En 2001, año del Manifiesto Ágil, DSDM publicó la versión 4.1 de su modelo, y se consideró una metodología ágil; y aunque mantuvo las siglas, cambió la denominación original (Dynamis Systems Development Method) por **Framework for Business Centred Development**. En mayo de 2003 se lanzó la versión 4.2 de DSDM que sigue siendo ampliamente utilizada y sigue siendo válida. En 2007 surge la versión más reciente de DSDM que se llama DSDM *Atern*. El nombre Atern es una abreviación de Arctic Tern - un pájaro de colaboración que puede viajar grandes distancias y resume muchos aspectos de la metodología que es su forma natural de trabajar, por ejemplo establecimiento de prioridades y la colaboración. [SitioNavegapolis]

DSDM Atern es un framework ágil, robusto y probado para la efectiva administración del proyecto y entrega [DSDM]. Se lo puede ver en forma gratuita y está complementado por un conjunto completo de materiales**,** handbook y plantillas – y el DSDM Consortium brinda el soporte.

El Atern Agile framework y sus prácticas han estado largo tiempo a la cabeza de la entrega de proyectos exitosos; experiencias del mundo real que asegura que los objetivos de tiempo, calidad y costos están siempre administrados y son alcanzables. Se diseñó Atern para que sea fácilmente adaptado y usado en conjunción con métodos de entrega (delivery methods) como

PRINCE2® y demostró que complementa y mejora apropiadamente sistemas de administración de calidad trabajados incluyendo aquellos que cumplen CMMI e ISO9001.

Atern es un framework de entrega de proyecto ágil que entrega la solución adecuada en el tiempo adecuado. Se entrega la solución de negocio correcta porque [CaineSitio]:

- El equipo y los stakeholders importantes se mantienen enfocados en el resultado del negocio
- Se entrega a tiempo asegurando un retorno temprano en la inversión
- Todas las personas involucradas en el proyecto trabajan colaborativamente para entregar la solución óptima
- Se prioriza el trabajo según la necesidad del negocio y la habilidad de los usuarios para acomodar los cambios en la escala de tiempo acordada
- Atern no compromete la calidad, por ejemplo la solución no está ni subdiseñadas ni sobrediseñada

El DSDM Consortium DSDM brinda el soporte y certifica Atern; la misión del DSDM Consortium sin fines de lucro es promover las mejores prácticas en la entrega de un proyecto ágil

Método de Desarrollo de Sistema Dinámico (DSDM)

Como se viene mencionando, DSDM se caracteriza por su rapidez de desarrollo atendiendo a las demandas de tecnología de forma eficaz y eficiente previendo que transcurra mucho tiempo y la tecnología cambie. Es una metodología ágil situada dentro de las RAD (Rapid Aplication Development) y es ideal para proyectos de sistemas de información cuyos presupuestos y agendas son muy apretados.

DSDM trata de evitar la falta de participación de los usuarios, la limitación en las oportunidades de cooperación y colaboración, sistemas de baja calidad que no cumplen con los requisitos de los usuarios, etc., todos éstos son problemas que los grupos han encontrado. DSDM consiste en técnicas de desarrollo y gestión del proyecto en la misma metodología.

Variables de un proyecto [DSDM]

La mayoría de los proyectos tienen cuatro parámetros – tiempo, costo, características y calidad. Tratar de ajustar todos los parámetros en la salida del proceso es imposible y es la causa de muchos problemas comunes.

En enfoques tradicionales de administración de proyecto (**Figura 8.2** Diagrama de la izquierda) se fijan las características de la solución y el tiempo y costo están sujetos a variación. Si el proyecto se sale de su curso generalmente se agregan más recursos o se extiende la fecha de entrega. Pero agregar recursos a un proyecto ya atrasado, lo atrasa más. No cumplir con un plazo es desastroso desde una perspectiva de negocio y puede dañar la credibilidad fácilmente. La calidad frecuentemente es una casualidad y se vuelve también una variable acompañada de entrega tardía y aumento de costos.

El enfoque Atern para administración de proyectos (**Figura 8.2** Diagrama de la derecha) fija el tiempo, el costo y la calidad en la Fase de fundamentación mientras la contingencia se maneja variando las características a entregar. Si es necesario, se descartan o difieren características de menor prioridad con el consentimiento de todos los stakeholders. Así un proyecto Atern siempre entrega una solución viable y se garantiza por lo menos la entrega de un conjunto mínimo de características en tiempo y presupuesto

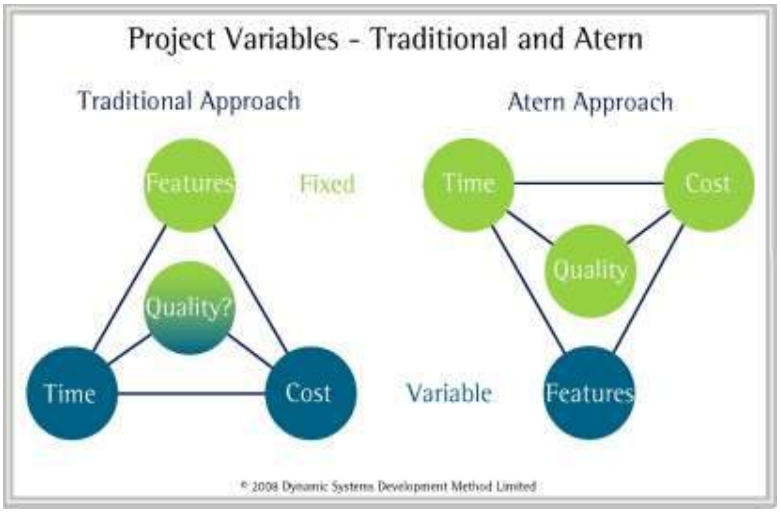


Figura 8.2: Variables de un Proyecto [DSDM Consortium]

En un proyecto Atern, se fija la calidad porque los criterios de aceptación se consensuan y establecen antes de comenzar el desarrollo A diferencia de los proyectos tradicionales, los costos, tiempos y calidad se fijan en etapas tempranas del proyecto. Contingencias, en la forma de características de menor prioridad, aseguran que se puede alcanzar a realizar la entrega a tiempo de una solución viable protegiendo un subconjunto mínimo de usabilidad y descartando o posponiendo características de menor prioridad, si es necesario.

Los principios de DSDM

DSDM tiene principios subyacentes que incluyen una interacción activa del usuario, entregas frecuentes, equipos autorizados, pruebas a lo largo del ciclo. Como otros métodos ágiles usan ciclos de plazos cortos de entre dos y seis semanas. Hay un énfasis en la alta calidad y adaptabilidad hacia requisitos cambiantes. DSDM es notable por tener mucha de la infraestructura de las metodologías tradicionales más maduras, al mismo tiempo que sigue los principios de los métodos ágiles. DSDM claramente evoca a las ideas de un enfoque de desarrollo exploratorio, remarca que los usuarios del sistema no pueden visualizar todos sus requerimientos al principio y recomienda un enfoque iterativo “el paso actual solo necesita completarse lo suficiente para pasar al siguiente paso” [Highsmith]

En Atern se usan los principios para proveer una guía a través del proyecto. Hay 8 principios y todo el framework completo puede derivarse a partir de ellos. En Atern existe una diferencia respecto a DSDM 4.2 ya que para la versión DSDM 4.2 se definieron 9 principios. Los principios están basados en las mejores prácticas en el verdadero sentido. Definen “la forma en que deben hacerse las cosas” [DSDM]. No cumplir con uno de ellos podría llevar al fracaso ya que son los bloques básicos sobre los que se apoya DSDM Atern [DSDM].

En el siguiente cuadro se detallan los principios de Atern y una breve descripción.

Icono	Principio	Descripción
	Enfocarse en las necesidades del negocio	Entregar lo que el negocio necesita cuando lo necesita. Las verdaderas prioridades del negocio deben entenderse.
	Entregar a tiempo	Planear los tiempos de duración anticipadamente y definir el marco de tiempo. Las fechas nunca se cambian; se varían las características dependiendo de las prioridades del negocio para cumplir con los plazos.
	Colaborar	Los equipos trabajan en un espíritu cooperativo y comprometido. La colaboración alienta a entender, ir más rápido y compartir la propiedad. El equipo debe tener poder de decisión e incluir representantes del negocio.
	Nunca comprometer la calidad	Una solución debe ser “suficientemente buena”. El nivel de calidad se define al principio. Los proyectos deben testear temprana y continuamente y revisar constantemente.
	Construir incrementalmente a partir de base sólidas	Los incrementos permiten que el negocio tome ventaja del trabajo antes de que el producto final esté completo, dándole confianza a los stakeholders y brindando retroalimentación. Esto está basado en hacer solo el análisis suficiente para proceder y aceptar que pueden surgir detalles más tarde.
	Desarrollar iterativamente	Aceptar que el trabajo no siempre está bien la primera vez. Usar plazos de tiempo fijos para permitir cambios y continuamente confirmar que la solución es la correcta.
	Comunicarse continua y claramente	Usar talleres, reuniones daily standups, modelado de prototipos, presentaciones y promover comunicación informal cara a cara.
	Demostrar control.	El equipo necesita ser proactivo al monitorizar y controlar el progreso respecto a las bases definidas. Constantemente necesitan evaluar la viabilidad del proyecto basado en los objetivos del negocio.

Tabla 8.1. Principios de DSDM Atern. Traducido de [CaineSitio]

Roles.

DSDM define tres grupos de roles: roles del proyecto, roles del desarrollo de la solución y otros roles El siguiente diagrama (**Figura 8.3**), conocido como “bebé alienígena” o “alien baby” es el diagrama estándar de DSDM Atern que ilustra estos tres grupos de roles.

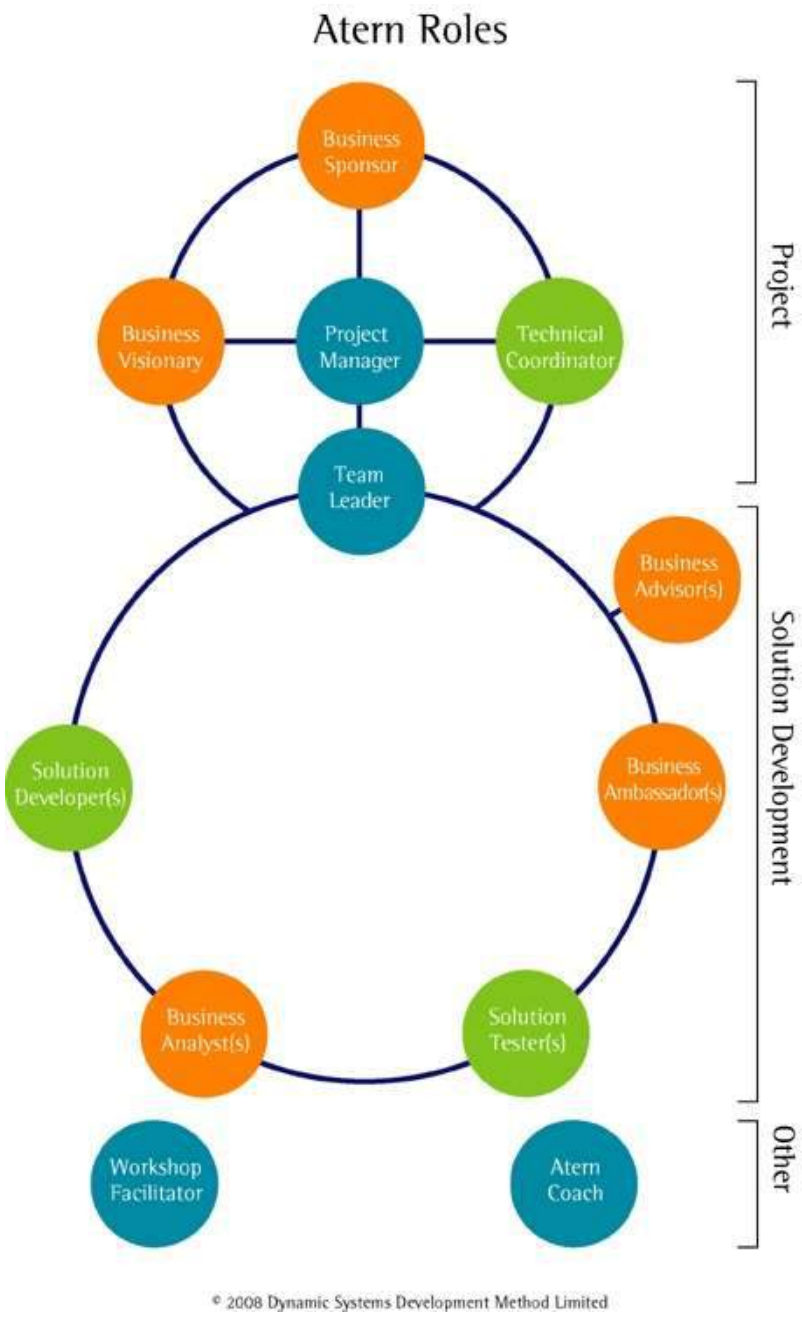


Figura 8.3. Roles en DSDM [CaineSitio]

En la siguiente tabla (ver Tabla 8.2) se describe cada uno de ellos.

Rol	Responsabilidades Claves
Sponsor de negocio (Business Sponsor)	Es dueño del negocio. Asegura los fondos y recursos. Garantiza la toma de decisiones efectiva y se ocupa de las priorizaciones rápidamente
Administrador de Proyecto (Project Manager)	Gobierna el proyecto. Planifica a alto nivel. Monitorea el progreso, recursos disponibles, configuración del proyecto, administra los riesgos y situaciones que puedan surgir.
Visionario del negocio (Business Visionary)	Tiene la visión del negocio y el impacto en cambios del negocio más amplios. Monitorea el progreso frente a la visión. Contribuye en las sesiones de los requerimientos claves, diseño y revisión.
Coordinador Técnico (Technical Coordinator)	Acuerda y controla la arquitectura técnica. Aconseja y coordina a los equipos. Identifica y administra riesgos técnicos. Asegura que se alcancen los requerimientos no funcionales.
Jefe del equipo (Team Leader)	Enfoca al equipo para entregar a tiempo. Alienta a la participación de todo el equipo. Maneja los tiempos fijados para actividades detalladas y actividades día a día. Garantiza que las actividades de prueba y revisión sean programadas y completadas.
Embajador de Negocio (Business Ambassador)	Contribuye en las sesiones de requerimientos, diseño y revisión. Provee la visión del negocio para realizar la toma de decisiones día a día. Describe escenarios de negocio para ayudar a diseñar y probar la solución. Provee la seguridad de que la solución es correcta. Coordina la aprobación del negocio.
Desarrollador de la solución (Solution Developer)	Crea la solución y participa por completo en todas las actividades de QA apropiadas.
Probador de la solución (Solution Tester)	Trabaja con otros roles del negocio para definir los escenarios de prueba para la solución. Realiza reportes completos de resultados de pruebas técnicas al líder de proyecto y al Coordinador técnico.
Analista del Negocio (Business Analyst)	Apoya la comunicación entre miembros técnicos y del negocio del equipo.Administra todos los productos requeridos relacionados a los requerimientos del negocio. Asegura que las implicancias del negocio de las decisiones diarias sean pensadas apropiadamente.
Asesor de Negocio (Business Advisor)	Provee entradas especializadas, por ejemplo un contador. Generalmente un futuro usuario de la solución.
Entrenador Atern (Atern Coach)	Ayuda a los equipos nuevos con Atern a sacarle el mayor provecho. Adapta Atern a las necesidades del proyecto.
Facilitador de taller (Workshop Facilitator)	Maneja y organiza talleres. Responsable del contexto no del contenido. Independiente.
Otros especialistas	Expertos requeridos por poco tiempo, posiblemente por cuestiones técnicas. Por ejemplo Especialistas en load-test, etc.

Tabla 8.2 roles en DSDM Atern [DSDM]

Los principales beneficios del enfoque DSDM se derivan de los usuarios del negocio (clientes) que participan activamente en los equipos de desarrollo.

Proceso Atern [CaineSitio]

Atern difiere como se mencionó anteriormente de otros enfoques ágiles en que cubre todo el ciclo de vida del proyecto. Incorpora disciplinas de la administración de proyectos, y provee mecanismos para asegurar que los beneficios del proyecto sean claros, la solución propuesta sea factible y haya bases sólidas antes de comenzar el trabajo detallado. En Atern existe una diferencia respecto a DSDM 4.2 sobre como mirar el proceso de desarrollo. Hay 7 fases en un proyecto Atern.

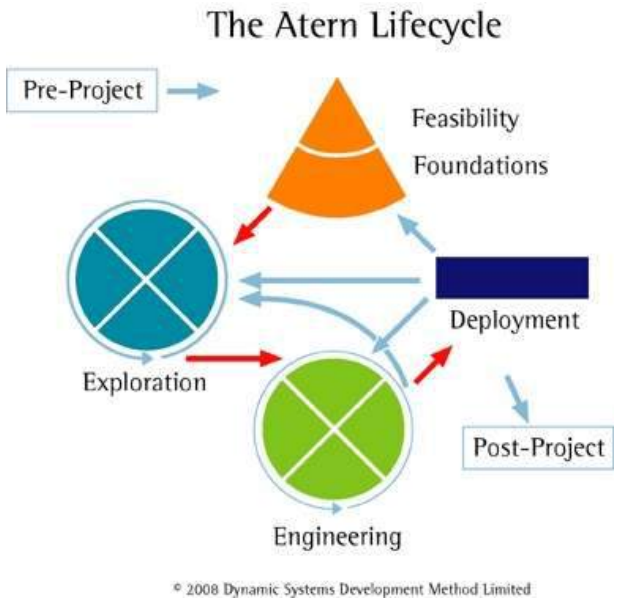


Figura 8.4: Fases de Atern [CaineSitio]

FASES	DESCRIPCIÓN
Pre-proyecto	Iniciación del proyecto, acordando los Términos de Referencia para el trabajo
Factibilidad	Generalmente una fase corta para evaluar la viabilidad y realizar un bosquejo del caso de negocio (justificación).
Bases (fundamentación)	Fase clave para asegurar que se entiende el proyecto y está definido lo suficiente para delinear el alcance a un alto nivel y los componentes tecnológicos y estándares acordados antes de que comiencen las actividades de desarrollo.
Exploración	Fase de desarrollo iterativo durante la que el equipo expande los requerimientos de alto nivel para mostrar la funcionalidad
Ingeniería	Fase de desarrollo iterativo donde se elabora la solución que puede usarse para la entrega
Despliegue	Por cada incremento (conjunto de plazos prefijados) del proyecto se obtiene una solución disponible
Post-proyecto	Evalúa los beneficios alcanzados.

Tabla 8.3 Fases de DSDM Atern. Traducido de [DSDM]

Las fases de exploración e ingeniería a menudo se combinan ya que el método es flexible, permitiendo a los equipos organizarse de la mayor forma para alcanzar la solución. Algunos ejemplos, presentados en [DSDM] se muestran a continuación.

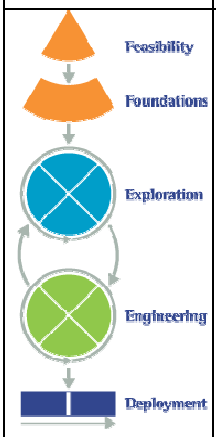
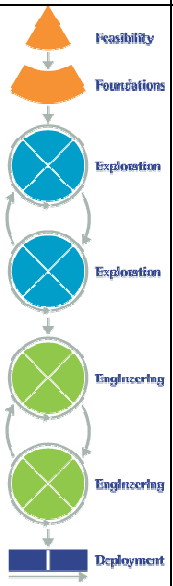
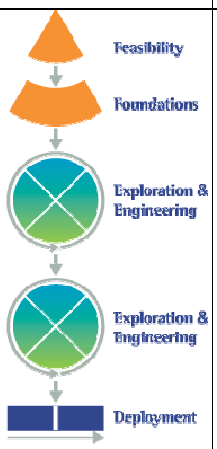
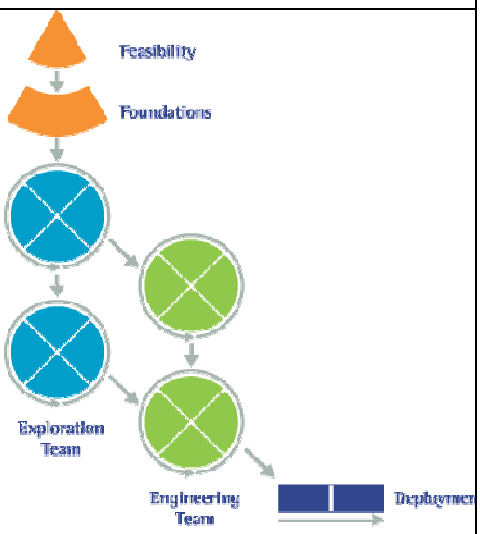
Ejemplo1	Ejemplo2	Ejemplo3	Ejemplo4
			
Realiza iteraciones entre las fases de exploración e ingeniería antes de producir un incremento	Completa todas las actividades exploratorias antes de comenzar con las de ingeniería	Se combina el trabajo de Exploración e Ingeniería	Escenario más complejo con dos equipos involucrados. Un equipo se centra en fase exploratoria y otra en la de ingeniería

Figura 8.5: Ejemplos de adaptaciones de DSDM Atern a un proyecto específico [DSDM]

Entregables

Los entregables están asociados con cada fase del ciclo de vida y se denominan productos. No se requieren todos los productos para cada proyecto y la formalidad dependerá del proyecto y la organización. Algunos productos son específicos de una fase particular y otros pueden seguir evolucionando a través de fases subsecuentes.

El flujo básico de los productos a través del ciclo de vida se muestra a continuación. Por ejemplo, el Estudio de Factibilidad (Feasibility Assessment) se completa con las Bases del negocio (Business Foundations) y la Lista priorizada de Requerimientos (PRL). De manera similar el Similarly, bosquejo de plan (Outline Plan) se refina en el Plan de Entrega (Delivery Plan) para el proyecto que el equipo va refinando por turnos para crear el Plan de Tiempo Prefijado individual y el plan de Despliegue para un incremento.

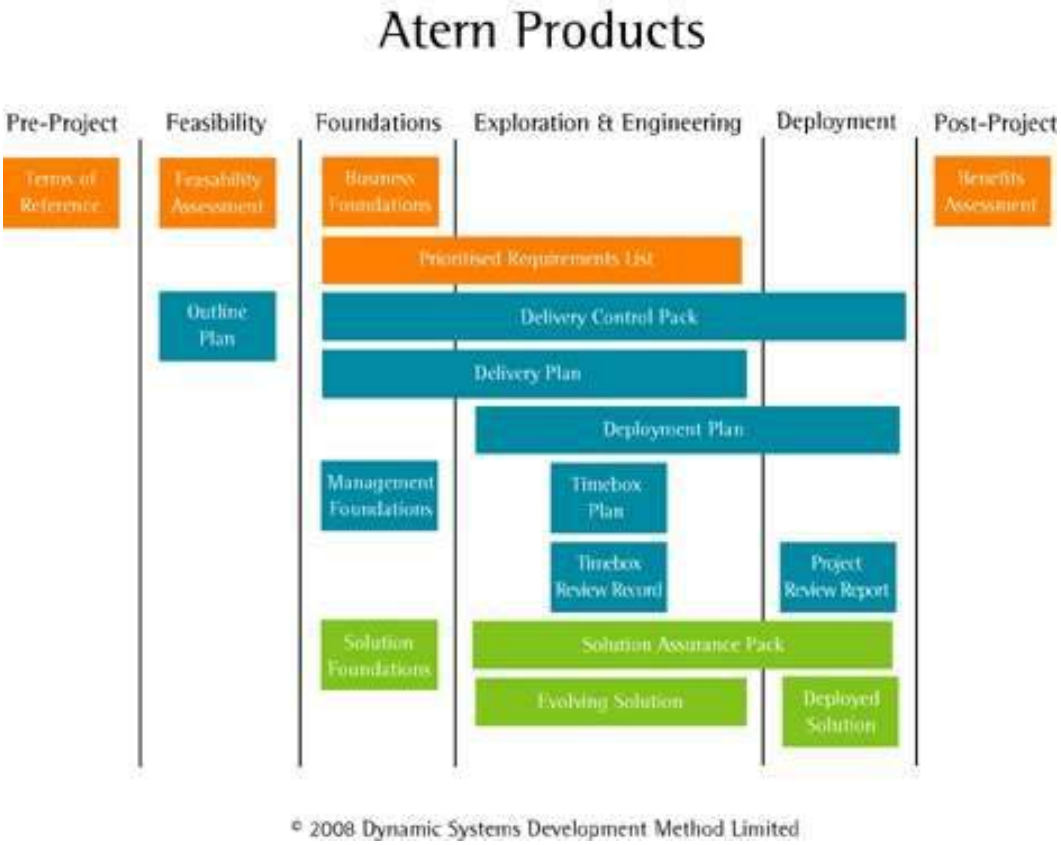


Figura 8.6: Productos Atern [DSDM]

Atern permite al equipo decidir qué productos construir o como deben verse, permitiendo adaptar los productos a la mayoría de los entornos. Algunos entornos podrían requerir todos los productos y otros solo el PRL y la Solución que va evolucionando (similar a Scrum).

Conclusiones

DSDM es una metodología ágil que abarca todo el ciclo de vida de un proyecto de desarrollo. Usa un ciclo iterativo para hacer evolucionar la solución apropiada para satisfacer los objetivos del proyecto. Al dividir el proyecto en períodos cortos de tiempo (timeboxes), cada uno con salidas esperadas muy claras, el Administrador del proyecto y los propios miembros del equipo pueden ejercer el control.

Se definen claramente los roles y se divide el trabajo en time-boxes con fechas de entrega inamovibles y resultados preacordados. El tamaño de los equipos en DSDM varía desde un mínimo de dos integrantes hasta seis, pudiendo coexistir múltiples equipos en un mismo proyecto. [Carvajal]

Atern puede usarse para complementar otras disciplinas de administración de proyecto como PRINCE2™ y PMI (Project Management Institute) sin duplicar esfuerzos. Atern puede incorporar otros enfoques ágiles, como XP y Scrum para proveer el framework de agilidad necesario para permitir una entrega controlada de proyectos ágiles.

9. CONCLUSIONES METODOLOGIAS AGILES

No existe una metodología universal para hacer frente con éxito a cualquier proyecto de desarrollo de software. Toda metodología debe ser adaptada al contexto del proyecto (recursos técnicos y humanos, tiempo de desarrollo, tipo de sistema, etc. [Letelier_b]

Históricamente, las metodologías tradicionales han intentado abordar la mayor cantidad de situaciones de contexto del proyecto, exigiendo un esfuerzo considerable para ser adaptadas, sobre todo en proyectos pequeños y con requisitos muy cambiantes. Las metodologías ágiles ofrecen una solución casi a medida para una gran cantidad de proyectos que tienen estas características. Una de las cualidades más destacables en una metodología ágil es su sencillez, tanto en su aprendizaje como en su aplicación, reduciéndose así los costos de implantación en un equipo de desarrollo. Esto ha llevado hacia un interés creciente en las metodologías ágiles. [Letelier_b]

Falta aún un cuerpo de conocimiento consensuado respecto a los aspectos teóricos y prácticos de la utilización de metodologías ágiles, así como una mayor consolidación de los resultados de aplicación. La actividad de investigación está orientada hacia líneas tales como: métricas y evaluación del proceso, herramientas específicas para apoyar prácticas ágiles, aspectos humanos y de trabajo en equipo.

Cualquiera sea la metodología ágil a utilizar existe una condición obligatoria que si bien puede resultar bastante obvia a veces no se logra fácilmente: Todo el equipo debe usar la metodología elegida, no usarla solamente como una serie de prácticas aisladas ya que no tendría ningún valor.

10. BASE DE CONOCIMIENTO

Nuevo Paradigma: Describir en vez de Programar [Gonda2007]

> *Se pretende "describir" en vez de "programar". Se pretende maximizar las descripciones declarativas y minimizar las especificaciones procedurales. [Gonda2010]*

Según Gonda Breogán y Jodal Nicolás, hasta el 2006 la complejidad de los sistemas había aumentado en un 2000% (datos al 2006) la productividad aunque los lenguajes de programación solo aumentaron un 150%. Esta situación hace que los costos crezcan de una manera desmesurada, considerando los costos como una composición de tiempo y dinero.

En la actualidad muchas empresas internacionales contratan personal para el desarrollo y mantenimiento de sus sistemas en países de bajos salarios, así influyen en la variable *dinero*. Pero, en realidad, el esfuerzo de realización de las tareas sigue siendo el mismo ya que el trabajo involucrado sigue siendo el mismo. Se podría considerar que simplemente se cambió la unidad de medida, en estas circunstancias el precio de la hora-hombre es mucho menor. Por lo tanto si se mide el costo en función del dinero, se produce una disminución. Pero, si se considera la dimensión tiempo, ésta no se modifica.

El *tiempo* es crucial a la hora de desarrollar sistemas y mantenerlos. Por otro lado, los sistemas de software necesitan responder a nuevas necesidades: nuevos dispositivos, nuevos usuarios, nuevas modalidades de uso, nuevas posibilidades de integración y, por todo esto, se hacen cada día más y más complejos y los negocios están sujetos a un “time to market” cada vez más crítico y que no pueden cambiar. Como consecuencia, cada día más negocios se están perjudicando por los tiempos inadecuados de desarrollo de las nuevas soluciones y de mantenimiento de las actuales. Por todo esto es evidente que es necesario un aumento de la productividad a través de tecnologías de muy alto nivel.

La solución no está relacionada tanto en mejorar más todavía los lenguajes de programación sino en la programación en sí. Hoy la enorme mayoría de los sistemas se desarrollan y mantienen con programación manual. Si se "describe" en vez de "programar", se pueden maximizar las descripciones declarativas y minimizar las especificaciones procedurales, haciendo *desarrollo basado en conocimiento* y no en programación. Esta pretensión constituye un cambio esencial de paradigma e implica un choque cultural.

Metodologías tradicionales de desarrollo y problemas asociados [Artech2008]

La forma tradicional de desarrollar aplicaciones parte de una premisa básica: es posible construir un modelo de datos estable de la empresa. Basándose en esa premisa, la primera tarea que se encara es el análisis de datos, donde se estudia la realidad en forma abstracta y se obtiene como producto el modelo de datos de la empresa. La segunda tarea es diseñar la base de datos. Es muy sencillo diseñar la base de datos partiendo del modelo de datos ya conocido. Una vez que se ha estudiado la realidad desde el punto de vista de los datos, se hace lo propio desde el punto de vista de las funciones (análisis funcional). Sería deseable que el estudio de la realidad tuviera como producto una especificación funcional que dependiera sólo de dicha realidad. Lo que se hace en las metodologías más usadas, sin embargo, es obtener una especificación funcional que se refiere a los archivos de la base de datos (o bien a las entidades del modelo de datos, lo que es esencialmente equivalente). Una vez que se tiene la base de datos y la especificación funcional, se pasa a la implementación de las funciones, existiendo tradicionalmente para ello varias opciones (lenguajes de 3ª. o 4ª generación, generadores, interpretadores). Sin embargo, todas las formas de implementación vistas tienen un problema común: parten de la enunciada premisa: es posible construir un modelo de datos estable de la empresa, y esta premisa es falsa.

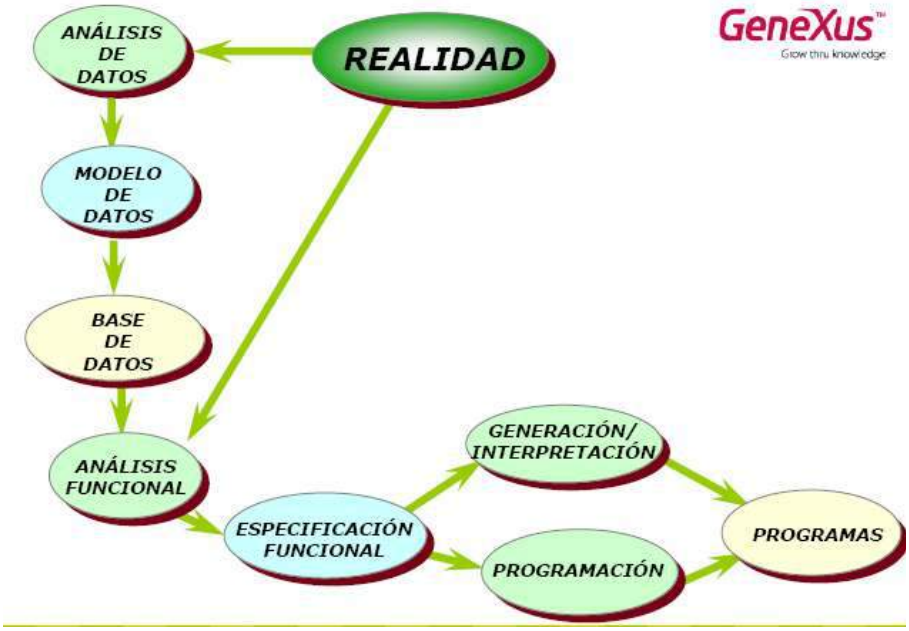


Figura 10.1: Metodología Tradicional [Genexus]

Realmente es imposible hacer, de una forma abstracta, un modelo de datos detallado de la empresa y con el suficiente nivel de detalle y objetividad, porque nadie la conoce como un todo. Por ello es necesario recurrir a múltiples interlocutores, y cada uno de ellos proyecta sobre el modelo, su propia subjetividad. Una consecuencia de esto es que, durante todo el ciclo de vida de la aplicación, se producen cambios en el modelo. Pero aún si se considerara la situación ideal, donde se conocen exactamente las necesidades y, entonces, es posible definir la base de datos óptima, el modelo no podrá permanecer estático porque deberá acompañar la evolución de la empresa. Todo esto sería poco importante, si la especificación funcional y la base de datos fueran independientes. Sin embargo, dado que la especificación funcional se refiere a la base de datos, las inevitables modificaciones en ésta implican la necesidad de modificaciones (manuales) en aquella.

La mayor consecuencia de lo anterior está constituida por los muy altos costos de mantenimiento: en la mayoría de las empresas que trabajan de una manera convencional se admite que el 80% de los recursos que teóricamente están destinados al desarrollo, realmente se utilizan para hacer mantenimiento de las aplicaciones ya implementadas. Cuando se trata de aplicaciones grandes la situación es aún peor: este mantenimiento comienza mucho antes de la implementación, lo que hace que los costos de desarrollo crezcan en forma hiperlineal con respecto al tamaño del proyecto.

Dado que es muy difícil, en este contexto, determinar y propagar las consecuencias de los cambios de la base de datos sobre los procesos, es habitual que, en vez de efectuarse los cambios necesarios, se opte por introducir nuevos archivos redundantes, con la consiguiente degradación de la calidad de los sistemas y el incremento de los costos de mantenimiento.

¿Paradigma orientado al conocimiento? [Gonda2003]

En los últimos años se han solidificado los sistemas de gerencia de base de datos relacionales: hoy es impensable hablar de otro tipo de Sistema de Gerencia de Base de Datos. Se han configurado estándares robustos que, utilizados estrictamente, permiten la portabilidad de las aplicaciones. La existencia de esos estándares dificulta la adopción de modificaciones sustanciales a los mismos. En particular, es muy poco probable que se tenga en un futuro previsible bases de datos cualitativamente mucho más evolucionadas.

Es previsible, en cambio, que se siga dando cada vez más solidez: disponibilidad, seguridad, eficiencia, escalabilidad, optimización de recursos y que se neutralicen las nuevas amenazas como las representadas por virus innovadores.

Paralelamente se ha generalizado la programación orientada a objetos (en particular fuertemente impulsada por el previsible auge de las plataformas Java y .net) y es vital que se resuelva rápidamente el problema de la vinculación natural de los programas con la base de datos (Object Relational Mapping).

Hay grupos de informáticos que creen en "bases de datos inteligentes" que permitan alimentar en forma declarativa todo el conocimiento necesario (fundamentalmente "reglas del negocio", "reglas de precedencia y/o de flujo" y "reglas de autorización" con toda la generalidad que esos conceptos representan) de manera que cualquier usuario sin la necesidad de conocimiento técnico alguno, de una manera simple, pueda hacer en cualquier momento todo aquello que quiera y esté autorizado a hacer, sin necesidad de programar. Este es un nuevo paradigma.

Nuevo Paradigma: desarrollo basado en conocimiento [Gonda2010]

En los últimos años se ha hablado mucho en la industria de la administración del conocimiento (Knowledge Management**)** y, dentro de este rótulo se han colocado muchas cosas que están bien distantes del Desarrollo Basado en Conocimiento a que se pretende referir en este trabajo

Generalmente la industria se ha referido a maneras de organizar y/o acceder el conocimiento para ser utilizado de una forma tradicional por los seres humanos. Provee los medios para la recolección, organización y recuperación computarizada de conocimiento Se trata de una versión actualizada, utilizando la tecnología actualmente disponible, de los libros (y que es de enorme utilidad para toda la humanidad): se accede a un cierto conocimiento leyendo un libro y, en nuestra mente, se hacen razonamientos sobre ese conocimiento lo que, eventualmente, determina acciones. Los buscadores de texto inteligentes que están disponibles desde hace pocos años hacen que este conocimiento sea cada vez más accesible a los seres humanos.

Como característica general, este conocimiento no es "entendible" por una máquina y, en consecuencia, no es operable. Adicionalmente, como el razonamiento de los seres humanos puede lidiar razonablemente (dentro de ciertos límites) con la ambigüedad y, aún, con la inconsistencia, este conocimiento muchas veces no es riguroso. [Artech2008].

Ya en el año 1983 Blazer y sus colegas [Barler] habían propuesto un modelo operacional que soporte la implementación automática a través de un CASE (asistente de desarrollo de software computarizado). La especificación de software operacional es obviamente una parte integral de la base de conocimiento de ingeniería de software. Crear este tipo de base de conocimiento, según los mencionados autores, requiere que:

- Exista un modelo de representación de base del conocimiento y tenga asociado un mecanismo de almacenamiento.
- La representación de la base del conocimiento esté embebida en un entorno de sistema de conocimiento que pueda ser accedido, modificado y ejecutado.
- La base del conocimiento está configurada con la versión inicial de un modelo operacional del ciclo de vida

Un sistema basado en conocimiento útil debe ser capaz de almacenar conocimiento de una manera persistente, una que pueda ser administrada y mantenida como se mantiene una base de datos. Una base de conocimiento persistente puede construirse acoplando sistemas de conocimiento y un administrador de base de datos relacional. [IEEE]

Entonces, es bueno restringir el concepto de conocimiento que se utilizará al mencionar el término Desarrollo Basado en Conocimiento adhiriendo a lo especificado por Artech [Artech2008]. Se trata de conocimiento que cumple las siguientes condiciones:

- Riguroso
- Representable en forma objetiva
- Operable

Una nueva manera de resolver el problema del desarrollo de sistemas pasa por la sustitución de la premisa básica enunciada anteriormente: asumir que no es posible construir un modelo de datos estable de la empresa y, en cambio, utilizar una filosofía incremental y hacer un Desarrollo Basado en Conocimiento. Un esquema incremental parece muy natural: no se encaran grandes problemas, sino que se van resolviendo los pequeños problemas a medida que se presentan. [Artech2008].

Este paradigma de desarrollo basado en conocimiento es completamente diferente a los usuales paradigmas de desarrollo de sistemas: no parte de un modelo de datos pre-existente ni de concepciones abstractas sobre lo que es importante para la empresa y lo que no lo es. [Gonda2010]

En todas las organizaciones hay múltiples usuarios (desde el Gerente General al cargo más bajo en el escalafón de la empresa). Pero, difícilmente exista alguien entre todos ellos que tenga el conocimiento suficiente sobre los datos de la organización. Tampoco es muy probable que alguno de ellos conozca estos datos con la adecuada objetividad y el suficiente detalle. Y este no es un problema que afecte exclusivamente a las grandes empresas, ocurre en empresas de todo tamaño. Por ello *se debe privilegiar lo concreto sobre lo abstracto*, porque los usuarios, que serán múltiples y tendrán los perfiles y roles más diversos en la empresa, se manejan bien con elementos concretos de su ambiente y mal con conceptos abstractos: la representación del conocimiento debe ser simple, detallada y objetiva.

La idea de este paradigma es partir de las diferentes visiones de sus usuarios. Cada usuario, perteneciente a cualquier nivel de la empresa, conoce bien la visión de los datos con los que trabaja a diario. Así, tomando como base este conjunto de visiones de datos, se encuentra el modelo de datos ideal derivado de ellas (puede probarse rigurosamente que, dado un número de visiones de usuarios, existe solo un modelo relacional mínimo que las satisface [Gonda2007]).

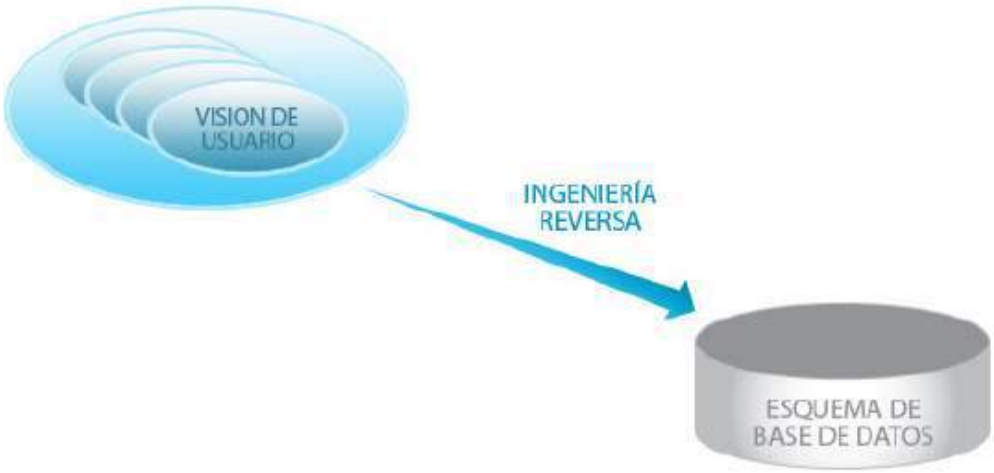


Figura 10.2: Proceso de Ingeniería Inversa [Gonda2007]

Se puede pensar entonces en un proceso de ingeniería inversa que, a partir de una serie de visiones de datos de diferentes usuarios, desarrolla el modelo ideal y la base de datos relacional correspondiente. Todo este conocimiento se puede sistematizar todo este conocimiento es lo que Gonda denomina una Base de Conocimiento. Además, como subproducto, también se puede sistematizar una buena descripción de las visiones de los usuarios y, partiendo de esto, se pueden generar, por ejemplo, los programas requeridos para operar con ellas.

Por lo tanto, resumiendo, este nuevo paradigma pretende capturar el conocimiento que existe en las visiones de los usuarios, y sistematizarlo en una base de conocimiento (todo ello en forma automática). La característica fundamental de esta base de conocimiento, que la

diferencia de los tradicionales diccionarios de datos, es su capacidad de inferencia: se pretende que, en cualquier momento, se puedan obtener de esta base de conocimiento, tanto elementos que se han colocado en ella, como cualquier otro que se pueda inferir a partir de ellos. Si este objetivo se logra, la base de datos y los programas de aplicación pasan a ser transformaciones determinísticas de dicha base de conocimiento y ello permite [Artech2008]:

- Generarlos automáticamente
- Ante cambios en las visiones de los usuarios determinar el impacto de dichos cambios sobre datos y procesos y propagar esos cambios generando:
    - los programas necesarios para convertir los datos;
    - los programas de la aplicación afectados por los cambios;
    - Aquellos programas de aplicación que no han sido afectados por los cambios pero que, ahora, podrían ser sustituidos por otros más eficientes

Modelo Externo y Base de Conocimiento [Gonda2007]

Históricamente la comunidad informática comenzó trabajando sólo con un modelo físico, que tenía muchas rigideces. Luego fue introduciendo otros modelos a los efectos de tener más flexibilidad y obtener cierta permanencia de las descripciones. Por ejemplo ANSI SPARC [ANSI-SPARC] introduce tres modelos:

- **Modelo Externo** donde se representan las visiones externas.
- **Modelo Lógico o Conceptual**: Es otro modelo (generalmente un modelo E-R) que se pretendía obtener por abstracción de la realidad.
- **Modelo Físico**: Se refería fundamentalmente al esquema de la base de datos.

La idea era desarrollar paralelamente los tres modelos para que los programas sólo interactuaran con el Modelo Externo y, a partir del Modelo Lógico se hiciera un mapeo entre esos programas y la base de datos (Modelo Físico) y que ello hiciera independientes a los programas de la base de datos.

El informe fue muy elogiado en su momento y, luego, casi olvidado. Más allá de todo esto, el informe ANSI SPARC sigue siendo válido hoy. La mayor diferencia es la tecnología disponible. En los años transcurridos, los cambios tecnológicos han sido muy importantes.

Analizando el asunto a la luz de la tecnología actual, los desarrolladores que trabajan con el nuevo paradigma de desarrollo basado en conocimiento, concluyen que pueden existir varios modelos, pero *el modelo realmente importante para los usuarios y los desarrolladores, es el Modelo Externo:* en él se recoge el conocimiento exterior y todo lo demás como, otros modelos auxiliares que pudieran ayudar, debe y puede inferirse automáticamente a partir de dicho Modelo Externo.

Ellos pone el énfasis en el *Modelo Externo*, porque es allí donde está el conocimiento genuino y donde reposa la esencia (el "*qué*") y es el realmente importante para los usuarios y los desarrolladores. En él se recoge el conocimiento exterior y todo lo demás, como otros modelos auxiliares que pudieran ayudar, puede inferirse automáticamente a partir de ese Modelo Externo. Un "Modelo Externo" no puede contener ningún elemento físico o interno como: archivos, tablas, entidades, relaciones entre entidades, índices o cualquier otro que pueda deducirse automáticamente del mismo. Ante nuevas posibilidades de la tecnología y necesidades de los clientes, lo primero a hacer es ampliar dicho Modelo Externo.

El Modelo Relacional está orientado a obtener una buena representación de los datos en una Base de Datos, obedeciendo algunas condicionantes muy deseables: eliminar la redundancia e introducir un pequeño conjunto de reglas, de manera de evitar las mayores fuentes de inconsistencia de los datos e introducir un conjunto de operadores que permitan manipular los datos a buen nivel.

El Modelo Externo se utiliza, entonces, para obtener y almacenar el conocimiento. Este modelo pretende ser la representación más directa y objetiva posible de la realidad, por ello, tomamos las visiones de los diferentes usuarios. Estas visiones son almacenadas en el modelo. Luego, se captura todo el conocimiento contenido en ellas y se lo sistematiza para maximizar las

capacidades de inferencia. Con eso se logra la independencia de la implementación, porque se tiene una descripción del sistema en alto nivel, descripción que solo cambiará si cambian las visiones sobre el mismo. Por ello se toman las visiones de los diferentes usuarios. Visiones que son almacenadas en el modelo. Luego se captura todo el conocimiento contenido en ellas y se lo sistematiza para maximizar las capacidades de inferencia. El Modelo Relacional se utiliza para representar y manipular los datos: es el modelo interno o físico, pero será inferido por alguna herramienta que trabaje con este paradigma a través de un procedimiento de ingeniería inversa. [MárquezLisboa]

Para definir el Modelo Externo, de manera que pueda aplicarse ingeniería inversa sobre ellas se realiza una descripción formal y rigurosa para que no haya ambigüedades y que un programa pueda "inferir" todo eso que los programadores hacen todavía a mano.

La Base de Conocimiento (Knowledge Base) inicialmente tiene asociado un conjunto mecanismos de inferencia y contiene reglas generales [MárquezLisboa]**,** como por ejemplo las que aseguran la consistencia, que son independientes de cualquier aplicación particular. Luego, el analista comienza a describir la realidad del usuario a través de objetos, estas descripciones (el Modelo Externo**)** son sistematizadas automáticamente y pasan a estar en la base de conocimiento. Además, sobre ese conocimiento, obtiene un conjunto de resultados que le ayudan a mejorar la eficiencia de las inferencias posteriores**.** Adicionalmente, sobre el conocimiento anterior, el sistema infiere lógicamente un conjunto de resultados que ayudan a mejorar la eficiencia de las inferencias posteriores.

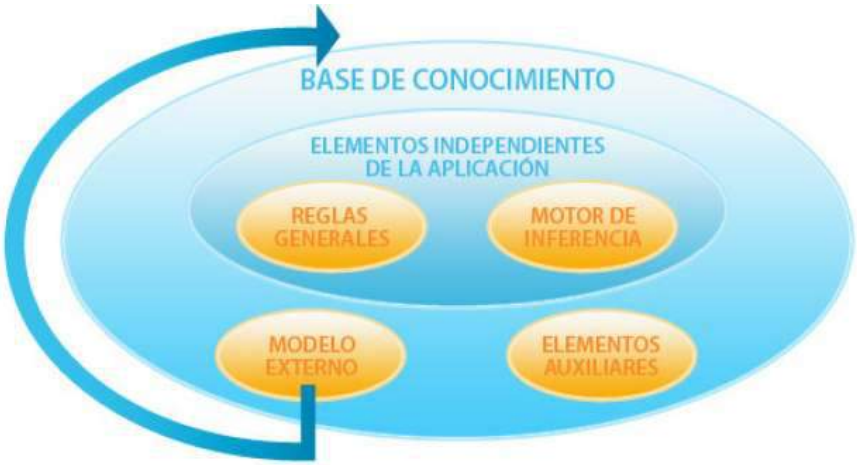


Figura 10.3: Base de conocimiento [Gonda2007]

Todo el conocimiento de la Base de Conocimiento es equivalente al contenido en el Modelo Externo (subconjunto de ella), ya que consiste en el propio Modelo Externo más reglas y mecanismos de inferencia independientes de dicho Modelo Externo y un conjunto de otros elementos que son automáticamente inferidos a partir del mismo.

El desarrollador puede alterar, modificando objetos de la realidad del usuario, el Modelo Externo y las modificaciones se propagarán automáticamente a todos los elementos que lo necesiten: otros elementos de la Base de Conocimiento, Base de Datos y programas de la aplicación. De la misma manera, el desarrollador no puede alterar directamente ningún elemento que no pertenezca al Modelo Externo.

A través de Herramientas Case Específicas creadas para tal fin se pretende dedicar la atención únicamente al Modelo Externo (el "qué") y abstenerse de tratar la Base de Conocimiento, que lo contiene y lo mantiene, (y que forma parte del "cómo": un sofisticado conjunto de herramientas matemáticas y lógicas sobre las que se basan los procesos de inferencia y resultados intermedios utilizados para aumentar la eficiencia de dichas inferencias). Si bien el "qué" y el "cómo" no funcionan uno sin el otro, si el "qué" no es correcto, todo está equivocado. De lo anterior se puede inferir algo muy importante: *todo* el conocimiento está contenido en el Modelo Externo y, por ello, mañana se podría soportar la Base de Conocimiento de una

manera totalmente diferente y el conocimiento de los clientes seguiría siendo utilizable sin problema alguno.

Se intenta, entonces, hacer un modelo que tenga las siguientes características:

- **Conocimiento adecuado para su tratamiento automático**. Modelo con el cual un software de computador pueda trabajar automáticamente. Esto implica que el conocimiento del modelo debe ser "entendible" y "operable" automáticamente.
- **Consistencia**. Modelo siempre consistente
- **Ortogonalidad**. Los objetos que en un determinado momento constituyen el modelo deben ser independientes entre sí. La adición de un nuevo objeto o su modificación o eliminación no implicarán la necesidad de modificar ningún otro objeto.
- **Proyecto, generación y mantenimiento automáticos de la base de datos y los programas**. Entre las cosas que un software de computador puede hacer automáticamente con el modelo, son esenciales el proyecto, generación y mantenimiento automáticos de la base de datos y los programas.
- **Escalabilidad**. Los ítems anteriores expresan las características cualitativas que debe tener el modelo. Pero se pretende resolver problemas grandes. Para ello se necesita que los mecanismos de almacenamiento, acceso e inferencia del modelo actúen eficientemente con independencia del tamaño del problema.

En resumen, la Base de Conocimiento y los mecanismos de inferencia asociados constituyen una gran "máquina de inferencia". Un buen ejemplo de esto es: dada una visión de datos se puede inferir en forma totalmente automática el programa necesario para manipularla.

Genexus

> *"El objetivo de GeneXus es [a través de la descripción de las visiones de los usuarios] conseguir un muy buen tratamiento automático del conocimiento de los sistemas de negocios." Breogán Gonda & Nicolás Jodal*

Genexus es una herramienta de desarrollo de software basada en conocimiento (Knowledge-based Development Tool), orientada principalmente a aplicaciones de clase empresarial para la web, plataformas Windows y Smart Devices producida en Uruguay por la empresa Artech. Su primera versión fue liberada al mercado para su comercialización en 1989. GeneXus Evolution 1, la versión actual, fue lanzada en el 2009. Sus creadores, Breogán Gonda y Nicolás Jodal, han sido galardonados con el Premio Nacional de Ingeniería en 1995 otorgado por el "Proyecto GeneXus" [Gonda1995][Artech2008].

Según los propios autores Genexus es, esencialmente, un sistema que permite una buena administración automática del conocimiento de los sistemas de negocios [Gonda2010]. GeneXus es una herramienta que parte de las visiones de los usuarios; captura su conocimiento y lo sistematiza en una *base de conocimiento*. A partir de esta última, GeneXus es capaz de diseñar, generar y mantener de manera totalmente automática la estructura de la base de datos y los programas de la aplicación, es decir, los programas necesarios para que los usuarios puedan operar con sus visiones. [Genexus] El desarrollador describe sus aplicaciones en alto nivel (de manera mayormente declarativa), a partir de lo cual se genera código para múltiples plataformas. GeneXus incluye un módulo de normalización, que crea y mantiene la base de datos óptima (estructura y contenido) basada en las visiones de la realidad descriptas por los usuarios utilizando un lenguaje declarativo. [Artech2008]

Figura 10.4: Base de Conocimiento [Gonda2007]

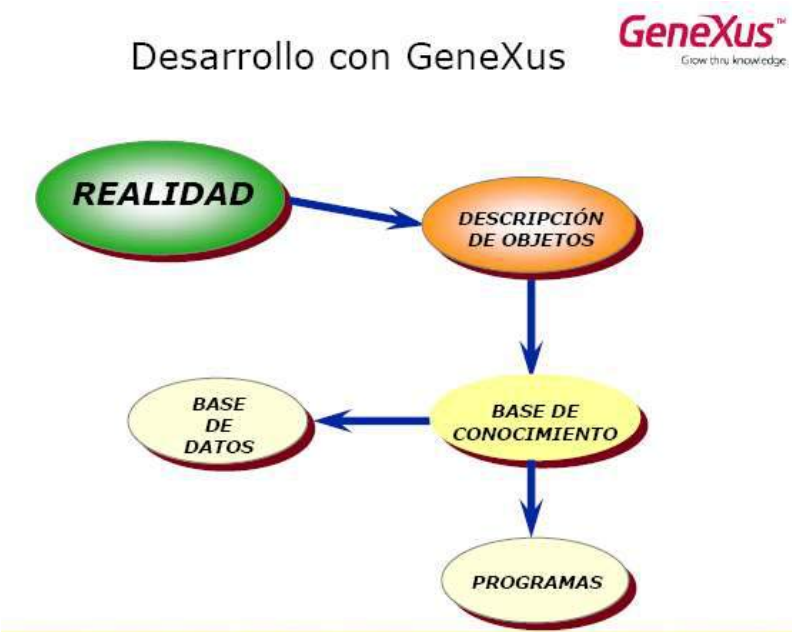


Figura 10.5: Desarrollo con Genexus[Gonda2007]

GeneXus [Genexus] es una herramienta de especificación de sistemas de información basada en la aplicación de un modelo con rigurosa fundación matemática que permite capturar e integrar las visiones de los usuarios en bases de conocimiento a partir de las cuales genera, mediante ingeniería inversa y procesos de inferencia, bases de datos normalizadas y aplicaciones completas; para diferentes plataformas de destino a partir de una misma especificación básica, pudiendo mantenerlas en forma automatizada ante cambios en los requerimientos [MárquezLisboa] [ANSI-SPARC] [Latorres] [Salvetto]. GeneXus modela la realidad mediante un conjunto de instancias de "objetos tipos Genexus", almacenados en las bases de conocimiento. Produce el código necesario para implementar las aplicaciones en diferentes lenguajes y plataformas de destino mediante la utilización de programas generadores. Se apoya en una metodología incremental e iterativa y trabaja sobre especificaciones, lo cual permite independencia tecnológica.

La base de conocimiento de Genexus tiene una gran capacidad de inferencia lógica: en cualquier momento es capaz de proporcionar cualquier conocimiento que se ha almacenado en

ella o que puede inferirse lógicamente de aquellos almacenados en ella. Basado en esta capacidad de inferencia es capaz de proyectar, generar y mantener, en forma 100% automática, la base de datos y los programas necesarios para satisfacer todas las visiones de usuarios conocidas en un determinado momento.

La versión X posee una arquitectura extensible, por lo cual pueden agregarse al producto paquetes de *software*, conocidos como extensiones, capaces de interactuar con la base de conocimiento, pudiendo ser desarrolladas por terceros [Gonda2010].

Una parte considerable del total de *software* producido en Uruguay se elabora con esta herramienta creada y mantenida por una empresa uruguaya, Artech y en Salta existen varias empresas del ámbito público y privado que también las utilizan.

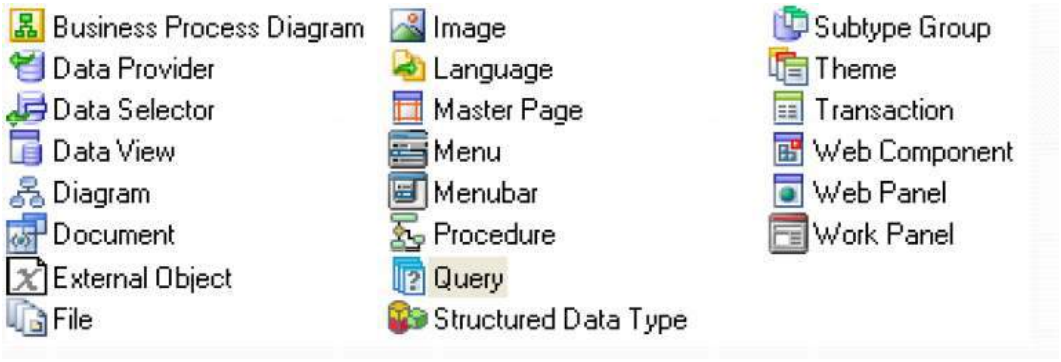


Figura 10.6: Objetos Genexus

Orígenes de Genexus [Gonda2007]

El primer propósito de GeneXus era establecer un robusto modelo de datos. Una vez construido con todo rigor dicho modelo, se pudo pensar en una ampliación del mismo incorporándole elementos declarativos como reglas, fórmulas, pantallas, etc., tratando de describir las visiones de datos de los usuarios y, de esta manera, abarcar todo el conocimiento de los sistemas de negocios. Genexus trató de abarcar todo el conocimiento de los sistemas de negocios porque es una condición necesaria para poder proyectar, generar y mantener en forma 100% automática los sistemas computacionales de una empresa cualquiera. Pero el propósito de Genexus actualmente *es conseguir un muy buen tratamiento automático del conocimiento de los sistemas de negocios.* Ésta es la esencia. Cumplido ese objetivo existe un gran conjunto de subproductos como, por ejemplo:

- Proyecto, generación y mantenimiento automáticos de la base de datos y los programas de aplicación necesarios para la empresa.
- Generación, a partir del mismo conocimiento, para múltiples plataformas.
- Integración del conocimiento de diversas fuentes para atender necesidades muy complejas con costos en tiempo y dinero muy inferiores a los habituales.
- Generación para las tecnologías futuras, cuando esas tecnologías estén disponibles, a partir del conocimiento que, hoy, atesoran los clientes GeneXus en sus Bases del Conocimiento: GeneXus trabaja con el conocimiento puro, cuya validez es totalmente independiente de las tecnologías de moda.

Tratamiento Automático del Conocimiento por Genexus [Gonda2010]

> *"GeneXus, al apoyarse en el paradigma de desarrollo basado en conocimiento, trabaja con conocimiento puro, cuya validez es totalmente independiente de las tecnologías de moda." [MárquezLisboa]*

GeneXus es una herramienta que parte de las "visiones de los usuarios"; captura su conocimiento y lo sistematiza en una base de conocimiento. GeneXus almacena en una Base de Conocimiento todos los elementos necesarios para construir la aplicación, y luego la utiliza para el desarrollo del sistema, construyendo automáticamente el modelo de datos en forma

normalizada, y también utilizando el lenguaje de programación y base de datos que se le indique. A partir de su base de conocimiento, GeneXus es capaz de diseñar, generar y mantener de manera totalmente automática la estructura de la base de datos y los programas de la aplicación (los programas necesarios para que los usuarios puedan operar con sus visiones). Así, permite obtener un proyecto con el conocimiento de la empresa. Todo ese conocimiento -reglas de negocio, diálogos, controles, reportes- que hoy están en un lenguaje de programación determinado, pueden ser convertidos a otros lenguajes sin necesidad de arrancar de cero. O sea que se reutiliza el conocimiento porque la Base de Conocimiento es conocimiento independiente de la plataforma.

Algunas características de esta herramienta CASE

- **Conocimiento puro**. GeneXus trabaja con conocimiento puro, y este conocimiento es independiente de la tecnología utilizada.
- **Mantenimiento 100% automático**. GeneXus *"conoce realmente"* la base de datos y los programas (porque posee el conocimiento para generarlos). Como consecuencia, es capaz de inferir un informe sobre el impacto causado por los cambios efectuados a los programas y a la base de datos, automáticamente y en cualquier momento. Y una vez que dicho reporte es aceptado, puede propagar automáticamente todos esos cambios a los datos y a los programas.
- GeneXus garantiza el mantenimiento 100% automático de las aplicaciones gracias a su tecnología única, y es único producto en todo el mundo capaz de hacerlo.
- **Independencia de plataforma, arquitectura y tecnología**. El conocimiento puro tiene un valor permanente, y es independiente de elementos de menor nivel tales como la plataforma (hardware, sistema operativo, servidor de base de datos, servidor de aplicaciones, etc.), la arquitectura (centralizada, cliente servidor de dos capas, cliente servidor de tres capas, multiservidor orientado a la red como Java o Microsoft .NET) y la tecnología disponible. Como consecuencia, el conocimiento que ha sido compilado en el desarrollo de un sistema con una plataforma y una arquitectura específicas y en un contexto tecnológico específico, puede usarse para generar sistemas para otras plataformas, arquitecturas y contextos tecnológicos (por ejemplo, las aplicaciones que hayan sido desarrolladas hace diez años para una plataforma centralizada y pantallas de formato texto, pueden ser tomadas ahora para plataforma Microsoft .NET o Java).
- GeneXus protege el conocimiento de todos los usuarios, independientemente de la tecnología utilizada. Cualesquiera que sean las tecnologías usadas en el futuro, el conocimiento será el mismo; por lo tanto, construyendo los generadores necesarios, este conocimiento será reutilizado para generar sistemas para las nuevas tecnologías.
- **El Negocio del Conocimiento**. Otra consecuencia del tratamiento automático del conocimiento mencionado anteriormente es que este conocimiento puede ser "fácilmente integrado" y, por lo tanto, comprado y vendido para facilitar y optimizar el desarrollo de sistemas.

Ciclo de desarrollo incremental Genexus [Artech2008]

Cuando una aplicación se desarrolla con GeneXus la primera etapa consiste en hacer el Diseño de la misma registrando las visiones de usuarios (a partir de las cuales el sistema captura y sistematiza el conocimiento). Posteriormente se pasa a la etapa de Prototipación en donde GeneXus genera la base de datos (estructura y datos) y programas para el ambiente de prototipo. Una vez generado el Prototipo debe ser puesto a prueba por el analista y los usuarios. Si durante la prueba del Prototipo se detectan mejoras o errores se retorna a la fase de Diseño, se realizan las modificaciones correspondientes y se vuelve al Prototipo. Llamaremos a este ciclo Diseño/Prototipo. Una vez que el Prototipo está aprobado, se pasa a la etapa de Implementación, en donde GeneXus genera, también automáticamente, la base de datos y programas para el ambiente de producción.

En resumen, una aplicación comienza con un diseño, luego se realiza el prototipo, posteriormente se implementa o pone en producción y en cualquiera de los pasos anteriores se puede regresar al diseño para realizar modificaciones.

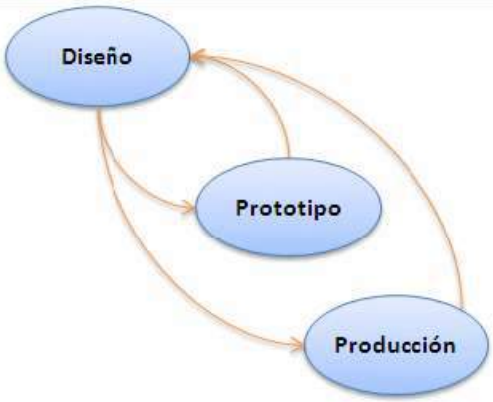


Figura 10.7: Ciclos Diseño-Prototipación y Diseño-Producción [Artech2008]

A continuación se describe cada una de estas tareas.

1. **Diseño**

Esta tarea es realizada conjuntamente por el analista y el usuario, y consiste en identificar y describir las visiones de datos de los usuarios. El trabajo se realiza en el ambiente del usuario. Este esquema permite trabajar con un bajo nivel de abstracción, utilizando términos y conceptos que son bien conocidos por el usuario final.

Una consecuencia muy importante, es que la actitud del usuario se transforma en francamente participativa. El sistema pasa a ser una obra conjunta y, como el usuario sigue permanentemente su evolución, su calidad es mucho mejor que la habitual.

De acuerdo a lo visto, GeneXus captura el conocimiento por medio de visiones de objetos de la realidad del usuario. Los tipos de objetos soportados por GeneXus son, entre otros: Transacciones, Reportes, Procedimientos, Work Panels, Web Panels, Data Views, Transacciones de BI.

La tarea de diseño consiste, fundamentalmente, en identificar y describir estos objetos. A partir de estas descripciones, y automáticamente, GeneXus sistematiza el conocimiento capturado y va construyendo, en forma incremental, la Base de Conocimiento. Esta Base de Conocimiento es un repositorio único de toda la información del diseño, a partir de la cual GeneXus crea el modelo de datos físico (tablas, atributos, índices, redundancias, reglas de integridad referencial, etc.), y los programas de aplicación. Así, la tarea fundamental en el análisis y diseño de la aplicación se centra en la descripción de los objetos GeneXus.

Desarrollo basado en el conocimiento

Partiendo de los objetos descritos, el modelo de datos físico es diseñado con base en la Teoría de Bases de Datos Relacionales, y asegura una base de datos en tercera forma normal (sin redundancia). Esta normalización es efectuada automáticamente por GeneXus. El analista puede, sin embargo, definir redundancias que, a partir de ello, pasan a ser administradas (controladas o propagadas, según corresponda), automáticamente por GeneXus.

El repositorio de GeneXus mantiene las especificaciones de diseño en forma abstracta, o sea que no depende del ambiente objeto, lo que permite que, a partir del mismo repositorio, se puedan generar aplicaciones funcionalmente equivalentes, para ser ejecutadas en diferentes plataformas.

2. **Prototipado**

En las tareas de diseño están implícitas las dificultades de toda comunicación humana:

- El usuario olvida ciertos detalles.
- El analista no toma nota de algunos elementos.

- El usuario se equivoca en algunas apreciaciones.
- El analista interpreta mal algunas explicaciones del usuario.

Pero, además, la implementación de sistemas es, habitualmente, una tarea que insume bastante tiempo, por lo que:

- Como muchos de estos problemas sólo son detectados en las pruebas finales del sistema, el costo (tiempo y dinero) de solucionarlos es muy grande.
- La realidad cambia, por ello, no es razonable pensar que se pueden congelar las especificaciones mientras se implementa el sistema.
- La consecuencia de la congelación de las especificaciones, es que se acaba implementando una solución relativamente insatisfactoria.

El impacto de estos problemas disminuiría mucho si se consiguiera probar cada especificación, inmediatamente, y saber cuál es la repercusión de cada cambio sobre el resto del sistema.

Una primera aproximación a ésto, ofrecida por diversos sistemas, es la posibilidad de mostrar al usuario formatos de pantallas, informes, etc. animados por menús. Esto permite ayudar al usuario a tener una idea de qué sistema se le construirá pero, posteriormente, siempre se presentan sorpresas.

Una situación bastante diferente sería la de poner a disposición del usuario para su ejecución, inmediatamente, una aplicación funcionalmente equivalente a la deseada, hasta en los mínimos detalles.

Esto es lo que hace GeneXus: *Un prototipo GeneXus es una aplicación completa, funcionalmente equivalente a la aplicación de producción*.

La diferencia entre prototipación y producción consiste en que la primera se hace en un ambiente de microcomputador, mientras que la producción se realiza en el ambiente objeto del usuario (IBM iSeries, servidor Linux, Cliente / Servidor, JAVA, .NET, etc.). El prototipo permite que la aplicación sea totalmente probada antes de pasar a producción. Durante estas pruebas, el usuario final puede trabajar con datos reales, o sea que prueba, de una forma natural, no solamente formatos de pantallas, informes, etc. sino también fórmulas, reglas del negocio, estructuras de datos, etc.

La filosofía de GeneXus está basada en el concepto conocido como *desarrollo incremental*. Cuando se trabaja en un ambiente tradicional, los cambios en el proyecto hechos durante la implementación y, sobre todo, aquellos que son necesarios luego de que el sistema está implantado, son muy onerosos (y raramente quedan bien documentados).

GeneXus resuelve este problema: construye la aplicación con una metodología de aproximaciones sucesivas que permite, una vez detectada la necesidad de cambios, prototiparlos y probarlos inmediatamente por parte del usuario, sin costo adicional.

3. Implementación

GeneXus genera automáticamente el código necesario para:

- Crear y mantener la base de datos;
- Generar y mantener los programas para manejar los objetos descritos por el usuario.

El proceso de generación puede ser considerado en dos etapas: *especificación y generación*. La especificación es totalmente independiente del ambiente objeto, pero la generación no. Esto significa que se puede ejecutar el mismo modelo en las diferentes plataformas de ejecución para las que se ha generado y cada una de estas versiones generadas puede ser optimizada de acuerdo con el ambiente en el cual correrá.

GeneXus genera código: para múltiples lenguajes, incluyendo: Cobol, RPG, Visual Basic, Visual FoxPro, Ruby, C#, Java; para múltiples plataformas incluyendo Smart Devices Devices Android o Blackberry y Objective-C para los Smart Devices Apple. Los DBMSs más populares son soportados, como Microsoft SQL Server, Oracle, IBM DB2, Informix, PostgreSQL y MySQL

Los ambientes y lenguajes más importantes actualmente soportados hasta la fecha de edición de este documento son:

Plataformas	• *Plataformas de ejecución:* JAVA, Microsoft .NET, Microsoft .NET Compact Framework • *Sistemas Operativos:* IBM OS/400, LINUX, UNIX, Windows NT/2000/2003 Servers, • Windows NT/2000/XP/CE y Windows Vista • *Internet:* JAVA, ASP.NET, Visual Basic (ASP), C/SQL, HTML, Web Services
Bases de Datos	IBM DB2 for iSeries y UDB, Informix, Microsoft SQL Server, MySQL, Oracle y PostgreSQL
Lenguajes	JAVA, C#, COBOL, RPG, Visual Basic
Servidores Web	Microsoft IIS, Apache, WebSphere, etc
Múltiples Arquitecturas	Arquitecturas de múltiples capas, basadas en web, Cliente/Servidor, centralizadas (iSeries).

Tabla 10.1: Ambientes y lenguajes soportados por Genexus [Genexus]

Recientemente el 15 de marzo de 2012 se lanzó GeneXus X Evolution 2 que acompaña los nuevos avances en la tecnología, incorporando un generador web basado en HTML5 y CSS3, Integración Automática en la Nube, y el generador de aplicaciones nativas para dispositivos móviles, con soporte para las plataformas más populares del mercado: Android, Blackberry e iOS.

La lista completa de tecnologías soportadas actualmente se encuentra dentro del sitio oficial de Genexus [Genexus]. Además GeneXus ofrece un conjunto de herramientas complementarias para:

- Workflow – GXflow (www.gxflow.com)
- Reporting – GXquery (www.gxquery.com)
- Business Intelligence – GXplorer (www.gxplorer.com)
- Portal Building – GXportal (www.gxportal.com)

4. Mantenimiento

Esta es una de las características más importante de GeneXus, y la que lo diferencia de manera más clara de sus competidores: el mantenimiento, tanto de la base de datos (estructura y contenido) como de los programas, es totalmente automático.

A continuación se explicará el proceso de mantenimiento ante cambios en la descripción de algún objeto GeneXus (visión del usuario):

- **Impacto de los cambios sobre la base de datos**

  Análisis de impacto

  Una vez descritos los cambios de las visiones de usuarios, GeneXus analiza automáticamente cual es el impacto de los mismos sobre la base de datos y produce un informe donde explica cómo debe hacerse la conversión de los datos y, si cabe, qué problemas potenciales tiene esa conversión (inconsistencias por viejos datos ante nuevas reglas, etc.). El analista decide si acepta el impacto y sigue adelante o no.

  Generación de programas de conversión

  Una vez que los problemas han sido solucionados o bien se ha aceptado la conversión que GeneXus sugiere por defecto, se generan automáticamente los programas para hacer la conversión (estructura y contenido) de la vieja base de datos a la nueva.

Ejecución de los programas de conversión

A continuación, se pasa al ambiente de ejecución que corresponda (prototipo, producción Internet, producción Cliente / Servidor, etc.) y se ejecutan los programas de conversión.

- **Impacto de los cambios sobre los programas**

Análisis de impacto

GeneXus analiza el impacto de los cambios sobre los programas, y produce un diagnóstico informando qué programas deben generarse o re-generarse y proporcionando también, para el nuevo programa, o bien el diagrama de navegación o bien un pseudo-código, a elección del analista.

Generación de nuevos programas

A continuación el sistema genera o regenera automáticamente todos los programas.

5. Documentación

Todo el conocimiento provisto por el analista, o inferido por GeneXus, está disponible en un repositorio activo, que constituye una muy completa documentación online, permanentemente actualizada.

La documentación incluye la descripción de objetos específicos e información sobre la base de conocimiento resultante y sobre la base de datos diseñada. La base de conocimiento de GeneXus no solamente le permite acceder al conocimiento que almacena siempre que el desarrollador lo desee sino que también le habilita el acceso a toda la información inferida lógicamente (una regla de integridad referencial, un mapa de navegación en la base de datos, un análisis de impacto de cambios, referencias cruzadas, diagramas de Entidad – Relación inferidos a partir del conocimiento almacenado, etc.).

Tendencias de Genexus

Genexus es un producto con una permanente evolución. Las tendencias de esta evolución se las pueden representar sobre cuatro dimensiones: *completitud, productividad, universalidad y usabilidad.*

- **Completitud**: Mide el porcentaje de código del sistema del usuario que es generado y mantenido automáticamente por Genexus. Desde 1992 la completitud alcanzada por Genexus es siempre de un 100%, Éste es un compromiso fundamental porque implica una propagación automática de los cambios, que implica una disminución dramática de los costos de desarrollo y mantenimiento.
- **Universalidad**: Mide la cobertura de las diferentes plataformas importantes disponibles en el mercado. Hoy soporta todas las plataformas “vivas”, entendiéndose por plataformas vivas aquellas que tienen una importante y creciente base instalada. De esta manera, brinda a los clientes una gran libertad de elección: si una aplicación es desarrollada con Genexus, el cliente puede siempre pasarla a otra plataforma soportada sin costos importantes.
- **Productividad**: Mide el aumento de productividad del desarrollo con Genexus sobre el desarrollo con programación manual. El objetivo de aumento de productividad de Genexus fue, durante muchos años, de un 500%, lo que era suficiente. Pero los sistemas que necesitan los clientes son cada día más complejos porque existen nuevos dispositivos y nuevas necesidades, sobre todo de sistemas que cumplan con las necesidades, por complejas que sean, pero se mantengan simples para los usuarios que, en general, no serán entrenados. Adicionalmente, esos sistemas deben ser desarrollados cada vez en menos tiempo: el “time to market” es cada vez más crítico. En el año 2004 se modificaron esos objetivos pasando al 1000% para la versión 9 y el 2000% para la versión Rocha.
- **Usabilidad**: Mide la facilidad de uso. Y pretende hacerlo con carácter general: facilitar el uso a los usuarios técnicos, pero también aumentar el alcance permitiendo que cada

vez más usuarios no técnicos puedan utilizar Genexus con provecho. El solo hecho de utilizar Genexus implica un gran aumento de la usabilidad: un desarrollador Genexus puede desarrollar excelentes aplicaciones para una determinada plataforma sin necesitar de un conocimiento profundo de esa plataforma lo que significa un nivel mínimo importante de usabilidad.

¿Por qué elegir este nuevo paradigma?

La razón principal aducida por Breogán Gonda y Nicolás Jodal para optar por este nuevo tipo de paradigma, desarrollo basado en conocimiento, es que en las grandes organizaciones no existe nadie que conozca los datos de la empresa con la adecuada objetividad y el suficiente detalle. Por lo tanto, el paradigma introducido por Genexus, que consiste en tomar el conocimiento partiendo de las visiones de los usuarios (de alguna manera, realizando un desarrollo isomorfo con la perspectiva), es mucho mejor que los paradigmas tradicionales, tal como se describió anteriormente.

Usualmente, los clientes de Genexus lo utilizan para desarrollar y mantener grandes aplicaciones de Misión Crítica. [Gonda2010]

Proyecto Altagracia

El gobierno de Venezuela prepara un generador de software libre que remplazará a Genexus denominado Altagracia. Este será un programa cuyo desarrollo es dirigido por el Centro nacional de Tecnología de información del gobierno bolivariano de Venezuela

Motivación del proyecto

Altagracia nace tras denuncias de las comunidades de Software Libre, luego de que el Estado venezolano firmara un convenio bilateral con Uruguay para la adquisición de licencias de Genexus que, por el decreto N° 3.390, debía ser totalmente libre, cumpliendo las cuatro libertades del software GPL. Genexus, que tiene muchas propiedades, pero que no es software libre.

El funcionario Carlos Figueira, presidente del Centro Nacional de Tecnologías de la Información de Venezuela, argumenta que de la reunión con directivos de Artech surgió el proyecto de crear un generador de código alternativo a Genexus que lo han llamado Altagracia, que será compatible con Genexus, tendrá sus cualidades, el mismo nivel de desarrollo de suite de aplicaciones, la misma potencia, con cambios mínimos. Según Figueira, Altagracia será una herramienta totalmente libre. Quien quiera tenerla, la adapta y la utiliza; serán las cuatro libertades puestas en su máxima expresión. Además estima que ahora la empresa Artech puede tener aplicaciones para ambos mundos: el libre y el propietario.

> *"Genexus tendrá algunas cosas distintas, en particular su capacidad para producir código para software propietario, Altagracia no hará nada para sistemas propietarios: base de datos, ERP y demás aplicaciones; esto permite que ellos mantengan su línea de negocios clásica del mundo propietario. Además tienen un esquema en el que ellos también se benefician, porque entran en el mundo del software libre; lo importante es que participarán empresas venezolanas, que serán formadas y participarán en el proceso". Carlos Figueira presidente del Centro Nacional de Tecnologías de la Información de Venezuela [Alvarado]*

Para el desarrollo de Altagracia, se tomó como base el generador Genexus. Altagracia, será compatible con Genexus, tendrá sus cualidades, el mismo nivel de desarrollo de suite de aplicaciones, la misma potencia, con cambios mínimos, pues el interés tampoco es violentar a la empresa uruguaya".

Altagracia

Altagracia apunta a obtener, a partir del trabajo en conjunto entre profesionales de la República Oriental del Uruguay y la República Bolivariana de Venezuela, una plataforma para la generación de códigos, construido bajo estándares abiertos. El nuevo software deberá cumplir

con las condiciones sobre software libre, inter-operable e independiente de la tecnología englobada en Genexus. Altagracia será un generador capaz de soportar el diseño e implementación de nuevas aplicaciones en software libre, así como la adaptación y el mantenimiento de las soluciones desarrolladas en estas. Con el generador de códigos libre, se podrán hacer todas las aplicaciones basadas en Genexus, como sistemas administrativos, gestión de documentación, logísticos, todas serán hechas en fuentes libres.

> *"Altagracia nos dará total autonomía sobre las aplicaciones y nos brindará una herramienta para el desarrollo de aplicaciones sofisticadas. El Estado se reserva el derecho de reservar la no liberación del código fuente creado en sectores críticos, si esto ocurre, se haría por razones estratégicas." Carlos Figueira presidente del Centro Nacional de Tecnologías de la Información de Venezuela [Alvarado]*

A inicio del año 2008 a través de diversos medios el Estado venezolano informó sobre este proyecto y anunció que tendría listo en un año el generador de código venezolano, de nombre código "Altagracia": un programa de computadora capaz de crear otros programas de computadora, ello a partir de diagramas introducidos por los programadores. *"Esto facilitará enormemente el desarrollo de aplicaciones de software libre para el Estado"*. Así lo había informado Carlos Figueira, presidente del Centro Nacional de Tecnologías de Información (Cnti, ente adscrito al Ministerio de Telecomunicaciones e Informática) en entrevista a Últimas Noticias.

El propio Figueira explicó en julio de 2010 que el proyecto estaba a punto de empezar y que "tardó el arranque porque involucra dos países y sin dudas hay complicaciones burocráticas, todo el tema tomará calor en pocas semanas y de aquí a un año estará el proyecto culminado" [Alvarado]. En entrevista realizada por Heberto Alvarado Vallejo, también, Carlos Figueira informó que en el 2011 se tendrá listo el generador de código libre Altagracia basado en Genexus. A más de tres años de iniciado el acuerdo con la empresa uruguaya Genexus, creadora del generador de código del mismo nombre, que permitiría la creación de un sistema con las mismas prestaciones, llamado Altagracia, pareciera que ahora sí hay fecha definitiva para adopción definitiva de Genexus. Tal como se puede ver en la página de administración de proyectos del Área Estratégica: CIENCIA, TECNOLOGÍA E INDUSTRIAS INTERMEDIAS de la Embajada de la República Bolivariana de Venezuela en la República Oriental del Uruguay [GenexusLibre]. El proyecto cuyo nombre completo es GENEXUS LIBRE. Proyecto Altagracia CNTI – Langecor figura que se terminaría a mediados de 2011.

Las Instituciones involucradas son: el Ministerio del Poder Popular para la Ciencia, la Tecnología y las Industrias Intermedias (http://www.mcti.gob.ve/), el Ministerio de Energía y Minería de la República Oriental del Uruguay (http://www.miem.gub.uy/portal/hgxpp001?5) y la empresa uruguaya Langecor, S.A. (Genexus Internacional). [CNTI]

Conclusiones

La idea de trabajar con bases de conocimiento para el desarrollo de software ha resultado positiva y ahorra tiempo y esfuerzo de desarrollo concentrando el trabajo en aspectos claves como entender lo que el cliente necesita.

Al trabajar con este tipo de desarrollo como Genexus no se plantea una metodología ágil. Si bien se menciona en [Genexus] como una de sus fortalezas que ésta permite el desarrollo ágil, no hay que entenderlo como un desarrollo siguiendo una metodología ágil, cumpliendo con el manifiesto ágil. Hay que entender la expresión simplemente como la posibilidad de acelerar los ciclos de producción y permitir responder rápidamente a los cambios del negocio, no implicando con esto que busca cumplir con los valores y principios enunciados en el manifiesto ágil.

Genexus es un producto con una permanente evolución. Tal como se mencionó anteriormente recientemente salió al mercado una nueva versión de Genexus que incorpora tecnología para Smart Devices. Una gran desventaja de Genexus es su característica de ser propietario pero el proyecto Altagracia reivindica la idea detrás de Genexus, buscando brindar las mismas características a través de una herramienta totalmente libre.

11. CONCLUSIONES GENERALES

Hace casi dos décadas que se comenzó a buscar una alternativa a las metodologías formales o tradicionales que estaban sobrecargadas de técnicas y herramientas y que se consideraban excesivamente “pesadas” y rígidas por su carácter normativo y fuerte dependencia de planificaciones detalladas previas al desarrollo. Las metodologías ágiles conllevan una filosofía de desarrollo de software liviana, debido a que hace uso de modelos ágiles. Se considera que un modelo es ágil o liviano cuando se emplea para su construcción una herramienta o técnica sencilla, que apunta a desarrollar un modelo aceptablemente bueno y suficiente en lugar de un modelo perfecto y complejo.

Existen actualmente una serie de metodologías que responden a las características de las metodologías ágiles y cada vez están teniendo más adeptos. Aunque los creadores e impulsores de las metodologías ágiles más populares han suscrito el manifiesto ágil y coinciden con sus postulados y principios, cada metodología ágil tiene características propias y hace hincapié en algunos aspectos más específicos.

A través de este trabajo de recopilación bibliográfica e investigación sobre dos temas que a primera vista parecen completamente disjuntos, Metodologías Ágiles y Base de Datos del Conocimiento, se puede ver que ambas metodologías de desarrollo buscan subsanar la deficiencia de contar con requerimientos claros por parte del cliente desde el principio.

Las metodologías ágiles, cada una diferente pero basadas en los principios y valores del manifiesto ágil, proponen distintas formas de trabajar en los proyectos de desarrollo de software. Los enfoques de cada una son diferentes y permiten adaptarlas a las necesidades del equipo y combinar el uso de más de una de ellas, logrando de esta forma una metodología de trabajo que permita a los equipos trabajar cómodamente y alcanzar los objetivos.

Por otro lado las bases de datos del conocimiento buscan dedicar mayor tiempo a definir el modelo externo del proyecto y no centrarse en el Interno, especificando como lograrlo. De esta forma se logra ahorrar tiempo y poder lograr resultados rápidamente. Si bien, como se viene destacando en el trabajo, dentro del propio sitio de Genexus se menciona que éste permite el desarrollo ágil, referido a responder rápidamente a los cambios de requerimientos y poder alcanzar los objetivos a tiempo no habiendo una concordancia con todos los principios del manifiesto ágil. Genexus propone una forma de trabajo que podría optimizarse para tomar las características de metodologías ágiles y poder realizar un desarrollo ágil a través del desarrollo basado en conocimiento.

Por lo tanto, en un futuro trabajo, se puede pensar en profundizar estos aspectos y tratar de delinear ciertos pasos que permitan incorporar las características de un desarrollo de un proyecto ágil a un desarrollo basado en conocimiento.

Lic Loraine Gimson

Mg. Gustavo Daniel Gil

Dr Gustavo Rossi

12. BIBLIOGRAFÍA

Metodologías ágiles

[Abrahamsson2002] Abrahamsson Pekka, Salo Outi, Ronkainen Jussi & Juhani Warsta. *Agile software development methods- Review and analysis.* ESPOO 2002 - VTT Publications 478 – Universidad de Oulu. ISBN 951-38-6009-4. http://www.vtt.fi/inf/pdf/publications/2002/P478.pdf

[AgilAlliance] Agile Alliance, http://www.agilealliance.org/. Acceso : 10/10/2011.

[Agile] *Principios del Manifiesto Ágil*. http://agilemanifesto.org/iso/es/principles.html. Acceso:10/10/2011.

[AgileData] Agile Data. Sitio de Amber Scott. Sitio: http://www.agiledata.org/. Accedido: 10/02/2012.

[AgileJournal] Revista electrónica sobre metodologías de desarrollo de software. *Agile Journal*, nº1, Marzo-2006. Http:// www.agilejournal.com. Accedido 05/12/2011.

[AgileSpain] Agil Spain. Sitio: http://www.agilspain.com. Accedido: 15/03/2012.

[Anderson] Anderson David. Kanban System for Sustaining Engineering. Publicado en encuetro de Agile2007. Sitio: http://www.agilemanagement.net/Articles/Papers/AKanbanSystemforSustainin.html. Accedido: 10/12/2011.

[AndersonSitio] Anderson David. Sitio Oficial. Sitio: http://agilemanagement.net/index.php. Accedido: 10/12/2011.

[Balduino] Balduino Ricardo. Año 2007. *Introduction to OpenUP (Open Unified Process)*. http://www.eclipse.org/epf/general/OpenUP.pdf Acceso:08/02/2012.

[BeedleSitio] Beedle Mike. Sitio oficial de Mike Beedle. Sitio: http://www.Xbreed.org. Accedido: 12/09/2011

[Beck] Beck Kent *Extreme Programming Explained: Embrace Change* Boston Addison Wesley. 2000.

[Bonilla] Bonilla David.. *Historias de Usuario vs. Casos de Uso*. Fecha: 8 Febrero 2010. Sitio: http://sixservix.com/blog/david/2010/02/08/historias-de-usuario-casos-de-uso/. Acceso: 14/11/2011

[Bonilla_b] Bonilla David. *Estructura de una Historia de Usuario*. Fecha: 10 Febrero 2010. Sitio: http://sixservix.com/blog/david/2010/02/10/estructura-historia-usuario/.. Acceso: 14/11/2011

[Burrows2010] Burrows, M. Learning together: Kanban and the Twelve Principles of Agile Software. Junio de 2010. Sitio: http://positiveincline.com/index.php/2010/06/learning-together-Kanban-and-the-twelve-principles-of-agile-software/. Acceso 11/06/2011.

[Caine] Caine. Matthew DSDM Atern –The biggest Agile Secret ? Bern – Swiss – ICS Scrum breakfast, 25 de enero 2012. Sitio: http://www.mcpa.biz/2011/11/dsdm-atern-%E2%80%93-the-biggest-agile-secret-%E2%80%93-bern-swissict-scrumbreakfast-january-25th-2012/. Accedido 08/02/2012.

[CaineSitio] Mathew Caine.*DSDM Atern.* Sitio Web de M.C. Partners & Associates – Crafting sustainable agility. Agosto/Octubre 2011. Sitio: http://www.mcpa.biz/2011/10/dsdm-atern-project-structure-overview/; http://www.mcpa.biz/2011/10/dsdm-atern-principals-overview/; http://www.mcpa.biz/2011/10/dsdm-atern-roles-and-responsibilities-an-overview/; http://www.mcpa.biz/2011/10/dsdm-atern-products-overview/; http://www.mcpa.biz/2011/10/an-overview-of-dsdm-atern/; http://www.mcpa.biz/2011/08/what-is-dsdm-atern/; http://www.mcpa.biz/2011/09/agile-project-management/. Accedido: 20/02/2012.

[Calero] Solís Manuel Calero**.** *Una explicación de la programación extrema (XP)*. V Encuentro usuarios xBase. Madrid. 2003. www.willydev.net/descargas/prev/Explicaxp.pdf

[Canós] Canós José H., Letelier Patricio y Penadés Mª Carmen. *Metodologías Ágiles en el desarrollo de software*. Universidad Politécnica de Valencia. { jhcanos | letelier | mpenades }@dsic.upv.es. http://www.willydev.net/descargas/prev/TodoAgil.Pdf. - 2003.

[CaraballoMaestre] Caraballo Maestre, Alejandro. *Scrum para Dummies. Desarrollo Software.* Sitio: http://caraballomaestre.blogspot.com/2009/05/scrum-para-dummies.html. Accedido: 05/05/2011

[Carvajal] Carvajal Riola, José Carlos - M*etodologías ágiles: herramientas y modelo de desarrollo para aplicaciones java ee como metodología empresarial* - Tesis Final de Máster - Septiembre 2008 - Sitio: http://upcommons.upc.edu/pfc/bitstream/2099.1/5608/1/50015.pdf

[CnetNews] Barnes Cecily - *More programmers going "Extreme* - CNet News. Abril, 2001. Sitio: http://news.cnet.com/2100-1040-255167.html. Accedido: 02/01/2012.

[Cockburn] Cockburn, Alistair - *Crystal Clear A Human-Powered Methodology for Small Teams* **–** Editorial: Addison Wesley Professional – Estados Unidos - Año 2004 - ISBN: 0-201-69947-8

[Cockburn2002] Cockburn, Alistair - **Agile Software Development –** Editorial: Pearson Edication Inc. Stoughton, MA – Estados Unidos - Año 2002 .ISBN:0-201-69969-9

[Cockburn2007] Cockburn, A. Agile Software Development The cooperative Game Second Edition. Alistair Cockburn. Addison Wesley. ISBN:0-321-48275-1 Año 2007 Boston. USA. (Fourth printing, Agust 2009)

[CockburnSitio] Cockburn, Alistair - Sitio oficial de Alistair Cockburn. Sitio: http://alistair.cockburn.us/ Accedido: 12/09/2011.

[CrispSitio] Tjänster - Sitio: http://www.crisp.se/Kanban. Último acceso: 03/02/2012.

[Cunningham] Cunningham, W. and Beck, K.: "A Diagram for Object-Oriented Programs," in Proceedings of OOPSLA-86, October 1986.

[Deemer] Deemer Peter, Benefield Gabrielle, Larman Craig, Vodde Bas. *Scrum Primer.* Versión 1.2. Año 2010. Sitio: http://www.goodagile.com/scrumprimer/scrumprimer.pdf. Acceso: 10/11/2011

[DeemerEs] Deemer Peter, Benefield Gabrielle, Larman Craig, Vodde Bas. Traducción al castellano de Leo Antoli. *Básica de Scrum* (*Scrum Primer).* Scrum Training Institute. Versión 1.1. Año 2009. Sitio: www.scrumalliance.org/resource_download/2481. Acceso: 10/11/2011

[DeSeta] De Seta, L. Los 7 principios del desarrollo Lean. Sitio Web Dos Ideas. Sitio: http://www.dosideas.com/metodologias/410-los-7-principios-deldesarrollo-lean.html. Último acceso: 10/09/2011.

[DSDM] DSDM Consortium. Sitio oficial. Sitio: http://www.dsdm.org/. Accedido: 08/02/2012

[DSDM_Ebook] - *DSDM Atern Enables More Than Just Agility* - Infonic AG, Zurich, Switzerland -- M.C. Partners and Associates, Zurich, Switzerland Abril 2011. Sitio http://www.dsdm.org/wp-content/uploads/2011/12/DSDM-Enables-More-Than-Just-Agility.pdf. Acceso: 08/02/2012.

[EclipseSitio] Dynamic Systems Development Method (DSDM) plugin for OpenUP. DSDM Consortium. Año 2006. Sitio: http://www.eclipse.org/epf/downloads/configurations/pubconfig_downloads.php. Accedido: 10/02/2012.

[Ferreiras2010] Ferreiras, M. - Abril de 2010. Sitio: http://es.scribd.com/doc/30651405/Curso-de-Conceptualizacion-Manufactura-Lean-Ud-1. Acceso 02/06/2011.

[Ferrer] Metodologías Ágiles. Sitio http://libresoft.dat.escet.urjc.es/html/downloads/ferrer-20030312. Jorge Ferrer – 2003 Acceso 02/06/2011.

[Fowler] Fowler, M. *The new methodology.* Trabajo. © Copyright Martin Fowler. Actualización Año 2003. Sitio:. http://martinfowler.com/articles/newMethodology.html. Actualización Año 2003. Acceso 02/10/2011

[Fowler_b] Fowler, M. *La nueva metodología.* Trabajo. © Copyright Martin Fowler. Traducción Alejandro Sierra. Sitio: http://www.programacionextrema.org/articulos/newMethodology.es.html. Actualización 2003. Acceso 10/10/2011

[Fowler_c] Fowler, M. *Ponga su proceso a dieta*. Trabajo. ThougthWorks, Inc.

[FowlerSitio] Fowler, M. Sitio personal de Martin Fowler. Artículo *"Is Design Dead?"* Sitio: www.martinfowler.com/ Acceso 10/10/2011

[FowlerSitio2] Fowler, M., Foemmel M. Sitio personal de Martin Fowler. Artículo "Continuous *Integration*". www.martinfowler.com/ Acceso 10/10/2011

[FowlerSitio3] Martin Fowler, Continous Integration. Mayo 2006. Sitio: http://martinfowler.com/articles/continuousIntegration.html.. Accedido: 10/02/2012

[FowlerSitio4] Martin Fowler, Test-Driven Development. Mayo 2006. Sitio: http://martinfowler.com/bliki/TestDrivenDevelopment.html. Accedido: 10/02/2012

[Frankel] Frankel David. sitio: http://www.lcc.uma.es/~av/MDD-MDA/publicaciones/p_2704-04%20COL%20MDSD%20Frankel%20-%20Bettin%20-%20Cook.pdf. Accedido: 08/02/2012.

[Giro] Consideraciones acerca de XP - Rodolfo Giro, Fernando Leiva, Oscar Mesa y Fernando Pinciroli – admin. De Proyectos – Mg en Ing Sofá – UNLP - 2002

[Gurley] Gurley Greg& Oster Nate de ATSC. Applying Agile OpenUp in a New Team: A Case Study . Trabajo presentado en junio de 2008 en Rational Software Conference Presentations. Sitio: http://www.atsc.com/documents/briefings/20080605-atsc-agileopenup-presentation.pdf. Accedido: 08/02/2012.

[Hartmann] Interview with Hakan Erdogmus by Deborah Hartmann. *Hakan Erdogmus on TDD Misunderstandings and Adoption Issues*. Julio 2008. http://www.infoq.com/interviews/TDD-Hakan-Erdogmus. Accedido: 12/12/2012.

[Higsmith] Highsmith Jim. *Agile Software Development Ecosystems*. AddisonWesley. 2002

[IBMSitio] Koll, Ben. OpenUP In a Nutshell . Septiembre 2007. Sitio: http://www.ibm.com/developerworks/rational/library/sep07/kroll/. Accedido: 10/02/2012.

[Jeffries] Jeffries Ron , Melkin Grigori *TDD: The Art of Fearless Programming* MAY/JUNE 2007 (Vol. 24, No. 3) pp. 24-30 © 2007 IEEE. Published by the IEEE Computer Society. Sitio: http://www.computer.org/csdl/mags/so/2007/03/s3024.html Accedido: 10/02/2012.

[JeffriesSitio] Sitio de Ronald E. Jeffries: *An Agile Software Development Resource*. http://www.xprogramming.com/, Acceso:10/12/2011.

[Joyce2009] Joyce, D. Pulling Value: Lean and Kanban. 27 de Junio de 2009. Recuperado el 9 de Junio de 2011, de Systems Thinking, Lean and Kanban: http://leanandKanban.files.wordpress.com/2009/06/pulling-value-lean-and-Kanban.pdf.

[Kniberg2010] Kniberg, H., & Skarin, M. Kanban vs Scrum: obteniendo lo mejor de ambos. 2010. Madrid, España: Proyectalis.

[Letelier] Letelier Patricio. *Metodologías Ágiles y XP.* Depto. de Sist. Informáticos y Computación. Univ. Politécnica de Valencia.

[Letelier_b] Letelier Patricio, Penadés Mª Carmen - *Métodologías ágiles para el desarrollo de software: eXtreme Programming (XP)* - Depto. de Sist. Informáticos y Computación.Universidad Politécnica de Valencia. Sitio: http://www.willydev.net/descargas/masyxp.pdf o bien Sitio: http://www.cyta.com.ar/ta0502/v5n2a1.htm Acceso: 13/10/2011

[Leteiler2003] Leteiler Patricio et. al., *An Experiment Working with RUP and XP, Extreme Programming and Agile Processes in Software Engineering*, 4th International Conference, 2003.

[LimitedWipSocieteSitio] Limited WIP Society,- The Home of Kanban Systems for Software Engineering. Sitio: http://www.limitedwipsociety.org/ Último acceso: 03/01/2012.

[Linden] Linden – Reed Janice. Quotable Kanban. Sabiduría recolectada de los participantes de Kanban Leadership Retreat. Reykjavik, Iceland, 2011. Sitio: http://agilemanagement.net/index.php/site/quotable/. Accedido: 11/12/2011.

[Lizeth] Lizeth Torres Flores, Carmina & Harvey Alférez Salinas, Germán - Establecimiento de una Metodología de Desarrollo de Software para la Universidad de Navojoa Usando OpenUP - Departamento de Sistemas, Universidad de Navojoa, A.C.& Facultad de Ingeniería y Tecnologia, Universidad de Montemorelos. Reporte Técnico año 2008

[ManifiestoAgil] Beck, K., et.al, *Manifesto for Agile Software Development*, http://agilemanifesto.org/.

[MendesCalo2010] Mendes Calo, K., Estevez, E. Fillottrani, P., *Evaluación de Metodologías Ágiles para Desarrollo de Software*, WICC 2010 - XII Workshop de Investigadores en Ciencias de la Computación

[MendesCalo2008] Mendes Calo, K., Estevez, E. Fillottrani, P., *Un Framework para Evaluación de Metodologías Ágiles* CACIC 2008 http://www.egov.iist.unu.edu/cegov/content/download/1878/47500/version/8/file/Un+Framework+para+Evaluacion+de+Metodologias+Agiles.pdf.

[Middleton] Peter Middleton David Joyce - Lean Software Management BBC Worldwide - Case Study - IEEE Transactions on Engineering Management. Febrero 2012. Sitio: http://leanandKanban.wordpress.com/. Acceso: 03/02/2012.

[Modezki2011] Modezki, M. Kanban para Gestión de Proyectos y otras Aplicaciones. 30 de Marzo de 2011. Sitio: http://www.cafeproyectos.com/gestion-de-proyectos/Kanban-para-gestion-de-proyectos-y-otras-aplicaciones/. Acceso 01/06/2011.

[OPENUP] OpenUP. Sitio Oficial. http://epf.eclipse.org/wikis/openup/index.htm Acceso: 10/02/2012.

[PalacioBañeres] Palacio Bañeres, Juan. *Navegapolis 2008.* Edición Navegápolis® Febrero 2008. http://www.navegapolis.net. Acceso: 05/04/2011

[Palacios2010] Palacios Bañeres Juan - Navegapolis 2010. Edición Navegapolis. Febrero 2010. e-book: http://www.navegapolis.net/files/navegapolis_2010.pdf. Descargado: 08/02/2012.

[Parasuraman] Parasuraman Narayan - *Understanding and Managing Agile Transitions* Sitio: http://www.modernanalyst.com/Resources/Articles/tabid/115/articleType/ArticleView/articleId/1926/Understanding-and-Managing-Agile-Transitions.aspx. Accedido: 15/12/2011.

[Poole] Damon B. Poole, *Do It Yourself Agile*, September 29th, 2009 http://www.accurev.com/whitepaper/pdf/Do-It-Yourself-Agile.pdf. Acceso: 08/08/2011.

[Poole_b] Damon B. Poole, *Do It Yourself Agile - Agile Development Material for the DIY'er* http://damonpoole.blogspot.com/. Acceso: 08/08/2011

[Prince2] *"PRINCE2 in Spanish"*: http://jlfr-prince2.blogspot.com/. Accedido: 04/04/2012.

[Programación-extrema] sitio web miembro de la Agile Aliance. Programación extrema. Sitio: Http:// www.programacionextrema.org. Accedido 05/12/2011.

[ProyectosAgiles] Proyectos ágiles. Sitio http://www.proyectosagiles.org/historia-de-scrum. Accedido:11/11/11.

[Rubio] Rubio Jimenez, M.A. Kanban: el método Toyota aplicado al software. Sitio: http://bosqueviejo.net/2009/06/22/Kanban-el-metodo-toyota-aplicadoal-software/. Último acceso: 10/09/2011.

[SandersSitio] Sanders Aaron. Constantly changing. Sitio: http://aaron.sanders.name/agile-fashion/naked-planning-explained-Kanban-in-the-small. Accedido: 10/12/2011.

[ScrumAllianceSitio] www.Scrumalliance.org Sitio oficial del Grupo de profesionales para compartir trabajo conScrum. Accedido: 10/09/2011. The Scrum Alliance is governed by a Board of Directors: Chairman Mike Cohn, Steve Fram, Treasurer Dan Hintz, Michele Sliger, Harvey Wheaton, Scott Duncan, and Mitch Lacey.

[Schmidkonz] Schmidkonz Christian, CSM & Jürgen Staader, SAP AG, Germany. "*Piloting of Test-driven Development in Combination with Scrum*" http://www.scrumalliance.org/resources/267. Acceso: 10/12/2012.

[SchwaberSitio] www.controlchaos.com/. Schwaber Ken - Sitio oficial de Ken Schwaber. Accedido: 10/09/2011

[ShoreSitio] Shore, James. Kanban-systems. Sitio: http://jamesshore.com/Blog/Kanban-Systems.html. Actualizado: 15 de octubre de 2008. Accedido: 10/12/2011.

[SitioNavegapolis] Palacios Bañeres Juan - *¿Qué es DSDM?* Navegapolis.net. Sitio: http://www.navegapolis.net/content/view/361/59/. Accedido: 08/02/2012.

[Stapleton] Stapleton, Jennifer. *DSDM Dynamic Systems Development Method – The method in practice.* DSDM Consortium. Edison Wesley Professional. 1997

[SutherlandSitio] Sutherland Jeff. Sitio oficial de Jeff Sutherland. Sitio: http://www.jeffSutherland.org. Accedido: 12/09/2011

[Sutherland2011] Sutherland Jeff & Schwaber Ken. *The Scrum Papers: Nut, Bolts and Origins o fan Agile Framework.* Enero 2011. París.

[ThoughtWorks] Barnes, Cecily *More Programmers Going "Extreme"* CNET News.. 3 abril 2001. http://news.cnet.com/2100-1040-255167.html. Accedido: 09/10/ 2011.

[Turley] Turley Frank - *El Modelo de Procesos PRINCE2® - Una magnífica introducción a PRINCE2*. Traducido por José Luis Fernández Ramírez. Año 2010.

[Vega] Vega Frank X. SCRUM, XP, AND BEYOND: One Development Team's Experience Adding Kanban to the Mix. PROCEEDINGS OF… LEAN SOFTWARE & SYSTEMS CONFERENCE. Atlanta. EEUU. 2010. Descargado de http://agilemanagement.net. Accedido: 12/12/2011.

[VersionOne] VersionOne, *State of Agile Development Survey Results2011,* realizado por VersionOne – Agile Made Easier. Sitio: http://www.versionone.com/state_of_agile_development_survey/11/. Accedido: 27/03/2012.

[XP2011] *Agile Processes in Software Engineering and Extreme Programming* - 12th International Conference, XP 2011- Madrid, Spain, Mayo de 2011. Proceedings. ISBN 978-3-642-20676-4.

[XP_sitio] Sitio Extreme Programming. http://www.extremeprogramming.org/ Acceso: 08/07/2011.

Referencias Bibliográficas

[Anderson2010] Anderson D.J.: Kanban: Successful Evolutionary Change for Your Technology Business. Blue Hole Press (2010)

[Ambler] Ambler, Scott W.. *When Is a Model Agile?.* Sitio: http://www.agilemodeling.com/essays/whenIsAModelAgile.htm

[Astels] David Astels. Test-Driven Development. A practical guide. Prentice Hall PTR 2003.

[Beck_b] K. Beck, Extreme Programming Explained: Embrace Change. Boston: Addison-Wesley, 1999.

[Beck_c] K. Beck, "Request for SCRUM Information," J. Sutherland, Ed. Boston: Compuserve, 1995.

[Beck2003] K. Beck, *Test Driven Development: By Example. Año 2002.* Addison-Wesley Professional. ISBN-10: 0321146530 | ISBN-13: 978-0321146533

[Bohem1976] Boehm, B., Software Engineering, IEEE Transactions on Computers, 1226-1241, 1976.

[Bohem2002] Boehm, B. (2002) Get ready for the agile Methods, with Care. Computer 35(1):64-69

[Cockburn1999] Cockburn, A., Characterizing People as Non-Linear, First-Order Components in Software Development, octubre 1999.

[Cockburn2001] Cockbun, A. "Agile Software Development". Addison-Wesley. 2001.

[Ehn88] Pelle Ehn - Work-Oriented Design of Software Artifacts – Editorial L. Erlbaum Associates Inc. Hillsdale, NJ, USA 1990 Año 1990. ISBN:0805807810

[Fowler_c] Fowler, M., Beck, K., Brant, J. "Refactoring: Improving the Design of Existing Code". Addison-Wesley. 1999

[eWorkshop2002] eWorkshop. *Resumen del segundo workshop en metodologías ágiles*. http://fc-md.umd.edu/projects/agile/secondeWorkshop/summary2ndeworksh.htm. Año 2002.

[Glass2000] Glass, R.L. (2001) Agile Versus traditional: Make Love, not War! Cutter IT Journal 14(12):12-18

[Glass2002] Glass, R. L., Facts and Fallacies of Software Engineering, 2002.

[Hawrysh2002] Hawrush, S. & Ruprecht, J. (2000). Light Methodologies: It's like Déjà Vu All over again. Cutter IT Journal. 13:4-12.

[Highsmith2001] Highsmith, J & Cockburn A. (2001). Agile Software Development: The Business of Innovation. Computer 34(5):120-122.

[IABG] IABG, The V-Model, http://www.vmodell.iabg.de/.

[Jacobson] Jacobson, I., et al, The Unified SoftwareDevelopment Process, Addison-Wesley. 1999

[Jeffires2001] Jeffries, R., Anderson, A., Hendrickson, C. "Extreme Programming Installed". Addison-Wesley. 2001

[Kroll] Kroll, P. and Kruchten, P. The Rational Unified Process Made Easy, Addison Wesley, 2003.

[Koll2006] Per Kroll, Bruce Macisaac, Agility and Discipline Made Easy: Practices from OpenUP and RUP, 2006, Addison-Wesley.

[Larman] Craig Larman, "Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, Third Edition", 2004, Prentice Hall.

[Martin2003] Martin, Robert C., *Agile Software Development, Principles, Patterns, and Practices*, Año 2002, Prentice Hall, ISBN-10: 0135974445, ISBN-13: 978-0135974445

[McConnell2003] Mc Connell, S., Professional Software Development: Shorter Schedules, Higher Quality Products, More Successful Projects, Enhanced Careers, Addison Wesley, 2003.

[McCourley2001] McCourley, R. (2001) Agile Development Methods Poised to Upset Status Quo. SIGCSE Bulletin 33(4):14-15

[Miller2001] Miller, D & Lee, J. (2001) The people make the process: commitment to employees, decision making, and performance. Journal of Managment 27:163-189

[Pfeffer1998] Pfeffer, J. (1988) The human equation: building profits by putting people first. Boston, MA, Harvard bussiness School Press.

[Ohno1988]. Ohno T.: Just-In-Time for Today and Tomorrow. Productivity Press (1988)

[Poppendieck2003]. Poppendieck, M., Poppendieck, T.: Lean software development: An agile toolkit. Addison Wesley, Boston (2003)

[Ridderstrale2000] Ridderstrale, J., et al, Business:Talent Makes Capital Dance. Prentice Hall, EEUU, 2000.

[Schwaber] Schwaber K., Beedle M., Martin R.C. "Agile Software Development with SCRUM". Prentice Hall. 2001.

[Schwaber_b] K. Schwaber, "Scrum Development Process," in *OOPSLA Business Object Design and Implementation Workshop*, J. Sutherland*, et al.*, Eds., London: Springer, 1997.

[Sutherland_b] J. Sutherland, "Agile Development: Lessons Learned from the First Scrum," *Cutter Agile Project Management Advisory Service: Executive Update,* vol. 5, pp. 1-4, 2004.

[Thomas] Thomas, Dave y Heinemeier Hansson, David. 2005. *Agile Web Development with Rails (2nd edition).* s.l. : The Pragmatic Programmers , 2005.

[Womack1991] Womack, J.P., Jones, D.T., Roos, D.: The Machine That Changed the World: The Story of Lean Production. Harper Business, New York (1991)

Base de datos del conocimiento

[Alvarado] Artículo "Gobierno garantizó la migración de las bases de datos del Estado" Hormiga analítica – Marcaibo-Venzuela – Martes13 de Julio de 2010- Año 2 Nro 56 Editor Heberto Alvarado Vallejo CNP 12270

[ANSI-SPARC] ANSI-SPARC: Final report of the ANSI/X3/SPARC DBS-SG relational database task group (portal ACM, citation id=984555.1108830)

[Artech2008] *Visión General de Genexus.* Artech Consultores S.R.L. . Uruguay. Octubre 2008. http://www.genexus.es/wp-content/uploads/2010/06/vision_general_gx_esp2009.pdf

[Balrer] R Balrer, 1 E Cheetham, Jr , and C. Green, "EoltwdreTechnologyin the 19905 intraunit and interunit inheritance reter- U5ing a New Paradrgm," *Computer,* Nov 1983, pp 19-45

[Bracci] Bracci Luigino. *YVKE mundial radio.* Gobierno Bolivariano de Venezuela / Ministerio de Poder Popular para la comunicación y la información **Últimas Noticias (Heberto Alvarado), YVKE Mundial.** DOMINGO, 24 DE FEB DE 2008. 12:42 PM. http://www.radiomundial.com.ve/yvke/noticia.php?3323

[CNTI] *Centro nacional de Tecnología de información*. Gobierno Bolivariano de Venezuela / Ministerio de Poder Popular para la comunicación y la información. **Sitio oficial:** . http://www.cnti.gob.ve/index.php?option=com_content&view=article&id=191&Itemid=63. Fecha de acceso: 16 de septiembre de 2011.

[Genexus] Sitio oficial de Genexus. http://www.genexus.com/. Fecha de último acceso: Febrero de 2012.

[GenexusLibre] *GENEXUS LIBRE. Proyecto Altagracia CNTI* - Langecor Proyecto Administrador de proyectos. Embajada de la republica Bolivariana de Venezuela en la Republica oriental del uruguay. Ministerio de Poder Popular para Relaciones exteriores. Sitio: http://173.83.84.126/buscar.aspx // http://173.83.84.126/ficha.aspx?id=68. Acceso 16 de septiembre de 2011.

[Giro] Giro Rodolfo, Leiva Fernando, Mesa Oscar y Pinciroli Fernando - *Consideraciones acerca de XP* - Admin. De Proyectos – UNLP - 2002

[Gonda1995] Gonda B.,Jodal N. "Proyecto GeneXus". Academia Nacional de Ingeniería, Uruguay, 1995 http://www.aiu.org.uy/gxpsites/agxppdwn?2,1,4,O,S,0,79%3BS%3B1%3B21.

[Gonda2003] Gonda Breogán - ¿Desarrollo orientado a procesos u orientado a datos? - Algunas reflexiones en el 40° aniversario de los Sistemas de Gerencia de Bases de Datos – Artech – 2003

[Gonda2007] Gonda Breogán y Jodal Nicolás - *Desarrollo Basado En Conocimiento - Filosofía Y Fundamentos Teóricos De Genexus* - Artech - Mayo de 2007 http://www.genexus.es/wp-content/uploads/2010/06/desarrollo_basado_en_el_conocimiento.pdf

[Gonda2010] Gonda, Breogán & Jodal, Nicolás. *GeneXus: Filosofía*. Artech Consultores S. R. L. Última actualización: 2010

[GxUnit] GxUnit: GxUnit1 y GxUnit2, http://www.gxopen.com/gxopenrocha/servlet/hproject?721, 2008.

[IEEE] Andrew J. Symonds, IBM Data System Division - *Creating a Software= Engineering Knowledge Base*. IEEExplore. Digital Library. Año 1988 http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=2011

[Latorres] Latorres E.,Salvetto P., Larre Borges U., Nogueira, J., "Una herramienta de apoyo a la gestión del proceso de desarrollo de software", IX Congreso Argentino de Ciencias de la Computación (CACIC 2003), La Plata, Argentina.

[MárquezLisboa] Márquez Lisboa Daniel, Fernández Cecilia - *Genexus Rocha – Episodio Uno* – Editorial Grupo Magro – Montevideo - Uruguay - 2007

[Pérez] Pérez, Beatriz - *Aplicando Extreme Programming* - VIII JIIO – Uruguay - Noviembre 2003.

[Salvetto] Salvetto P. "Modelos Automatizables de Estimación muy Temprana del Tiempo y Esfuerzo de Desarrollo de Sistemas de Información", Tesis de Doctorado, Directores: Segovia, F., Nogueira J., Fac.Informática, Univ.Politécnica de Madrid, Doctorado conjunto en ingeniería informática UPM-ORT, 2006 http://oa.upm.es/367/01/PEDRO_SALVETTO_LEON.pdf