MANUAL PARA ELEGIR UNA METODOLOGÍA DE DESARROLLO DE SOFTWARE DENTRO DE UN PROYECTO INFORMÁTICO

Arnaldo Espinoza-Meza

Piura, febrero de 2013

Facultad de Ingeniería

Área departamental de Ingeniería Industrial y de Sistemas

Espinoza, A. (2013). *Manual para elegir una metodología de desarrollo de software dentro de un proyecto informático* (Tesis de pregrado no publicado en Ingeniería Industrial y de Sistemas). Universidad de Piura. Facultad de Ingeniería. Programa Académico de Ingeniería Industrial y de Sistemas. Piura, Perú.

U N I V E R S I D A D D E P I U R A
FACULTAD DE INGENIERIA

"Manual para elegir una metodología de desarrollo de software dentro de un proyecto informático"

Tesis para optar el Título de
Ingeniero Industrial y de Sistemas

Arnaldo Andres Espinoza Meza

Asesor: Ing. Omar Armando Manuel Hurtado Jara

Piura, Febrero 2013

Dedicatoria

A Dios, por haberme dado la fortaleza espiritual para lograr mis objetivos, además de su infinita bondad y amor.

A mi familia, en especial mis padres, hermanos y abuelos, por su apoyo completo e incondicional.

A todas las personas que han hecho posible mi realización como persona y como profesional.

Prólogo

En la última etapa de mis estudios de pregrado, tomé la decisión de concretar una oportunidad de mejora e innovación, y desarrollarla a través de una tesis. Tuve la iniciativa de buscar asesoría académica con profesores universitarios, quienes me brindaron el soporte para perfilar las posibles ideas.

Paralelamente, inicié mi experiencia profesional, donde pude constatar la importancia que tienen los sistemas informáticos dentro de las organizaciones y su incidencia en el flujo de información y la toma de decisiones. De esta forma opté por orientar mi investigación hacia las metodologías de desarrollo de software, pues son marcos que contribuyen al éxito del producto final y a la satisfacción de sus clientes y usuarios. Así, con la ayuda de mi asesor llegué a formular la elaboración de un manual para la elección de la alternativa de metodología idónea de un proyecto de sistemas informáticos.

La importancia de esta tesis radica en que facilitará a los ingenieros de software la búsqueda de la metodología que será la escogida para aplicar en el proyecto de software que lo requiera, a partir del filtro y reducción consecutiva de metodologías alternativas.

Personalmente, quiero agradecer al Ing. José Luis Calderón Lama, quien me apoyó en la búsqueda inicial del área de investigación en la que podría orientar, y a mi asesor Ing. Omar Hurtado Jara, quien con su amistad, dedicación y paciencia me guió en la definición del tema final del proyecto y en el desarrollo de la tesis, y me motivó a su culminación.

Resumen

Los proyectos de software, por su complejidad, necesitan un conjunto de patrones que permitan su gestión y ejecución. Numerosas metodologías de desarrollo de software ofrecen ese marco. Sin embargo, aún no existe una aplicable en la totalidad de proyectos informáticos. Los ingenieros de software tienen la responsabilidad de escoger una metodología adaptable al proyecto de software que lo requiera. No obstante, si no hay una buena elección, costaría más ajustar la metodología a la realidad del proyecto, generándose obstáculos que pondrían en riesgo la consecución de objetivos.

Se elaboró un manual para la selección de la alternativa de metodología idónea para un proyecto de sistemas informáticos. Para ello, se recopiló información que permita elaborar un estudio comparativo de un grupo de metodologías previamente escogidas. A partir del estudio se definió los elementos que serían incluidos en el contenido del manual.

El uso del manual no garantiza un proyecto exitoso, pero permite una buena elección que se acerca más a las características del proyecto, creándose los elementos y conexiones necesarias para cumplir con los requisitos del producto solicitado por los clientes.

Índice General

Introducción

Los proyectos de software, así como otros tipos de proyectos, por su complejidad necesitan de un conjunto de patrones que permitan su gestión y ejecución. Las metodologías de desarrollo de software surgieron en sus inicios adoptando conocimientos de otras disciplinas. Últimamente, están partiendo de conceptos y características propias e inherentes de los proyectos informáticos.

En la actualidad, existen numerosas metodologías de desarrollo de software, producto de las numerosas investigaciones de diversos especialistas en el tema. Sin embargo, no existe una metodología que pueda aplicarse a la totalidad de proyectos informáticos. Por ello, los responsables de la planificación de un proyecto escogen cuál de las numerosas metodologías es la más adecuada para emplear y así obtener el producto solicitado. No obstante, si no se realiza una buena elección, costaría más adaptar la metodología a la realidad del proyecto, generándose obstáculos que pondrían en riesgo la consecución de objetivos.

Esta tesis presenta la elaboración del **"Manual para elegir una metodología de desarrollo de software dentro de un proyecto informático"**. El objetivo de este manual consiste en dar soporte a los planificadores en la selección a través de pasos que filtren sucesivamente un grupo de metodologías hasta hallar la metodología a aplicar. El uso del manual permite la mejor elección, que si bien no garantiza un proyecto exitoso, se acerca más a las características del proyecto, creándose los elementos y conexiones necesarias para cumplir con los requisitos del producto solicitados por los clientes.

El primer capítulo señala conceptos y antecedentes históricos de sistemas de información y metodologías de desarrollo de software, necesarios para comprender el contexto del tema al cual nos orientamos.

En el segundo capítulo, se describen las metodologías de desarrollo de software que serán incluidas en nuestro manual. Además, explica la definición de aspectos a considerar en la descripción y la determinación de las metodologías.

Se elabora un estudio comparativo de metodologías de desarrollo de software. Los criterios definidos, descripción comparativa y resultados de este estudio son abordados en el capítulo tres.

El capítulo cuatro presenta el contenido del manual. Se definen los elementos incluidos en él, siendo el elemento central un procedimiento de selección de metodologías de desarrollo se software.

El capítulo cinco muestra casos de proyectos informáticos con necesidad de una metodología de desarrollo de software para cada una. Se utilizará el manual elaborado para hallar la solución de cada situación.

Complementario a los capítulos, se detallan conclusiones del proyecto de tesis y recomendaciones para investigaciones futuras. También se presenta una bibliografía de consulta para el desarrollo de esta tesis.

Capítulo 1

MARCO TEÓRICO

El presente capítulo definirá algunos conceptos, aspectos generales e ideas importantes para comprender el dominio del estudio a realizar. Mostrará, además, la evolución del contexto de las metodologías de desarrollo de software hasta la actualidad, presentando los avances logrados y aspectos aún por resolver.

1.1. Sistemas de Información

1.1.1. Definición

Conjunto de componentes interconectados cuya finalidad es recolectar, transformar, generar, almacenar y suministrar datos e información que sirven de soporte a las actividades y procesos de una organización y de esta manera lograr el mejor desempeño para la consecución de objetivos [20] [51].

1.1.2. Importancia dentro de las organizaciones

Hasta hace unos 50 años se daba énfasis a la mano de obra, a la maquinaria y a la producción a gran escala como pilar fundamental de desarrollo. En la actualidad, sin desmerecer a los anteriormente mencionados, se considera a la información, y en concreto, su creación, manejo y distribución, elemento esencial en la realización de nuestras actividades, en el plano cultural, económico y empresarial.

Los sistemas de información son considerados el centro neurálgico de toda organización, porque:

- El flujo eficiente de la información, permite un mejor control y la toma de decisiones en todos sus niveles (Operativo, Administrativo – Funcional y Estratégico – Gerencial) [58].
- Facilitan y disminuyen el tiempo en la búsqueda y recopilación de información, aumentando la productividad [33] [56].
- Permiten estar más preparados para actuar en un escenario que evoluciona constantemente Implica un cambio en la forma de ver los negocios dentro de una empresa [33] [53].

- Intensifican y propician las relaciones entre miembros de la empresa. Ya sea en la formación de grupos de trabajo e investigación, o en la solución de falta de comunicación entre instancias no inmediatas [56].
- Son medio de innovación y desarrollo de nuevas oportunidades de negocio que satisfagan las necesidades del cliente, y de interacción con ellos, adquiriéndose una nueva ventaja estratégica y competitiva [53].

1.1.3. Sistemas Informáticos

Son sistemas de información cuyos componentes son: hardware (computadoras y dispositivos de entrada y salida), software (programas) y recursos humanos (el hombre como usuario del hardware y software). Éste término, a diferencia de sistemas de información, está más relacionado a los temas abordados en nuestra investigación[1].

1.2. Ingeniería de Software

1.2.1. Definición

Ingeniería de software es una disciplina que se encarga de la aplicación de procedimientos, métodos, técnicas, tecnologías y herramientas, provenientes de diversos estudios (computación, gestión de proyectos, ingeniería, el entorno y otras ciencias), de manera sistémica, en el proceso de desarrollo y mantenimiento de software [41] [45][2].

1.3. Metodologías de desarrollo de software

1.3.1. Definición

Actividades, procedimientos, técnicas, herramientas y documentos, en su conjunto, normados y comprendidos en un marco de trabajo. Sirven de soporte en la estructuración, planificación y control requeridos para lograr la conversión de una necesidad o un grupo de necesidades a un sistema de información de manera eficiente [24][2] .
Las metodologías nos indican un plan adecuado de gestión y control del proyecto de software[2]: definición de etapas, ingresos y salidas, restricciones, comunicaciones, tareas ordenadas y distribución de recursos.

1.3.2. Evolución histórica [24]

En los años 50, aparecen los primeros lenguajes de alto nivel para programación. Los programadores, dada la limitada capacidad y velocidad de las computadoras, centraban su esfuerzo en economizar líneas de código y optimizar recursos.

Hacia 1968, las mini-computadoras habían ingresado al mercado. Aumentó la capacidad de memoria y las velocidades de cálculo. La incursión del sistema IBM/360 y la variedad de lenguajes de alto nivel dificultaban el traslado de un programa de un equipo a otro. Los

[1] En adelante, se utilizará la palabra “sistemas”, como sinónimo de “sistemas informáticos”.

[2] Los términos apropiados serían Ingeniería de Sistemas Informáticos, Metodologías de desarrollo de Sistemas Informáticos y Proyectos de Sistemas Informáticos, pues en la práctica también abarcan el medio físico que contiene al software (hardware) y las personas que lo manipulan (clientes). En nuestro estudio, por convención, mantendremos la palabra “software” en dichos términos.

costos de software ya sobrepasaban a los de hardware, debido a la reducción de estos últimos.

Ante esta situación, no se podía seguir asumiendo la durabilidad y permanencia en el tiempo de los programas. Se plantearon criterios para concluir con éxito un proyecto de software, de acuerdo a su código:

- Bajo costo de desarrollo inicial.
- Facilidad de mantenimiento, para realizar mejoras y correcciones.
- Portabilidad, a nuevo hardware.
- Satisfacción de requisitos del cliente.
- Satisfacción de parámetros de calidad (como seguridad, confiabilidad).

Cumplir dichos criterios significaba establecer una reglamentación que sirva como pauta, ya que, primero se codificaba, en base a requisitos ambiguos y poco específicos.

En la década de los 70, la Programación Estructurada, intentó dar solución a esta problemática. Se incentivó la estrategia *Top Down*[3], se formalizó el uso constructores (Secuencia, selección e iteración), y la descomposición de problemas siguiendo una rutina de pasos. Se propuso para el modelado de estructura de programa a los diagramas de flujo, pseudocódigo, diagramas de Nassi – Schneiderman (híbrido entre flujos y pseudocódigo), diagramas de decisión, etc.

En este sentido, trasplantar los planteamientos de la programación estructurada a un análisis y diseño conlleva al nacimiento de la Ingeniería de Software como disciplina.
En consecuencia, surgen las metodologías SA/SD (Metodologías de diseño y análisis estructurado). En el análisis se diferencia el modelado de procesos del modelado de datos con distintas notaciones según el método. En el diseño del sistema se toma el modelado del análisis para complementar detalles y notaciones (Cartas de estructura), y finalmente describir el código.

Modelos de ciclo de vida fueron publicitados, afirmando aminorar costos y acrecentar confiabilidad. Primó el modelo cascada, que formulaba una secuencia lineal de fases de desarrollo. Sin embargo, este modelo fracasó debido a que el software se hacía más complejo y ya no se podía manejar cambios de manera adecuada a través de fases sucesivas. Los modelos prototipo, *Rapid Aplication Development* y espiral intentaron también resolver este problema.

En los 80, CASE (*Computer Aided Software Engineering*) e IDE (*Integration Development Enviroment*), aunque, útiles herramientas de apoyo, no estaban relacionados con el concepto de metodología que se buscaba desde el inicio.

La programación orientada objetos, en contraste con la programación estructurada tiende a asemejarse a un escenario real, en que los elementos interaccionan entre sí. Alcanzó su conceptualización completa entre los 70 y 80. Sus propulsores aseveraban que permitiría mayor flexibilización y reutilización de objetos. Las metodologías basadas en la nueva

[3] *Top Down*. Enfoque de desarrollo de software, que promueve la planificación del proyecto de software y el análisis del problema desde un nivel de detalle bajo, hasta alcanzar el mayor detalle posible donde recién se puede diseñar y construir el software.

filosofía primero se pensaron para programación, luego diseño, y por último análisis. Pese a ello, muchas no alcanzan todo el ciclo de vida.

En la década de los 90 se logra unificar y estandarizar una notación, obteniéndose el Lenguaje de Modelamiento Unificado (UML), promovida por Rational[4]. Ahora independientemente de la metodología que se utilice, todos entenderían y podría comunicar sus diseños e ideas. *Rational Unified Process* (RUP), perteneciente a la misma Rational, fue introducida en el mercado en 1998. Propiamente, RUP agrupa elementos de otras metodologías en una metodología universal que pueda adaptarse a las particularidades intrínsecas de cada proyecto.

A finales de dicha década, en contraposición a las metodologías predictivas, consideradas burocráticas y lentas, nace el concepto de metodologías ágiles, en las que, la ausencia de documentación y equipos multidisciplinarios son su principal carta de presentación. XP (Programación Extrema), *Crystal Clear*, Método de Desarrollo de sistemas dinámicos (DSDM), el Proceso Unificado Ágil, etc., por citar algunos.

1.3.3. Tipos de metodologías

1.3.3.1. Metodologías predictivas

Las metodologías predictivas de software surgieron por la necesidad de hallar parámetros reguladores del desarrollo de software. Así se intentó en base a metodologías pertenecientes a otros campos de especialización y de trabajo, en los que ya se tenía años de experiencia (gestión de proyectos de ingeniería) [13] [23].

A. Características

- División de etapas ordenadas y secuenciales, cada una con objetivos y reglas prefijadas [13] [41].
- Administración de documentación específica y detallada de la planificación del proyecto de software, elaborada antes de iniciar las actividades [13].
- Al final de cada etapa o sub- etapa definida del proyecto (hito) se materializa un entregable y se realiza retroalimentación [13].
- Predomina el control del proceso. Específica roles, funciones, tareas y herramientas que pueden ser completadas por personal idóneo. No es importante quién, ni cómo lo hace, sino qué rol cumple [41].
- Las comunicaciones entre el equipo y con los interesados son planeadas de manera formal y mayormente sirven para anunciar avances. Prefiere la interacción con el cliente, para recolección de requisitos y sugerencias, al inicio del proyecto [13].
- Se asume que las estimaciones de alcance, tiempo y costos, no son variables, y que se deben respetar y cumplir [23].

B. Críticas y desventajas

Si bien las metodologías tradicionales han sido efectivas, no lo eran tanto para nuevos proyectos de software, afectados por un entorno cambiante [23]. Especialistas en

[4] Rational ó Rational Software, es una división de IBM encargada de la investigación y creación de metodologías y herramientas para el desarrollo de software.

desarrollo de software han analizado las metodologías tradicionales de software, hallando puntos en contra, siendo los más resaltantes los siguientes:

- Poca capacidad para adaptarse a los cambios. Muchos proyectos, por sus características van evolucionando en cortos períodos de tiempo. Es difícil conocer el rumbo que tomarán los requisitos y las variables del entorno, desestimando la previsión de costos, tiempo y alcance preliminarmente planteada [7] [23].
- Toma de decisiones erradas y dificultades para corregir errores. En la primera etapa de un proyecto de software es donde se presume tener todos los requisitos y objetivos claros, puesto que serán fuente de la toma de decisiones que guiarán en adelante el curso del proyecto, pero aún existen vacíos. La retroalimentación debería servir de cuantiosa ayuda inmediata. Más bien, esto significa regresar al comienzo y rehacer todo, con un costo alto [23] [44].
- Estas limitaciones, resultan en una merma en la calidad del proceso, suponiendo un grave riesgo para garantizar la misma calidad del producto, clave para cumplir las expectativas del cliente.

1.3.3.2. Metodologías ágiles

Son el resultado de un nuevo enfoque que se basa en la pronta entrega de software incremental, proveniente de un desarrollo iterativo durante todo el ciclo de vida del software. Son ideadas en 1990, oponiéndose a las metodologías de desarrollo tradicional. Sus propulsores argumentaban que para obtener un software eficiente se debe buscar una flexibilización en las restricciones y en la burocratización del trabajo y de las comunicaciones [7] [41].

En el año 2001, un grupo de desarrolladores, expertos y escritores sobre el tema, entre los que encontramos a Kent Beck, Alistair Cockburn, Martin Fowler y Jim Highsmith, firmaron el “Manifiesto para el desarrollo ágil de software” [3]. En él se describe lo citado a continuación:

“Estamos descubriendo formas mejores de desarrollar software tanto por nuestra propia experiencia como ayudando a terceros. A través de este trabajo hemos aprendido a valorar:

- 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.

Esto es, aunque valoramos los elementos de la derecha, valoramos más los de la izquierda”. Significa que, en las metodologías ágiles predominan los individuos, las entregas funcionales, la colaboración y la respuesta al cambio.

A. Características

- Adaptación a entornos y a parámetros cambiantes

Es imposible predecir las variaciones en los requisitos a través del tiempo, por lo que es imprescindible que un proceso permita ingresar esas variaciones con facilidad en el transcurso su ciclo de vida [7] [17].

- Prevalecen la personas en vez de los roles y funciones

Para planificar un proyecto de software se conforma primero el grupo de trabajo, y que éste defina sus requisitos y entorno, en vez de definirlo primero y que luego el equipo se ajuste. Por esta razón la documentación se limita a toma de decisiones críticas, en vez de detallar un planeamiento [23].

- Iterativo e incremental

Puesto que debe concurrir un progreso o tendencia al perfeccionamiento, la mejor forma de lograrlo es a través de retroalimentación y ciclos de corta duración y repetitivos [7] [44].

- Cooperación activa del cliente

La retroalimentación viene por parte del cliente, con quien se interactúa presentándosele avances tangibles (prototipos totales o parciales) del sistema [23] [44].

- Fáciles de aprender.

No demanda considerable tiempo de capacitación en aprender el manejo de herramientas complejas [44].

B. Críticas y desventajas

La explosión y la visión positiva que se ha podido percibir alrededor del desarrollo ágil, ha opacado, los cuestionamientos, dudas y críticas que ha recibido. Entre las principales destacan:

- Escasa innovación, por ser ideas que en el pasado ya se utilizaban. Lo que se logró a partir del manifiesto ágil en adelante fue, una conciliación [17].
- No hay un enfoque hacia la arquitectura de software, perdiendo esta última calidad [17].
- No es apropiado para algunos proyectos: aplicaciones distribuidas, aplicaciones que requieren seguir un diseño estricto (sistemas operativos, software de telecomunicaciones), sistemas militares, médicos o industriales, obligados a originar una documentación a profundidad, aplicaciones basadas en interfaces gráficas de usuario, o proyectos muy grandes (entorpece la comunicación entre los miembros del equipo) o aplicaciones donde la escalabilidad o la eficacia sean importantes [44].
- No se conoce el verdadero beneficio de estas metodologías. Las pruebas y estudios de incremento de calidad y relación entre la satisfacción del usuario y el uso de estas prácticas no son definitivos [17].
- Un estudio elaborado por los académicos Dyba y Dingsoyr, hallaron cansancio entre los clientes cuando se utilizan rutinariamente y la no intercambiabilidad de personas, dado que es un proceso de personas y no cuenta con documentación más amplia [18].

1.3.4. Comentario de la situación actual de las metodologías de desarrollo de software

Existe un debate abierto, con la finalidad de mantener una forma de trabajo disciplinada y rígida dentro de las organizaciones, o implantar una filosofía de pronta adaptación a los cambios, en lo que el proceso de desarrollar sistemas de información y software concierne.
Las metodologías de desarrollo de software, según sus características, no son aplicables para todos los proyectos. Ante la aparición de diversas metodologías, la tarea de los responsables de la planificación de un proyecto es escoger cuál de las numerosas metodologías es la más adecuada para aplicar y obtener el producto solicitado.
Por este motivo, en esta tesis se ha decidido elaborar un manual de soporte a ingenieros de software para la selección de la alternativa de metodología idónea de un proyecto de sistemas informáticos. La elaboración del manual será abordada en los siguientes capítulos.

Capítulo 2

METODOLOGÍAS DE DESARROLLO DE SOFTWARE A ESTUDIAR

Como se ha mencionado en el primer capítulo se ha decidido elaborar un manual de soporte a ingenieros de software para la selección de la alternativa de metodología idónea de un proyecto de sistemas informáticos. Para elaborar ese manual, nos basaremos en un estudio comparativo de metodologías de desarrollo de software. Por ende, lo primero que se realizó fue definir los aspectos importantes de las metodologías que nos ayudarán en el estudio comparativo y las metodologías que formarán parte del estudio. Ambos, aspectos y metodologías serán presentados en el presente capítulo.

2.1. Definición de aspectos a considerar en descripción de metodologías

Para definir los aspectos que se considerarán en la descripción de las metodologías de desarrollo de software, primero se ha recopilado información, de libros, estudios, revistas, publicaciones y eventos con respecto al ámbito del tema a tratar en la tesis. Luego, se ha analizado la documentación para separar la información relevante útil.

Se ha recurrido a fuentes bibliográficas para la búsqueda de información porque se realizará la comparación de características formales e inherentes de las metodologías.

Se revisó información de diversas metodologías de desarrollo de software, encontrándose aspectos comunes entre ellas. Se concluyó que dichos aspectos generales servirían para una primera agrupación de características, de las que posteriormente podrían hallarse criterios de comparación. Finalmente, los aspectos definidos son los siguientes:

A) Descripción
B) Principios
C) Características
D) Ciclo de vida
   - Roles
   - Fases/Etapas
   - Artefactos y/o Herramientas

Debido a las particularidades propias de cada metodología en los aspectos podrían contener algunas variaciones en sus contenidos. En el caso del aspecto del ciclo de vida, los ítems mencionados (Roles, Fases/Etapas, Artefactos y/o Herramientas) tratan de darnos una idea de los temas que deberán ser abordados en ese aspecto, mas no del esquema que se adoptará en su descripción.

2.2. Determinación de metodologías de desarrollo de software a estudiar

Atendiendo a la cantidad de información hallada con respecto a los aspectos de evaluación, se determinó las metodologías que serán incluidas en el estudio. Se decidió incluir cinco metodologías en nuestro estudio de tesis.

Dentro de las metodologías predictivas, se escogió a RUP por ser la más completa respecto a sus elementos, lo que le permite adaptarse a diferentes proyectos de desarrollo de software. Para elegir las metodologías ágiles, utilizamos diferentes criterios, cada uno por separado:

- Metodologías ágiles más usadas

  Nos valimos de la Encuesta sobre el estado de desarrollo ágil 2011 realizada por la organización VersionOne [25], donde los resultados, como lo muestra la **figura 2.1.**, son los siguientes:

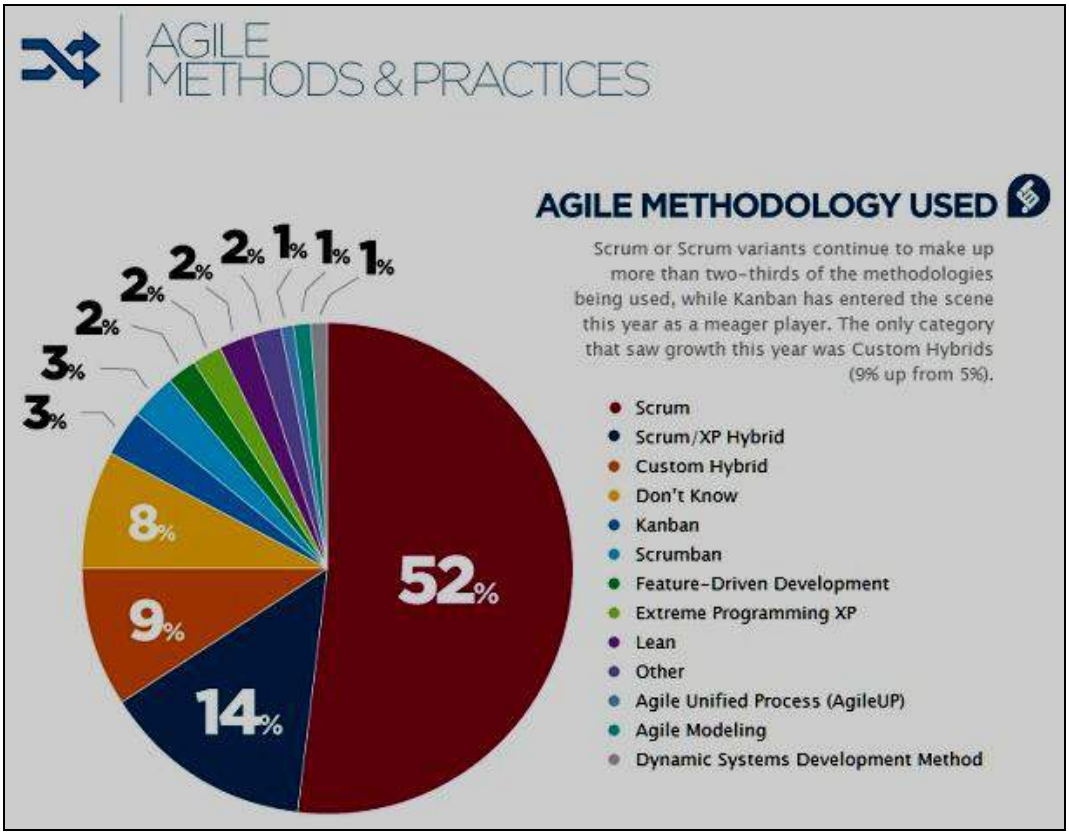


Fig. 2.1. Metodologías más usadas según Encuesta sobre el estado de desarrollo ágil. Fuente: de Garzás, J. [25]

Del cuadro circular podemos concluir que *Scrum*, ya sean por separado y combinada con otra metodología, es la más utilizada. También podemos observar que si bien *Extreme Programming* por sí sola tiene un 2% de preferencia de uso, obtiene un 14% de aceptación al utilizarse combinada con *Scrum*.

- Metodologías ágiles con contenido fundamental

Es decir cuyos conceptos han sido definidos y propuestos para el diseño y evaluación de metodologías ágiles de desarrollo de software.

Esto no quiere decir necesariamente que a partir de esta metodología se crearon todas las demás metodologías ágiles. En cambio, presenta conceptos que permiten diseñar nuevas metodologías ágiles o evaluar de entre todas las existentes (anteriores o posteriores a la que esté en investigación) cual se ajustaría más a un proyecto informático.

De las metodologías de las que se buscó información se halló lo descrito en la **tabla 2.1.**:

Tabla 2.1. Resultados de búsqueda de metodologías con conceptos y principies para diseño y evaluación de metodologías. Diseño propio.

METODOLOGÍAS AGILES	¿SE ENCONTRO CONCEPTOS Y PRINCIPIOS PARA DISEÑO?	¿SE ENCONTRO CONCEPTOS Y PRINCIPIOS PARA EVALUACIÓN?
SCRUM	NO	SI
XP	NO	NO
ICONIX	NO	NO
LEAN SOFTWARE DEVELOPMENT	NO	NO
CRYSTAL	**SI**	**SI**
AUP	NO	NO

Alistair Cockburn creó el grupo de metodologías *Crystal* en base a conceptos que le permitieron desarrollar *Crystal Clear* o *Crystal Orange*, cada uno de acuerdo a diferentes necesidades. Sin embargo, según el mismo Cockburn, esos mismos conceptos se pueden utilizar para crear nuevas o para evaluar las ya existentes metodologías [9].

- Metodologías ágiles cuyos fundamentos hayan sido aplicados con éxito en otras áreas de la ingeniería

Encontramos dos metodologías que provienen de fundamentos de producción desarrollados y utilizados inicialmente en la empresa Toyota: *Lean Manufacturing* y *Kanban*. De ellos surgen sus pares para desarrollo de software: *Lean Software Development* y *Kanban Software Development.*

Para tomar una decisión sobre cuál de estas dos metodologías será utilizada en nuestro estudio, se comparó la finalidad que tienen los conceptos originales en manufactura. En producción, mientras que *Lean Manufacturing* es una filosofía que promueve la optimización de procesos a través de la eliminación de actividades carentes de valor, *Kanban* es una de las diversas herramientas utilizadas por la primera [40]. Por este motivo, se concluye que *Lean Manufacturing* tiene una mayor amplitud de enfoque, principios y técnicas, lo que sugiere la elección de *Lean Software Development* para el estudio a realizar.

Finalmente las metodologías que han sido incluidas en nuestro estudio son las siguientes:

- *Rational unified process* (RUP)
- *Scrum*.
- *Extreme Programming* (XP).
- *Crystal*.
- *Lean Software Developemnt* (LSD).

2.3. Descripción de metodologías de desarrollo de software elegidas

Teniendo las metodologías seleccionadas, se procedió a ordenar la información de cada una según los aspectos considerados inicialmente. El resultado es una descripción de cada metodología, el cual es presentado a continuación.

2.3.1. *RATIONAL UNIFIED PROCESS*

2.3.1.1. Descripción

RUP es un proceso de ingeniería de software, desarrollado y mantenido por *Rational Software Corporation* [46]. Es un conjunto de actividades agrupadas en un marco adaptable a distintos tipos de organizaciones, áreas de aplicación, niveles de aptitud y tamaños de proyectos para producir software a partir requerimientos iniciales [52].

Sus inicios se remontan a 1997, cuando la *Rational* desarrolla *Rational Objectory Process* (ROP), siendo UML el lenguaje de modelado. Posteriormente, la organización liderada por Grady Booch, Ivar Jacobson y James Rumbaugh, adaptó a ROP nuevos elementos, tomando forma de *Rational Unified process* y lanzándolo para junio de 1998 [47].

2.3.1.2. Características

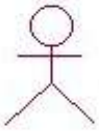
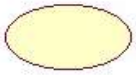
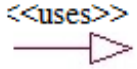
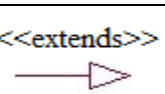
A. Manejo de requisitos

Requerimientos son las necesidades funcionales, las ventajas que le otorgan, así como los límites y lo que no será tomado en cuenta para desarrollar un software. La dirección que tomará el proyecto depende de la especificación de requisitos y se logra ello a través de los casos de uso[5] [47]. Los modelos de casos de uso, cuya función es la de plasmar los requerimientos definidos, guiarán el análisis, diseño, implementación y las pruebas, y otras actividades de modo tal que finalmente cumpla con las

[5] Caso de uso. Representación gráfica de las actividades realizadas por algún proceso.

necesidades del usuario [34]. La **tabla 2.2**. nos muestra los elementos de un modelo de casos de uso.

Tabla 2.2. Elementos de un modelo de caso de uso. Fuente: página web Casos de uso [8].

<table>
<tr><th colspan="2">Elementos</th><th>Definición</th><th>Representación</th></tr>
<tr><td colspan="2">Actor</td><td>Rol o labor que un usuario desempeña en un sistema.</td><td>  </td></tr>
<tr><td colspan="2">Caso de uso</td><td>Función o servicio que proporciona un sistema y es ejecutado por el actor, ya sea de manera autónoma o por una petición externa a él.</td><td>  </td></tr>
<tr><td rowspan="3">Relaciones</td><td>Asociación</td><td>Relación entre actor y caso de uso. El actor participa en el caso de uso.</td><td>  </td></tr>
<tr><td rowspan="2">Generalización</td><td>Uso: <<<i>uses</i>>>Inclusión, un caso de uso está incluido en la ejecución de otro.</td><td> 
</td></tr>
<tr><td>Herencia: <<<i>extends</i>>>Especialización, un caso de uso es una situación particular de otra.</td><td> 
</td></tr>
</table>

B. Centrado en la arquitectura

La arquitectura de software otorga una serie de estándares que encaminarán la organización y estructura de un software, sus elementos componentes, en lo posible agrupados de acuerdo a un comportamiento, subproceso, características o relaciones en común [47].

En RUP, la arquitectura es representada a través de modelos, que contribuirán a tener una descripción de los elementos más importantes de la arquitectura. Cada modelo proporcionará sólo una visión parcial de la arquitectura, ya sea estática, funcional o dinámica. Los modelos se plasman a través de lenguajes, es decir de diagramas. La **figura 2.2.** presenta los distintos modelos según cada vista de arquitectura. En el caso de RUP, como lo hemos mencionado líneas arriba, trabaja con el Lenguaje Unificado de Modelado (UML) [34] [47].

Fig. 2.2. Vistas de Arquitectura. Adaptado de Reynoso, C. [48]

Los casos de uso fundamentales son el punto de partida en el consecuente desarrollo de una estructura de arquitectura. A este armazón inicial se le denomina línea base de la arquitectura [52]. A su vez, la arquitectura debe crear las condiciones necesarias para desarrollar los casos de uso a lo largo del ciclo de vida del producto. Podemos deducir entonces la estrecha conexión entre ambas [47].

C. Desarrollo iterativo e incremental

El desarrollo deberá dividirse en proyectos más pequeños y más manejables. Cada uno de estos será considerado como una iteración que incluirá flujos de trabajo, y abarcará una parte funcional creciente. Es decir que los entregables de una iteración servirán como punto de apoyo retroalimentativo subsecuente [47]. Se afinarán los requerimientos (ingresando nuevos o modificando los actuales) plasmados en los casos de uso, alcanzando un progreso continuo más realista, que con otros modelos [46]. En cada iteración se efectuará incrementos progresivos en la planificación, especificación, diseño, implementación, integración prueba y ejecución.

Con este tipo de desarrollo iterativo e incremental se adquiere una mejor gestión del proyecto [52]:

- Gestión de cambios. Los cambios no son traumáticos, puesto que existe un marco de trabajo que reacciona mejor ante su aparición.
- Gestión de costos. Se incurren a menos costos por corrección, dado que no es lo mismo corregir desde el inicio del proyecto, que solo desde el inicio de la iteración.
- Gestión del trabajo. Incremento de la eficacia del trabajo por parte del equipo, debido a la forma solo trabajar lo planificado en dicha iteración: las actividades se asignan de acuerdo a prioridades, tiempo establecido y a la cantidad de recursos disponibles.

- Gestión de estructura y funciones de software. Las iteraciones, como mini proyectos, crean condiciones de equilibrio entre casos de uso y arquitectura, pues ambas ensamblan mejor cuando se avanza gradualmente.

2.3.1.3. Ciclo de Vida

A. Elementos

a) Artefactos

Es un producto que cumple funciones dentro del proceso. La primera función es servir como punto de partida y herramienta para realizar un trabajo o para él mismo ser modificado (Artefactos de entrada). La segunda, es ser resultado de un grupo de actividades (Artefactos de Salida) [52]. Corresponde, a creaciones planificadas en las fases hasta culminar el proyecto. Comprende resultados tangibles: documentos, modelos, manuales, gráficos, código y versiones ejecutables de un programa [34].

b) Hitos

Un hito en un proyecto es el momento en que se tiene listos artefactos previamente concebidos para concretar en una fase [46]. En ellos, como parte de un seguimiento, se debe evaluar el cumplimiento de objetivos del producto. En cuanto la respuesta sea positiva, se puede continuar con la siguiente fase. Caso contrario, será necesario mejorar el direccionamiento que está siguiendo el proyecto [52].

c) Roles

Es el comportamiento y las responsabilidades que adoptan un individuo o un equipo. El rol no se refiere a una persona en sí, sino una simbolización de cómo debería desenvolverse ante el trabajo. Por ese motivo que una persona puede representar varios roles y un rol ser representado por varios individuos. Un rol es responsable de un grupo de actividades y de un grupo de entregables (Artefactos) [34] [46] [52].

B. Fases

El componente dinámico en el tiempo del proceso RUP se divide en ciclos, en los cuales se genera una nueva versión del producto. Estos ciclos están organizados en cuatro fases continuas. Dentro de cada una se llevan a cabo un número “n” de iteraciones planificadas [46]. En la **figura 2.3.** hemos graficado las fases de RUP de acuerdo a las características antes mencionadas.

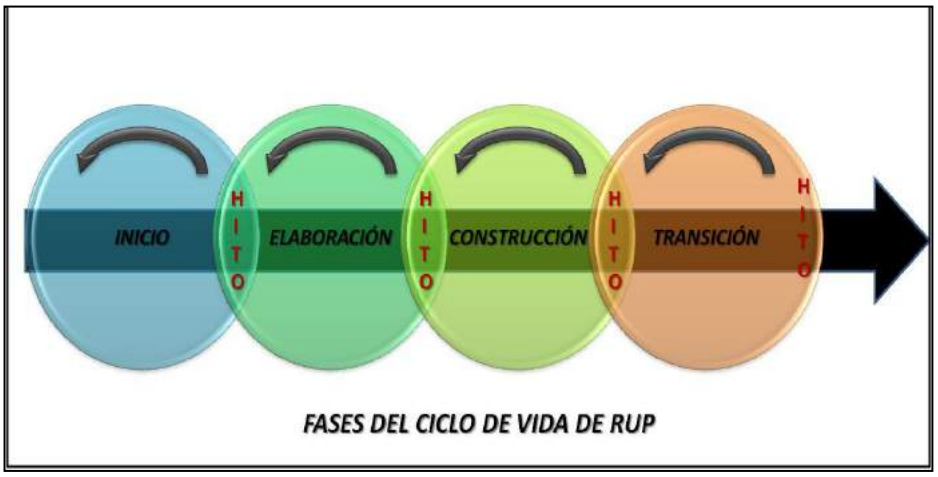


Fig. 2.3. Representación gráfica de las fases de ciclo de Vida de RUP. Diseño Propio.

a) Fase de inicio

En esta fase se debe evaluar si es factible dar inicio al proyecto. Para ello se formulará los objetivos, una estimación del alcance, los requisitos funcionales principales y su correspondiente descripción, un primer esquema de la arquitectura del proyecto y, de las limitaciones halladas, los riesgos más significativos, data fundamental para el análisis [52].

Debe ir acompañado de una planificación de recursos, hitos a completar, criterios de evaluación y aceptación, cronograma e iteraciones tentativas [46].

Tener en cuenta que este período no debe tomar un tiempo mayor de dos semanas, ni pretender obtener todos los puntos mencionados anteriormente al detalle, pues se irán puliendo [34].

b) Fase de elaboración

En la fase de elaboración consolidan los objetivos, los requerimientos y la arquitectura del proyecto. Para tal caso se buscará poseer una visión total del problema en cuestión, los requisitos deben estar bien definidos y plasmados en los casos de usos, y los riesgos críticos, que podrían amenazar seriamente la realización, mitigados o eliminados [34][46]. Con esta información, al término de esta fase obtendremos una línea base de arquitectura, conformada por un prototipo de arquitectura más adecuada para soportar el sistema, casos de prueba, casos de uso al 80% y un plan con las siguientes iteraciones [52].

Se ejecutarán un conjunto de iteraciones previstas anticipadamente, en las que progresivamente se elaborará un plan para la construcción del software [46]. Es importante que los ítems implicados sean resueltos de una manera adecuada y fiel a los objetivos reales del proyecto. Es decir, una arquitectura robusta y que admita una construcción viable y un plan comprensible, puesto que en la siguiente fase el riesgo es mayor debido al costo envuelto en su realización [34] [46].

c) Fase de construcción

Según *Rational*, la fase de construcción "es un proceso de manufactura donde el énfasis se asienta en gestionar recursos y controlar operaciones para optimizar costos, tiempos y calidad" [46]. Todo el material intelectual: Requerimientos, abstracciones, características y modelos, inclusive los no tomados en cuenta anteriormente, es integrado en un aplicativo ejecutable, una versión casi lista del producto que será entregado a los usuarios finales [34] [46].

La gestión y optimización de costos, tiempo y calidad incluye, en la medida de lo posible, utilizar técnicas o prácticas adecuadas para alcanzar aquellas, de forma más eficaz y eficiente, como la práctica de trabajos paralelos, en el transcurso de las iteraciones [34] [46].

d) Fase de transición

La fase de transición se enfoca en poner a disposición del usuario el sistema y los elementos de ayuda necesarios para que interactúe con él de manera satisfactoria [46]. Abarca las actividades que llevan a generar versiones actualizadas del producto, como completar los documentos de guía y llevar a cabo sesiones de adiestramiento y capacitación [34]. Si se desea corregir errores o incluir características nuevas, se contemplarán iteraciones semejantes como en la construcción, ejecutadas mientras que el usuario hace uso del sistema, sirviendo de retroalimentación continua y consiguiente mejora [52].

C. Áreas del conocimiento implicadas

El componente estático del proceso RUP concierne a las disciplinas y flujos de trabajo. Una disciplina corresponde a un conjunto de actividades relacionadas y que se integran en torno a las áreas del conocimiento en las que se divide un proyecto [52]. Mientras que los flujos de trabajo se refieren a la organización de roles, entregables y secuencialización de actividades agrupadas en una disciplina [46][52].

a) Modelado del negocio

El modelado de negocio es un flujo de trabajo donde se documenta el proceso donde se desea establecer un software. Lo que se desea con este grupo de actividades es una comprensión mayor de las necesidades del cliente al entender de manera más amplia la organización, los trabajadores, sus actividades o trabajos críticos, la forma como lo realiza y con quien interactúan, y hallar que procesos o partes del proceso necesitan de una mejora informática, como para ser incluidos dentro de nuestro proyecto. No se trata de estudiar a una organización por completo, sino el escenario que lo requiera. La finalidad de un software es que su uso propicie una forma de trabajo óptima de la que se puede venir haciendo, de modo que nuestro producto debe otorgar una facilidad en vez de una dificultad para los usuarios, a las metas que quieran obtener [46]. La **figura 2.4.** muestra las actividades del modelado del negocio.

Fig. 2.4. Actividades del modelado de negocio. Fuente: Torossi, G. [52].

Este flujo se puede realizar o no, teniendo en cuenta si es necesario o no para nuestro desarrollo. Por ejemplo, podría ocurrir que nuestro cliente ha elaborado una definición de requisitos en lenguaje no basado en objetos. Inclusive, algunos autores lo incluyen dentro del flujo siguiente flujo, puesto que de su ejecución es posible partir hacia la definición de requisitos que englobará el sistema [46][52].

Para formalizar nuestro negocio, nos valemos de dos modelos:

- Modelo de negocio

Representación de los procesos de negocio en una organización. Incluye dos tipos de modelos:

El modelo de casos de uso de negocio. Se vale de diagramas de caso de uso para figurar procesos implicados en casos de uso y actores, respectivamente [52].

El modelo de objetos de negocio. Presenta como clases al trabajador de negocio quienes muestran los roles que un trabajador o un grupo de trabajadores personifican, y a la entidad del negocio, referido a artefactos que los trabajadores utilizan, manejan, modifican, controlan o crean en un negocio y que están agrupados en unidades de trabajo [52].

- Modelo de dominio

Es una representación de conceptos importantes relacionados con el dominio del negocio. Utiliza como notación los diagramas de clases. Los objetos del modelo son clases que definen atributos (características propias de cada clase) y las relaciones entre ellas. Se pueden mencionar en las clases objetos tangibles o conceptos utilizados, modificados, controlados o generados; o eventos que han ocurrido u ocurrirán [52].

b) Requisitos

La captura de requerimientos, es uno de los flujos más significativos dentro del desarrollo de software. RUP enfoca la toma de decisiones sobre el curso del desarrollo, en la especificación de requisitos. Su objetivo principal es establecer lo que hará el producto, definir límites [46]. El modelo de casos ordenará los requerimientos funcionales recolectados (en actores y casos de uso) [52]. Dicho ordenamiento abrirá las puertas para llegar a un acuerdo, por medio de entrevistas con los interesados y usuarios, e iniciar un plan de acción de iteraciones, y una estimación de tiempo y costos. Estos consensos serán discutidos en entrevistas. También en este flujo se perfila una interfaz gráfica, que capte los requerimientos de usabilidad que indiquen, esta vez los usuarios [34]. La **figura 2.5**. presenta un flujo de las actividades de la toma de requisitos.

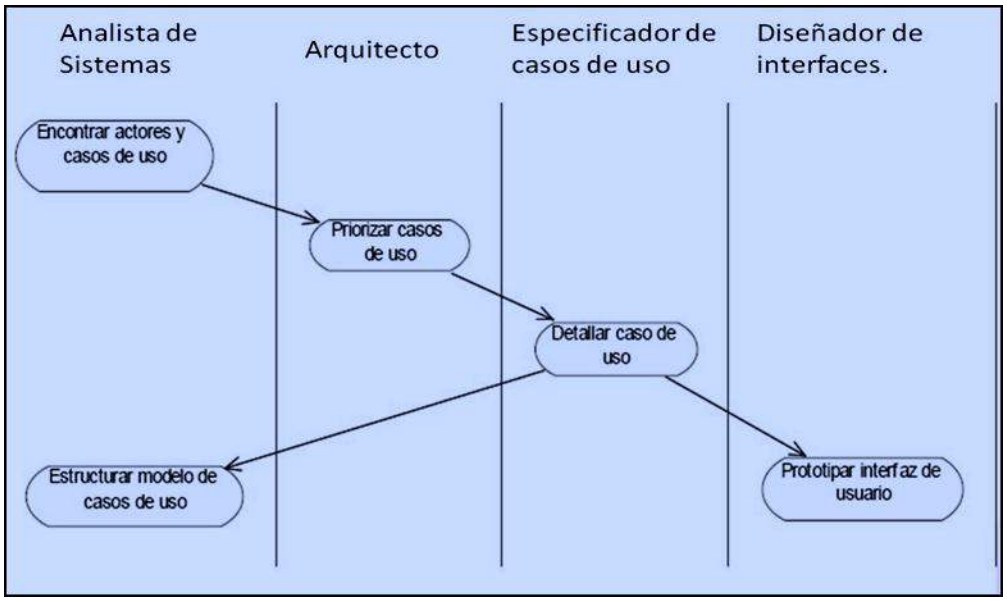


Fig. 2.5. Diagrama secuencial de Flujo de Trabajo de Toma de Requisitos. Fuente: Torossi, G. [52].

c) Análisis

En el flujo de trabajo de análisis, se convierte los requerimientos funcionales recolectados anteriormente en una guía capaz de mostrar una visión acerca de qué funciones se deberá incluir en el software [34]. El resultado final de la conversión consiste en un modelo de análisis de la funcionalidad interna del sistema, escrita en un lenguaje formal de desarrollo, donde los requisitos han sido refinados hasta alcanzar, en comparación con los casos de uso, una mayor precisión, comprensión y estructuración, facilitando su manejo y mantenimiento [52].

De acuerdo a [52], el modelo de análisis, "esboza cómo llevar a cabo la funcionalidad dentro del sistema, incluida la funcionalidad significativa para la arquitectura; sirve como una primera aproximación al diseño". Del modelo de análisis se da forma a la arquitectura. Se organiza en clases (obtenidos a partir de los casos de uso) y paquetes, y se eliminan redundancias e incongruencias.

El modelo de análisis será un artefacto de entrada base para realizar el diseño y la implementación [34] [52]. La **figura 2.6.** grafica el flujo de las actividades del análisis.

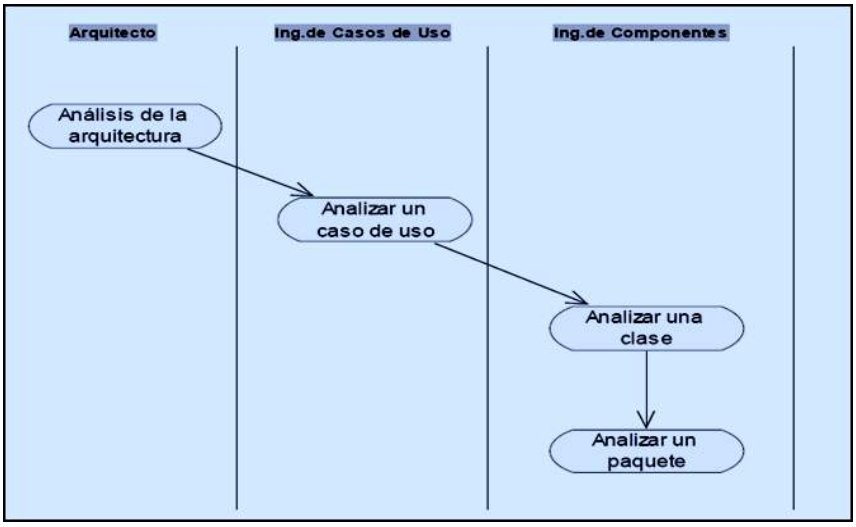


Fig. 2.6. Diagrama secuencial de Flujo de Trabajo de Análisis. Fuente: Torossi, G. [52].

d) Diseño

Para este flujo, se toman en cuenta los requerimientos no funcionales y el modelo de análisis obteniendo un modelo que especifica cómo implementar el sistema. El tránsito hacia el modelo de diseño supone una conversión desde lo conceptual hacia lo físico. Dispone la implementación, de modo que es única para este proyecto, cuando el modelo de análisis aún puede ser diseñado de distintas maneras. En el diseño se cumplen las iteraciones de la fase de elaboración, por lo que se llega a un punto preciso de formalidad. Como es último paso a la fase de construcción, el artefacto modelo de diseño es elaborado para su continuidad a lo largo del ciclo de vida del producto, por lo que tiene una mayor tipología de clases, la arquitectura consta de mayor cantidad de capas y su dinamismo se enfoca completamente en la secuencia las funciones [52]. La **figura 2.7.** grafica el flujo de las actividades del diseño.

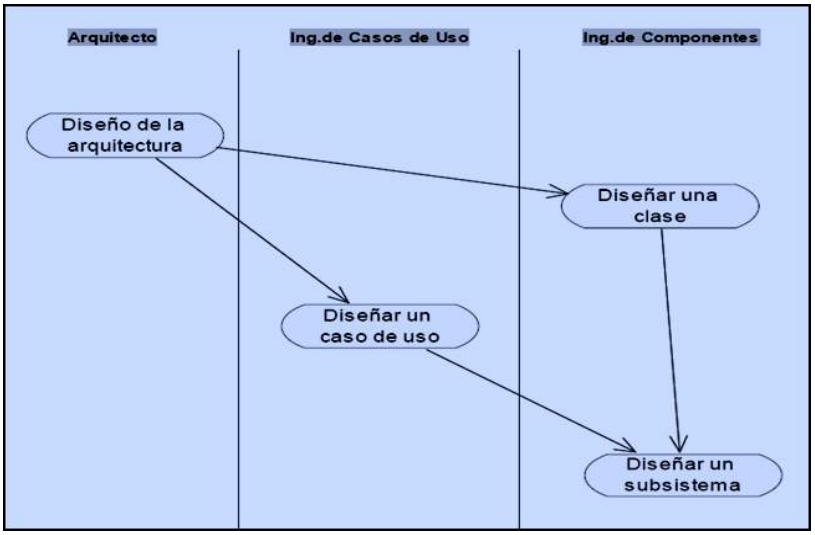


Fig. 2.7. Diagrama secuencial de Flujo de Trabajo de Análisis. Fuente: Torossi, G. [52].

e) Prueba

Las pruebas son procedimientos que nos permiten evaluar si un producto cumple con ciertas condiciones que regulan su calidad [34]. Para este proceso el flujo debe mantenerse constante durante todo el ciclo de vida del producto, al término de cada iteración, de modo que se pueda identificar deficiencias y plantear mejoras desde las primeras fases del proyecto, cuando aún el coste de desarrollo es bajo [30][47].

RUP propone tres criterios de evaluación [46]:

- Fiabilidad

Probabilidad de que durante la ejecución de las operaciones de un sistema no ocurra un fallo [57].

- Funcionalidad

El software debe actuar de manera apropiada, en concordancia con sus requisitos funcionales [34].

- Rendimiento

La rapidez en que el software realiza una tarea ante una determinada situación y condiciones predefinidas de trabajo [43].

A partir de la implementación de acuerdo al objetivo logrado que se desee evaluar, las pruebas son tituladas [34]:

- Pruebas de unidad, si se enfoca en las unidades o subsistemas de implementación.
- Pruebas de integración, a un grupo de unidades previamente integradas.
- Pruebas del sistema, cuando ya se tiene el sistema finalizado.
- Pruebas de aceptación, en el momento que el sistema entra en contacto con los usuarios finales.

Para que las pruebas se hagan efectivas, antes se deberá definir los objetivos que se quiera lograr con cada una de ellas. Luego se precisará en qué iteraciones y cuáles pruebas se incluirán en la planificación de pruebas. Se diseñarán e implementarán los artefactos herramientas para efectuar los test, como lo son los casos de prueba, procedimientos de prueba y componentes de prueba, incluidos en el modelo de casos de prueba [34]. En la **figura 2.8.** se muestra el flujo de actividades de la implementación.

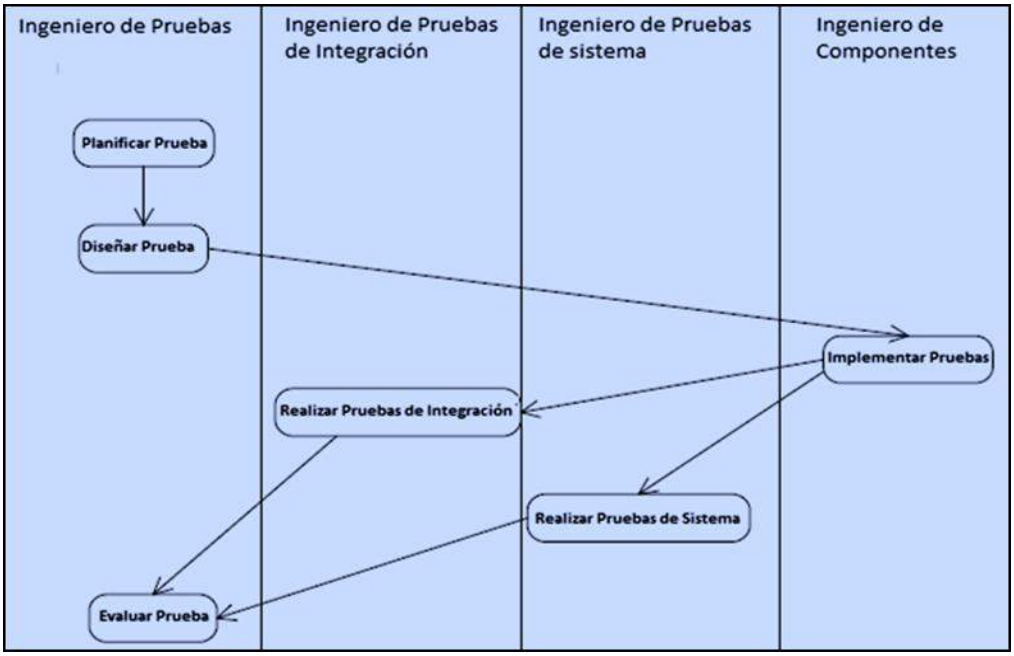


Fig. 2.8. Diagrama secuencial de Flujo de Trabajo de Pruebas. Fuente: Torossi, G. [52].

f) Implementación

Este flujo se ejecuta a lo largo de las fases del ciclo de vida, inicialmente implementando prototipos, para la línea base de la arquitectura o para reducir efectos tardíos. Sin embargo, sus actividades se enfocan más en la fase de construcción [34] [52].

En la implementación, las clases y objetos definidos se agrupan en unos paquetes llamados componentes (Ver **tabla 2.3.**) que formarán parte del sistema final [52]. El trabajo se distribuye en unidades que deben ser desarrolladas y probadas por sus responsables, para finalmente integrar incrementalmente cada fragmento en un todo y obtener un ejecutable completo [34].

Tabla 2.3. Tipo de Componentes. Fuente: Torossi, G. [52].

TIPO DE COMPONENTE	DESCRIPCIÓN
<<executable>>	Programa ejecutable en un nodo.
<<file>>	Fichero que contiene código fuente o datos.
<<library>>	Librería estática o dinámica.
<<table>>	Tabla de base de datos.
<<document>>	Documento.

Los componentes son agrupados en subsistemas de implementación. De acuerdo a una planificación de integración determinada por los implementadores, se decidirá qué subsistemas y en qué orden se procederá a construir e integrar [34]. La **figura 2.9.** presenta el flujo de las actividades de la implementación.

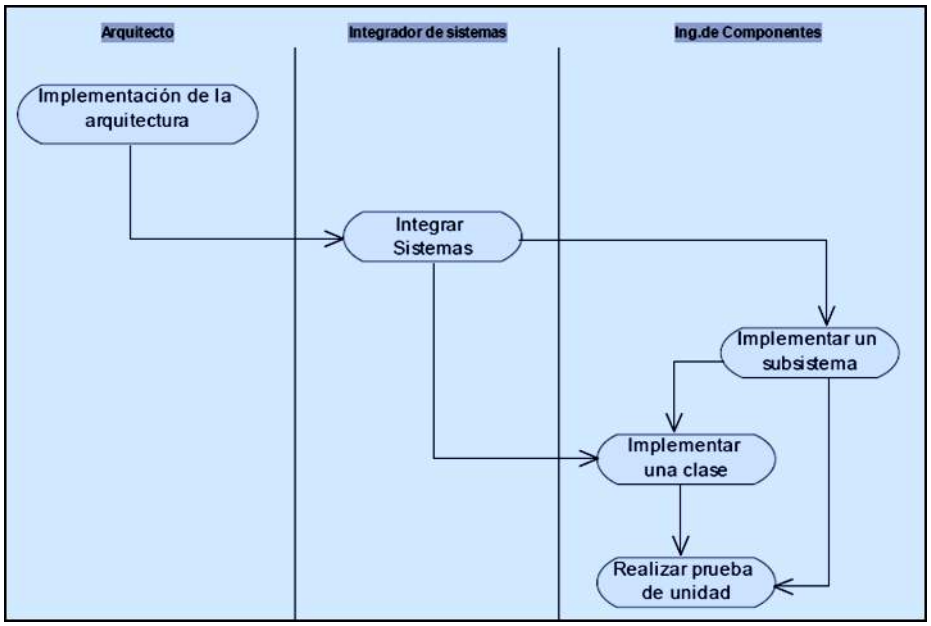


Fig. 2.9. Diagrama secuencial de Flujo de Trabajo de Implementación. Fuente: Torossi, G. [52].

g) Despliegue

Su objetivo es la producción de lanzamientos para la distribución del producto de software a los usuarios finales. Comienza en las primeras fases del proyecto, con la planificación y con la elaboración y mejora continua de manuales de usuario u otra herramienta que permita al usuario un desenvolvimiento e interacción exitosa con el sistema, hasta la fase de transición, donde se materializan las actividades establecidas en el plan de despliegue [34] [46]. Aunque RUP no profundiza en este flujo como en otros, dada la diversidad de forma de manejo, u orientación acondicionada a las características únicas de un producto, sí menciona actividades generales que deberían ser incorporadas [34] [46]:

- Empaque de software.
- Distribución de software.
- Instalación de software.
- Asistencia a usuarios.
- Migración de software o datos.
- Pruebas y aceptación a nivel de usuario.

h) Gestión del proyecto

La gestión de proyecto en RUP plantea un enfoque que si bien no garantiza el éxito del proyecto de software, aumenta las probabilidades de encaminarlo hacia él [46]. El éxito está estrechamente relacionado con la calidad del producto, en términos del cumplimiento de los requerimientos del usuario. Estas premisas se apoyan en la consecución de un equilibrio entre el logro de objetivos, la administración de riesgos y la superación de limitaciones [34]. Este flujo, permanente en la total extensión del ciclo del producto, concentra sus actividades en tres aspectos [34] [46]:

- Planificación del Proyecto

La planificación se divide en dos niveles:

- ✓ Planificación general

  Consiste en definir los recursos, el cronograma y el alcance en sentido del proyecto de manera concisa. Se atenderán puntos como provisión de personal, herramientas, programación de fechas de entrega de hitos y subhitos principales. Es el plan inicial que nos acompañará a lo largo del proyecto y es susceptible a actualizaciones y ajustes.

- ✓ Planificación específica

  Planificación de una iteración. Se elabora para cada una, aproximadamente en la mitad del curso de su antecesora.

- Administración de riesgo

Es la gestión de situaciones de incertidumbre donde existe probabilidad que el proyecto se obstaculice o fracase. Se identificarán (listado de riesgos, como documento de administración) y luego se elaborará un plan de contingencias, que contendrá las estrategias para atenuarlos.

- Monitoreo del proyecto

Actividad propia del control de las actividades, en orden de alineación a los objetivos y a la calidad requerida. Se vale de métricas, parámetros de medición de lo que deseemos evaluar. El plan de control constará con una justificación de cada métrica y una planificación de acciones frente a cambios.

Adicionalmente, la gestión del proyecto estipula la obtención del caso de negocio. Contiene información sobre las consideraciones del proyecto en cuanto a su factibilidad de ponerlo en marcha. Explicará las circunstancias del entorno, criterios de éxito, el financiamiento del mismo como inversión y las ganancias de una posible comercialización del producto (ROI- *Return of investment* - Retorno de inversión) [34].

i) Configuración y gestión de cambios

Cuando un grupo de personas está abocado a un proyecto o a una parte del mismo, en común, pueden ocurrir inconsistencias debido a la falta de integridad en la evolución de artefactos. Tres casos son documentados [46]:

- Actualización Simultánea

Error de sobre escritura. Dos o más personas/equipos trabajan por separado sobre un artefacto. El último en actualizar, elimina lo hecho por el primero.

- Notificación Limitada

Error de comunicación. Cambios hechos en artefactos compartidos por varios roles no son informados a la totalidad de quienes los manipulan.

- Versiones Múltiples

Error de propagación. Cuando una versión del producto es modificada en distintos flujos a la vez, los cambios deberían ser implementados, bajo un marco de monitoreo, en la versión de todos los flujos.

Este flujo complementario evita estas confusiones y conflictos, protegiendo la integridad del software [46]. Envuelve tres aspectos dependientes entre sí [34]:

- Gestión de Configuración

Identifica los componentes estructurados de una configuración (versiones de artefactos, productos, dependencias entre artefactos y elementos interrelacionados, etc.) para su seguimiento, y los espacios de trabajo que garanticen una labor coordinada.

- Gestión de solicitudes de cambio

Realiza la recepción de solicitudes de cambio, estudia posibles impactos de las modificaciones en la planificación y realiza un seguimiento de la ejecución del cambio. Asimismo, este aspecto se encarga de precisar la infraestructura organizacional evaluadora de impactos y definir las funciones del comité de gestión de cambios.

- Métricas y status

Provee de parámetros para captar información útil a la gestión del proyecto, proveniente de las herramientas y actividades de los dos primeros aspectos (estado del producto, nivel de avance, estado de entregas, áreas críticas, etc.).

j) Entorno

El flujo de entorno enmarca de las actividades encargadas de la configuración, adaptación y perfeccionamiento del proceso en relación proyecto a realizar. Es decir, determinar y proporcionar las herramientas, procesos, soporte y demás condiciones ambientales, adecuándolas e incluso mejorándolas de acuerdo a las necesidades del equipo de trabajo [34] [46]. Las funciones se plasman en las siguientes actividades [34]:

- Preparar el entorno para el trabajo. Ajustar RUP al proyecto de manera global y definir herramientas.
- Preparar el entorno para una iteración. Garantizar condiciones de proceso y herramientas para la ejecución de iteraciones.
- Preparar líneas de guía para una iteración.

- Brindar soporte al entorno durante la iteración. Actividad de asistencia paralela a una iteración.

El caso de desarrollo, artefacto primordial de manejo, puntualiza como se adoptará el proceso unificado, herramientas, productos y modos de uso, lineamientos guía con secuenciación y políticas de desempeño de labores y creación y modelado de artefactos [34].

2.3.2. *SCRUM*

2.3.2.1. Descripción

SCRUM es un proceso de desarrollo que sigue los fundamentos establecidos en el manifiesto ágil [2]. Toma sus principios de los estudios realizados en los años 80 por los japoneses Hirotaka Takeuchi e Ikujijo Nonaka, quienes investigaron nuevas prácticas de producción, inicialmente para productos tecnológicos. Jeff Sunderland en 1993 adaptó dichas ideas para desarrollo de software para la empresa Easel Corporation y en 1996, junto con Ken Schwaber presentó el proceso formal [37] [36].

2.3.2.2. Principios

Scrum se sostiene sobre tres principios obtenidos de la teoría del control empírico de procesos [36]:

A. Transparencia

Todos los integrantes de la administración del proceso deben conocer los aspectos y resultados que inciden sobre aquél. Estos aspectos deben tener estados definidos, con términos concretos y no ambiguos.

B. Inspección

La frecuencia de inspección debe ser suficiente como para identificar variaciones que podrían afectar de manera negativa un proyecto. El éxito de las actividades dependerá de la habilidad, cautela, eficiencia y experiencia de los inspectores.

C. Adaptación

La adaptación rápida permite ajustar el proceso cuando, a través de la inspección, se hallan aspectos fuera de los límites y que producen desviaciones al objetivo principal del proceso.

Para inspeccionar y adaptar, el proceso se vale de reuniones diarias, revisión de iteraciones y reuniones de planificación [50].

2.3.2.3. Características

SCRUM posee características de los procesos agrupados como de desarrollo ágil, es decir [3] [36]:

- Modo de desarrollo de carácter adaptable más que predictivo.
- Orientado a las personas más que a los procesos.
- Estructura de desarrollo ágil: incremental basada en iteraciones y revisiones.

Dichas características se disgregan en un conjunto de prácticas de control [37] [36]:

A. Revisión de iteraciones

Revisión al final de una iteración (llamada también *Sprint*), reuniendo a los implicados en el proyecto. Se identifican desviaciones y se realizan los ajustes necesarios para una adaptación adecuada del proyecto o del producto.

B. Desarrollo incremental

No se trabaja con diseños, modelos, abstracciones o documentación excesiva, sino que se enfoca en el producto, directamente con módulos, fracciones ejecutables del producto, para su inspección y evaluación. Los módulos van progresando hasta obtener un software final.

C. Desarrollo evolutivo

Para entornos inestables, la incertidumbre impide una predicción. Es decir, no se puede definir previamente, en una fase inicial, los requisitos completos para desarrollar una arquitectura y un diseño para producir un producto de software. *Scrum* propone técnicas de trabajo que logren una refactorización de las tareas de estructuración y diseño sin alterar negativamente la calidad del software.

D. Auto-Organización

La responsabilidad no recae sólo sobre una persona (Gestor de proyectos). En cambio, a todos los miembros se les concede una potestad ideal de toma decisiones en sus correspondientes áreas y niveles, y así reaccionar de mejor manera ante las situaciones que demanden cambios.

E. Colaboración

Para un entorno de trabajo y prácticas ágiles tal como las ofrece *Scrum*, donde prevalece la auto-organización, se propicia un ambiente en la que todos sus miembros cooperan de acuerdo a sus conocimientos y capacidades, en vez de roles. Un control eficaz implica que los integrantes estén en abierta, dinámica y constante colaboración.

2.3.2.4. Ciclo de vida

El proceso *SCRUM* divide a sus componentes en los siguientes grupos principales: Elementos, Roles, Bloques de tiempo.

A. Elementos

a) *Product Backlog* (Pila del producto)

Lista de los requerimientos del sistema, representados por características, funciones, tecnologías, mejoras y correcciones de errores que debe contener el software para ser exitoso [37] [50]. Cada elemento debe constar de una descripción, una prioridad, y una estimación de esfuerzo (magnitud de tiempo que tomará realizar una tarea determinada) [50]. No se debe pretender tener todos los requisitos desde el inicio. Se irá actualizando a través del tiempo, según los cambios del entorno. Se trabajará con el *Product Backlog* teniendo como actividades inmediatas las correspondientes a obtener los requisitos con mayor prioridad y, por consiguiente, detalladas con mayor nivel [37] [50].

El responsable del *Product Backlog* es el propietario del producto. Los miembros del equipo tienen acceso a él y participan en su elaboración [37].

b) *Sprint Backlog* (Pila de iteración)

Lista de tareas que se realizarán para alcanzar a hacer un incremento [37]. Las tareas son agregadas al *Sprint Backlog* tomando como referencia el *Product Backlog* en la reunión de Planificación del *Sprint*. Luego estos elementos son descompuestos en tareas más pequeñas hasta lograr un entendimiento mejor de cada una de ellas [50]. A cada tarea se debe asignar a personas miembros del equipo encargadas, con un tiempo y unos recursos para completarlas [37].

El *Sprint Backlog* puede ser cambiado exclusivamente por el equipo, ya sea modificando tareas y sus características (estimaciones de tiempo), agregando tareas, o eliminando las que no aportan valor [50].

c) Incremento

Ejecutable resultado de un *Sprint*. Se trata de un módulo operativo. Según Schwaber y Sutherland [50], la principal característica de un incremento es que debe estar "hecho". Para estos mismos autores [50], esto es:

- Tener un pleno compromiso por parte del equipo en hacer un elemento del *Product Backlog* en un *Sprint*.
- Que los elementos del *Product Backlog* y el incremento en su totalidad hayan completado el análisis, diseño, refactorización, programación, documentación y pruebas (testeo unitario, de sistema, de usuario y de regresión, pruebas no funcionales, pruebas de rendimiento, de estabilidad, de seguridad y de integración).
- Cualquier internacionalización que permita adaptarse a hábitos lingüísticos y culturales del grupo de usuarios que lo operará.

B. Roles

A todos los participantes se les clasifica de acuerdo dos conceptos principales:

Responsables. Son los encargados de que *Scrum* funcione dentro de la organización, del desarrollo y de que se logre el producto. Forman un equipo (Equipo *SCRUM*), donde están definidos los siguientes roles:

- Propietario del producto.
- Equipo de desarrollo.
- *Scrum Manager*.

Interesados. Personas que si bien son afectados por el proyecto y participan en él, no forman parte del equipo, ni de la toma de decisiones en el proyecto. Ej. Gerente general, jefes áreas, usuarios.

a) Propietario del producto

Responsable del *Product Backlog*, es decir, que el producto del desarrollo satisfaga los requerimientos de los clientes. El Propietario conoce a plenitud el entorno del negocio del cliente y de la visión del producto (las características que se demandan y que se espera que posea). Buscará maximizar, dentro de lo posible, el resultado del producto, para lo cual debe difundir claramente el contenido del *Product Backlog* y las prioridades de sus elementos entre todo el equipo *Scrum*. Además, en base a su conocimiento del proyecto, se encargará de su financiamiento [37] [50].

Es el nexo entre los interesados y el equipo de trabajo, pero, si bien puede recibir sugerencias por parte de miembros de la organización, sus decisiones deben ser respetadas. Él es el único acreditado a tomar decisiones con respecto al *Product Backlog* y sus prioridades. El equipo no seguirá decisiones distintas a las que el propietario manifieste [50].

b) Equipo de desarrollo

Grupo de personas que desarrollan el producto. Se encargan de la conversión del *Product Backlog* a un incremento ejecutable. Son grupos auto-organizados y auto-gestionados, porque ellos deciden la forma como llevarán a cabo esta conversión [37]. Está formado por personas con las diversas habilidades especializadas, para completar los incrementos. Estamos refiriéndonos entonces de un equipo multidisciplinar o multifuncional. Sin embargo, cada miembro no se concentra sólo en las actividades correspondientes a su disciplina, sino que deben manejar las demás, puesto que el equipo completo debe estar involucrado en todas las tareas. Esto implica compartir conocimientos entre miembros y un aprendizaje de nuevas habilidades [37] [50].

Para que el trabajo del equipo sea óptimo, y evitar limitaciones interacción (en el caso de pocos integrantes) o aumento en la complejidad y coordinaciones (cuando hay muchos integrantes, problemas que afectan la productividad, los autores proponen que el número de personas componentes sea de 7 +/- 2 personas [50].

c) *Scrum Manager*

Encargado de entrenar al equipo y garantiza que éste trabaje según el modelo *Scrum* [37]. El *Scrum Manager* instruirá al equipo con respecto a los valores, prácticas y

normas *Scrum*, su rol como miembros del Equipo *Scrum* (autogestión, multifuncionalidad, entrega) e irá guiándolos y resolviendo sus dudas. Su rol de líder permitirá que el equipo que mejore paulatinamente la calidad de su desempeño (productividad) y del software desarrollado [50].

El *Scrum Manager* puede ser un miembro del equipo, pero nunca el Propietario del Producto [37]. El rol de *Scrum Manager*, dependiendo de las características de la organización puede ser confiado a una persona en exclusivo, o asumido por los encargados de áreas como procesos, calidad, o gestión de proyectos [50].

C. Bloques de Tiempo

El proceso *Scrum*, para su mejor entendimiento, es dividido en bloques de tiempo (Ver **figura 2.10.**), intervalos donde se agruparán una serie de actividades, en pasos necesarios para construir un producto. En ellos se puede identificar reuniones de planificación y monitoreo, y el desarrollo en sí.

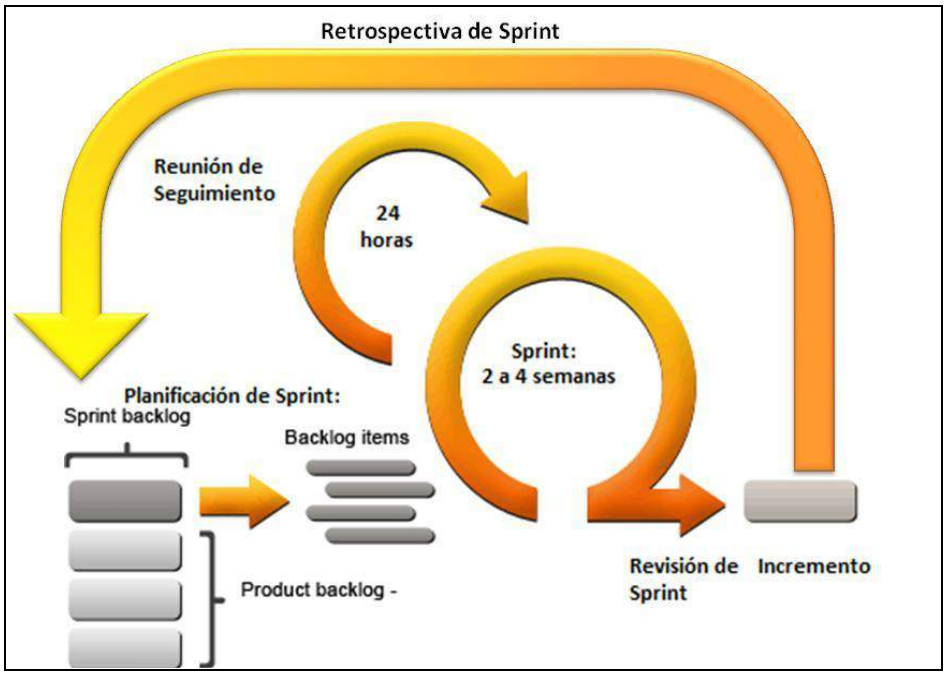


Fig. 2.10. Esquema de ciclo de vida de Proceso *Scrum*. Fuente: Metodología *Scrum* [35]

a) Planificación de *Sprint*

Jornada de trabajo previa al *Sprint*, donde de tomará decisiones acerca del trabajo y objetivos que se engloban en la iteración [37]. Según Kniberg [29], en concreto de debe fijar:

- Meta(s) a conseguir en el *Sprint*.
- Una lista de miembros participantes del *Scrum* (y su nivel de dedicación)
- Un *Sprint Backlog*.
- Una fecha fija para la revisión del *Sprint*.
- Un lugar y momento definidos para el *Scrum* Diario o Reunión de seguimiento del *Sprint*.

b) Seguimiento de *Sprint*

Reuniones diarias realizadas al inicio de cada jornada, donde los miembros del equipo, con soporte del *Scrum Manager* discuten acerca de [37] [50]:

- Lo avanzado desde la última reunión.
- Qué se avanzará hasta la siguiente.
- Problemas que les cueste realizar sus labores con normalidad y sus respectivas soluciones.

c) *Sprint*

Un *Sprint* corresponde a una iteración que durará un período de tiempo, que es recomendable que no sea mayor a un mes, donde se desarrollará el incremento ejecutable. La duración será constante para todos los *Sprint*s del proyecto. En el transcurso del *Sprint*, el *Scrum Manager* es responsable que los objetivos del *Sprint* y los integrantes del Equipo permanezcan invariables [50].

d) Revisión de *Sprint*

Esta reunión informal consiste en la presentación, utilizando lenguaje de negocio, efectuada por el equipo, del incremento funcional generado en la iteración hacia al propietario del producto y demás interesados [50], y de un debate donde se analiza y revisa [37] los siguientes puntos [50]:

- Avance y lo que aún falta por completar.
- Dificultades halladas y soluciones aplicadas.
- Preguntas y dudas acerca del trabajo completado presentado.
- Estado actual del *Product Backlog*.
- Proyección de fechas tentativas para culminación con distintos. Se exponen probables casos dependiendo de la velocidad.
- Lo que posiblemente se realizaría en el siguiente *Sprint*.

e) Retrospectiva de *Sprint*

Es una reunión en la cual, el *Scrum Manager* y el Equipo analizan, en base al marco de proceso y prácticas de *Scrum*, cómo se ha llevado el trabajo, las interrelaciones, el manejo de herramientas y el mismo proceso en el último *Sprint*. Se hallarán las fortalezas y los aspectos a mejorar o modificar a fin de incrementar la productividad en la siguiente iteración [50].

D. Herramientas de apoyo

Existen herramientas que sirven como soporte y guía en algunos aspectos del proceso. Tenemos los siguientes:

a) Pizarra *Kanban*

También conocida como pizarra de trabajo (Ver **figura 2.11.**), es una pizarra con elementos físicos removibles, que permiten una mejor visualización del avance de la iteración. Posibilita la comunicación e interacción en las reuniones de seguimiento de *Sprint*.

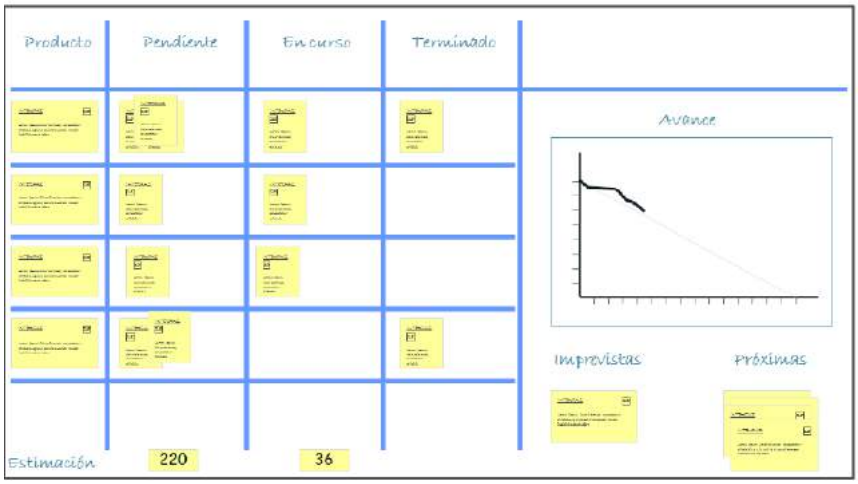


Fig. 2.11. Esquema de Pizarra *Kanban*. Fuente: Palacio, J. [37]

b) Gráfico *Burn-up*

Es un gráfico que permite al Propietario del Producto hacer un seguimiento del avance alcanzado por el Equipo hasta el último *Sprint*, comparado con la estimación de esfuerzo total necesario previsto en el *Product Backlog* para culminar el proyecto [37] (Ver **figura 2.12.**).

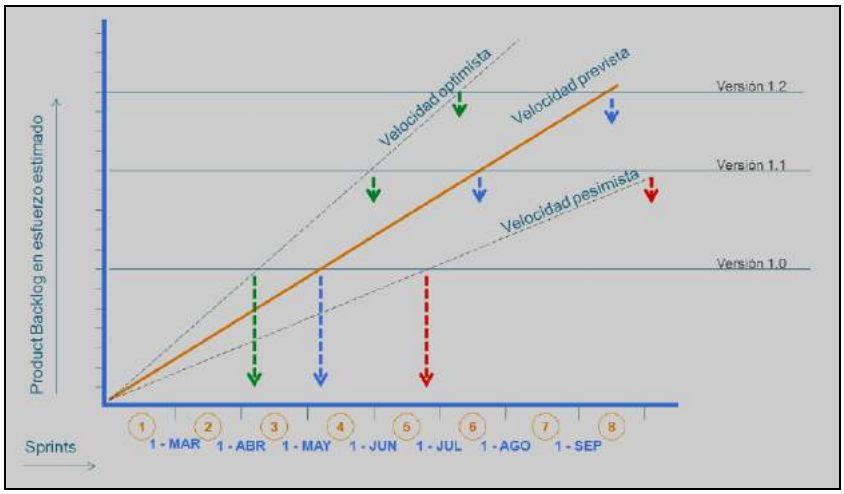


Fig. 2.12. Gráfico *Burn-up*. Fuente: Palacio, J. [37]

c) Gráfico *Burn-down*

Gráfico de ayuda para el Equipo, quien realiza un rastreo del avance diario en el *Sprint*, y además revisando el esfuerzo faltante para completar las tareas de la iteración. Se actualiza en las reuniones de seguimiento de *Sprint* (Ver **figura 2.13.**). Las desviaciones significativas del avance real con el estimado, permiten identificar si ha habido sub valoración o sobrevaloración del esfuerzo estimado para el *Sprint*,

pudiéndose tomar la decisión si se aumenta o disminuye tareas en el *Sprint Backlog* [37].

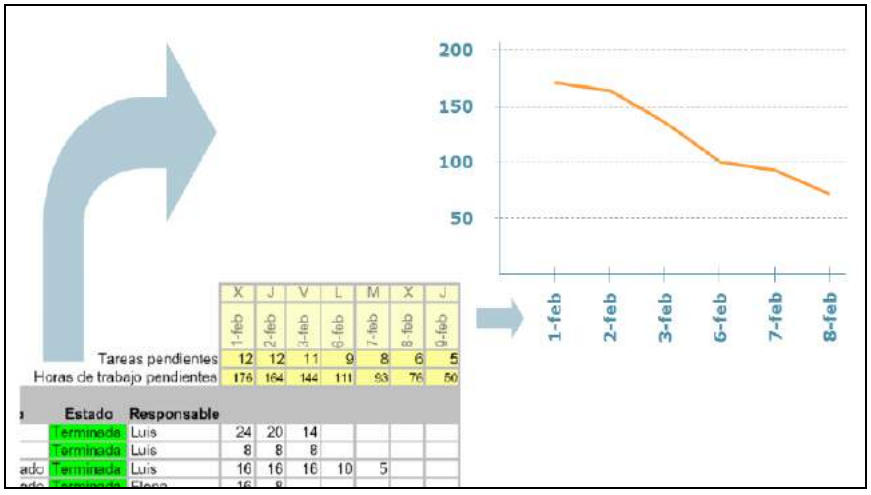


Fig. 2.13. Gráfico *Burn-down*. Fuente: Palacio, J. [37]

d) Juegos y protocolos de decisión

Estimación de Póquer: protocolo de desarrollo ágil, útil para agilizar y dinamizar la labor de asignación de estimación de esfuerzo a tareas de un *Sprint Backlog* (Ver **figura 2.14.**). En resumen, cada participante, de acuerdo a determinados criterios de trabajo del equipo, escoge del número de cartas en su poder, una que a su juicio, corresponde con la estimación de esfuerzo que él asigna a una tarea. Los resultados son consensuados y finalmente se obtiene una estimación final. En caso se decida que el esfuerzo de una tarea es mayor al límite de tamaño de una tarea, se procede a replantearla o desglosarla [29] [50].

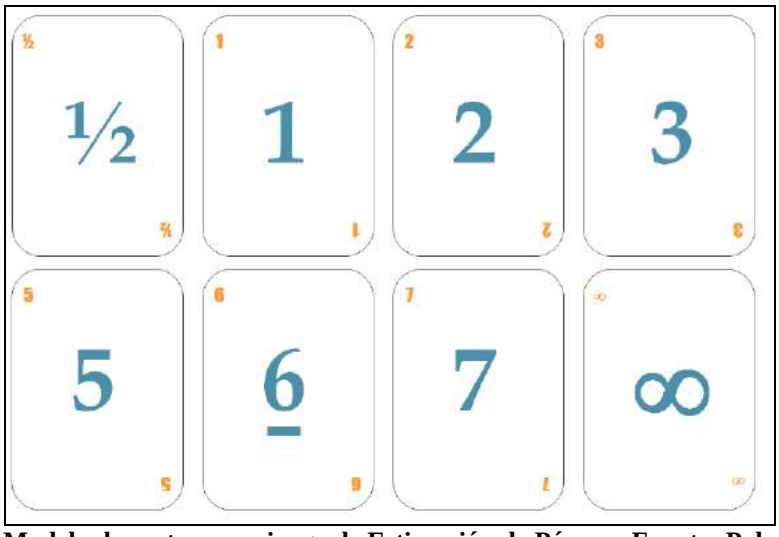


Fig. 2.14. Modelo de cartas para juego de Estimación de Póquer. Fuente: Palacio, J [37]

2.3.3. *EXTREME PROGRAMMING* (XP)

2.3.3.1. Descripción

Extreme Programming es una metodología de desarrollo de software ligera planteada por Kent Beck en el año 1999 [5]. Hacía un año él la había puesto en práctica en la ejecución de un proyecto llamado C3 (*Chrysler Comprehensive Compensation*) para la compañía automotriz Chrysler [28].

Afirma que se puede modificar la curva de costo de cambios en el desarrollo de programas a lo largo del ciclo de vida, de tipo creciente en metodologías tradicionales. Para obtener una gráfica opuesta, propone un cambio sustancial en la forma hacer un sistema [6].

XP unifica prácticas conocidas desde los inicios del desarrollo de software [49], que reunidas buscan satisfacer al cliente con software de desarrollo sencillo [42], facilitar la respuesta a los cambios que pueden experimentar las necesidades del cliente en el tiempo, y maximizar la productividad del grupo de trabajo [6].

Para conseguir tales objetivos, se requiere que gerentes del proyecto, desarrolladores y clientes (como un miembro más del equipo de desarrollo de software) sean participantes activos e involucrados en el proyecto [6].

2.3.3.2. Principios

A. Valores

Antes de abordar los principios, debemos partir desde atrás. Para sentar las bases primordiales del éxito de un proyecto, que en esencia deberá lidiar con los cambios del entorno a través de su ciclo de vida, XP presenta 4 valores [5] [6] [28] [49]:

a) Comunicación

Los integrantes del grupo de proyecto deben mantener una comunicación constante, puesto, que varias de las prácticas empleadas en XP, requieren de comunicación entre clientes, gerentes y desarrolladores. A su vez, las prácticas propician la comunicación. Muchas veces la comunicación se resiente o simplemente no existe, producto de la personalidad, mecanismos de defensa, y hábitos de las personas, es por ello que se debe contar con un coach que se percate de estas fallas para retomar el entorno comunicativo.

b) Simplicidad

Este valor incentiva a desarrollar en el día lo que se tiene planificado, sin tener que adelantarnos con funcionalidad que haría más complejo lo programado. Es mejor hacer la funcionalidad planificada para hoy y esperar para el futuro, que intentar hacer más funcionalidad, y luego tener que deshacer porque un requerimiento no se necesitó, con la demanda de costo, tiempo y esfuerzo que ha implicado en total.

c) Retroalimentación

Dentro del proyecto, debe existir un flujo de retroalimentación permanente. En la planificación, los clientes reciben retroalimentación sobre cómo posteriormente elaborar historias de usuario (Ver ítem **2.3.3.4**) o corregirlas. Las pruebas de unidad y funcionales otorgan retroalimentación a los desarrolladores sobre el estado del sistema y a los clientes de su fiabilidad y el avance para un posterior ajuste del plan general y planificación de las iteraciones.

La relación de la retroalimentación con la comunicación y con la simplicidad es directa con respecto al beneficio, pues mientras mayor sea la retroalimentación, aumentan los otros dos valores y viceversa.

d) Valentía

En XP se requiere de valentía para afrontar los problemas que se susciten, tomando decisiones incluso radicales, como el desechar parte del avance alcanzado, y por consiguiente del esfuerzo invertido, pero que a la larga resultaría más beneficioso para el proyecto.

La comunicación facilitará el consenso y el aval de toma de decisiones valientes, la simplicidad incentiva la valentía de obtener un producto tan simple como sea posible. La retroalimentación brinda a la valentía la tranquilidad de que la toma de decisiones conllevará a que el sistema pase las pruebas de sistema al culminar el proyecto.

B. Descripción de Principios

De estos valores genéricos se desprenden **principios específicos**, guía de la metodología XP y sirven de ayuda para determinar las prácticas que serán utilizadas, siendo los más importantes [5] [49]:

a) Retroalimentación Rápida

Al recibir la retroalimentación, se debe procesar y aplicar al desarrollo tan rápido como sea posible, en las magnitudes de tiempo de días y semanas.

b) Asumir Simplicidad

Enfrentar el desarrollo y la resolución de problemas, como si sus soluciones fuesen lo más simples. Se debe unir esfuerzos en realizar bien el trabajo planificado para hoy. Si se deseara añadir elementos adicionales o una mayor complejidad, deberán ser pospuestos para cuando realmente sea necesario.

c) Cambios Incrementales

Optar por pequeños incrementos sucesivos que permitan manejar cambios en las necesidades y en el entorno del proyecto, en vez de incrementos grandes que luego sean difíciles y costosos de modificar.

d) Afrontar el cambio

Se debe conservar las opciones no utilizadas para desarrollar una solución de software, en caso un cambio futuro amerite utilizarlas.

e) Trabajo de calidad

En XP son definidas cuatro variables para proyectos de desarrollo: costo, tiempo, alcance y calidad. La calidad, no es, como las otras variables, completamente negociable, debido a que no admite valores que reflejen una mala calidad. El motivo es que nadie le gusta realizar un trabajo catalogado como de calidad pobre. Es por ello que se toman las previsiones de estrategias, técnicas, herramientas y prácticas para lograr un trabajo bueno.

f) Principios complementarios

Sirven de guía dentro de situaciones particulares del proyecto. Estos son [5]:

- Enseñar aprendiendo

XP no es un conjunto de estrategias que dicen sobre lo que se debe hacer, sino unas que te permitirán captar el aprendizaje sobre cuánto hacer de cada cosa.

- Pequeña inversión inicial

El tener pocos recursos, en vez de tener todos al instante, motiva a hacer una mejor distribución de ellos.

- Jugar para ganar

No se trata de aplicar una metodología de desarrollo y actuar por cumplir con sus normas y requisitos, sino para obtener un producto exitoso a través de un desempeño que vaya mejorando a lo largo del proyecto.

- Concretar experimentos

Cada funcionalidad que se desee diseñar en el sistema, debe primero tener una unidad de pruebas, antes de ser implementada, con la que se experimentará cada abstracción a concretar. Si no se prueba, al final se obtendrá un sistema con mayores riesgos de fallar.

- Comunicación abierta y honesta

Aún siendo malas noticias, los miembros del proyecto deben sentirse en libertad de transmitirlas, sin el temor de sufrir represalias por parte de quienes son comunicados. Sólo así se podrá resolver todas las dificultades que surjan en el desarrollo.

- Trabajar con los instintos de la gente, no en contra de ellos

Las personas desean ver resultados en el corto plazo. XP posee prácticas que permiten obtener soluciones, resultados, entregables funcionales, etc. en cada una de ellas en períodos cortos de tiempo (minutos, horas, días, o pocas semanas).

- Responsabilidad aceptada

Las tareas a realizar no son asignadas a dedo y por obligación a los miembros del equipo, sino que son aceptadas por ellos.

- Adaptación local. Se debe adaptar la filosofía XP a los distintos entornos de trabajo

XP es una guía de buenas prácticas para desarrollar software. Sin embargo, no tomará las decisiones sobre qué tareas hacer para lograr un producto. Esa responsabilidad recae sobre los miembros del equipo del proyecto.

- Viaje con poco equipaje

En oposición a otras metodologías, XP no contiene una gran cantidad de artefactos complicados de elaborar y que no tienen valor más que en su formalidad. En cambio, los pocos que tiene son sencillos, otorgan valor al cliente. Mientras menos artefactos hay, menos confusión al reaccionar frente a cambios.

- Medidas honestas

No se pide estimaciones, ni medidas con un alto nivel de detalle para controlar el proceso. Basta con aproximaciones generales acordes a los instrumentos que se usarán en Programación Extrema.

2.3.3.3. Características

Extreme Programming posee características de los procesos agrupados como de desarrollo ágil, esto es [3]:

- Modo de desarrollo de carácter adaptable más que predictivo.
- Orientado a las personas más que a los procesos.
- Estructura de desarrollo ágil: incremental basada en iteraciones y revisiones.

Las características propias del proceso son descritas en 12 prácticas que deben ser aplicadas en su totalidad para aseverar que en una organización ha adoptado XP [49]:

A. El juego de la planificación

El cliente es el responsable de la elaboración de historias de usuario (requerimientos del sistema plasmados en tarjetas) [21]. Luego, en reuniones de planificación, con participación de la parte del negocio (cliente) y la parte técnica (desarrolladores), teniendo como herramienta las historias de usuario, se acordará los puntos que serán incluidos en el plan de entregas [5] [28]. En la **tabla 2.4.** se muestra los ítems del plan de entregas según sean determinados por el negocio o por los técnicos.

Tabla 2.4. Ítems determinados en el Plan de Entregas. Fuente: Beck, K. [5]

Plan de entregas	
Ítems determinados por el negocio.	**Ítems determinados por parte técnica.**
• Alcance (historias de usuario).	• Estimaciones.
• Prioridad.	• Consecuencias de toma de decisiones.
• Componentes de entregas.	• Procesos (Organización de tareas y equipo).
• Fechas de entrega.	• Programación detallada (Cronograma de tareas).

B. Pequeñas entregas

Entregas funcionales sencillas realizadas en iteraciones de pocas semanas, con valor de negocio para el cliente. Es preferible esta práctica pues cualquier cambio puede ser realizado fácilmente, a diferencia de los proyectos cuyas iteraciones duran de 6 meses a un año y luego el costo de realizar un cambio es muy alto [5].

C. Metáfora

Una metáfora es una historia de usuario que explica, a través de comparaciones con objetos reales, los objetivos, alcance, funcionamiento e interrelaciones de un sistema [6] [42]. Está escrita en lenguaje de negocio, pues su finalidad es que todos tengan una visión unificada de lo que se desea del producto en desarrollo [49]. Sentará las bases para nombrar las entidades a implementar, cuidando que guarden concordancia con la metáfora y que se pueda entender la conexión parte-todo [5] [21].

D. Diseño simple

Se refiere al diseño más sencillo que cumpla con los requerimientos del cliente [42]. En caso nos encontremos con características de diseño complejo, deberíamos reformularlas y dividirlas [21]. Obtendremos diseños más fáciles de modificar, que pasen todas las pruebas, sin duplicidades, ejecutando lo que el programador desea que haga y con menos clases y métodos [5] [6].

E. Pruebas

Para el programador, cada característica a implementar del programa debe ir a la par con su correspondiente prueba de unidad, diseñada antes de la implementación. Además, cuando se tenga las versiones entregables operativas, el cliente se encargará de realizar las pruebas funcionales del sistema. Las pruebas hacen más seguro (libre de defectos) a un programa y adaptable a cambios [5]. Se busca que los programadores exploten sus habilidades creando código con la menor cantidad de errores, en vez de invertir recursos con actividades de búsqueda de errores [49].

F. Refactorización (Recodificación)

Para mantener un código simple, se exhorta recodificar, es decir, reescribir una porción de código, sin afectar la funcionalidad, eliminando duplicidades o ineficiencias, reestructurando, mejorando y volviendo más entendible el diseño, [21] [28] [49]. Su utilidad se verifica al avanzar en las iteraciones, pues se ahorra tiempo en generación de código repetitivo y corrigiendo errores, y favorece la adición de mejoras, cambios o más funcionalidades [6] [28].

G. Programación por parejas

XP promueve el trabajo en parejas, rotativas incluso en la misma jornada diaria de trabajo, en una sola computadora. A cada uno le corresponde un rol [5]:

- Diseñar y generar el código que mejor resuelva una funcionalidad.
- Idear estrategias (pruebas, recodificaciones, identificación de errores) para maximizar la calidad del código escrito.

Es una práctica muy discutida, pero sus defensores garantizan las siguientes ventajas:

- Mejora la distribución del conocimiento del trabajo por parte de los integrantes, reduciendo los riesgos por retiro de uno de los miembros [49].
- Entregables con mejor calidad, con menos fallas, por las revisiones que realiza una persona mientras la otra programa [21].
- Se comparte conocimiento sobre áreas que un compañero no maneja y el otro sí [5].
- Reducción de costos y de tiempo en el proyecto [28].

H. Propiedad colectiva

En XP, el código generado es propiedad de todos los miembros de desarrollo, así como también la responsabilidad sobre aquel. Cualquier pareja que encuentre una oportunidad de realizar una mejora que beneficie al producto final puede agregarla, sin tener que pedir permiso a algún autor único [5]. Como resultado, todos están al corriente, si quiera en términos generales, de todo el trabajo. Esta práctica fomenta la recodificación, correcciones, y adecuación al cambio, y también apoya a reducir el riesgo de pérdida de información por retiro de un integrante del grupo [28] [6].

I. Integración continua

La integración del código desarrollado debe integrarse de manera frecuente, cuanto menos una vez al día, y por sólo una pareja [5]. De este modo, se tendrán siempre

versiones del software actualizado con las últimas características de software construidas, para sobre ellas mejorar o modificar [28]. Incluye las pruebas de integración que verifican que el avance funciona y no tiene errores. Se evita, entonces, la acumulación de componentes para integrar en la última parte de la integración, encontrar errores que echarían por la borda el trabajo de semanas [42].

J. 40 horas semanales

La Programación Extrema sostiene que una adecuada distribución del trabajo diario y semanal, impedirá una sobrecarga, un sobreagotamiento y una consiguiente reducción del desempeño de los miembros del proyecto. Se afectaría seriamente la calidad del software con trabajadores cansados y desmotivados [21]. Algunos autores amplían el campo de esta práctica al trabajo a ritmo sostenido, independientemente del número de horas semanales definidas con buen criterio [28]. Según Beck [5], horas extras constantes son un síntoma de existe algún problema, por ese motivo la regla en XP es, como máximo dos semanas seguidas de sobre-tiempo. Una salida preferible sería solicitar un reajuste al plan de entregas [28].

K. Cliente en casa

La característica novedosa en esta práctica es la inclusión del cliente como miembro del equipo [5]. El rol del cliente será de definir las historias de usuario con sus respectivas estimaciones y prioridades, en la reunión de plan de entregas con los técnicos [21]. Como los requerimientos y funcionalidades en las historias son escritos en lenguaje de negocio, es necesario que en la etapa de desarrollo el cliente sea quien resuelva las dudas y detalle cada característica, de manera presencial, al lado de los programadores [28].

Tener en cuenta que esta práctica no implica que el cliente deje de lado sus labores habituales [5]. Más bien, reduce la cantidad de documentación y permite descifrar en el momento adecuado a los programadores qué es lo que se pretende realmente del sistema [42].

L. Estándares de codificación

Debe existir un patrón regulador de la codificación, de la unidad del sistema y que además sustente la propiedad colectiva. Ese patrón se basa en la comunicación y la no duplicación de código. De lo contrario todos aportarían según su criterio, convirtiendo al programa en un grave desorden [5] [21] [42].

2.3.3.4. Ciclo de Vida

A. Roles

Beck [5] propone los siguientes roles para el proceso XP:

a) Programador

A la tarea habitual de diseñar y generar código, se suma la de diseñar pruebas de unidad e integración para verificar lo implementado. El programador debe tener la

habilidad de hallar oportunidades de mejora al diseño. La simplicidad es una meta, y la recodificación es una actividad clave constante. Su rol en XP, implica, además comunicación permanente con el equipo [5] [32].

b) Cliente

Como miembro del equipo de desarrollo, el cliente tiene un rol definido. Es el responsable de escribir historias de usuario, fijándoles prioridades y un orden de inclusión en las entregas, y de diseñar pruebas funcionales para las versiones ejecutables. Puede personificar a un grupo de personas interesadas en el sistema, sin embargo, es su obligación representar fielmente sus necesidades. Debe tener la valentía de tomar decisiones cuando se necesite incluir, priorizar o cambiar las historias de usuario. Para el aprendizaje y retroalimentación de cómo realizar sus labores, el contacto con el equipo técnico es fundamental [5] [32].

c) Encargado de pruebas

Apoya al cliente en el diseño de pruebas funcionales. Es responsable de ejecutar pruebas, publicar resultados y asegurar el funcionamiento correcto de las herramientas de prueba [5] [32].

d) Encargado de seguimiento

Recopila información permanentemente en el mismo curso del desarrollo del proyecto, realiza cálculos analíticos, en base a ello, brinda retroalimentación al equipo de proyecto respecto comparaciones, estadísticas, desviaciones entre los tiempos y recursos estimados y reales. Finalmente, recomienda soluciones o cambios para corregir defectos, optimizar la productividad o adaptarse a los cambios de objetivo, requerimientos o entorno en las siguientes iteraciones [5] [32].

e) Entrenador

Miembro del proyecto con amplio conocimiento y experiencia sobre valores, principios, prácticas y proceso de *Extreme Programming*, responsable de capacitar, guiar al equipo, detectar desviaciones, ofrecer soluciones dentro de este marco. El entrenador tratará de orientar buscando la madurez del grupo en XP, interviniendo lo menos que se pueda en el desarrollo y de la manera más cauta posible. Se pretende que el equipo, a partir de ideas, trabaje en soluciones, en vez que el entrenador lo resuelva todo, evitando de este modo el riesgo de dependencia [5] [32].

f) Consultor

Miembro experto en un área del conocimiento, requerido por el equipo de proyecto, cuando se necesite información especializada para resolver un problema. Cabe aclarar que, al igual que con el entrenador, el consultor da pautas acerca de cómo se podría llegar una solución, sin llegar a depender de este último para alcanzar la respuesta [5] [32].

g) Gestor

El gestor es el gerente del proyecto, nexo entre el cliente y el equipo de desarrollo. Garantiza que las condiciones de trabajo del equipo sean las óptimas para su mejor desempeño [32]. En virtud del cumplimiento de esta función, mantiene contacto con el equipo recogiendo sus posiciones con respecto a cómo afectaría al proyecto un cambio de escenario, estudiándolas y finalmente tomando una decisión. Se mantiene al tanto de los avances y resultados. Identifica la relación entre éstos con las prácticas y formas de trabajo adoptadas por el equipo de desarrollo. Un buen gestor debe poseer valentía para aprobar dichas prácticas y formas de trabajo, o de detenerlas para reconsiderarlas, en caso que no estén marchando ni dando resultados [5].

B. Proceso

El ciclo de vida ideal en XP se encuentra dividido en las siguientes fases (Ver **figura 2.15**):

a) Exploración

En esta fase inicial, cliente y equipo de desarrollo realizan actividades de preparación, estrechamente interrelacionadas (los clientes y programadores mantendrán constante comunicación, recibiendo retroalimentación) para la producción del sistema [5]:

- Cliente

Elaboración de historias de usuario primarias, en las que el cliente mejorará su habilidad y conocimiento para redactarlas [28] [32].

Para mejorar la composición de las historias de usuario, el cliente se retroalimentará de las necesidades de los desarrolladores. Al término de la fase, las historias de usuario estarán listas para las reuniones de planificación [5].

- Equipo de desarrollo

Experimentación con diferentes tipos de tecnologías, técnicas para ejecutar tareas y arquitectura, evaluando y comparando características, desempeño, límites y riesgos y escogiendo las mejores para desarrollar el software. Con esta información y la presentada en las historias de usuario, podrán incrementar su capacidad para estimar el tiempo que tomarán las tareas de programación de este proyecto en particular [5].

El equipo de desarrollo estimará el tiempo de implementación de las historias de usuario en un mínimo de una semana y en un máximo de tres. Fuera de este intervalo, se deberá replantear, juntar con otras historias, o dividir la historia en unas más pequeñas [21] [28].

Al término de la fase, **estimaciones de tareas y un plazo total estimado** estarán listos para las reuniones de planificación [28].

La duración de esta fase es de aproximadamente 2 a 4 semanas, obedeciendo al conocimiento previo de los programadores de la tecnología a aplicar [5] [32].

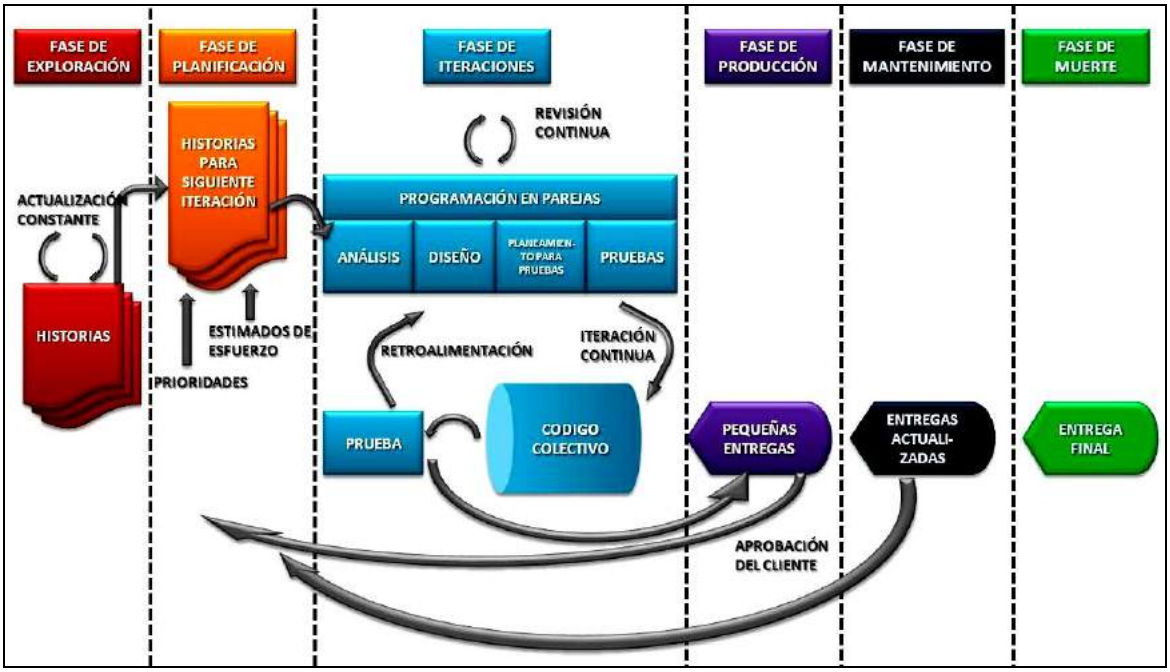


Fig. 2.15. Fases de *Extreme Programming*. Fuente: Página web Agile Methodologies [2].

b) Planificación

Reuniones permanentes donde se acuerda los puntos especificados en el Juego de planificación [5].

- Plan de entregas

En la reunión para la creación del plan de entregas, el cliente, gerentes del proyecto y equipo técnico, los clientes asignan prioridades a las historias de usuario [21]. Luego, en conjunto con las estimaciones de tiempos realizados por los programadores, se confecciona un cronograma de entregas, base para el plan de iteraciones [21] [28].
Posterior a algunas iteraciones, se pueden observar desviaciones en los objetivos o en los avances del plan de entregas, que precisarán de ajustes [28].

- Plan de iteraciones

Antes de una iteración, se convoca a una reunión para planificar el trabajo a realizar. Las historias que han sido incluidas para esta entrega y las historias que no pasaron las pruebas de aceptación de la última iteración, son analizadas por los desarrolladores y divididas en tareas y luego estimadas en días de trabajo (idealmente entre 1 y 3 días) [21] [28].

De acuerdo a la velocidad de desarrollo recolectada de iteraciones anteriores, se puede renegociar el plan de entregas [21].

- Reuniones diarias de seguimiento

Reuniones diarias de corta duración de todo grupo de proyecto, donde los participantes comunican sus avances y las dificultades que van hallando. Se analizan las posibles

soluciones [28]. Para entender mejor los problemas se pueden valer de una computadora y una pantalla [21]. Para agilizar la dinámica y acortar su duración, el modo en que se lleva a cabo es teniendo todos de pie [28].

c) Iteraciones

Aquí las historias de usuario son implementadas en versiones ejecutables. Para la primera iteración, se recomienda que los programadores elijan las historias que permitan desarrollar de la arquitectura primero, siempre que el cliente no decida algo distinto (priorizar historias que agreguen valor al negocio) [5] [32]. El cliente, como miembro del equipo de desarrollo, profundiza en el detalle de las historias que se trabajarán según el plan de entregas [28].

Antes de culminar la iteración, se ejecutan las pruebas funcionales por parte del cliente, concluyendo cuáles historias son completadas con éxito, o requieren ser enviadas a la siguiente iteración para su corrección [32].

d) Producción

Al terminar toda la fase de iteraciones, el software se deriva a producción [32]. En iteraciones planificadas, de generalmente una semana, se realizan pruebas que certifiquen que las funciones se ejecuten adecuadamente y los ajustes que demanden los cambios que aparezcan [5] [32] [28]. El riesgo es mayor con respecto a las mejoras, adiciones y modificaciones. Entonces, se debe evaluar con cautela si los complementos agregan valor o no al producto [5].

e) Mantenimiento

Durante esta parte del ciclo de vida, se exploran nuevas tecnologías, se idean más recodificaciones, por medio de la resolución de solicitudes de soporte al cliente (incluso elaborando nuevas historias de usuario), para enfrentar los cambios que aparecen mientras la primera versión del software se encuentra en funcionamiento. Se reestructura el equipo de desarrollo, debido al ingreso de tareas de soporte, que requieren rotación de personal. Miembros del equipo se retiran y otros ingresan. Estos últimos se capacitarán constantemente durante la misma realización de sus labores hasta que se hallen al mismo nivel de afianzamiento al proyecto que sus otros compañeros [5].

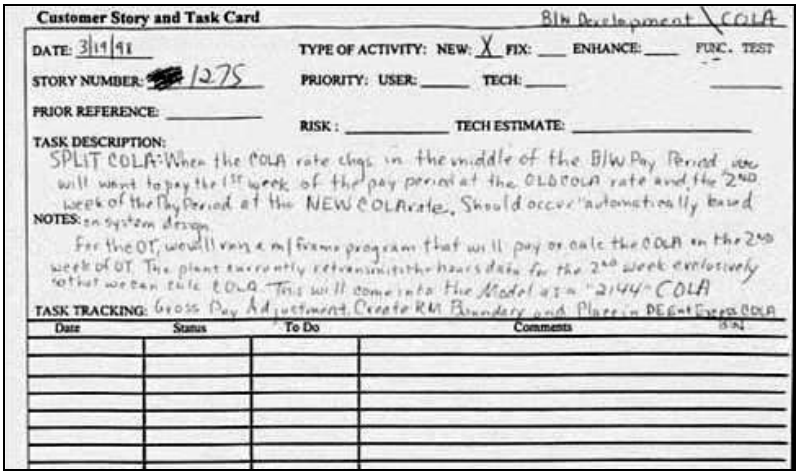
f) Muerte

El ciclo de vida se puede dar por finalizado por razones positivas, cuando el cliente está satisfecho con el producto y no tiene más características por agregar o no tiene más historias de usuario que entregar. En ese caso se debe redactar un documento guía de todo el sistema para proyectos futuros sobre el programa. Razones negativas para finalizar el proyecto son que el sistema tenga muchos errores y sea imposible hacer mejoras o cuando el costo de mantenimiento del software es insostenible para la empresa porque no devuelve beneficios económicos por su uso a dicha organización [5] [32].

C. Herramientas

a) Historias de usuario

Las historias de usuario [21] [28] son características funcionales que el cliente requiere del sistema (Ver **figura 2.16.**). Están escritos en lenguaje de negocio por el cliente. La diferencia con los casos de uso u otra documentación de requerimientos es que no necesitan un nivel de especificación compleja al inicio del proyecto. Los detalles serán dialogados con el equipo de desarrollo en la misma etapa de implementación.


Customer Story and Task Card

B/W Development \ COLA

DATE: 3/19/91

TYPE OF ACTIVITY: NEW: X FIX: ___ ENHANCE: ___ FUNC. TEST

STORY NUMBER: [redacted] 1275

PRIORITY: USER: _____ TECH: ____

PRIOR REFERENCE: _____

RISK: _____ TECH ESTIMATE: _____

TASK DESCRIPTION:

SPLIT COLA: When the COLA rate chgs in the middle of the B/W Pay Period we will want to pay the 1st week of the pay period at the OLD COLA rate and the 2nd week of the Pay Period at the NEW COLA rate. Should occur automatically based on system design.

NOTES:

For the OT, we will run a mainframe program that will pay or calc the COLA on the 2nd week of OT. The plant currently retransmits the hours data for the 2nd week exclusively so that we can calc COLA. This will come into the Model as a "2144" COLA

TASK TRACKING: Gross Pay Adjustment. Create RM Boundary and Place in PEE at Exec COLA

Date	Status	To Do	Comments	B/W

Fig. 2.16. Tarjeta de historias de usuario. Fuente: Beck, K. [5].

b) Estimación de póquer

Herramienta de estimación de esfuerzo para tareas. Creado para apoyar en *Extreme Programming*, fue luego adoptado por otras metodologías como *Scrum* (para mayor explicación, ver *Scrum* – **2.3.3.4. Ciclo de vida - D) Herramientas de apoyo**).

c) Tarjetas CRC

Las reuniones de programadores utilizan Tarjetas Clase – Responsabilidad – Colaboración. Cada tarjeta corresponde a un objeto, en la cual se agrega su responsabilidad y las clases con las que interactúa para cumplir con dicha responsabilidad [21]. Su utilidad reside en que todo el equipo de desarrollo puede participar del diseño, sin necesidad de profundizar (al menos de manera previa) en análisis más complejos [21] [49]. La **figura 2.17.** nos muestra el esquema para llenado de las tarjetas Clase – Responsabilidad.

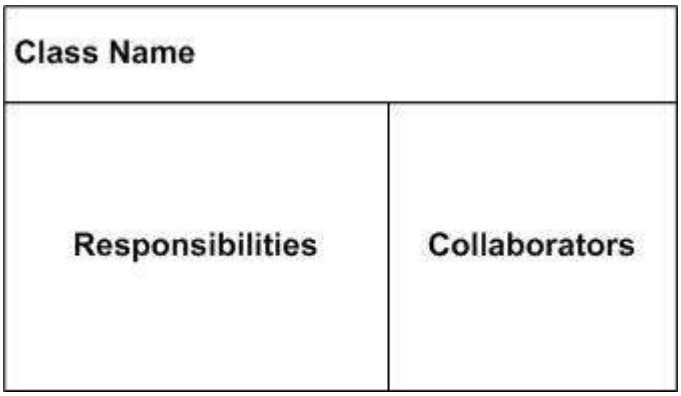


Fig. 2.17. Esquema de tarjetas Clase – Responsabilidad – Colaboración. Fuente: Amber, S. [4].

2.3.4. *CRYSTAL*

2.3.4.1. Descripción

Crystal es una familia de metodologías, resultado de las investigaciones realizadas por Alistair Cockburn. El autor plantea que dependiendo de ciertos criterios, se opta por uno u otro método.

Crystal es considerada dentro del grupo de metodologías de desarrollo ágil. De ello se puede deducir su enfoque hacia las personas. A esto se suma que todos los proyectos son distintos. Poseen particularidades que han sido observadas y documentadas. De ambas premisas, se puede partir que a medida que ciertas características de un proyecto varíen [55]:

- La agilidad debe ser expresada de manera distinta.
- Se da prioridad a una comunicación accesible.
- Se deben ajustar los procesos, las políticas, procedimientos, y actividades en el trabajo.

2.3.4.2. Principios

A. Valores

a) Enfoque en la comunicación y la persona

Esta metodología enfoca al componente humano y las relaciones de comunicación existentes entre los miembros de un grupo de proyecto como primordiales para el éxito del software. Las herramientas, artefactos y los procesos, sin dejar de ser importantes, pasan a un segundo plano, como elementos de soporte [9].

b) Alta tolerancia

Indica que *Crystal* es adaptable a diferentes entornos culturales. Además, puede ser utilizada por sí misma, o en combinación con prácticas provenientes de otras metodologías (Ej. XP o *Scrum*) [9].

B. Descripción de principios

Después de muchos años de experiencia Cockburn [9] formuló principios que, afirma, son útiles para diseñar o evaluar una metodología.

a) La comunicación interactiva y cara a cara es el canal más rápido y barato para intercambiar información

Una comunicación de contacto personal, implica una reducción de costos en el proyecto, pues facilita el desarrollo. Sin embargo, mientras mayor sea el tamaño del proyecto, se dificulta la comunicación de este tipo, perdiendo calidad, afectando al desarrollo [9].

b) El exceso en las metodologías pesadas es costoso

Una metodología pesada comprende una gran cantidad de artefactos y documentación, que por las características del proyecto probablemente no sean necesarios, entorpeciendo el enfoque en las actividades que en realidad darán valor al producto, mermando la productividad y, por consiguiente, los costos. Cabe resaltar que se trata de evitar excesos, no de eliminar elementos deliberadamente, pues algunos son críticos para garantizar la calidad del proyecto [9].

c) Equipos más grandes necesitan metodologías más pesadas

Utilizar una metodología ligera, para un equipo de desarrollo mayor (y muchas veces mucho mayor) de 10 personas, es impráctico, debido a las características de comunicación y de espacio de trabajo (todos juntos en un mismo ambiente). Se necesita, entonces de una metodología más pesada que permita las coordinaciones de comunicación entre más personas, sin caer en los excesos de las actividades burocráticas que provocan ineficiencia en el trabajo importante [9].

d) Mayor ceremonia en las metodologías, para proyectos de mayor criticidad

La criticidad expresa el daño potencial que produce un fallo en el sistema. Las pérdidas identificadas son las siguientes:

- Pérdida de comodidad. Las fallas sólo producen una pequeña molestia momentánea en las personas.
- Pérdida de dinero discrecional. Los errores suelen ser resueltos con coordinaciones simples y soporte del personal propio de una organización.
- Pérdida de dinero esencial. Fallos no son tan simples de resolver, y ponen en riesgo la existencia de una organización.
- Pérdidas de vidas humanas. Un error en el sistema puede afectar la integridad de personas involucradas, e incluso matarlos. Ej. Software de control de fugas en central nuclear o en refinerías.

A medida que la criticidad aumenta, se añade un mayor costo para impedir estos errores y daños, agregando más controles rigurosos y menos tolerancias a las metodologías [9].

e) El incremento de retroalimentación y comunicación reduce la necesidad de entregables/artefactos intermedios

Los entregables intermedios son documentos que coordinan las decisiones entre los miembros del grupo de desarrollo y comprometen a los mismos ante los clientes, a cumplir un contrato de satisfacción de requerimientos. Existen dos formas de resolver el problema de los entregables intermedios [9]:

- Reducir el tiempo de entrega de versiones, recibiendo mayor retroalimentación por parte del cliente.
- Reducir el número de integrantes del grupo. Entonces, todos cabrán en un único ambiente, donde podrán comunicarse personalmente, sin necesidad de documentos.

f) Disciplina, habilidades y comprensión contra proceso, formalidad, y documentación

Ambos conjuntos son radicalmente opuestos. Mientras la disciplina, habilidades y comprensión, propias de metodologías livianas, permiten explorar para adaptarse mejor a situaciones cambiantes, el proceso, formalidad y documentación, comunes en las metodologías pesadas, buscan optimizar costos en un entorno estático [9].

g) La eficiencia es prescindible en actividades que no son cuello de botella

Las actividades de cuello de botella son las que definen el tiempo que tomará un proyecto de software. Este principio indica que las personas que cumplen actividades cuello de botella no deben ser sobrecargadas con trabajo extra proveniente de las actividades que lo alimentan. Más bien, estas últimas tomarán tiempo para revisar sus entregables y darles mayor calidad [9].

C. Conclusiones

En base a los anteriores principios, se determina las siguientes conclusiones [9]:

a) Añadir personal es costoso

Se dificulta la comunicación y se pierde eficiencia [9].

b) El equipo incrementa en saltos largos

Si se desea aumentar la productividad de un equipo. El incremento de personal, en conjunto con el incremento de peso de metodología, debe ser casi el cuadrado del número inicial, lo cual muchas veces es económicamente poco factible [9].

c) Los equipos deberían ser mejorados, en vez de agrandados

Trabajar con sus habilidades, comunicación, interrelaciones personales, unidad y ciertos reemplazos resulta en una mayor productividad, que sólo agregar miembros [9].

d) Diferentes proyectos necesitan diferentes metodologías

Teniendo como indicadores el número de personas, la criticidad y las prioridades de desarrollo, Cockburn elabora una gráfica (Ver **figura 2.18.**) donde cada celda requerirá una metodología con diferentes pesos y rigurosidad en sus prácticas [9].

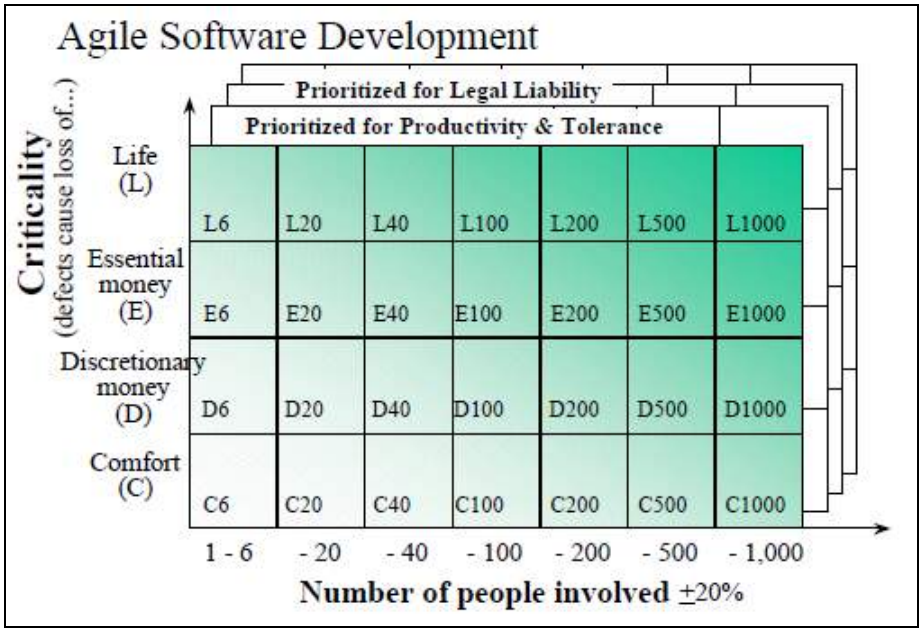


Fig. 2.18. Gráfica de Selección de Metodologìas de acuerdo a: Tamaño – Criticidad – Prioridad. Fuente: Cockburn, A. [9].

e) Las metodologías ligeras son mejores, hasta que se quedan sin combustible

Metodologías ligeras con pocas personas pueden desarrollar un producto igual o mejor que una metodología pesada. Pero tienen sus límites, donde será necesario agregar a saltos más personal, a la par con una metodología más pesada [9].

f) Las metodologías deben estirarse para adaptarse

Es más práctico y tomará menos tiempo de proyecto elegir una metodología más ligera aunque cercana a nuestras condiciones de desarrollo y agregarle unos pocos elementos para obtener logros, que seleccionar una más pesada y eliminar elementos para adaptarla [9] [12].

En base a estas premisas, Cockburn, plantea un conjunto de metodologías que se ajustarán a las variables que las determinen. Esas son las metodologías *Crystal*. De todas ellas, *Crystal Clear*, *Crystal Orange* y su modificación *Crystal Orange*/*Web* son las únicas metodologías desarrolladas y puestas en práctica hasta la fecha, siendo *Crystal Clear*, la expuesta con mayor profundidad [12] [54]. En la **figura 2.19.** se observa una adaptación de la gráfica tamaño, criticidad prioridad, se puede observar dentro de que valores de parámetros está enmarcada cada metodología.

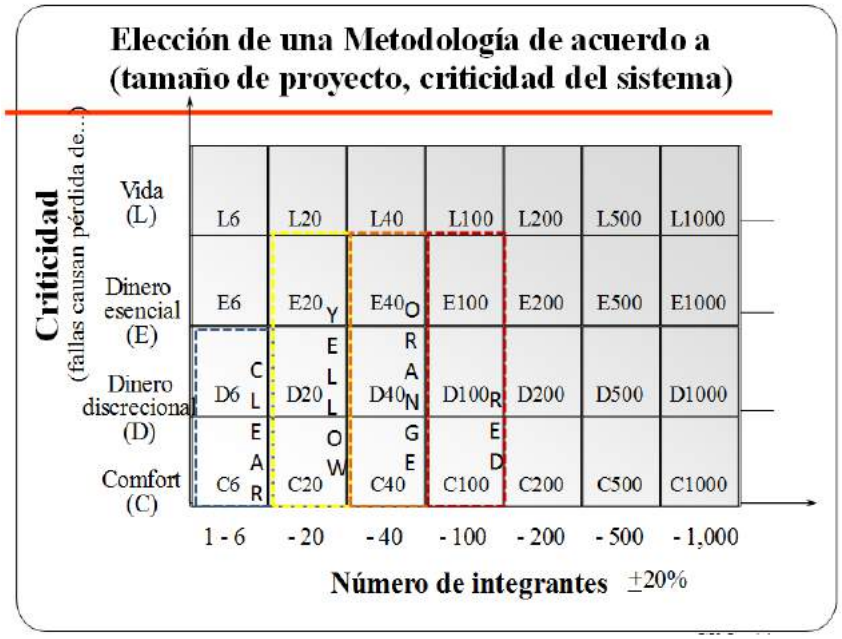


Fig. 2.19. Gráfica de Metodologìas *Crystal* según: Tamaño – Criticidad. Fuente: Cockburn, A. [12].

2.3.4.3. Características

Crystal especifica 7 características, comunes para *Crystal Clear* y *Crystal Orange* que definen a un proyecto de software exitoso:

A. Entregas frecuentes

Consiste en entregar versiones ejecutables, producto de código implementado y debidamente testeado, al cliente, con una periodicidad establecida según el tipo de

proyecto. Para *Crystal Clear*, el tiempo entre entregas es de 1 a 2 meses, mientras que para *Crystal Orange*, entre 1 a 3 meses [9] [10] [55].

B. Comunicación cercana y osmótica

Todos los miembros de un equipo, reunidos de manera presencial en un mismo ambiente, son capaces de captar información relevante. Esta práctica reduce los costos de comunicación y además propicia un clima de debate y retroalimentación, donde las ideas, recomendaciones y conocimientos en general se actualizan y ajustan a los objetivos de desarrollo [9] [10] [55].

C. Mejora reflexiva

El equipo convocará a reuniones, de unas pocas horas en la semana o mes, donde se analiza y discute problemas que creen obstáculos en el desarrollo del proyecto, a fin de concretar soluciones y ponerlas en práctica en la siguiente iteración [9] [10] [55].

D. Seguridad personal

Es la capacidad de enviar y a la vez recibir, crítica honesta con respecto al proyecto de desarrollo, en un ambiente que no genere temores ni represalias en la interna del grupo, con la finalidad de descubrir y corregir debilidades [9] [10] [55].

E. Enfoque

Esta característica, establece que el equipo de proyecto, en cada miembro y en su conjunto, debe saber en qué está trabajando actualmente, y estar en un ambiente que favorezca la dedicación a sus actividades de desarrollo [9] [55]. Se necesitan de dos aspectos para conseguir que una persona se encuentre enfocada [10]:

- Conocer los objetivos y las prioridades del desarrollo del software requerido, de modo que se evitará realizar tareas innecesarias y que no otorgan valor.
- Recibir garantía de los administradores del proyecto que las interrupciones serán reducidas de manera que se optimice la concentración en el trabajo. Se aconseja a los desarrolladores no embarcarse en más de dos proyectos a la vez.

F. Acceso a usuarios expertos

El contacto directo con el usuario es crucial para recibir mayor detalle de los requerimientos y retroalimentación de diseño, funcionalidad y calidad a través de pruebas. Mientras más rápida sea la respuesta del usuario, de mejor manera se abordará el problema en cuestión. Si no pudiese estar de manera perenne junto al grupo de desarrollo, se buscará que este el mayor tiempo posible a disposición del proyecto [9] [10] [55].

G. Ambiente técnico con pruebas automatizadas, gestión de la configuración e integración frecuente

- Pruebas automatizadas. Ofrecen confianza para diseñar la funcionalidad que se desea implementar y mayor facilidad de búsqueda de errores a corregir [10].
- Gestión de la configuración. Soporte de revisión y ajuste del desarrollo avanzado y de los recursos y herramientas utilizados, a los cambios que se presentan en el ciclo de vida de proyecto, sin afectar la unidad ni integridad del software [10].
- Integración frecuente. Mientras con mayor frecuencia se integre los componentes creados (se recomienda varias veces al día), de manera más eficiente se encontrarán y corregirán errores, en vez de dejarlo como una actividad al final del desarrollo [10].

2.3.4.4. Ciclo de Vida

A. Roles

Estos son los roles comunes para *Crystal Clear* y *Crystal Orange*:

a) Patrocinador

Responsable de toma de decisiones de negocio en concreto sobre asignación de dinero para el proyecto, basándose en la información relevante que recibe del equipo. Debe mantener un equilibrio de los objetivos del proyecto con los objetivos a corto y largo plazo de la organización. Es el punto de nexo entre el equipo y los clientes [10].

b) Usuario

El usuario tiene conocimiento directo de los procesos, actividades, interrelaciones y de las necesidades del dominio de negocio en que se enfocará la solución de software [10].

c) Diseñador líder

El diseñador líder es el desarrollador que posee mayores competencias y experiencia en proyectos de software y en la metodología a aplicarse. Es responsable de diseño y programación y de encaminar el trabajo del equipo hacia los objetivos del desarrollo. En el grupo técnico debe encontrarse al menos un diseñador líder [10].

d) Diseñador – Programador

Encargado de convertir los requerimientos del usuario en componentes que unificados, finalmente resultarán en un sistema operativo. La persona adecuada para este rol debe reunir ambas competencias, ya que mientras menos estén separadas ambas responsabilidades, no será necesario un eslabón más de comunicación y como quien diseña también programa, la calidad del resultado es mayor [10].

e) Roles comunes en *Crystal Clear* y *Crystal Orange*, a tiempo parcial en *Crystal Clear*

Pueden ser cumplidos por la misma persona que tiene a cargo los cuatro anteriores roles [9], y son:

- Experto de negocio

Garantiza una alineación de las prácticas y procesos del proyecto con el entorno de negocio, las políticas, procedimientos y estrategias de la organización, y los cambios que aparecieran en ellas [10].

- Coordinador

Ofrece a los patrocinadores una visión del estado de avance del proyecto, efectuando, en adición, el papel de conciliador entre sponsors y técnicos en las reuniones de información de avance. En *Crystal Clear* suele ser un miembro del equipo de desarrollo (muchas veces el diseñador líder) [10].

- Encargado de Pruebas

Responsable de las pruebas de unidad, de integración y de aceptación [10].

- Encargado de Documentación

Responsable de elaborar los documentos guía para el manejo del producto (Manual de usuario) [10].

f) Roles adicionales en *Crystal Orange* [9]:

- Facilitador técnico.
- Analista y diseñador de negocio.
- Gestor de proyecto.
- Arquitecto.
- Mentor de diseño.
- Diseñador de interfaz de usuario.
- Punto de reuso.

B. Prácticas

Crystal propone las siguientes prácticas:

g) Planificación

Bajo del concepto de desarrollo incremental, se realiza la planificación del proyecto subdividido en iteraciones. Se analiza los requerimientos, fijan sus prioridades y

desglosa en tareas de cada iteración. Versiones ejecutables son entregadas cada 2 a 3 meses en *Clear*, y 3 a 4 meses en *Orange* [9] [27].

h) Revisión

Al finalizar cada iteración se debe evaluar que el avance cumple con los objetivos establecidos en la planificación. Para cada elemento construido deben existir pruebas para detectar fallas y corregirlas a tiempo [9] [27].

i) Monitorización

La toma de datos de los avances del proceso y del cumplimiento de hitos permite, por medio de cálculos matemáticos y estadísticos, mantener un control de la estabilidad de casa fase [9] [27].

j) Afinamiento

Reuniones de retroalimentación, de retrospectiva y talleres, son muy útiles para mejorar puntos fuertes, hallar debilidades o problemas y darles solución [9] [27].

k) Paralelismo

El paralelismo, manifiesta que para que el software alcance un nivel adecuado de estabilidad, se puede mantener las actividades de construcción y revisión en simultáneo. Es una práctica propia de *Crystal Orange* [27].

l) Diversidad

Se refiere a dividir un grupo grande personas, en pequeños grupos multifuncionales, con el objetivo de incrementar las habilidades específicas de las personas. Es una práctica propia de *Crystal Orange* y los equipos en mención son [9] [27]:

- Planeamiento del sistema.
- Monitorización del proyecto.
- Arquitectura.
- Tecnología.
- Funciones.
- Infraestructura.
- Pruebas Externas.

m) Revisión de usuarios

Se recomienda, dos usuarios revisando cada entrega, para *Clear*. En cambio, en *Orange*, un usuario, debería revisar el ejecutable tres veces en cada iteración [9] [27].

C. Proceso

Crystal, comparte con otras metodologías ágiles el desarrollo concurrente, es decir diferentes trabajos ejecutados y alimentándose en forma paralela [10]. Como lo presentan las **figuras 2.20. y 2.21.**, estipulan ciclos generales independientes, embebidos uno dentro del otro, estos son [10]:

- Ciclo de proyecto.
- Ciclo de entrega.
- Ciclo de iteración.
- Ciclo de integración.
- Semana y día.

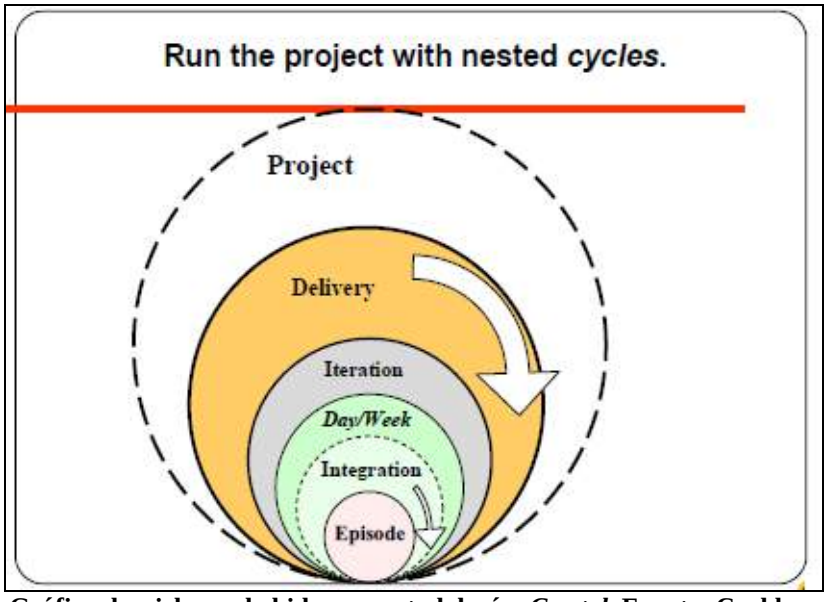


Fig. 2.20. Gráfica de ciclos embebidos en metodologías *Crystal*. Fuente: Cockburn, A. [12].

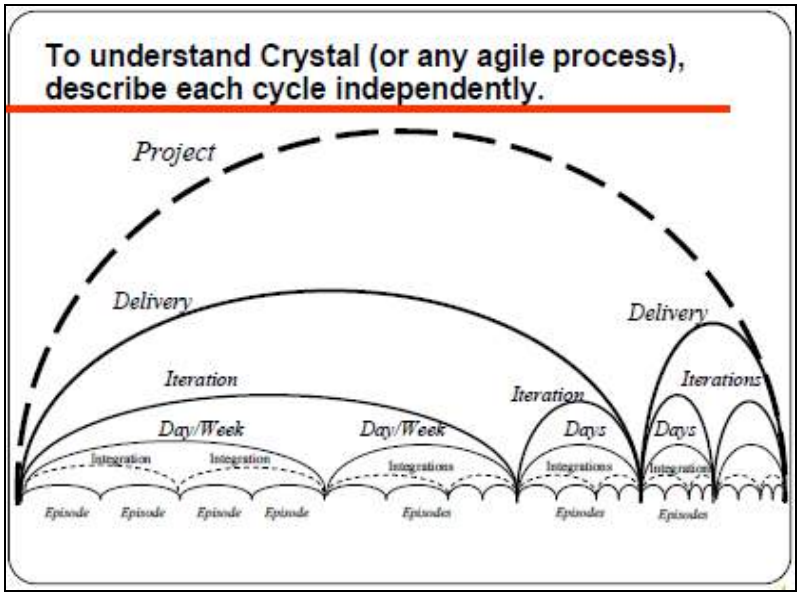


Fig. 2.21. Gráfica secuencial de ciclos embebidos en metodologías *Crystal*. Fuente: Cockburn, A. [12].

D. Artefactos

A continuación se presenta la **tabla 2.5.** con la relación de artefactos incluidos en las metodologías *Crystal Clear* y *Crystal Orange*.

Tabla 2.5. Listado de artefactos en metodologías *Crystal*. Adaptado de [9]

Artefactos en *Crystal*	
Crystal Clear	***Crystal Orange – Orange/Web***
• Secuencia de lanzamiento • Cronograma de revisions de usuario y entregas • Casos de uso o descripciones de requerimientos • Esquemas y notas de diseño • Bocetos de interfaces de usuario • Modelo de objetos común • Código ejecutado • Código de migración • Casos de prueba • Manual de usuario	• Documento de requerimientos • Secuencia de lanzamiento • Cronogramas • Reportes de estado • Documento de diseño de interfaz de usuario • Modelo de objetos común • Especificaciones entre equipos • Manual de usuario • Código Fuente • Casos de prueba • Código de migración

2.3.5. *LEAN SOFTWARE DEVELOPMENT* (LSD)

2.3.5.1. Descripción

Lean Software Development es una metodología de desarrollo de software propuesta por Mary y Tom Poppendick. Es una adaptación de los conceptos y prácticas de *Lean Manufacturing* aplicados en 3M y Toyota [31] [19]. Está enfocado principalmente en las prácticas y herramientas que una organización debe poner en práctica para gestionar equipos de desarrollo de software [16].

El objetivo de la filosofía *Lean* es la mejora de la productividad y la construcción de productos de calidad. Para conseguirlo, se enfocan en el personal como elemento principal de la organización, centro de los procesos de una empresa, y una fuente de habilidades, conocimientos y nuevas ideas que deben ser tomados en cuenta [31].

Se utilizó los conceptos de liderazgo y autogestión para mejorar el desempeño y se formularon principios que debían ser implantados radicalmente en la cultura organizacional de las empresas [31] [26]:

- El valor del producto está definido por el cliente.
- Cadena de valor, eliminando desperdicios, es decir lo que no agregue valor al producto.
- Diseñar un flujo continuo para el proceso.
- Producción impulsada, por las necesidades del cliente.
- Perfección, tendiendo a la eficiencia y buscando continuamente la mejora y el valor.

2.3.5.2. Principios

Los Poppendick adecuaron los principios de *Lean Manufacturing* para desarrollo de software, resultando los siguientes:

A. Eliminar desperdicio

Lean considera desperdicio a todas las actividades que no agregan valor al desarrollo ni al producto desarrollado [39]. En otras palabras, sólo se tendrá en cuenta e invertirá recursos en lo que otorgue un valor real al proyecto [31]. Se debe analizar qué partes del proceso no cumplen con esta característica para eliminarlas [19].

B. Amplificar aprendizaje

Debido a los cambios que ocurren en el entorno, es imposible recolectar todos los requerimientos al inicio del proceso. Por esta razón, el aprendizaje implica que el equipo del proyecto mantenga constante contacto con el cliente durante todo el ciclo de vida del software [31].

El aprendizaje es un proceso repetitivo, de forma iterativa. Se recomienda las iteraciones debido a que al final de cada una, se muestra una versión incremental al cliente, recibiéndose retroalimentación y mayor detalle. Mientras más frecuentes las iteraciones, mayor es el aprendizaje [38] [19]. Así, se llegará a reaccionar mejor frente a los cambios y obtener, de manera progresiva un sistema que se ajuste a las expectativas del cliente [40].

C. Postergar las decisiones

En un escenario cambiante constantemente, se prefiere tomar decisiones críticas cuando se tenga suficiente información concreta en que basarnos. Esta información se obtiene a través de retroalimentación producto del desarrollo iterativo e incremental [19]. A medida que continúa el aprendizaje, elaboramos diferentes opciones para mantener en espera la toma de decisión, hasta alcanzar la más adecuada [38].

D. Entregar rápidamente

Realizar entregas rápidas que abarquen las funcionalidades prioritarias y oportunas en ese instante para los clientes [31]. Cuando se entrega más rápido se elimina desperdicio, debido a que sólo se ejecutarán actividades y tareas que produzcan lo que ofrezca valor al cliente en ese período [40]. El desarrollar un entregable de calidad implica no dejar de hacer tareas que resultarían más costosas si se realizaran en últimas fases (como las pruebas y la integración) [38].

El usuario retroalimenta con mayor frecuencia, y resulta en mayor confiabilidad [31] [38]. A cambio de ello, hay mayor flexibilidad para los clientes, pudiendo cambiar de idea y de curso de los proyectos en el momento que lo requieran. En conclusión, clientes y desarrolladores ganan, pues, mientras más rápido se entregue, más capacidad para postergar decisiones [39].

E. Facultar al equipo

El líder de proyecto es responsable de guiar al equipo dentro de los objetivos del proyecto, la filosofía y prácticas de LSD, y confía al equipo la responsabilidad de su autogestión y auto-organización [31]. Esta autonomía para tomar decisiones, viene acompañada de unas habilidades y conocimientos previos, que son anteriores al proyecto o son fruto de un entrenamiento dentro en el mismo. La delegación maximiza las destrezas, incentiva el aprendizaje y mejora continua para actuar posteriormente ante situaciones y dificultades que se presenten [38].

F. Construir integridad

Existen dos tipos de integridad [38]:

- Integridad Perceptiva. Es la percepción que tiene el cliente respecto al cumplimiento del software de sus requerimientos.
- Integridad Conceptual. Significa que todos los componentes del software trabajan como una unidad funcional. La integridad perceptiva es un requisito indispensable para la integridad perceptiva.

Un software bien integrado es adaptable, es decir, es más fácil de mantener y modificar [31] [19].

G. Ver el todo

En un sistema, cada uno de sus componentes interactúa para conseguir un objetivo. Una situación que afecte uno solo de esos componentes, definitivamente comprometerá el trabajo de los demás [39]. Eso mismo sucede cuando tratamos de mejorar una de las partes sin pensar en sus interacciones. En vez de optimizar, suboptimizamos el sistema [31] [38] [19].

Cuando el equipo o uno de los miembros del proyecto analicen un error, o intenten realizar una mejora, debe hacer uso del pensamiento sistémico. Así mismo, el equipo del proyecto debe ser evaluado en cuanto a los resultados conseguidos por todo el equipo y no por el desempeño personal [38].

2.3.5.3. Características

Esta metodología como características posee prácticas propuestas, a las que llama herramientas, que ayudarán a lograr satisfactoriamente los principios establecidos. Son divididas de acuerdo a los principios a los que brinda soporte:

A. Eliminar desperdicio

a) Herramienta 1. Identificar los desperdicios

Aprender a reconocer "aquellos pasos que no contribuyen directamente con el producto final y en vez de ello elevan su costo" [39]. A continuación, una lista de los desperdicios adaptados a la versión de *Lean* para desarrollo de software [38] [40]:

- Trabajo parcialmente hecho (inventario).
- Procesos extra, para generar una sobrecarga de documentación (burocracia).
- Funcionalidades extra que no se encuentran entre los requerimientos del cliente.
- Cambio en las tareas, ya que se requiere tiempo para reenfocarse en una nueva tarea, problema típico en la asignación de proyectos múltiples.
- Tiempo de espera, de información, de decisiones, o de artefactos de entrada al rol.
- Transferencia de conocimiento a través de documentación.
- Defectos que no son prontamente detectados en tests iniciales, desperdician tiempo y recursos al analizarlos posteriormente.

b) Herramienta 2. Mapeo de la cadena de valor

Analizar valiéndonos de mapas de cadena de valor los tiempos de espera y los tiempos de valor de un proceso. Aquellas actividades cuyos tiempo de espera sean mayores deben ser estudiados para hallar las fuentes de demora, planificar cómo reducirlos y diseñar un nuevo mapa de cadena de valor [39] [19] [40].

B. Amplificar aprendizaje

a) Herramienta 3. Retroalimentación

Mantener una constante comunicación entre componentes del equipo, clientes e interesados. La retroalimentación constante permite reaccionar frente a cambios, detectar fallas en el avance del sistema, mejoras en la forma de realizar el trabajo y la gestión del proyecto, y una mejor toma de decisiones [39] [19] [40].

b) Herramienta 4. Iteraciones

Períodos cortos de tiempo donde se produce incrementos ejecutables del software final. Estos incrementos son mejorados al avanzar las iteraciones. Cada iteración es planificada, incluyendo en ellos las funcionalidades requeridas por prioridad, las tareas y los recursos para lograr las versiones [39].

c) Herramienta 5. Sincronización

Varias personas realizan tareas sobre una misma actividad, lo que hace necesario que exista sincronización para evitar inconsistencias, incongruencias e incompatibilidades entre avances, se unifiquen y que al final del día se tenga como avance un todo integrado. Las pruebas de integración son muy útiles para detectar fallas, corregir y mejorar la construcción terminada en la jornada [39] [19].

d) Herramienta 6. Desarrollo concurrente

En desarrollo de software, para tomar una decisión, deben diseñarse una serie de límites que encaminen los acuerdos, y así, las discusiones no se tornen un círculo sin término. En la solución de un problema, estos límites se llaman soluciones y en base a como funcionen, se elegirá la alternativa de manera más ágil [39].

C. Postergar las decisiones

a) Herramienta 7. Pensamiento de opciones

El tomar una decisión crítica al inicio, tiene mucho riesgo, pues hay incertidumbre sobre qué sucederá. En cambio, las opciones son experimentos para el aprendizaje. Necesitamos aprender, para recabar información y así tomar una decisión adecuada [39].

b) Herramienta 8. Último momento responsable

El último momento responsable según [39], es "el momento en que fallar al tomar una decisión, se elimina una alternativa importante". Este autor también presenta algunas tácticas para tomar decisiones en el último momento [39]:

- Compartir información de diseño completada parcialmente, y descubrir la demás durante el proceso.
- Organizar para una colaboración directa entre miembros, compartiendo información entre clientes y desarrolladores.
- Desarrollar un sentido de cómo absorber cambios, esto se gana cuando se obtiene experiencia.
- Desarrollar un sentido de qué es críticamente importante en el dominio, y considerarlo, según prioridades, desde el inicio.
- Desarrollar un sentido de cuando se deben tomar las decisiones, aprendiendo a desarrollar algunas necesidades de soporte y a aprovechar las oportunidades.
- Desarrollar capacidad de respuesta rápida, desarrollando de manera rápida en iteraciones, sólo decidiendo sobre lo que se hará en ese período y postergando otras decisiones.

c) Herramienta 9. Toma de decisiones

Para un entorno cambiante, se requiere un enfoque principal de amplitud, en la cual la persona tiene experiencia para enfrentar los cambios e identificar oportunidades para la toma de decisiones. Un enfoque principal, por el contrario, de profundidad requeriría alguien que pueda tomar decisiones desde el inicio y la seguridad que el entorno es estable [39].

D. Entregar rápidamente

a) Herramienta 10. Sistemas impulsados

En un sistema impulsados, los clientes entregan sus requerimientos en forma de historias de usuario, para que el equipo, a través de un grupo de criterios y reuniones

definidas, acuerden una planificación en interacciones (de duración no mayor al mes) de acuerdo a la capacidad de los miembros, auto-organización de tareas correspondientes a las historias, y retroalimentación para mejorar aspectos del desarrollo y solución de problemas, manteniendo permanentemente, además, una visión del avance del trabajo [39] [40].

b) Herramienta 11. Teoría de colas

Los conceptos de teoría de colas y de tiempos de ciclo, son importantes para identificar los cuellos de botella y las colas de espera en un proceso de desarrollo de software. Luego, se debe de mejorar la forma de alimentación entre subprocesos y tareas dependientes, reordenar la organización y frecuencia de paquetes de alimentación y reducir los tiempos de ciclo, para incrementar la rapidez de las entregas [39] [40].

c) Herramienta 12. Costo de demora

Construcción de modelos financieros que muestren a los interesados del proyecto el impacto que produciría agilizar o demorar la introducción del producto de software. Los beneficios o las pérdidas presentadas, crearán conciencia en la organización de la importancia de implantar un enfoque ágil al proceso de desarrollo de software [39].

E. Faculta al equipo

a) Herramienta 13. Auto-determinación

Un proyecto debe estar enfocado en las personas que desarrollan el producto. Ellos deben ser los encargados de gestionar, organizar y distribuir su trabajo, en vez que un gerente o un administrador les diga qué hacer [39].

b) Herramienta 14. Motivación

Para formar la motivación en los miembros del equipo de proyecto se necesita [39]:

- Dar a conocer el sentido y el propósito del cumplimiento de sus tareas, y que tengan la autonomía de gestionarlo.
- Cultivar en ellos una sensación de pertenencia, seguridad, competencia y progreso de su trabajo.

c) Herramienta 15. Liderazgo

No es lo mismo administración que liderazgo. Un administrador se encarga de controlar que el proyecto y sus componentes trabajen dentro de unos estándares o acuerdos. Un líder, en cambio, ha realizado un trabajo conceptual inicial sobre un proyecto, poseyendo un sentido de pertenencia sobre el producto, lo que le permite dar a conocer al equipo los objetivos del proyecto y a la vez maximiza las habilidades de cada miembro. Entonces, se requiere líderes que motiven a su equipo por su compromiso y dedicación [39] [40].

d) Herramienta 16. Pericia

En un proyecto se debe incluir un grupo de personas especializadas y expertas en distintas áreas del conocimiento (técnicas y de domino), de modo tal, que se forme una ventaja competitiva sobre otros competidores. Debe ir acompañada de una estructura organizacional que favorezca el desarrollo de la experiencia [39].

F. Construir integridad.

a) Herramienta 17. Integridad percibida

De acuerdo a [39], la integridad percibida se consigue estableciendo flujos de información, y, para ello se consideran las siguientes técnicas:

- Para sistemas pequeños, equipos simples de desarrollo que mantengan contacto directo con los clientes, y usen cortos periodos de tiempo en las iteraciones.
- Incluir pruebas de usuario para retroalimentación.
- Lenguaje entendible para los clientes y útil para los desarrolladores en el diseño.
- Para sistemas grandes un desarrollador experto con vasta experiencia en requerimientos de usuario, que dentro del equipo sea un representante de los intereses del cliente.

b) Herramienta 18. Integridad conceptual

Según los Poppendieck [39], la integridad conceptual se consigue considerando los siguientes aspectos:

- Entender y resolver un requerimiento o problema de software al mismo tiempo (Ej. desarrollo centrado en pruebas).
- Información previa al inicio y el flujo de información completa en el transcurso del desarrollo, cuando se haya recabado totalmente.
- Información enviada en pequeños paquetes, en vez de todo a la vez.
- Flujo de información bidireccional.
- Preferencia por la comunicación cara a cara, en vez de documentaria.

c) Herramienta 19. Refactorización

Identificar oportunidades de mejora y simplificación del código escrito para una funcionalidad (Ver XP).

d) Herramienta 20. Pruebas

El código escrito para una funcionalidad debe ir acompañado de pruebas de unidad, funcionales, de integración cuando se unifican sus componentes y de clientes al final del desarrollo. Éstos asegurarán que el sistema ejecuta las funciones requeridas de forma esperada, sin errores y como un todo. La pruebas se diseñan y ejecutan durante todo el desarrollo, pues son útiles para comunicar que debe hacer una función, recibir retroalimentación, y como fuente de mejoras y cambios [39].

G. Ver el todo

a) Herramienta 21. Medidas

Enfocarse en medidas que representen logros grupales en lugar de individuales. Si aplicamos mejoras sólo locales, se afectará la organización del proyecto, y sub optimizará su desempeño como un todo. Para lograr métricas que evalúen el todo se cuenta con técnicas como la agregación y las métricas de información [39] [19].

b) Herramienta 22. Contratos

Se necesita contratos donde la responsabilidad del riesgo por los cambios sea compartida por cliente y proveedor, donde ambos se fijen metas de manera unificada y trabajen en conjunto para conseguir calidad y beneficios según un marco. Los tradicionales contratos de precio fijo no cumplen con estas características, como si lo hacen sus pares de costo-objetivo, costo-cronograma y beneficios compartidos [39] [40].

2.3.5.4. Ciclo de Vida

LSD no define ningún ciclo de vida para desarrollar software. Sin embargo, dentro de sus principios, y prácticas identifica los siguientes pasos para lograr entregas rápidas [39]:

- El cliente elabora historias de usuario.

- Reunión de planeamiento

Los desarrolladores estiman el tiempo que tomaría implementar cada historia, mientras los clientes otorgan prioridad a estas últimas. Según los tiempos estimados y las prioridades, las historias de usuario son organizadas en iteraciones.

- Reunión de iteración

El equipo de desarrollo define las tareas necesarias para completar cada historia. Los desarrolladores eligen las tareas sobre las cuales trabajarán (Autogestión).

- Iteración

Incluye reuniones diarias de 15 minutos, entre el equipo de desarrollo, donde se discute los avances realizados, cómo se realizará el trabajo del día, y las dificultades halladas, para encontrar soluciones adecuadas. El avance es controlado visualmente utilizando pizarras kanban, gráficos burn – down y gráficos de aceptación de pruebas. Al final de cada iteración se realizan pruebas de aceptación del usuario, donde se evalúa cuáles son las funcionalidades que pasan las pruebas, o las que requieren mejoras o cambios y son enviadas a la siguiente iteración.

Como parte de facultar al equipo de proyecto en su autogestión y auto-organización, deja a su criterio los roles y artefactos que adoptará [15].

Capítulo 3

ESTUDIO COMPARATIVO DE METODOLOGÍAS DE DESARROLLO DE SOFTWARE

En el presente capítulo realizaremos un estudio comparativo que evaluará las semejanzas y diferencias existentes entre las metodologías de desarrollo de software analizadas en el capítulo anterior.

3.1. Establecimiento de criterios de comparación

Los criterios a comparar han sido determinados después de haber recopilado y ordenado la información de las metodologías estudiadas en el capítulo anterior. Para ello se ha tenido en cuenta principalmente particularidades encontradas en las metodologías en las cuales se podrían establecer paralelos. De manera complementaria, se han tomado como guía el estudio realizado por Abrahamsom, Salo, Ronkkainen y Warsta [1], y los criterios de gestión flexible en el software propuestos por Palacio [37].

A partir de lo mencionado anteriormente, las particularidades se organizado en criterios. Estos criterios se han reunido según sus similitudes, en tres grupos destinados a facilitar el estudio:

- Procesos.
  - ✓ Predicción – Agilidad.
  - ✓ Frecuencia de flujos de trabajo.
  - ✓ Definición de prácticas, roles y tareas.
    - o Prácticas.
    - o Roles y tareas
  - ✓ Definición de entregables/artefactos.
  - o Cantidad y complejidad de entregables/artefactos.
  - o Propiedad de entregables.
- Personas.
  - ✓ Equipo de trabajo.
    - o Número de integrantes.
    - o Comunicación entre integrantes.
    - o Cambio de integrantes.

- ✓ Clientes.
  - o Participación en el proyecto.
- Organización y proyectos.
  - ✓ Tamaño de proyectos.
  - ✓ Manejo de contratos.

3.2. Comparativa entre metodologías

En este apartado, se explicarán las similitudes y diferencias entre cada metodología por criterio. Dentro de la descripción de cada criterio se especificarán las particularidades que permitieron identificar dichas relaciones de semejanza y contraste.

3.2.1. Procesos

A. Predicción – Agilidad

RUP es una metodología predictiva. Se elabora la planificación al inicio del proyecto, debido a la estabilidad y poca incertidumbre de las variables y del entorno del proyecto [46].

Scrum, XP, *Crystal* y *Lean Software Development*, metodologías ágiles, dadas las características inestables del proyecto, necesitan reaccionar frente a cambios. Por tanto, su planificación es continua e incremental, al inicio y en el transcurso de cada iteración corta [37].

B. Frecuencia de flujos de trabajo

RUP pretende que los flujos de trabajo se desarrollen permanentemente durante todo el ciclo de vida del software. Sin embargo, cada uno alcanza mayor prioridad en una fase específica y una a continuación de otra, asemejándose entonces a un proceso cascada [46].

En las metodologías ágiles estudiadas, los flujos son desarrollados continuamente, alimentándose permanentemente unos y otros en ciclos iterativos. Los flujos de planificación-seguimiento y desarrollo se ejecutan juntos incluso en niveles de unidades mínimas de tiempo de proyecto (jornada diaria) [37].

C. Definición de prácticas, roles y tareas.

a) Prácticas

En RUP, las prácticas se orientan al desarrollo de software teniendo como punto de partida una definición inicial de requerimientos completa y a partir de ella una generación secuencial de modelos [46].

Con *Scrum* y XP sucede un tema particular. Ambas centran la producción de software de calidad basándose en buenas prácticas. La diferencia entre ambas es que, mientras la primera prioriza las prácticas de gestión, la segunda hace lo mismo con sus similares de desarrollo. Ambas han sido utilizadas en conjunto, pues por esta característica son consideradas como complementarias [29].

Crystal, basa sus prácticas en la creación de software teniendo en cuenta ciertas características del proyecto para realizar una elección de una metodología específica [9].

Lean Software Development, finalmente enfoca la producción de software en la aplicación de principios para la mejora continua y, por consiguiente, la optimización de recursos para obtener productos de calidad [39] [40].

En las metodologías ágiles, además, resaltan, las prácticas de autogestión y auto-organización. Indican que los equipos de trabajo deciden los roles, y tareas a realizar, en concordancia a las demás prácticas de la metodología utilizada.

b) Roles y tareas

RUP [37] define una variedad de roles, cada uno con respectivas tareas específicas para cada flujo de trabajo. Una persona puede ejecutar varios roles, y un rol puede ser compartido por varias personas. La asignación de roles está muy relacionada con especialización de habilidades, experiencia que tenga un miembro del equipo sobre un área de conocimiento.

En las metodologías ágiles, la cantidad de roles y tareas indicados por las metodologías son bajas, a excepción de *Crystal*, donde, propone una cantidad de roles y tareas que aumenta proporcionalmente al tamaño y criticidad del proyecto.

En XP [5] las tareas y los roles son complementadas con las prácticas de programación por parejas y rotación incluso en una misma jornada de trabajo. La rotación juega un papel importante, pues es base para la propiedad colectiva: todos los miembros conocen el desarrollo total del producto. La responsabilidad aceptada, precisa que no se asigna tareas planificadas para la iteración, sino que cada miembro elige y acepta la responsabilidad sobre una.

En *Crystal Clear*, los roles y tareas de desarrollo podrían rotarse y compartirse debido a la poca cantidad de personal. En *Orange* se prefiere la maximización de habilidades específicas, definiendo grupos multifuncionales [9] [10].

LSD [39] [40], si bien no menciona roles específicos, en sus prácticas indica cualidades críticas que deben tener los miembros del equipo de desarrollo. Como tareas indica pasos de manera general, que son muy similares a los presentados en los procesos de *Scrum* y XP, quedando a criterio del grupo la profundización de dichos pasos.

D. Definición de entregables/artefactos

a) Cantidad y complejidad de entregables/artefactos.

En RUP [52] se utiliza una gran variedad de artefactos intermedios antes de llegar al software final. Los principales son modelos complejos y profundamente detallados que reúnen las necesidades de los clientes. Estos continúan evolucionando a medida que se suceden los flujos de trabajo (Requerimientos, Análisis, Diseño), hasta que

tengan la información necesaria para realizar la implementación que construya el sistema ejecutable completo.

Scrum, XP y *Lean Software Development* [37] [5] [39] [40] no detallan de manera inicial los requerimientos del cliente. Se prefiere la comunicación cara a cara entre los miembros del equipo, reduciéndose drásticamente la cantidad de artefactos intermedios. Los artefactos se limitan a sencillas historias de usuario (o *Product backlog*) y otras herramientas de seguimiento. En el mismo desarrollo, el único artefacto a realizar es el ejecutable incremental. XP y LSD además proponen la simplicidad y refactorización, por lo que el código creado se busca sea lo más simple posible.

Crystal [9] [10], propone artefactos que son utilizados a medida que se incremente el tamaño o la criticidad del proyecto. Mientras mayor sea el tamaño o criticidad, más entregables serán usados para modelar el sistema, las comunicaciones y la documentación de seguridad ante errores posteriores.

b) Propiedad de entregables/artefactos

En el proceso unificado de *Rational* [52], la metodología asigna a cada rol artefactos bajo su responsabilidad. La propiedad de cada entregable recae sobre el rol, quien tiene la potestad, capacidad y habilidades para crearlos o modificarlos. De manera similar ocurre en *Crystal* [9] [10], pero, debido a la comunicación permanente, más personas pueden tener conocimiento y apoyar en el desarrollo de dichos artefactos aparte del responsable.

En el caso de *Scrum*, XP y *Lean Software Development* [37] [5] [39] [40], los únicos artefactos asignados son los de planeamiento, requerimientos o historias de usuario y las entregas. El grupo de funcionalidades a desarrollar en la iteración se dividen, con la finalidad de asignar tareas (o aceptar en el caso de XP). De todas maneras, las reuniones diarias, la retroalimentación y la cooperación permiten que todos los miembros conozcan, al menos de manera general, todas las funcionalidades desarrolladas. XP va más allá, dando énfasis a la propiedad colectiva, donde todos trabajan el mismo código generado y pueden modificarlo y mejorarlo.

3.2.2. Personas

A. Equipo de trabajo

a) Número de integrantes

RUP [46] [52] no fija un determinado número de miembros, pero sí refiere que es una metodología diseñada para manejar proyectos con equipos de trabajo grandes. Esto se puede constatar en los múltiples roles que requieren una cierta especialización, aparte del número de artefactos que recaen bajo su responsabilidad. Esto no quiere decir que grupos pequeños no puedan ponerlo en práctica, pero deberán adaptar el proceso a las necesidades del grupo.

Buscando una optimización y una mejora de la productividad, *SCRUM* [50] sugiere que el número de integrantes sea 2 integrantes de gestión y 7 +/- 2 de desarrollo. Por

debajo o por encima de este intervalo se afectaría la comunicación e interacciones entre los miembros.

Para el desarrollo de software aplicando Programación extrema, expertos recomiendan grupos reducidos de trabajo de entre 10 a 20 personas [14] [28]. Fuera del intervalo podría traer problemas de coordinación para llevar a cabo prácticas como programación en pares, en el manejo y tiempo de las reuniones diarias, y dificultad de retroalimentación rápida.

Crystal [9] posee desarrolladas dos tipos de metodologías que definen números de integrantes de equipo. *Clear* requiere de 8 a 10 personas, mientras que *Orange*, de 10 a 40 integrantes.

Lean Software Development en su filosofía [39] fomenta la retroalimentación, la mejora en el aprendizaje y el desarrollo rápido en iteraciones cortas. Para que ello suceda con éxito, los miembros deben poder trabajar frente a frente, de modo que el flujo de información sea de primera mano e inmediato. Por ese motivo para el desarrollo esbelto se prefiere equipos pequeños de trabajo, aunque no específica un número determinado. Para efectos prácticos del estudio, se considerará un número de integrantes de entre 5 a 10 personas.

b) Comunicación entre integrantes

En las metodologías evaluadas se pueden observar dos vertientes marcadas respecto a las comunicaciones entre integrantes:

Por un lado tenemos RUP [46] [34], que trata las comunicaciones entre miembros de manera indirecta. Establece como únicos puntos de retroalimentación, las reuniones en la planificación, los hitos o alguna reunión adicional formal. Las interacciones establecidas son las transferencias de información por medio de artefactos.

Caso contrario ocurre en las otras metodologías estudiadas, consideradas como ágiles. Ellas promueven la comunicación entre miembros cara a cara en diferentes grados:

Scrum, XP y LSD [37] [9] [39] facilitan las reuniones de frecuencia diaria entre miembros del equipo. Además, dan mucho énfasis en mantener al equipo trabajando en un mismo espacio. XP incluso va más allá, planteando la práctica de programación en parejas. El hecho de tener un equipo pequeño posibilita que los miembros se comuniquen personalmente en pleno desarrollo. Las ventajas radican en la retroalimentación, fortalecimiento de la propiedad colectiva y reducción de tiempos de transporte de las personas para comunicarse.

Crystal [9] [10] afirma que para mantener una comunicación cercana y osmótica, los desarrolladores deben estar lo más cerca posible, en un mismo cuarto (en *Crystal Clear*) o en un mismo edificio (para *Crystal Orange*). Introduce más coordinaciones y artefactos (mayor peso de metodología). La frecuencia de las reuniones varía a medida que ya no sea posible mantener a tantos miembros es una sola habitación.

c) Reemplazo de integrantes

En RUP, una metodología predictiva, un cambio de integrantes en el proyecto, o que el proyecto sea retomado por un equipo distinto al inicial, no debería afectar el curso del ciclo de vida de manera dramática. Se posee la documentación y artefactos necesarios para continuar con el rumbo del desarrollo [9].

En desarrollo ágil los cambios hacen que la documentación pierda su valor en cortos intervalos de tiempo. Se corre el riesgo que se pierda conocimiento importante de las personas con propiedad exclusiva sobre artefactos o entregables, y por tanto mayores costos por capacitación [9].

Beck en Programación Extrema [5] recomienda que los cambios de integrantes a lo largo del proyecto sean de manera paulatina. Un nuevo integrante se empapará poco a poco del conocimiento y pericia en las habilidades necesarias para continuar con sus labores, con apoyo de los integrantes antiguos. Después de dos a tres iteraciones este miembro estará a la par con sus colegas. El cambio gradual permitiría que en aproximadamente un año el equipo completo pudiese ser completamente distinto al inicial sin dificultades.

XP es, dentro de las metodologías ágiles estudiadas, la única que clarifica el cambio de integrantes. Se ayuda mucho de la propiedad colectiva para mitigar riesgos por reemplazo de integrantes. Extrapolándolo a *Scrum*, *Crystal* y *Lean Software Development*, se puede lograr a mitigar riesgos, debido a que incentivan la cooperación entre los miembros de proyecto, aunque no se puede asegurar la efectividad que sí logra XP.

B. Clientes y usuarios

a) Participación en el proyecto

En RUP [46] [52] el rol de cliente representa a la empresa, con respecto al proyecto. Con el cliente se acuerda el contrato, la planificación, a la que se consulta el giro del negocio y los requerimientos, y con la que se aprueba el cumplimento de los objetivos en los hitos. Los usuarios participan en el levantamiento de información para definir requerimientos, en la evaluación de hitos, y en la ejecución de pruebas de usuario y distribución.

En cuanto a *Scrum* [37] [50], el propietario del producto representa a los clientes. En la planificación de iteraciones, participa en las reuniones de elaboración del product backlog, asignando prioridades y detallando los requerimientos. En el desarrollo, el equipo de proyecto puede contactar a menudo al propietario para resolver dudas y finalmente, participa en las reuniones de revisión de *sprint*. Los usuarios son solicitados en la ejecución de pruebas de usuario de las iteraciones.

Para XP [5] el cliente es un miembro más del equipo de desarrollo. Escribe las historias de usuario y participa en las reuniones de planificación. Además, se encuentra integrado al día a día del desarrollo sentado junto a los desarrolladores, siendo parte activa del detalle de los requerimientos, de la programación, del diseño y de la ejecución de pruebas de funcionalidad.

Crystal [9] encarga a los representantes del cliente la redacción de requerimientos, participación en la planificación y la retroalimentación al final de las iteraciones y entregas. No indica una permanencia del cliente y usuarios en el ciclo de desarrollo, pero sugiere que, mientras mayor tiempo dedicado al proyecto, mejor retroalimentación y alineación del producto con las necesidades a satisfacer.

Por último en *Lean Software Development* [39] los clientes y usuarios, participan en la definición de requisitos, en la planificación y en el desarrollo como colaboradores permanentes y componentes de los flujos de retroalimentación y aprendizaje. No menciona frecuencia de disponibilidad, pero las características de las prácticas, como la búsqueda continua de conocimiento y de detalle, los hace participantes constantes y frecuentes dentro del proyecto.

3.2.3. Organización y proyectos

A. Criticidad de proyectos

Para analizar esta característica, se tendrá en cuenta la definición y niveles de criticidad expuestos en el apartado de la metodología *Crystal* [9].

RUP es una metodología que en su totalidad ha sido creada para proyectos de criticidad alta (nivel 4), dada la cantidad de flujos, roles, actividades, y artefactos que define. Sin embargo, también se puede ajustar a proyectos de otros niveles utilizando los elementos que el equipo considere aplicables [9].

Scrum, XP y *Lean Software Development* poseen muchas características comunes con *Crystal Clear*, por lo que podrían considerarse metodologías indicadas para proyectos de hasta nivel 2 de criticidad [10].

Crystal [9] [10] se divide en distintas metodologías que abarcan toda la gama de tamaños y criticidades de proyectos. Sin embargo, hasta ahora sólo han sido desarrolladas dos: *Crystal Clear* y *Crystal Orange*, cada una para distintas criticidades.

B. Manejo de contratos

Beck y Poppendieck [5] [39] afirman que, para metodologías predictivas, como en el caso de RUP, donde se definen completamente los objetivos, planificación y requerimientos al inicio del proyecto, los contratos utilizados son los de costo-fijo. En este tipo de contratos se dividen las responsabilidades entre el cliente y el equipo de desarrollo, con respecto al proyecto y al resultado final. Incidirá de manera positiva en la calidad final de producto que la innovación, incertidumbre del entorno e inestabilidad de los requisitos sean bajos.

También precisan que en las metodologías ágiles, donde los índices de innovación, incertidumbre e inestabilidad son altos, se requiere reacción frente al cambio que los contratos fijos no podrían proporcionar. Ante la menor variación, ambos responsables (cliente y equipo de desarrollo) intentarían obtener mayores beneficios por su lado prescindiendo de la calidad del software. Contratos de costo-objetivo, cronograma-

objetivo o de beneficios compartidos propician que clientes y la empresa prestadora de servicios trabajen juntos con la finalidad de llegar a un objetivo en común [39].

3.3. Gráficos de resultado del estudio comparativo

Se ordenó la información recabada en las descripciones de la comparativa entre metodologías, y diseñaron las **tablas 3.1.a, 3.1.b, 3.2.** y **3.3.** que presentan los resultados finales del estudio comparativo.

Tabla 3.1.a Tabla comparativa de grupo de criterios según procesos.

Grupo de Criterios	Criterios	Sub-Criterios	Ítems	RUP	*SCRUM*	XP	*CRYSTAL*	*LEAN SOFTWARE DEVELOPMENT*
Proceso	**Predicción - Agilidad**			**Metodología predictiva.** **Metodología ágil en casos excepcionales.**	**Metodología ágil.**	**Metodología ágil.**	**Metodología ágil.**	**Metodología ágil.**
	Frecuencia de Flujos de Trabajo		Prioridad en todas las fases del proyecto	No	Sí	Sí	Sí	Sí
			Simultaneidad en ejecución de flujos de trabajo	Baja	Alta	Alta	Alta	Alta
	Definición de Prácticas, Roles y Tareas	Prácticas	¿Prácticas definidas?	Sí	Sí	Sí	Sí	Sí
			Orientación de prácticas	Detalle de requerimientos para la creación y, mantenimiento de artefactos, que posibiliten el desarrollo de un software	Gestión para la generación directa de entregables funcionales e incrementales.	Desarrollo directo de entregables funcionales e incrementales.	Generación de entregables funcionales e incrementales, según las características del proyecto.	Optimización de los flujos de trabajo y recursos para obtener software de calidad
			¿Predomina Autogestión y Auto organización de roles y tareas?	No	Sí	Sí	Sí	Sí

Tabla 3.1.b Tabla comparativa de grupo de criterios según procesos.

Grupo de Criterios	Criterios	Sub-Criterios	Ítems	RUP	*SCRUM*	XP	*CRYSTAL*	*LEAN SOFTWARE DEVELOPMENT*
Proceso	Definición de Prácticas, Roles y Tareas	Roles y Tareas	Cantidad de roles	Alta.	Baja.	Media.	Media.	No indica. Cualidades críticas del equipo mencionadas en prácticas.
			Cantidad de tareas propuestas por metodología	Alta.	Baja.	Baja.	Media.	Baja.
			Responsabilidad sobre roles y tareas	Asignada	Asignada	Aceptada	Asignada	Asignada
			Rotación de roles y tareas	No	No indica	Sí	Solo en Clear	No indica
	Definición de entregables/ artefactos	Cantidad y complejidad de entregables/ artefactos		Alta.	Baja.	Baja (Lo más simple posible).	Media a Alta.	Baja.
		Propiedad de entregables/ artefactos		Propiedad recae sobre el rol.	Colaboración permite conocer de manera general todos los entregables.	Propiedad colectiva.	Propiedad recae sobre rol. Colaboración permite conocer de manera general todos los entregables.	Colaboración permite conocer de manera general todos los entregables.

Tabla 3.2. Tabla comparativa de grupo de criterios según personas.

Grupo de Criterios	Criterios	Sub-Criterios	Ítems	RUP	*SCRUM*	XP	*CRYSTAL*	*LEAN SOFTWARE DEVELOPMENT*
Personas	Equipo	**Tamaño de proyecto.**		**Cualquier tipo de tamaño**	**7 a 11 personas**	**10 a 20 personas.**	***C. Clear*: 8 a 10 personas.** ***C. Orange*: 10 a 40 personas.**	**5 a 10 personas.**
		Comunicación entre integrantes.	Tipo de comunicación predominante	Indirecta	Cara a cara	Cara a cara	Cara a cara en Clear Aumento de comunicación indirecta en Orange	Cara a Cara
			Formalidad de reuniones	Alta	Baja	Baja	Media	Baja
			Importancia de la documentación	Alta	Baja	Baja	Baja en Clear, aumenta a media en Orange.	Baja
			Frecuencia de retroalimentación	Baja	Alta	Alta	Alta (Clear) Media (Orange)	Alta
		Cambio de integrantes.	¿Tratado por la metodología?	Sí	No	Sí	No	No
			Práctica utilizada	Documentación	No lo refiere (Retroalimentación es asumible).	Aprendizaje y retroalimentación colectiva.	No lo refiere (Retroalimentación es asumible).	No lo refiere (Retroalimentación es asumible).
	Clientes y usuarios	Participación en el proyecto.	Tipo de participación	Interesado	Miembro parcial	Miembro del equipo	Miembro parcial	Miembro parcial
			Disponibilidad a lo largo del proyecto	Baja	Media	Alta	Media	Media - Alta

Tabla 3.3. Tabla comparativa de grupo de criterios según organización y proyectos.

Grupo de Criterios	Criterios	Sub-Criterios	Ítems	RUP	*SCRUM*	XP	*CRYSTAL*	*LEAN SOFTWARE DEVELOPMENT*
Organización	**Criticidad de proyectos**			**Criticidad 1 a 4**	**Criticidad 1 a 2**	**Criticidad 1 a 2**	**Criticidad 1 a 3**	**Criticidad 1 a 2**
	Manejo de contratos		Responsabilidad delimitada en contrato	Responsabilidad dividida.	Responsabilidad compartida.	Responsabilidad compartida.	Responsabilidad compartida.	Responsabilidad compartida.
			Tipos de contratos a utilizar	Contratos de costo fijo.	Contratos de costo-objetivo, cronograma-objetivo o de beneficios compartidos.	Contratos de costo-objetivo, cronograma-objetivo o de beneficios compartidos.	Contratos de costo-objetivo, cronograma-objetivo o de beneficios compartidos.	Contratos de costo-objetivo, cronograma-objetivo o de beneficios compartidos.

Capítulo 4

DESCRIPCIÓN DE MANUAL DE ELECCIÓN

A continuación presentamos y describimos el contenido del manual de elección de metodologías de desarrollo de software. El eje del manual es un procedimiento que ayuda a seleccionar la metodología de desarrollo de software adecuada para llevar a cabo un proyecto de sistemas informáticos.

4.1. Procedimiento de selección de metodología más adecuada para proyecto de software

4.1.1. Establecimiento y descripción de filtros

Los siguientes filtros han sido elegidos de los criterios de comparación del estudio comparativo. En esta elección se ha tomado en cuenta los estudios realizados por autores de desarrollo de software. El aporte de estos autores será explicado en la descripción de cada filtro.

4.1.1.1. Predicción – Agilidad

Palacio [37] nos explica que en una organización, para tomar una decisión sobre la metodología de desarrollo de software conveniente para llevar a cabo un proyecto de software, se deben evaluar los siguientes criterios:

- El valor innovador que se espera del sistema.
- La incertidumbre del entorno de negocio del cliente.
- La inestabilidad prevista de los requisitos.

Si se concluye la necesidad de utilizar una metodología predictiva, RUP será la alternativa a escoger. Según Rational [46], RUP puede adaptarse a cualquier tamaño y criticidad de proyecto de software.

En cambio, si la necesidad es ágil, ocurrirá una situación particular. *Scrum*, XP, *Crystal* y LSD, por sus principios y características son consideradas metodologías ágiles. Sin embargo, según las consideraciones descritas en la primera parte del estudio comparativo,

las cuatro metodologías juntas, no cubren las necesidades de tamaño y criticidad. Para hallar una solución, se recurre a dos premisas expresadas por especialistas en el tema:

- Allistair Cockburn [9], afirma que ante un aumento de tamaño y criticidad se requiere mayor peso de metodología.
- Martin Fowler [23], indica que RUP puede ser utilizado de manera ágil, siempre que la metodología se ajuste a los principios establecidos en agilidad, es decir el Manifiesto Ágil.

Se deduce entonces, que RUP se puede utilizar para proyectos de software ágiles, si:

- El proyecto posee características de tamaño y/o criticidad no cubiertas por SCRUM, XP, CRYSTAL ni LSD.
- RUP es adaptado a los principios establecidos por el Manifiesto Ágil.

4.1.1.2. Tamaño de proyecto

El tamaño del proyecto es el número de integrantes interrelacionados que conforman el equipo de proyecto de software. Cuando se tiene número de integrantes reducido es posible ubicar a todos en un solo ambiente y mantener contacto directo para la comunicación. Sin embargo, a medida que aumenta el tamaño del proyecto, la comunicación directa se reduce y se empieza a necesitar otros medios de comunicación (documentación, internet, comunicación telefónica), es decir, mayor peso de metodología [9].

Los encargados de la planificación de proyectos de software estiman la cantidad de personal necesario. Para tal fin, utilizan herramientas matemáticas, juicio de expertos o simplemente se ajustan a las limitaciones de recursos en las organizaciones. Aun así, los valores estimados no son siempre exactos. Sin embargo, lo importante para el filtro, más bien es que las estimaciones se encuentren ubicadas dentro del rango de tamaño de alguna metodología.

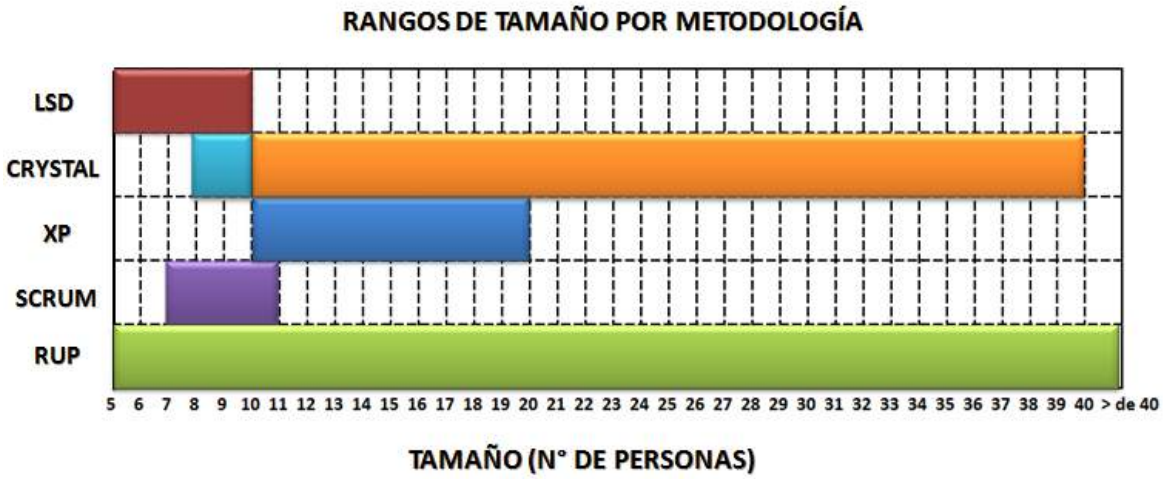


Fig. 4.1. Rangos de tamaño por metodología de desarrollo de software. Elaboración propia.

4.1.1.3. Criticidad

La criticidad es el daño potencial causado por algún error no identificado en el software. Mientras mayor sea la criticidad del proyecto, se necesitará mayor peso de metodología, puesto que se requiere mayor documentación formal y protocolos para identificar, eliminar y reducir riesgos en el desarrollo y en la utilización del software [9].

Cockburn [9] propone 4 niveles de criticidad, los cuales también se utilizarán para este filtro:

- Pérdida de comodidad. Las fallas sólo producen una pequeña molestia momentánea en las personas.
- Pérdida de dinero discrecional. Los errores suelen ser resueltos con coordinaciones simples y soporte del personal propio de una organización.
- Pérdida de dinero esencial. Fallos no son tan simples de resolver y ponen en riesgo la existencia de una organización.
- Pérdidas de vidas humanas. Un error en el sistema puede afectar la integridad de personas involucradas, e incluso matarlas. Ej. Software de control de fugas en central nuclear o en refinerías.

Los planificadores del proyecto tendrán en cuenta estos niveles para definir la criticidad del software a desarrollar.

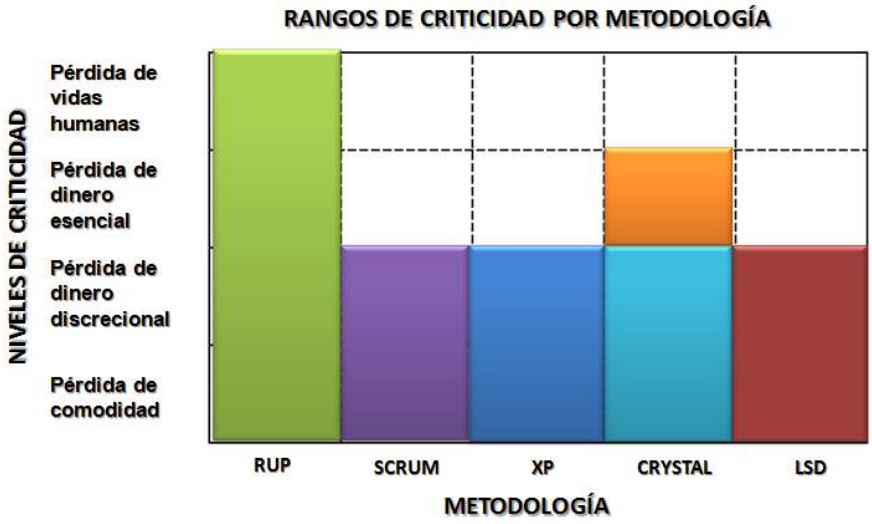


Fig. 4.2. Rangos de criticidad por metodología de desarrollo de software. Elaboración propia.

4.1.2. Consideraciones a tener en cuenta en el procedimiento de selección

- No se pretenderá tener toda la información para hallar la solución desde el inicio del problema, sino que ésta irá apareciendo a medida que se avance la búsqueda de la metodología.
- Aplicaremos el concepto de autogestión mencionado en metodologías ágiles para la realización de evaluaciones en el procedimiento. Dentro de la descripción del

procedimiento se indicará cuales evaluaciones pueden ser gestionadas por el equipo de planificación.

4.1.3. Descripción del procedimiento de selección

A continuación se presenta el procedimiento de selección elaborado para nuestra herramienta.

4.1.3.1. Evaluación de necesidad de Predicción – Agilidad

A. Entradas

- Información del entorno del proyecto.

B. Descripción

- Para determinar la necesidad de predicción – agilidad se recurrirá a los tres criterios mencionados en la descripción del filtro predicción – agilidad, es decir:

  - ✓ El valor innovador que se espera del sistema.
  - ✓ La incertidumbre del entorno de negocio del cliente.
  - ✓ La inestabilidad prevista de los requisitos.

- Estos tres criterios están fuertemente relacionados, por lo que sería inusual que sólo el resultado de uno de ellos sea alto. En todo caso, se recomienda empezar a considerar una necesidad de agilidad si se obtiene resultados altos en al menos dos criterios.
- Las herramientas, actividades y la definición de niveles de resultados necesarios para evaluar la necesidad de predicción – agilidad, serán fijadas y gestionadas por el equipo de planificación.

C. Salidas

- Resultado de determinación de necesidad de predicción – agilidad.

4.1.3.2. Aplicación de primer filtro: Predicción – Agilidad

A. Entradas

- Metodologías estudiadas en el estudio comparativo
- Resultado de determinación de necesidad de predicción – agilidad.

B. Descripción

- Si el resultado del estudio de necesidad de predicción – agilidad, es predicción, la única metodología que aprueba el filtro es RUP.

- Si el resultado del estudio de necesidad de predicción – agilidad, es agilidad, las cinco metodologías aprueban el filtro, siendo RUP un caso especial[6].

C. Salidas

- Metodologías que han aprobado el primer filtro.
- Si el número de metodologías es mayor que 1, se continua con el siguiente filtro. Caso contrario, culmina el procedimiento y la metodología aprobada será la escogida para el proyecto de desarrollo de software.

4.1.3.3. Determinación de tamaño de proyecto

A. Entradas

- Información del entorno del proyecto.

B. Descripción

- Las herramientas y actividades necesarias para evaluar la necesidad de predicción – agilidad, serán fijadas y gestionadas por el equipo de planificación.

C. Salidas

- Resultado de determinación de tamaño.

4.1.3.4. Aplicación de segundo filtro: Tamaño de proyecto

A. Entradas

- Metodologías aprobadas en el primer filtro.
- Resultado de determinación de tamaño.

B. Descripción de Filtro

- Ingresar una metodología aprobada a filtro.
- Si el resultado de determinación de tamaño está incluido dentro del rango de tamaño de la metodología filtrada, la metodología aprueba el filtro, caso contrario dicha opción queda descartada.
- El filtro culmina cuando el total de las metodologías de entrada son filtradas.

[6] Si la necesidad del proyecto es de una metodología ágil, RUP será utilizado como metodología alternativa de entrada en los siguientes pasos que lo requieran sólo en la situación particular explicada en la descripción del filtro Predicción – Agilidad de este capítulo.

C. Salidas

- Metodologías que han aprobado el segundo filtro.
- Si el número de metodologías es mayor que 1, se continua con el siguiente filtro. Caso contrario, culmina el procedimiento y la metodología aprobada será la escogida para el proyecto de desarrollo de software.

4.1.3.5. Determinación de criticidad de proyecto

A. Entradas

- Información del entorno del proyecto.

B. Descripción

- Para determinar la criticidad del proyecto se recurrirá a los niveles de criticidad mencionados en la descripción del filtro criticidad de proyecto, es decir:

  - ✓ Pérdida de comodidad.
  - ✓ Pérdida de dinero discrecional.
  - ✓ Pérdida de dinero esencial.
  - ✓ Pérdidas de vidas humanas.

- Las herramientas y actividades necesarias para evaluar la necesidad de predicción – agilidad, serán fijadas y gestionadas por el equipo de planificación.

C. Salidas

- Resultado de determinación de criticidad.

4.1.3.6. Aplicación de tercer filtro: Criticidad

A. Entradas

- Metodologías aprobadas en el segundo filtro.
- Resultado de determinación de criticidad.

B. Descripción

- Escoger una metodología a filtrar.
- Si el resultado del estudio de criticidad está incluido dentro del rango de criticidad de la metodología filtrada, la metodología aprueba el filtro, caso contrario dicha opción queda descartada.
- El filtro culmina cuando el total de las metodologías de entrada son filtradas.

C. Salidas

- Metodologías que han aprobado el tercer filtro.
- Si el número de metodologías es mayor que 1, se continua con el siguiente filtro. Caso contrario, culmina el procedimiento y la metodología aprobada será la escogida para el proyecto de desarrollo de software.

4.1.3.7. Evaluación y decisión final de selección

A. Entradas

- Metodologías aprobadas en el tercer filtro.
- Estudio comparativo de metodologías de desarrollo de software.
- Información de entorno del proyecto.

B. Descripción

- Contrastar las metodologías de entrada para este paso. Se utilizará el estudio comparativo para tal caso.
- Evaluar qué metodología, según el contraste de metodologías realizado, se acerca más a satisfacer las necesidades del proyecto.
- Aquella metodología que, según las evaluaciones de los planificadores, sea la más cercana a las necesidades del proyecto, será finalmente la escogida.
- Las herramientas y actividades necesarias para evaluar y elegir la metodología a aplicar al proyecto de software serán fijadas y gestionadas por el equipo de planificación. Se recomienda utilizar los resultados del estudio comparativo de metodologías de desarrollo de software, ya sea como una herramienta de evaluación, o dentro de un conjunto.

C. Salidas

- La metodología aprobada, es la elegida para adaptar al proyecto de desarrollo de software.

4.2. Resumen gráfico del procedimiento de selección

Se presenta en las **figuras 4.3., 4.4., 4.5., 4.6.** un diagrama de flujo con el procedimiento de selección de metodologías de desarrollo de software.

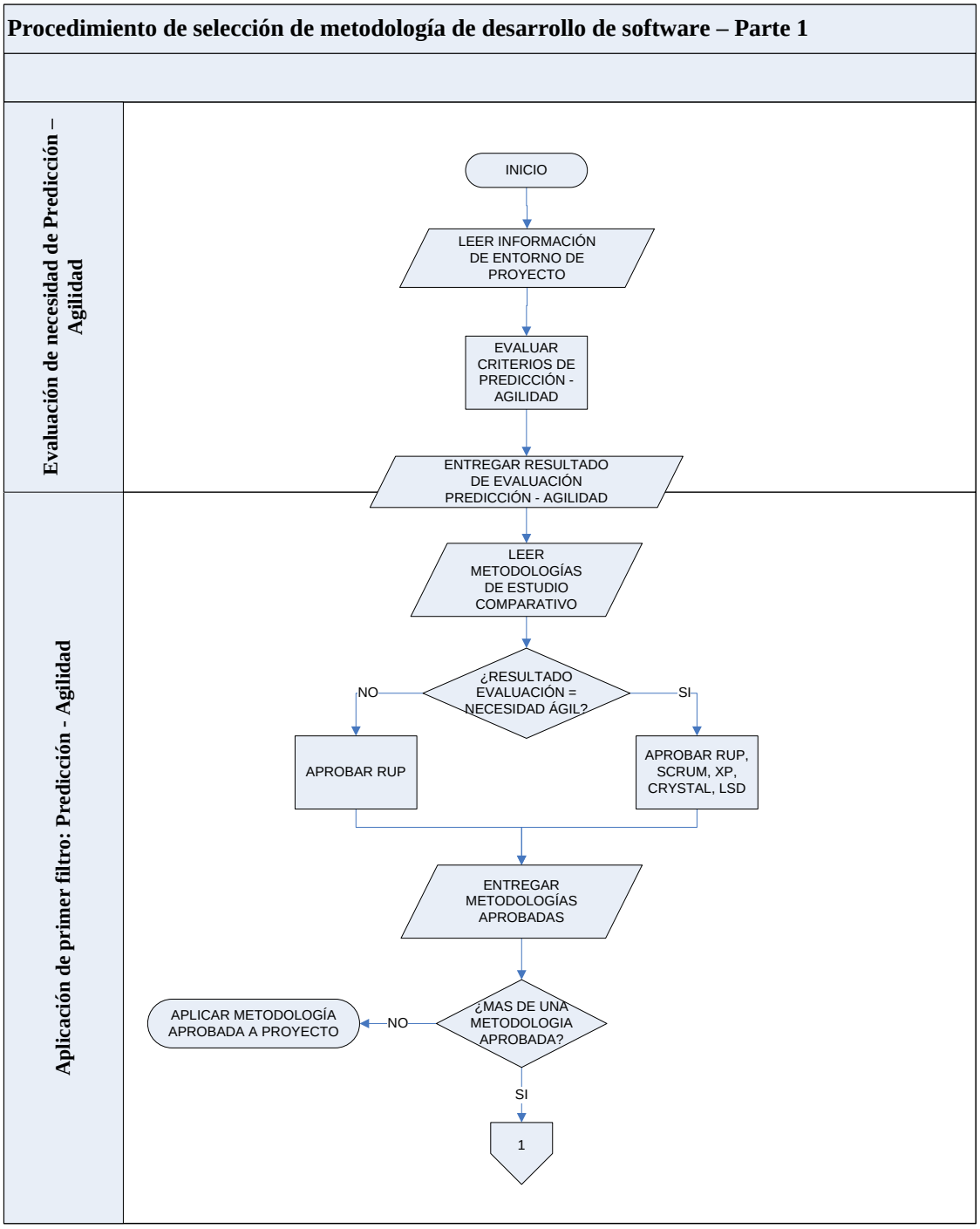


Fig. 4.3. Procedimiento de selección de metodología de desarrollo de software – Parte 1. Elaboración propia.

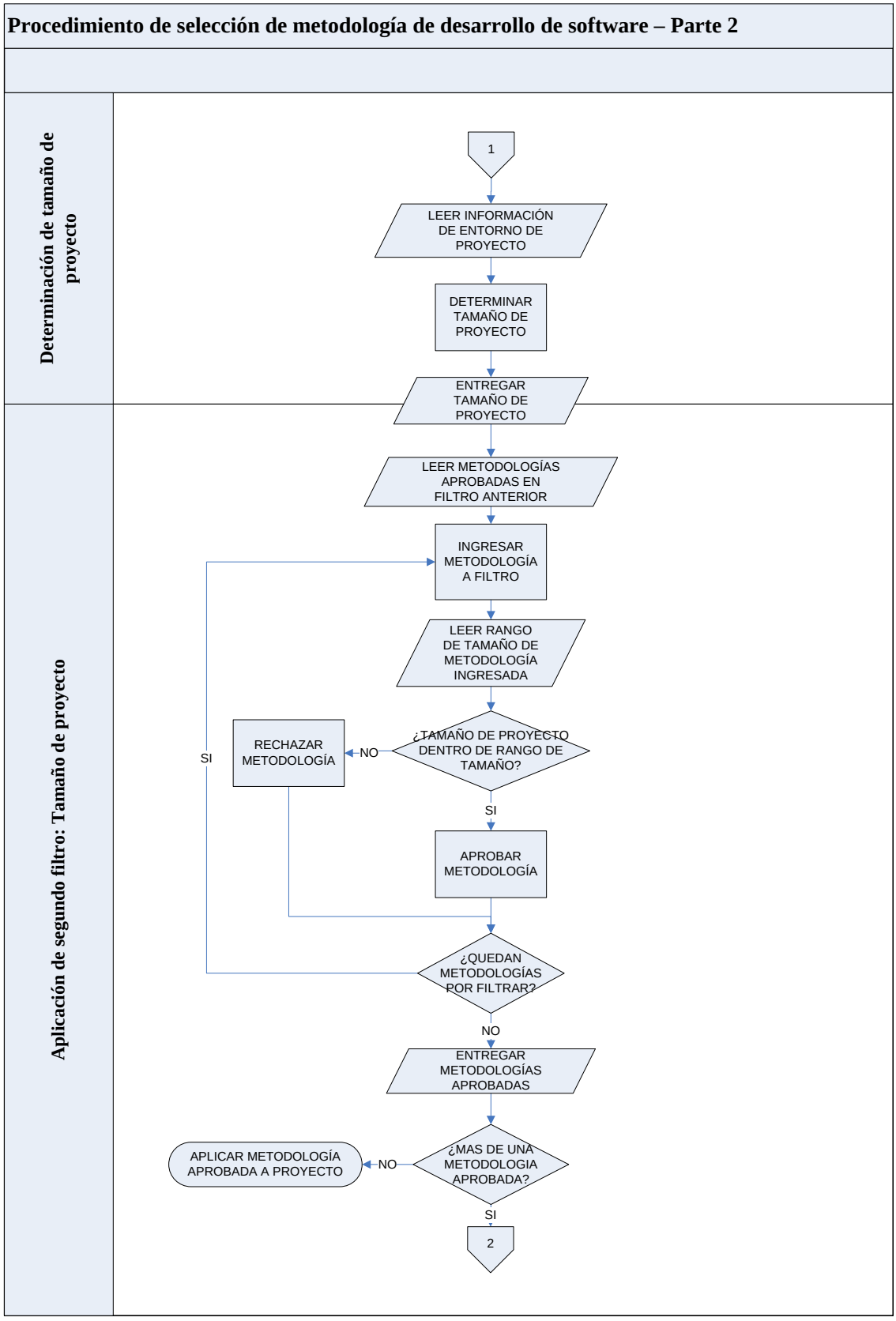


Fig. 4.4. Procedimiento de selección de metodología de desarrollo de software – Parte 2. Elaboración propia.

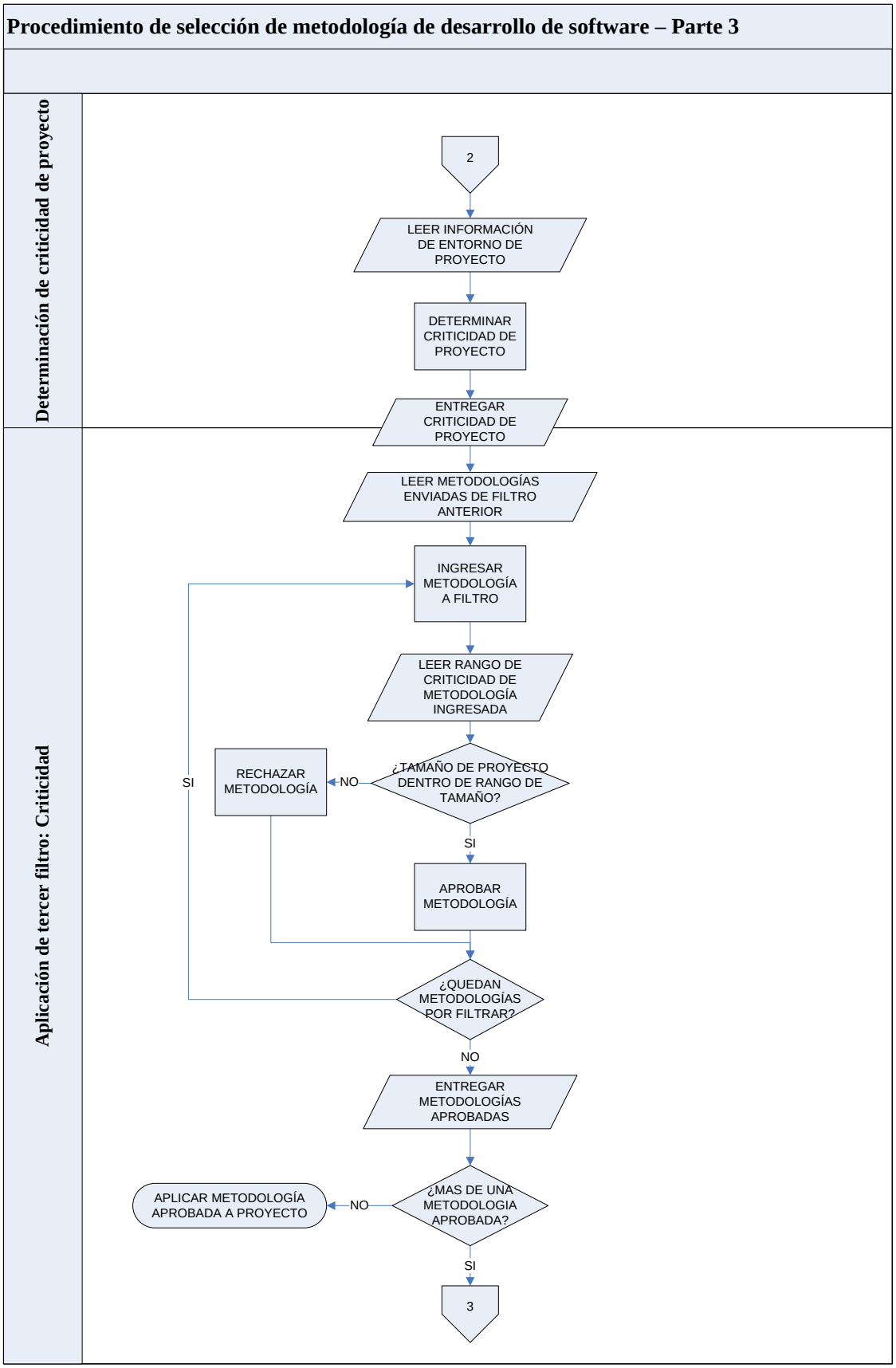


Fig. 4.5. Procedimiento de selección de metodología de desarrollo de software – Parte 3. Elaboración propia.

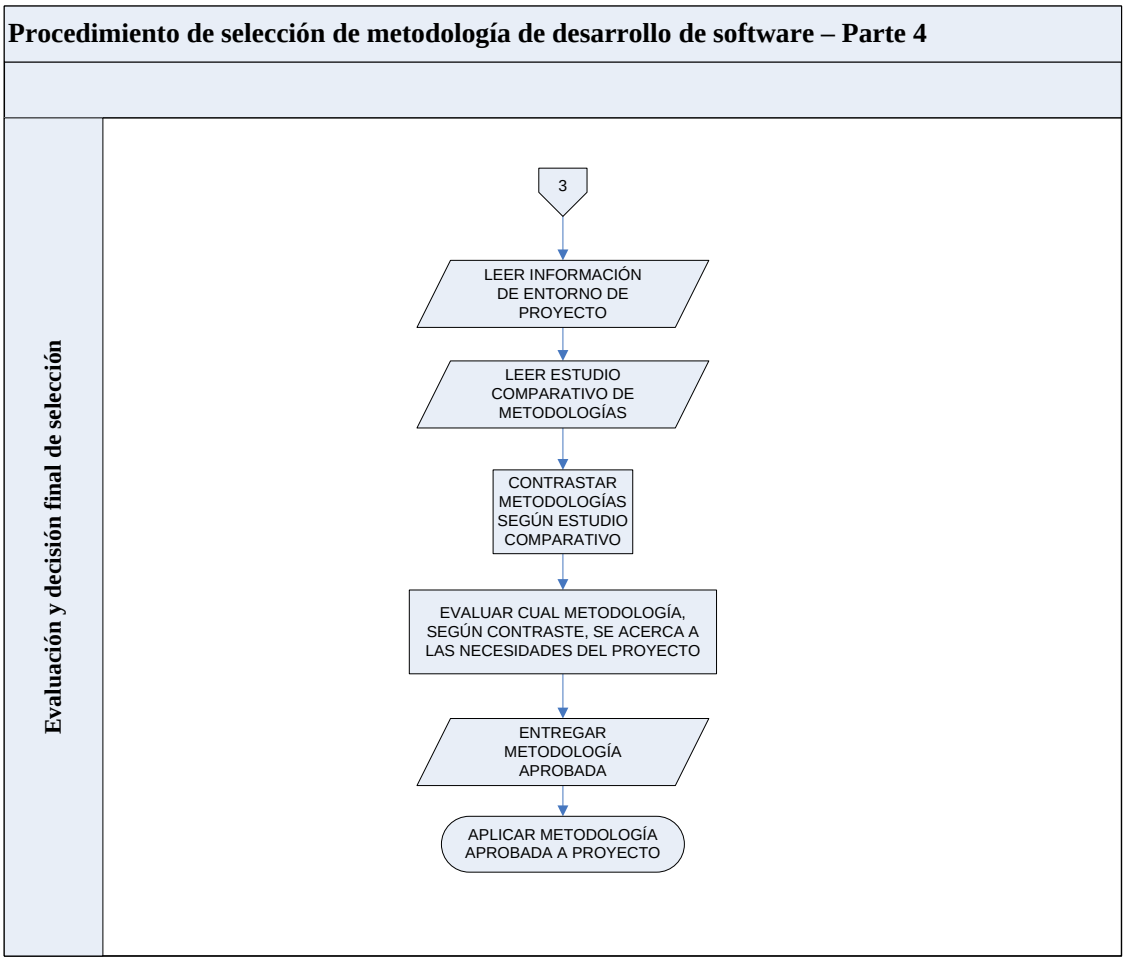


Fig. 4.6. Procedimiento de selección de metodología de desarrollo de software – Parte 4. Elaboración propia.

Capítulo 5

CASOS DE APLICACIÓN

Este capítulo presenta tres casos situacionales de proyectos de desarrollo de software. La problemática central de los casos será la elección de una metodología de desarrollo de software para cada una. Para hallar la solución a esta problemática se utilizará nuestro manual de elección descrito en el capítulo anterior. Los detalles de la búsqueda de esta solución también serán descritos dentro de cada caso.

5.1. Consideraciones de los casos de aplicación.

En la obtención de un caso real se requiere de toda una metodología de levantamiento y análisis de información para luego llegar a las conclusiones alcanzadas para aplicar el manual de la tesis. Esta consistió en las siguientes actividades:

- Recopilación y análisis de información a partir de fuentes presenciales y orales.
- Selección de información relevante para los objetivos de la elaboración del manual.
- Adaptación de información de acuerdo a los objetivos de la elaboración del manual.
- Selección de información útil como entrada para cada uno de los pasos del procedimiento de aplicación del manual.
- Detalle de resultados de la aplicación del manual a cada caso descrito.

Por lo tanto, se debe tener en cuenta a lo largo de la lectura de los casos que:

- Se hará uso del manual de elección para resolver los casos. Esto quiere decir que se aplicará principalmente el procedimiento de selección de metodologías y, según como se requieran, los elementos adicionales contenidos en el manual.

- Los casos son descripciones que recogen algunas situaciones reales adaptadas con el fin de ofrecer una visualización didáctica de la aplicación de nuestro manual. La información del caso se limitará a la necesaria para aplicar el contenido del manual de elección. No se pretende detallar con amplitud ni a nivel técnico dichas características.

- Las evaluaciones y determinaciones se considerarán como realizadas dentro de la descripción de los casos. Para la ejecución de pasos que incluyan evaluaciones o determinaciones, en la descripción de los casos se incluirán sólo los resultados de dichas actividades.

5.2. Descripción de los casos de aplicación.

5.2.1. Caso 1

Una empresa estatal productora y comercializadora de combustibles hidrocarburos desea desarrollar un software. El software le permitirá registrar y generar reportes de análisis de los diversos controles que se les realiza a las cisternas que transportan el producto desde la planta comercializadora central, a su ingreso a la planta comercializadora en la ciudad de Piura.

La jefatura de la planta comercializadora comunica al área de informática la evaluación de su solicitud. Realizaron diversos estudios y visitaron la planta para observar el problema de cerca. Aprobaron el desarrollo del software y además concluyeron lo siguiente:

- Software de control ya había sido desarrollado en diversas áreas de la empresa, así como en plantas y terminales comerciales de otras empresas del mismo rubro.

- No han habido cambios sustanciales en las prácticas ni en los cálculos analíticos de control de cisternas y tanques de almacenamiento desde su implementación. Estas prácticas son conocidas en la industria y están basadas en estándares y reglamentos de organizaciones internacionales y gubernamentales.

- No se prevé cambios futuros en los requerimientos, por ser las características del proceso de control estables y bien definidas.

5.2.1.1. Desarrollo y solución de caso

Paso1. **Evaluación de necesidad de Predicción – Agilidad**

Para esta evaluación se tiene en cuenta la información hasta ahora recopilada por el área informática, que será contrastada con los tres criterios de evaluación de necesidad:

- El valor innovador que se espera del sistema. El caso explica que el desarrollo de este tipo de software ya había era conocido por la empresa y por el sector. En consecuencia, el valor innovador buscado en el sistema es bajo.

- La incertidumbre del entorno de negocio del cliente. La incertidumbre del entorno también es baja, puesto que “no ha habido cambios sustanciales en las prácticas ni en los cálculos analíticos de control de cisternas y tanques de almacenamiento desde su implementación”.

- La inestabilidad prevista de los requisitos. Según el texto "No se prevé cambios futuros en los requerimientos, por ser las características del proceso de control estables y bien definidas". Por lo tanto la inestabilidad es baja.

Nuestros tres indicadores obtuvieron resultados bajos. De esta manera se concluye que el resultado final del primer paso es que el proyecto tiene una necesidad de predicción, es decir, de una metodología predictiva.

Paso2. Aplicación de primer filtro: Predicción – Agilidad

El resultado del paso anterior indica una necesidad de predicción. Por lo tanto la única metodología que aprueba el filtro es RUP. Como es una sola alternativa, culmina el procedimiento y se empieza a adaptar RUP al proyecto de software en mención.

5.2.2. Caso 2

Una empresa agroindustrial productora de mangos, Agroman S.A., ubicada en la ciudad de Lambayeque, tiene la necesidad de implementar un software que le permita gestionar las actividades de sus áreas de soporte (contabilidad, compras, finanzas, recursos humanos, legal, desarrollo tecnológico, etc.). Para ello, contactan a una empresa de desarrollo de software, Minisystems, para que visite la planta de mangos y se pueda llegar a un acuerdo para llevar a cabo la construcción del sistema. En las primeras entrevistas y reconocimiento previo de la situación de la empresa agroindustrial, los representantes de Minisystems identifican las siguientes características:

- Los sistemas de gestión son utilizados por muchas empresas. Su desarrollo no es desconocido, por lo que no implica una innovación a lo que se ha venido construyendo.

- La empresa agroindustrial ha iniciado sus operaciones hace tres años. Actualmente cuenta con un reducido personal dividido en las áreas más básicas: administrativas y operativas. Actualmente, las actividades y procesos de la organización no están definidos completamente, ya que aún se realizan según las necesidades del momento.

- El gerente de Agroman estima que dentro de 3 años su producción aumentará 10 veces la actual, introduciéndose sus productos en nuevos sectores del mercado nacional y extranjero. Esto conllevará a que se especialicen las actividades, se creen más áreas y sub áreas operativas y de soporte, e incremente el número de personal. Además, para ese entonces, ya se tendrá un conocimiento y formalización de las actividades.

- El gerente solicita el desarrollo de un sistema que vaya evolucionando, hasta convertirse, dentro de 3 años en un sistema que incluya los nuevos progresos de esta área, acercándose a un sistema de gestión de procesos.

- Minisystems es una empresa que actualmente cuenta con 100 miembros en la ciudad de Chiclayo. Para el software en cuestión, esta empresa ha estimado que se

necesitará entre 15 a 20 personas. Este rango incluye al encargado de sistemas de Agroman que sería asignado para también conformar el grupo de trabajo.

- Al evaluar los errores críticos que podrían ocurrir en el software, se determinó que los errores de mayor importancia que podría ocurrir serían los siguientes:

  ✓ Falla en la información de planillas, en el módulo para recursos humanos. De haber una demora en los pagos a los trabajadores, ellos acudirían a su sindicato, el cual iniciaría un cese de actividades hasta que se solucione el problema. Una demora en las actividades de producción y en la entrega de pedidos afectaría los lotes previamente asignados a pedidos de clientes. Y un incumplimiento de pedido, dadas las condiciones de competencia y exportación del mercado, afectaría gravemente la imagen de la empresa, perdiéndose clientes importantes.

  ✓ Falla en la información de los módulos de contabilidad y finanzas. No se tendría un seguimiento adecuado de las transacciones financieras de la empresa, ocasionando errores en la toma de decisiones para la inversión. Además, estas fallas serían detectadas en auditorías internas o externas. En el último caso si fuesen realizadas por organismos gubernamentales reguladores provocarían sanciones graves por presentar estados financieros incorrectos.

Ambos errores ocasionarían pérdidas de las cuales Agroman difícilmente podría recuperarse, debido a la enorme inversión realizada para cumplir el plan de crecimiento en 3 años anteriormente mencionado, comprometiendo seriamente la subsistencia de la empresa.

5.2.2.1. Desarrollo y solución de caso

Paso1. **Evaluación de necesidad de Predicción – Agilidad**

Esta evaluación tomará la información identificada por Minisystems, que será contrastada con los tres criterios de evaluación de necesidad:

- El valor innovador que se espera del sistema. El caso explica que "Los sistemas de gestión son utilizados por muchas empresas. Su desarrollo no es desconocido, por lo que no implica una innovación a lo que se ha venido construyendo.". En consecuencia, el valor innovador buscado en el sistema es bajo.

- La incertidumbre del entorno de negocio del cliente. El caso menciona que el entorno de negocio cambiará, puesto que se prevé para tres años un aumento en la producción, produciéndose cambios dentro de la organización, en el entorno de la empresa (clientes) y en el flujo de actividades. Estos cambios aun son desconocidos y se infiere que irán clarificándose en el transcurso del tiempo. Entonces la incertidumbre del entorno es alta.

- La inestabilidad prevista de los requisitos. Finalmente el texto indica que el gerente busca un software que solucione sus problemas de información actuales y que evolucione junto con los cambios que durante tres años experimentará la productora de mangos. Por lo tanto la inestabilidad es alta.

Dos de tres indicadores obtuvieron resultados altos. De esta manera se concluye que el resultado final del primer paso es que el proyecto tiene una necesidad de agilidad, es decir, de una metodología ágil.

Paso2. Aplicación de primer filtro: Predicción – Agilidad

El resultado de la evaluación de necesidad de Predicción – Agilidad, concluye que el proyecto tiene una necesidad ágil. Por consiguiente las metodologías aprobadas son *Scrum*, XP, *Crystal* y LSD. RUP será reservado en caso ocurra la situación particular explicada en el capítulo anterior.

Paso 3. Determinación de tamaño de proyecto

De acuerdo al caso, el tamaño de proyecto estimado se encuentra entre 20 a 25 personas. Por lo tanto, la salida de este paso es este intervalo de miembros.

Paso 4. Aplicación de segundo filtro: Tamaño de proyecto

En este filtro sólo serán aprobadas aquellas metodologías provenientes del anterior filtro cuyo rango de tamaño de proyecto contenga al resultado de la determinación de tamaño del proyecto del caso (20 a 25 personas). Entonces, esa condición es cumplida por XP, *Crystal*.

Paso 5. Determinación de criticidad de proyecto

Según lo mencionado en la segunda parte del caso se describen dos fallas potenciales, cuyas consecuencias comprometerían de manera grave la existencia de la empresa. El nivel de criticidad en el que estaría incluido este tipo de daños es el nivel 3 "Pérdida de dinero esencial", que sería el resultado de este paso.

Paso 6. Aplicación de tercer filtro: Criticidad de proyecto

En este filtro sólo serán aprobadas aquellas metodologías provenientes del anterior filtro cuyo rango de nivel de criticidad contenga al resultado de la determinación de criticidad del proyecto del caso ("Pérdida de dinero esencial"). Esa condición es cumplida por *Crystal* y RUP. RUP es descartado, pues según lo explicado en el capítulo 4, sólo será utilizado cuando no haya metodología ágil que cubra la necesidad de tamaño o criticidad del proyecto.

Finalmente la metodología aprobada será *Crystal* en su versión *Orange*. Como es una sola alternativa, culmina el procedimiento y se empieza a adaptar *Crystal Orange* al proyecto de software en mención.

5.2.3. Caso 3

Como parte del programa de mejora en Unión Acerera Inti, un grupo de empleados de las áreas de sistemas de información y de gestión de calidad de la sede central en Lima, han decidido formar un grupo de círculos de calidad y llevar a cabo un proyecto. El proyecto consiste en desarrollar una red social privada vía intranet para los miembros del área de gestión de calidad de todas las sedes de la empresa (5 en total). Su propósito es incentivar la comunicación e interacción entre ellos y una mejora del clima laboral del área.

Para desarrollar este software, el equipo de desarrollo tendrá en cuenta los siguientes aspectos:

- Se buscará innovar continuamente. Además de las ya conocidas características comunes en las redes sociales, se crearán y renovarán constantemente la apariencia, funciones y aplicaciones con temática propia de la empresa que atraigan a los usuarios. Se explorará nuevas formas de comunicación dentro del software.

- Con respecto al entorno del negocio, esta área es relativamente joven, teniendo sólo dos años de creada. Paulatinamente se están implementando programas que permitan implantar en los demás miembros de la organización los conceptos y prácticas concernientes al *Total Quality Managment* (TQM). Estos programas ofrecerán cambios dentro del área, los cuales aún no son conocidos en su totalidad, sino que se irán detectando en el tiempo. Según lo concluido por el equipo de desarrollo, estos cambios serán fuente importante de nuevas ideas en la innovación del software.

- Los nuevos programas conllevarán a que nuevas necesidades de los clientes y usuarios sean descubiertas. Se puede concluir entonces que la innovación, entorno y necesidades de los usuarios se encuentran fuertemente relacionados.

- Unión Acerera Inti inició la implementación del programa de equipos de progreso y círculos de calidad desde hace 10 años, cuando la unidad de Sistemas Integrado tenía a cargo su ejecución. El asesor es un analista de TQM que es el responsable de difundir los conceptos y principios del programa, de formalizar la constitución de los equipos, y de realizar seguimiento a los avances de los proyectos. El encargado no se encuentra de manera permanente junto a los círculos y equipos. Sin embargo, estos últimos eligen a un miembro que hace las veces del asesor de forma permanente y, por lo general, se escoge al trabajador que tenga mayor participación y experiencia en este programa. En pocas palabras, se ha dado énfasis a la gestión dentro del programa, dejando el desarrollo de los proyectos a los mismos equipos.

- Los resultados del programa han sido positivos y le han permitido a la empresa estar entre los tres primeros puestos del concurso de programas de calidad organizado por la Sociedad Nacional de Industrias, justamente de los últimos 10 años. Para esta ocasión, el equipo de desarrollo de este software buscará una metodología que oriente sus prácticas a la gestión de proyectos y se logre un mejor alineamiento con los objetivos del programa.

- Según las reglas del programa de mejora, un grupo de círculos de calidad deberá tener como mínimo 6 y máximo 8 integrantes. El grupo de nuestro proyecto cuenta con 5 miembros del área de informática y con 2 de gestión de calidad. Estos cuentan con experiencia y conocimientos similares en desarrollo de software y pueden ejecutar de manera indistinta cualquier actividad implicada en la construcción de la red social.

- El equipo ha definido que el problema más crítico del software sería una falla en sus funcionalidades y aplicaciones. Esto causaría incomodidad en los usuarios y los instaría temporalmente a usar otros medios web para comunicarse, de ser necesario. Luego de solucionado el error se retornaría al uso normal de la red social.

5.2.3.1. Desarrollo y solución de caso

Paso1. **Evaluación de necesidad de Predicción – Agilidad**

Esta evaluación tomará la información presentada, que será contrastada con los tres criterios de evaluación de necesidad:

- El valor innovador que se espera del sistema. Del caso se concluye que el valor de innovación es alto, puesto que habrá creación y renovación continua en las funcionalidades de la red social.

- La incertidumbre del entorno de negocio del cliente. La aparición de nuevos programas de mejora dentro de la empresa, según el equipo serán fuente de cambios en el área. Entonces, existe una incertidumbre alta en el entorno.

- La inestabilidad prevista de los requisitos. Finalmente el texto indica que el gerente busca un software que solucione sus problemas de información actuales y que evolucione junto con los cambios que durante tres años experimentará la productora de mangos. Por lo tanto la inestabilidad es alta.

Los tres indicadores obtuvieron resultados altos. De esta manera se concluye que el resultado final del primer paso es que el proyecto tiene una necesidad de agilidad, es decir de una metodología ágil.

Paso2. Aplicación de primer filtro: Predicción – Agilidad

El resultado de la evaluación de necesidad de Predicción – Agilidad, concluye que el proyecto tiene una necesidad ágil. Por consiguiente las metodologías aprobadas son *Scrum*, XP, *Crystal* y LSD. RUP será reservado en caso ocurra la situación particular explicada en el capítulo anterior.

Paso 3. Determinación de tamaño de proyecto

De acuerdo al caso, el tamaño de proyecto definido es de 7 personas. Por lo tanto, la salida de este paso es este número de miembros.

Paso 4. Aplicación de segundo filtro: Tamaño de proyecto

En este filtro sólo serán aprobadas aquellas metodologías provenientes del anterior filtro cuyo rango de tamaño de proyecto contenga al resultado de la determinación de tamaño del proyecto del caso (7 personas). Entonces, esa condición es cumplida por LSD, *Scrum* y RUP.

Paso 5. Determinación de criticidad de proyecto

Según lo mencionado en la segunda parte del caso se observa una posible falla, cuyas consecuencias sólo causaría incomodidad en sus usuarios. El nivel de criticidad en el que estaría incluido este tipo de daños es el nivel "Pérdida de comodidad", que sería el resultado de este paso.

Paso 6. Aplicación de tercer filtro: Criticidad de proyecto

En este filtro sólo serán aprobadas aquellas metodologías provenientes del anterior filtro cuyo rango de nivel de criticidad contenga al resultado de la determinación de criticidad del proyecto del caso ("Pérdida de comodidad"). Esa condición es cumplida por LSD, *Scrum* y RUP. RUP es descartado, pues según lo explicado en el capítulo 4, sólo será utilizado cuando no haya metodología ágil que cubra la necesidad de tamaño o criticidad del proyecto. Entonces LSD y *Scrum* son los elegidos para pasar a la siguiente fase.

Paso7. Evaluación y decisión final de selección

Utilizaremos como herramienta de evaluación el estudio comparativo de desarrollo de software en este paso. Para tomar la decisión final se evaluará el sub criterio Prácticas, del criterio Definición de Prácticas, Roles y Tareas, perteneciente al grupo de criterios de Procesos. El ítem Orientación de prácticas menciona que *Scrum* se centra en la "gestión para la generación directa de entregables funcionales e incrementales". De otro lado, para LSD afirma que busca la "optimización de los flujos de trabajo y recursos para obtener software de calidad". Se constata que *Scrum* coincide mejor con la búsqueda del equipo de una metodología que se ajuste al enfoque de gestión en el que se encuentra orientado el programa de equipos de progreso y círculos de calidad al que pertenece el proyecto.

Finalmente, la metodología aprobada será *Scrum*. Culmina el procedimiento y se empieza a adaptar *Scrum* al proyecto de software en mención.

CONCLUSIONES Y RECOMENDACIONES

A continuación, se describe las conclusiones originadas por los resultados del desarrollo de esta tesis. Además se menciona recomendaciones útiles para estudios e investigaciones posteriores ligadas a este proyecto o a su ámbito de estudio.

6.1. Conclusiones

- Ante la inexistencia de una metodología única para la totalidad de proyectos, el manual elaborado para la elección de una metodología de desarrollo de software es una ayuda para ingenieros de software en la planificación de un proyecto informático. Como se ha probado en los casos descritos, el manual guía la búsqueda a través de un procedimiento que entrega como resultado una metodología con características que se acoplen mejor a las de los proyectos.

- El estudio comparativo fue importante para la elaboración de nuestro manual de elección de metodologías de desarrollo de software. Del estudio comparativo se detectaron similitudes y diferencias. Éstas fueron elementos determinantes en el diseño del manual, ya que ayudan a definir criterios de selección de la mejor metodología para un proyecto específico.

- El manual elaborado puede ser utilizado para cualquier tipo de proyecto de software. Los criterios de predicción – agilidad, criticidad y tamaño clasifican a los proyectos de software en niveles. Las metodologías escogidas poseen características que en su conjunto abarcan por completo dichos niveles.

- Se puede realizar un manual de elección con un grupo diferente de metodologías de desarrollo de software. Es posible obtener un estudio comparativo con un grupo diferente de metodologías. Además, los filtros definidos en el manual para el procedimiento de selección concuerdan con criterios considerados por autores de desarrollo de software en la elección o elaboración de metodologías de desarrollo de software. Entonces, estudio y filtros se pueden utilizar y adaptar para desarrollar manuales con un grupo de metodologías de entrada distinto en cantidad y en características.

6.2. Recomendaciones

- Se recomienda realizar pruebas de uso del manual en situaciones reales. Las observaciones servirán de retroalimentación para hallar mejoras que incidan en los resultados de la aplicación del manual.

- En caso se decida realizar un manual con grupo de metodologías de entrada distinto en cantidad y en características, se sugiere utilizar el método aplicado para elaborar el manual de esta tesis (efectuando un estudio comparativo y utilizando los criterios definidos para nuestro manual). Se debe tener en cuenta que las metodologías deben poseer características distintas entre sí. Si se utiliza más de cinco metodologías, de ser necesario se recomienda definir filtros más específicos y adicionales a los originales. Esto permitirá que se conserve la mecánica principal del manual: contener un procedimiento central, basado en un estudio comparativo, cuyos pasos son una sucesión de determinaciones y filtros hasta ingresar a un último paso con un número muy reducido de alternativas para facilitar la evaluación y decisión final.

BIBLIOGRAFÍA

1. **ABRAHAMSON, P., SALO, O., RONKAINNEN, J., WARSTA, J.** (2002). Agile software development methods. Review and analysis. Primera Edición. Editorial VVT Publications.

2. **AGILE METHODOLOGIES** (s.f) Extraído el 01 de Febrero de 2012 desde http://www.umsl.edu/~sauterv/analysis/6840_f09_papers/Nat/Agile.html

3. **AGILEMANIFESTO** (2001) Manifesto for agile software development. Extraído el 07 de Julio de 2011 desde http://www.agilemanifesto.org/

4. **AMBER, S.** (s.f.) Class Responsability Colaborator (CRC) Models. Extraído el 01 de Febrero de 2012 desde http://www.agilemodeling.com/artifacts/crcModel.htm

5. **BECK, K.** (1999) Extreme Programming Explained. Primera edición. Editorial Addison- Wesley.

6. **CALERO, M.** (2003) *Una explicación de la programación extrema (XP)*. Trabajo presentado en el V Encuentro usuarios xBase, Madrid. Extraído el 23 de Enero de 2012 desde http://www.willydev.net/descargas/prev/ExplicaXP.pdf

7. **CANÓS, J. H., LETELIER, P., PENADÉS, M.C**. (2003) *Metodologías ágiles en el desarrollo de software.* Trabajo presentado en el Taller Metodologías ágiles en el desarrollo de software de la VIII Jornada de Ingeniería del Software y Bases de Datos, JISBD, Noviembre, Alicante.

8. **CASOS DE USO.** Extraído el 02 de Enero de 2012 desde http://www.dcc.uchile.cl/~psalinas/uml/casosuso.html

9. **COCKBURN, A.** (2001) Agile Software Development. Primera edición. Editorial Addison- Wesley Professional.

10. **COCKBURN, A.** (2004) Crystal Clear: A Human-Powered Methodology for Small Teams. Primera Edición. Editorial Addison- Wesley Professional.

11. **COCKBURN, A.** (2006) Agile Software Development: The Cooperative Game. Segunda Edición. Editorial Addison- Wesley Professional.

12. **COCKBURN, A.** (2007) *The Crystal Methods, or How to make a methodology fit.* Trabajo presentado en el Agile Development Conference, Washington DC. Extraído el 04 de Febrero de 2012 desde http://agile2007.agilealliance.org/downloads/handouts/Cockburn_818.pdf

13. **COLUSSO, R., GABARDINI, J.** (2011). *Desarrollo ágil de software.* Extraído el 10 de Junio de 2011 desde http://blacknemessis.wordpress.com/2011/03/01/desarrollo-agil-de-software/

14. **CORTIZO, J., EXPOSITO, D., RUIZ, M.** (2003) Extreme Programming. Extraído el 04 de Junio de 2012 desde http://www.esp.uem.es/jccortizo/xp.pdf

15. **DESARROLLO DE SOFTWARE SIN DESPERDICIOS: LEAN SOFTWARE.** (2008). Extraído el 03 de Febrero de 2012 desde http://everac99.wordpress.com/2008/03/27/desarrollo-de-software-sin-desperdicios-lean-software/

16. **DÍAZ, J.** (2009) Desarrollo de Software Lean. Extraído el 03 de Febrero de 2012 desde http://najaraba.blogspot.com/2008/02/desarrollo-software-lean.html

17. **DOLADO J., RODRÍGUEZ D.** (2011) Utilidad de los procesos ágiles en el desarrollo de software. *Novática. Revista de Asociación de Técnicos de Informática – España, 209,*73 – 74.

18. **DYBA, T., DINGSØYR, T.** (2008) Empirical studies of agile software development: A systematic review. Editorial Butterworth-Heinemann.

19. **ELIZALDE, E.** (2007) *Desarrollo de software esbelto. ¿Se puede?* Trabajo presentado en el Congreso Software Guru 2007, Octubre, Ciudad de México .Extraído el 03 de Febrero de 2012 desde http://www.sg.com.mx/sg07/presentaciones/Mejora%20de%20procesos/SG07.P03.Desarrollo%20Esbelto.pdf

20. **FERNÁNDEZ ALARCÓN, V.** (2006) Desarrollo de sistemas de información. Una metodología basada en el modelado. Primera edición. Editorial Edicions UPC.

21. **FERNANDEZ, G.** (2002) Introducción a Extreme Programming. Extraído el 23 de Enero de 2012 desde http://www.dsi.uclm.es/asignaturas/42551/trabajosAnteriores/Trabajo-XP.pdf

22. **FERNÁNDEZ, J.** (2008) Crystal Methodologies y los equipos de desarrollo. Extraído el 04 de Febrero de 2012 desde http://sistemasdecisionales.blogspot.com/search/label/Crystal%20Methodologies

23. **FOWLER, M.** (2005, Diciembre 15) The new methodology. Extraído el 7 de Julio de 2011 desde http://martinfowler.com/articles/newMethodology.html

24. **GACITÚA BUSTOS, R.** (2003). Métodos de desarrollo de software: el desafío pendiente de la estandarización. *Theoria, Universidad del Bío Bío, 12*,23 – 42.

25. **GARZÁS, J.** (2012). Encuesta sobre el estado de lo ágil. Extraído el 08 de noviembre de 2012 desde http://www.javiergarzas.com/2012/02/encuesta-agil-2011.html

26. **GOMES, N.** (2010) Los principios Lean y los 7 desperdicios. Extraído el 23 de Febrero de 2012 desde http://nolimitsquality.blogspot.com/2011/02/los-principios-lean-y-los-siete.html

27. **GORAKAVI, P.** (2010) What You Should Know about Crystal Orange Methodology. Extraído el 04 de Febrero de 2012 desde http://www.asapm.org/asapmag/articles/A6_CrystalOrange.pdf

28. **JOSKOWICZ J.** (2008) Reglas y Prácticas en Extremme Programming. Extraído el 23 de Enero de 2012 desde http://iie.fing.edu.uy/~josej/docs/XP%20-%20Jose%20Joskowicz.pdf

29. **KNIBERG, H.** (2007) Scrum y XP desde las trincheras. Primera edición. Editorial C4 Media.

30. **KRUTCHEN P.** (2001) The Rational Unified Process. An Introduction. Editorial Addison Wesley.

31. **LEAN – FILOSOFÍA ESBELTA.** (2011) Extraído el 03 de Febrero de 2012 desde http://talento2011.weebly.com/uploads/8/3/9/1/8391024/lean_tecmkt.pdf

32. **LETELIER, P., PENADÉS, M.** () Métodologías ágiles para el desarrollo de software: eXtreme Programming (XP). Extraído el 01 de Febrero de 2012 desde http://www.willydev.net/descargas/masyxp.pdf

33. **MARKER, G.** (s.f.) *La importancia de los sistemas de información en la empresa*. Extraído el 21 de Junio de 2011 desde http://www.informatica-hoy.com.ar/informatica-tecnologia-empresas/La-importancia-de-los-sistemas-de-informacion-en-la-empresa.php

34. **MARTÍNEZ, A., MARTÍNEZ R.** (s.f.) Guía a Rational Unified Process. Sexta edición. Extraído el 02 de Enero de 2012 desde http://www.info-ab.uclm.es/asignaturas/42551/trabajosAnteriores/Trabajo-Guia%20RUP.pdf

35. **METODOLOGÍA SCRUM** (s.f.) Extraído el 18 de Enero de 2012 desde http://www.clubdesarrolladores.com/articulos/mostrar/63-metodologia-scrum

36. **PALACIO, J.** (2006) El modelo Scrum. Extraído el 13 de Enero de 2012 desde http://www.navegapolis.net/files/s/NST-010_01.pdf

37. **PALACIO, J.** (2007) Flexibilidad con Scrum. Segunda Edición. Editorial Safe Creative.

38. **POPPENDIECK, M.** (2003) Lean Software Development. Extraído el 03 de Febrero de 2012 desde http://documents.scribd.com/docs/1vxr3x2enyvvndb2m4cs.pdf

39. **POPPENDIECK, M., POPPENDIECK T.** (2003) Lean Software Development: An Agile Toolkit. Primera edición. Editorial Addison- Wesley Professional.

40. **POPPENDIECK, M., POPPENDIECK T.** (2006) Implementing Lean Software Development From Concept to Cash. Primera edición. Editorial Addison-Wesley Professional.

41. **PRESSMAN, R. S.** (2005) Ingeniería de software: Un enfoque práctico. Sexta edición. Editorial McGraw Hill.

42. **PROGRAMACIÓN EXTREMA** (s.f.) Extraído el 23 de Enero de 2012 desde http://eisc.univalle.edu.co/materias/WWW/material/lecturas/xp.pdf

43. **PRUEBAS DE RENDIMIENTO** (s.f.) Extraído el 05 de Enero de 2012 desde http://www.corporacionsybven.com/portal/index.php?option=com_content&view=article&id=246

44. **QUALITRAIN** (s.f.) Metodologías Ágiles de Desarrollo de Software (Primera Parte). Extraído el 8 de Julio de 2011 desde http://www.qualitrain.com.mx/Metodologias-Agiles-de-Desarrollo-de-Software-Primera-Parte.html

45. **QUISPE-OTAZU, R.** (*2007, Mayo 13). ¿Qué es la Ingeniería de Software?* Extraído el 21 de Junio de 2011 desde http://www.rodolfoquispe.org/blog/que-es-la-ingenieria-de-software.php

46. **RATIONAL** (2001) Rational Unified Process. Best Practices for Software Development Teams. Extraído el 02 de Enero de 2012 desde http://www.ibm.com/developerworks/rational/library/content/03July/1000/1251/1251_bestpractices_TP026B.pdf

47. **RATIONAL UNIFIED PROCESS (RUP)** (s.f.) Extraído el 02 de Enero de 2012 desde http://www.utim.edu.mx/~mgarcia/DOCUMENTO/ADSI2/RUP.pdf

48. **REYNOSO, C.** (2004) Introducción a la Arquitectura de Software. Extraído el 02 de Enero de 2012 desde http://www.librostonic.com/pdf/Introduccion-a-la-Arquitectura-de-Software

49. **ROBLES, G., FERRER J.** (2002) *Programación extreme y Software Libre.* Trabajo presentado en el V Congreso Hispalinux, Octubre, Madrid. Extraído el 23 de Enero de 2012 desde http://es.tldp.org/Presentaciones/200211hispalinux/ferrer/robles-ferrer-ponencia-hispalinux-2002.pdf

50. **SCHWABER, K., SUTHERLAND J.** (2010) Scrum Guide. Extraído el 13 de Enero de 2012 desde

http://www.scrum.org/storage/scrumguides/Scrum%20Guide%20-%20ES.pdf#view=fit

51. **SORBERAMURINA, V**. (2007, Octubre 30). *Sistemas De Información: Concepto Y Aplicaciones.* Extraído el 21 de Junio de 2011 desde http://www.editum.org/Sistemas-De-Informacion-Concepto-Y-Aplicaciones-p-128.html

52. **TOROSSI, G.** (s.f.) El Proceso Unificado de Desarrollo de Software. Extraído el 02 de Enero de 2012 desde http://www.ecomchaco.com.ar/UTN/disenodesistemas/apuntes/oo/ApunteRUP.pdf

53. **TORRES COVARRUBIAS, V. J.** (2005) Importancia de los sistemas de información en la administración y la economía de las organizaciones. *Encuentros, Revista Semestral de la Unidad Académica de Economía, Universidad Autónoma de Nayarit, 2.*

54. **TORRICO, R.** (2010) Metodología Crystal. Extraído el 04 de Febrero de 2012 desde http://www.cs.umss.edu.bo/rep_materia_doc.jsp?doc_mat=130

55. **TORRICO, R.** (2010) Metodología de desarrollo ágil Crystal. Extraído el 04 de Febrero de 2012 desde http://www.cs.umss.edu.bo/rep_materia_doc.jsp?doc_mat=130

56. **TRIPOD.COM.** (s.f.) *Importancia y Beneficios de los Sistemas De Información.* Extraído el 21 de Junio de 2011 desde http://wilbercalles.tripod.com/impyben.html

57. **VALIDO, E.** (2006) Software reliablilty methods. Extraído el 04 de Enero de 2012 desde http://is.ls.fi.upm.es/doctorado/Trabajos20052006/Valido.pdf

58. **VEGA BRICEÑO, E. A.** (2005, Junio). *Los sistemas de información y su importancia para las organizaciones y empresas.* Extraído el 21 de Junio de 2011 desde http://www.gestiopolis.com/Canales4/mkt/simparalas.htm