UNIVERSIDAD TECNOLÓGICA EQUINOCCIAL

FACULTAD DE CIENCIAS DE LA INGENIERÍA

INGENIERÍA INFORMÁTICA Y CIENCIAS DE LA COMPUTACIÓN

TEMA: "ANÁLISIS COMPARATIVO DE LAS METODOLOGÍAS EXISTENTES ORIENTADAS AL DESARROLLO ÁGIL DE APLICACIONES".

TESIS PREVIA LA OBTENCIÓN DEL TÍTULO DE INGENIERA EN INFORMÁTICA Y CIENCIAS DE LA COMPUTACIÓN

AUTORA:
VERÓNICA GABRIELA NORIEGA BORJA

DIRECTOR:
ING. VÍCTOR HUGO GÁLVEZ CAZA

QUITO, JUNIO DE 2011

DECLARACIÓN

Del contenido del presente trabajo se responsabiliza el autor.

_____

Verónica Gabriela Noriega Borja.

C.I.: 1720690245

Quito, 14 de enero del 2011

Señor

Ingeniero Jorge Viteri MBA.- MSc

DECANO DE LA FACULTAD DE CIENCIAS DE LA INGENIERÍA

Presente.

De mi consideración:

Por medio del presente hago llegar el informe final de la tesis de la estudiante **Verónica Gabriela Noriega Borja**, con el tema, **"Análisis Comparativo de las Metodologías Existentes Orientadas al Desarrollo Ágil de Aplicaciones"**, el mismo que a continuación detallo:

- La tesis cumple con el objetivo general propuesto en el plan.
- La tesis cumple con los objetivos específicos planteados al inicio del trabajo de investigación.
- Se ha utilizado una metodología para el desarrollo del proyecto de tesis.
- Se ha realizado la respectiva capacitación a los estudiantes de quinto semestre de la escuela de Informática.

Hago propicia la ocasión para desearle éxitos en sus funciones.

Atentamente,

Ing. Víctor H. Gálvez C.

Director de Tesis

UTE
EQUADOR
FACULTAD DE CIENCIAS DE LA INGENIERÍA
SECRETARÍA
2011-01-13

DEDICATORIA

A mi familia, por ser mi apoyo y estar siempre a mi lado en los momentos en que los necesité, por brindarme su cariño, su amor y el oportuno consejo que me ha permitido realizarme como persona y profesional.

AGRADECIMIENTO

A la Universidad Tecnológica Equinoccial por haberme brindado la oportunidad de educarme en sus aulas.

A quienes en su momento fueron mis profesores, porque ellos han sido el vehículo que me ha permitido llegar a mi meta.

Al Ing. Víctor Hugo Gálvez quien ha dedicado su tiempo y experiencia para guiarme en la consecución de mi trabajo de tesis.

A Jaime por brindarme su apoyo y compartir conmigo su conocimiento y experiencia, lo que me ha permitido ver el mundo desde una perspectiva más amplia.

A mis compañeros de estudio y amigos por brindarme su amistad y compartir conmigo momentos inolvidables.

Y especialmente a Dios, ya que sin su ayuda nada de esto hubiera sido posible.

Índice General

DECLARACIÓN III

CARTA DEL DIRECTOR IV

DEDICATORIA V

AGRADECIMIENTO VI

Índice General VII

Índice de Contenidos IX

Índice de Cuadros XV

Índice de Figuras XVI

Índice de Tablas XVII

Índice de Anexos XVIII

RESUMEN XIX

SUMMARY XX

CAPÍTULO I

EL PROBLEMA DE LA INVESTIGACIÓN. 1

CAPÍTULO II

PROCESO DE DESARROLLO DE SOFTWARE 15

CAPITULO III

METODOLOGÍAS DE DESARROLLO ÁGIL 36

CAPITULO IV
ANÁLISIS COMPARATIVO 98

CAPITULO V
GUÍA DE APLICACIÓN 119

CAPITULO VI
CAPACITACIÓN 138

CAPITULO VII
CONCLUSIONES Y RECOMENDACIONES 147

GLOSARIO 153

BIBLIOGRAFÍA 155

LINKS DE PÁGINAS WEB CONSULTADAS 157

VIDEOS DE AYUDA CONSULTADOS 159

ANEXOS 161

Índice de Contenidos

CAPÍTULO I 1

EL PROBLEMA DE LA INVESTIGACIÓN. 1

1.1 INTRODUCCIÓN. 1

1.2 FORMULACIÓN DEL PROBLEMA. 3

1.3 OBJETIVOS 3

1.3.1 Objetivo General 3

1.3.2 Objetivos Específicos 4

1.4 JUSTIFICACIÓN 4

1.5 ALCANCE 5

1.6 FACTIBILIDAD 5

1.6.1 Factibilidad Técnica 5

1.6.2 Factibilidad Económica 8

1.7 MARCO DE REFERENCIA 9

1.7.1 Marco Conceptual 10

1.8 HIPÓTESIS 12

1.8.1 Hipótesis General 12

1.8.2 Hipótesis Específicas. 12

1.8.3 Variable Independiente 12

1.8.4 Variables Dependientes 12

1.8.5 Indicadores 13

1.9 METODOLOGÍA 13

1.9.1 Metodología de la Investigación 13

1.9.2 Metodología informática. 13

1.9.3 Técnicas e Instrumentos para obtener los datos 14

1.9.4 Fuentes de Información. 14

CAPÍTULO II 15

PROCESO DE DESARROLLO DE SOFTWARE. 15

2.1 Historia de los Métodos de Desarrollo de Software. 15

2.1.1 Inicios: 16

2.1.2 El cambio: *Programación Estructurada* 17

2.1.3 Diseño y Análisis Estructurado 19

2.1.4 Una nueva medida de éxito: el factor humano 20

2.1.5 Orientación a Objetos. 21

2.1.6 Llega un estándar 23

2.1.7 Una nueva alternativa 25

2.2 Modelos de Desarrollo de Software 26

2.2.1 El Modelo en Cascada 27

2.2.2 El Modelo D.R.A. 29

2.2.3 Modelo de Construcción de Prototipos 32

2.2.4 El Modelo en Espiral 34

CAPITULO III 36

METODOLOGÍAS DE DESARROLLO ÁGIL 36

3.1 ¿Qué es una metodología? 36

3.2 Necesidades de una metodología 36

3.3 Definición de Desarrollo Ágil de Aplicaciones 39

3.1.1 Manifiesto Ágil 42

3.2 Metodologías de Desarrollo Ágil de Aplicaciones 46

3.2.1 *XP* (*eXtreme Programming*) 46

3.2.2 *AUP* (*Agile Unified Process*) 65

3.2.3 Crystal Clear 71

3.2.4 SCRUM 80

3.3 Métodos que colaboran al desarrollo ágil de aplicaciones. 88

3.3.1 Agile Modeling 88

3.3.2 Programación Pragmática (*Pragmatic Programming*) 92

3.3.3 Planning Poker 94

CAPITULO IV 98

ANÁLISIS COMPARATIVO 98

4.1 Evaluación de las Metodologías. 98

4.1.1 Fortalezas y Debilidades de *XP* 98

4.1.2 Fortalezas y Debilidades de *AUP* 100

4.1.3 Fortalezas y Debilidades de *CRYSTAL* 102

4.1.4 Fortalezas y Debilidades de *SCRUM* 103

4.2 Tablas Comparativas 105

4.2.1 Comparativa del Ciclo de vida. 107

4.2.2 Comparativa del tipo de proceso (ajustable o prescriptivo). 109

4.2.3 Comparativa de tamaño del equipo. 110

4.2.4 Comparativa de existencia de herramientas colaborativas dentro de la descripción de la Metodología. 111

4.2.5 Comparativa de requisitos iniciales. 114

4.2.6 Comparativa del Estado Actual de la Metodología. 116

CAPITULO V 119

GUÍA DE APLICACIÓN 119

5.1 Clasificación de Proyectos de Desarrollo de Software 119

5.1.1 Por el entorno de Aplicación. 119

5.1.2 Por el Tamaño del Proyecto 120

5.1.3 Por la Criticidad 121

5.1.4 Por la Calidad Requerida 123

5.1.5 Por el tiempo Disponible para el Desarrollo del Proyecto. 124

5.1.6 Por la definición de Requerimientos Iniciales 125

5.2 Guía de aplicación 126

5.2.1 Cuándo Usar Extremme Programming 127

5.2.2 Cuándo Usar Agile Unified Process 128

5.2.3 Cuándo Usar Crystal Clear 129

5.2.4 Cuándo Usar SCRUM 130

5.3 Análisis por casos de aplicación. 131

Caso uno 131

Caso dos.- 132

Caso tres.- 132

Caso cuatro.- 133

Caso cinco.- 134

Caso seis.- 134

Caso siete.- 135

Caso ocho.- 136

CAPITULO VI 138

CAPACITACIÓN 138

6.1 Guía de Capacitación. 139

CAPITULO VII 147

CONCLUSIONES Y RECOMENDACIONES 147

7.1 Conclusiones 147

7.2 Recomendaciones 149

GLOSARIO 153

BIBLIOGRAFÍA 155

LINKS DE PÁGINAS WEB CONSULTADAS 157

VIDEOS DE AYUDA CONSULTADOS 159

ANEXOS 161

Índice de Cuadros

Cuadro 1. Cuadro de Factibilidad Técnica. 6

Cuadro 2. Empresas que usan metodologías ágiles. 7

Cuadro 3. Factibilidad Económica. 8

Cuadro 4. Modelo Propuesto de Tarjeta para recolectar historias de usuario. 59

Cuadro 5. Modelo Propuesto para prueba de aceptación. 61

Cuadro 6. Modelo Propuesto de Tarjeta para asignación de tareas de ingeniería. 62

Cuadro 7. Modelo Propuesto de Tarjeta CRC. 63

Cuadro 8. Clasificación de la familia Crystal 72

Cuadro 9. Resumen de las Metodologías 118

Cuadro 10. Cuándo usar Extremme Programming 127

Cuadro 11. Cuándo usar Agile Unified Process 128

Cuadro 12. Cuándo usar Crystal Clear 129

Cuadro 13. Cuándo usar SCRUM 130

Índice de Figuras

Figura 1. Modelo Cascada resumido 27

Figura 2. Modelo D.R.A. 30

Figura 3. Modelo Basado en prototipos 33

Figura 4. Modelo en Espiral 34

Figura 5. Prácticas de *XP* y sus interrelaciones. 51

Figura 6. Prácticas de *XP* y sus interrelaciones 57

Figura 7. Ciclo de vida de AUP (*Agile Unified Process*). 68

Figura 8. Tiempo entre versiones de AUP 70

Figura 9. Modelo de procesos de *SCRUM*. 85

Figura 10. Cartas para jugar Planning Poker. 95

Índice de Tablas

Tabla Comparativa 1. Comparativa del Ciclo de vida. 108

Tabla Comparativa 2. Comparativa del tipo de proceso. 110

Tabla Comparativa 3. Comparativa del tamaño del equipo. 111

Tabla Comparativa 4. Comparativa de la existencia de herramientas colaborativas. 113

Tabla Comparativa 5. Comparativa de requisitos iniciales. 115

Tabla Comparativa 6. Comparativa del Estado de la Metodología. 117

Índice de Anexos

Anexo 1. Video de la Capacitación realizada. 161

Anexo 2. Fotografías de la Capacitación realizada. 161

Fotografía 1. Capacitación. 162

Fotografía 2. Capacitación. 162

Fotografía 3. Capacitación. 163

Fotografía 4. Capacitación. 163

Fotografía 5. Capacitación. 164

Fotografía 6. Capacitación. 164

Anexo 3. Evaluación Aplicada. 165

RESUMEN.

El presente trabajo de investigación realiza un análisis comparativo entre cuatro Metodologías orientadas al Desarrollo Ágil de Aplicaciones de Software (Metodologías Ágiles), estas son: Extremme Programming (XP), Agile Unified Process (AUP), Crystal Clear (CC) y SCRUM, la investigación se la realiza mediante el método deductivo que permite analizar a cada una de las metodologías usando tablas comparativas para llegar a una conclusión general, como resultado de este análisis se obtienen las Fortalezas y Debilidades de cada una de las Metodologías, también se realiza un estudio de casos de aplicación los mismos que representan diferentes escenarios cotidianos, para según los parámetros especificados recomendar la metodología que mejor se adapte a las condiciones presentadas, y de esta forma facilitar el entendimiento.

Para llegar al análisis este trabajo incluye un estudio previo que describe en detalle cada una de las metodologías antes mencionadas, es decir, sus elementos, sus fases, sus roles y/o participantes, sus artefactos y sus prácticas, esto permite conocer a fondo a la metodología para poder compararla.

El propósito general de este documento es convertirse en una guía que permita al desarrollador elegir una metodología de acuerdo a sus necesidades y de esa forma asegurar el éxito de su aplicación durante y después del desarrollo.

SUMMARY.

This research work presents a comparative analysis of four Methodologies oriented to the Agile Application Development Software (AgileMethodologies), these are: Extremme Programming (XP), Agile Unified Process (AUP), Crystal Clear (CC) and SCRUM, the research is done by the deductive method to analyze each of the methodologies using comparative tables to reach a general conclusion, as a result of this analysis we obtain the strengths and weaknesses of each methodology, also performed a study of Application Cases, that represent different daily scenarios, and according to the parameters specified recommend the methodology that best suits the conditions presented, and thus ease understanding.

To get the analysis this job includes a preliminary study that describes in detail each of the above methods: elements, phases, roles and / or participants, their artifacts and practices, it allows knowing the methodology to compare.

The general purpose of this document is to become a guide for the developer to choose a methodology according to their needs and thus ensure the success of its implementation during and after development.

Ing. Víctor Hugo Gálvez Caza.

CAPÍTULO I

CAPÍTULO I

EL PROBLEMA DE LA INVESTIGACIÓN.

1.1 INTRODUCCIÓN.

El software con el pasar de los años se ha convertido en parte indispensable en el cotidiano desenvolvimiento de algunas actividades del ser humano debido a que la tecnología tiende a automatizar los procesos con la finalidad de hacerlos más rápidos, fáciles y de resultados confiables.

Así podemos notar que el software ahora es usado en varios sectores: la educación, la ciencia, la ingeniería, la medicina, los negocios, las telecomunicaciones, el entretenimiento, el transporte, la industria y un sinnúmero de áreas que tardaríamos años en citarlas. Muchas de estas actividades son llevadas a cabo solamente con ayuda del software, es decir sin supervisión humana, situación que exige que el software sea confiable, porque un pequeño error podría significar daños de hardware, pérdidas económicas o peor aún pérdida de vidas humanas.

Por estas razones el software debe ser desarrollado bajo parámetros estrictos que aseguren calidad y sobre todo exactitud de resultados, esto se logra a través de procesos ordenados y monitoreados, estos procesos en su conjunto son llamados métodos o metodologías de desarrollo de software.

Los métodos de desarrollo de software le indican al programador los pasos a seguir para desarrollar un software estable, el desarrollo puede dividirse en etapas o en actividades dependiendo de su naturaleza, ya que no hay que olvidar que un software debe contar con documentación acerca de su estructura, un manual de uso, etc.

Las metodologías que se han venido usando casi desde los inicios de los años 70, se están volviendo cada vez mas ineficaces, y a decir de varios profesionales de la informática, estos métodos en lugar de agilitar las cosas entorpecen el normal flujo de actividades dentro del entorno de desarrollo, y se van quedando obsoletos por que las exigencias de desarrollo ha cambiado diametralmente desde ese entonces hasta acá.

Dada la necesidad de una metodología que se ajuste a las nuevas tendencias, surgen las metodologías ágiles como una extensión a las metodologías tradicionales para mejorar el desarrollo de sistemas debido a que se ajustan según el tipo de proyecto y empresa, aumentando y optimizando las practicas de desarrollo de software.

La filosofía de las metodologías ágiles, es centrarse en dimensiones poco usuales, como por ejemplo el factor humano o el producto de software, las cuales dan mayor valor al individuo, a la colaboración con el cliente y al desarrollo incremental del software con iteraciones muy cortas. Este enfoque está mostrando su efectividad en proyectos con requisitos muy cambiantes y cuando se exige reducir drásticamente los tiempos de desarrollo pero manteniendo una alta calidad.

1.2 FORMULACIÓN DEL PROBLEMA.

La existencia de varios métodos que sirven de guía para el desarrollo ágil de aplicaciones de software hace dudar a los desarrolladores hacia cuál de ellos inclinarse ya que cada uno tiene sus ventajas y desventajas, y muchos coinciden en que la metodología que se debe usar depende de la aplicación que se va a crear.

Existe mucha información sobre este tipo de metodologías debido a que están en apogeo, pero no existen documentos formales que especifiquen para que tipo de aplicaciones de software sirven, o cuál de ellas es más recomendable para ciertos requerimientos, por tal motivo se hace necesario realizar un análisis que permita disgregar al desarrollo de aplicaciones de software acorde al método que se debe usar, de esta manera podremos clasificar correctamente las metodologías.

1.3 OBJETIVOS

1.3.1 Objetivo General

Realizar un análisis comparativo de las metodologías de desarrollo ágil de aplicaciones: *AUP, Extremme Programming, Crystal Clear y SCRUM.*

1.3.2 Objetivos Específicos

- Definir el tipo de aplicaciones para las que son útiles cada una de las metodologías propuestas.
- Describir las fortalezas y debilidades de cada una de las metodologías propuestas.
- Capacitar a los docentes de Informática acerca de estas metodologías y su campo de aplicación.

1.4 JUSTIFICACIÓN

El desarrollo de aplicaciones de software es la estructura central de un sistema, pero muchas veces se elige de una manera equivocada la metodología que se va a usar, lo que conlleva problemas de adaptación de los procesos a las etapas definidas, procesos engorrosos innecesarios (*burocráticos*), gasto excesivo de recursos, disminución de la calidad del software, entre otros.

Es por eso que se vuelve necesaria la existencia de un documento que defina que metodología usar de acuerdo a las características de la aplicación a desarrollar, porque de esta manera se logrará mejorar el tiempo y optimizar recursos, dado que se tendrá una base referencial previa que permitirá elegir adecuadamente la metodología a usar, lo que se traducirá en la disminución de problemas.

1.5 ALCANCE

En ésta investigación se analizarán las metodologías de desarrollo de software más difundidas: *AUP, Extremme Programming, Crystal Clear y SCRUM,* para en base a éste análisis construir un documento guía que permita elegir la metodología apropiada de acuerdo a las características de la aplicación a desarrollar.

Este documento contendrá además las fortalezas y debilidades de cada una de las metodologías, lo que permitirá tener una concepción más clara del campo de aplicación.

Además para complementar este trabajo de investigación se realizará el plan de capacitación en el manejo de las metodologías analizadas en el presente estudio que estará dirigido a los docentes y alumnos de la Carrera de Ingeniería Informática y Ciencias de la Computación.

1.6 FACTIBILIDAD

1.6.1 Factibilidad Técnica

Dado que el presente estudio es meramente investigativo, por lo tanto la factibilidad técnica, en este caso, se refiere a la tecnología disponible para realizar la investigación.

Se ha definido que se necesitaran los siguientes elementos para realizar el presente trabajo:

Cuadro 1. Cuadro de Factibilidad Técnica.

Actividad	Herramienta a usar
Trabajo de Documentación	Un computador con Internet
Trabajo de Investigación	Libros, Páginas de Internet, Revistas, Folletos.
Tabulación de Datos	Excel
Procesamiento de Datos	Word
Recolección de Datos	Papel, Fichas Técnicas, Grabadora de voz.
Capacitación	Power Point, Proyector, Papel.

Elaborado por: Verónica Noriega.

Por otro lado, dentro de la factibilidad técnica de esta investigación cabe mencionar acerca de la factibilidad de aplicar estas metodologías en el mundo real, para de esta manera saber si son útiles o no. A continuación un resumen de las empresas que actualmente están usando metodologías de desarrollo ágil de aplicaciones, y que han obtenido excelentes resultado:

Cuadro 2. Empresas que usan metodologías ágiles.

Sectores	Ejemplos de empresas que utilizan metodologías ágiles
Medios y Telecomunicaciones	BBC, BellSouth, British Telecom, DoubleYou, Motorola, Nokia, Palm, Qualcomm, Schibsted, Sony/Ericsson, Telefónica I+D, TeleAtlas, Verizon
Software, Hardware	Adobe, Autentia, Biko2, Central Desktop, Citrix, Gailén, IBM, Intel, Microsoft, Novell, OpenView Labs, Plain Concepts, Primavera, Proyectalis, Softhouse, Valtech, VersionOne.
Internet	Amazon, Google, mySpace, Yahoo
ERP	SAP
Banca e Inversión	Bank of America, Barclays Global Investors, Key Bank, Merrill Lynch
Sanidad y Salud	Patientkeeper, Philips Medical
Defensa y Aeroespacial	Boeing, General Dynamics, Lockheed Martin
Juegos	Blizzard, High Moon Studios, Crytek, Ubisoft
Otros	3M, Bose, GE, UOC

Fuente: http://www.proyectosagiles.org/historia-de-scrum

En consecuencia, el presente proyecto en el que se pretende realizar una investigación y posterior análisis de las metodologías de desarrollo ágil de aplicaciones, es técnicamente viable, ya que está a fácil alcance todo el material requerido, y además se comprobó que la aplicación de las metodologías objeto de este estudio es posible y tiene buenos resultados.

1.6.2 Factibilidad Económica

El presente proyecto, al ser un estudio investigativo no requiere de herramientas ni materiales específicos, por lo tanto tiene un bajo costo de realización, esto quiere decir que el proyecto es económicamente viable, dado a que los gastos que necesita son mínimos. A continuación se presenta un cuadro que resume los gastos posibles:

Cuadro 3. Factibilidad Económica.

CANT.	DESCRIPCIÓN	PRECIO UNITARIO	PRECIO TOTAL
MATERIALES			
4	Libros de Investigación	45,00	180,00
8	Resmas de papel bond	3,50	28,00
6	Meses de Internet	30,00	180,00
6	Meses de Energía Eléctrica	20,00	120,00
800	Impresiones (200 por copia)	0,02	16,00
RECURSOS HUMANOS			
1	Director de Tesis	400,00	400,00
		SUBTOTAL	924,00
1	15% de imprevistos	138,60	138,60
		TOTAL	1062,60

Elaborado por: Verónica Noriega.

Fuente: Índice de precio al consumidor INEC.

Presupuesto en dólares americanos

Se estima que para la realización de este trabajo de investigación se necesitarán aproximadamente 1062,60 dólares.

1.7 MARCO DE REFERENCIA

Los procesos de desarrollo de software tradicionales están caracterizados por ser lentos, poco adaptables y en su mayoría dirigidos por la documentación que se genera en cada una de las actividades de cada una de las etapas del desarrollo.

Existen muchos métodos (metodologías) de desarrollo ágil; la mayoría se enfoca en minimizar riesgos desarrollando software en cortos lapsos de tiempo. El software desarrollado en una unidad de tiempo es llamado una iteración, la cual debe durar de una a cuatro semanas. Cada iteración del ciclo de vida incluye: planificación, análisis de requerimientos, diseño, codificación, revisión y documentación. Una iteración no debe agregar demasiada funcionalidad para justificar el lanzamiento del producto al mercado, pero la meta es tener un demo (sin errores) al final de cada iteración. Al final de cada iteración el equipo vuelve a evaluar las prioridades del proyecto.

Los métodos ágiles enfatizan las comunicaciones cara a cara en vez de la documentación, porque es más fácil y rápido. La mayoría de los equipos ágiles están localizados en una simple oficina abierta, a veces llamadas "plataformas de lanzamiento". La oficina debe incluir revisores, escritores de documentación y ayuda, diseñadores de iteración y directores de proyecto. Los métodos ágiles también enfatizan que el software funcional es la primera medida del progreso. Combinado con la preferencia por las comunicaciones

cara a cara, generalmente los métodos ágiles son criticados y tratados como *indisciplinados* por la falta de documentación técnica.

Los grupos de trabajo de desarrolladores que se organizan autónomamente, obtienen mejores resultados y mejores logros, debido a que se conocen entre sí y por ende forman un entorno bastante apto para trabajar, además las interacciones con el cliente son bastante buenas. Por otro lado los equipos que son conformados sistemáticamente enfrentan problemas en el inicio, desarrollo y fin de un proyecto, hasta que se acoplen al equipo, esto ocurre porque desconocen el manejo del nuevo grupo de trabajo.

1.7.1 Marco Conceptual

Para un mejor entendimiento del presente documento, es preciso hacer primero una revisión de algunos términos que serán usados con frecuencia, para saber su significado dentro del contexto.

- **Ágil.-** según el diccionario de la Real Academia de la Lengua ágil es: Ligero, pronto, expedito, también hace referencia a una persona o animal que se mueve o utiliza sus miembros con facilidad y soltura. Pero en el ámbito de la Ingeniería de Software esta definición no es útil, en este ámbito una acepción más acertada sería así; ágil: es la forma de desarrollar software aprovechando al máximo los recursos disponibles, con una excelente comunicación entre los miembros de un equipo y tratando de mostrar resultados en el menor tiempo posible.

- ***Feedback.-*** en español retroalimentación, en la ingeniería de software es un proceso por medio del cual se recogen sugerencias, críticas positivas y negativas de parte varios miembros del equipo de desarrollo y de los clientes acerca de la mejoras necesarias para proseguir en el desarrollo del producto de software; ésta práctica es propia de las metodologías ágiles y facilita la socialización de errores y malos funcionamientos.

- **Refactorización *(refactoring).-*** En ingeniería del software, el término refactorización se usa a menudo para describir la modificación del código fuente sin cambiar su comportamiento, lo que se conoce informalmente por limpiar el código. La refactorización se realiza a menudo como parte del proceso de desarrollo del software: los desarrolladores alternan la inserción de nuevas funcionalidades y casos de prueba con la refactorización del código para mejorar su consistencia interna y su claridad. Las pruebas aseguran que la refactorización no cambia el comportamiento del código. (Wikipedia, 2009)

- ***Release.-*** en español esta palabra significa lanzar, más en el mundo de la Ingeniería Informática un release es una versión con leves mejorías de un producto de software. Cuando se habla en específico del desarrollo de software un release es la presentación al cliente de una versión funcional a la que poco a poco se le van añadiendo funciones hasta cumplir con todos los requerimientos. Cuando los cambios en una versión son muy grandes entonces se habla de una nueva versión y ya no de un release.

1.8 HIPÓTESIS

1.8.1 Hipótesis General

A falta de una guía precisa del uso de metodologías de desarrollo ágil de aplicaciones se vuelve necesario realizar un análisis de las diferentes opciones existentes, para de acuerdo a los requerimientos de la aplicación elegir una metodología que permita tener procesos óptimos y eficaces.

1.8.2 Hipótesis Específicas.

- La investigación permitirá establecer cuáles son los procesos más óptimos de cada una de las metodologías, dependiendo de su enfoque.
- Se optimizará el uso de recursos para desarrollo de aplicaciones, si se elige una metodología adecuada.

1.8.3 Variable Independiente

- Comparar las metodologías de desarrollo ágil de aplicaciones: *AUP, Extremme Programming, Crystal Clear y SCRUM.*

1.8.4 Variables Dependientes

- Agilizar los procesos de desarrollo de software.
- Ajustar los procesos a las necesidades.

- Recomendar la metodología adecuada para cada tipo de aplicación.

1.8.5 Indicadores

- Reducción de tiempo de desarrollo.
- Optimización de recursos.
- Mayor calidad del producto de software.

1.9 METODOLOGÍA

1.9.1 Metodología de la Investigación

Para el desarrollo de esta investigación se usará el método deductivo que estudia los fenómenos o problemas desde las partes hacia el todo, es decir analiza los elementos del todo para llegar a un concepto general, este método es adecuado ya que se pretende analizar varias metodologías de desarrollo ágil de aplicaciones para identificar las cualidades de cada una y así crear un documento que sirva de guía al momento de elegir una metodología.

1.9.2 Metodología informática.

Dado que este es un trabajo meramente investigativo y de documentación no requiere una metodología informática que se apegue a los paradigmas de programación existentes.

1.9.3 Técnicas e Instrumentos para obtener los datos

Para realizar este trabajo investigativo, se usarán herramientas como encuestas de campo y observaciones, que permitirán conocer las necesidades de los desarrolladores.

1.9.4 Fuentes de Información.

- Páginas *Web.*
- Libros Especializados.
- Profesionales en Informática.
- Expertos en Metodologías de Desarrollo de Software.
- Desarrolladores.
- Videos.

CAPÍTULO II

CAPÍTULO II

PROCESO DE DESARROLLO DE SOFTWARE.

2.1 Historia de los Métodos de Desarrollo de Software.

El desarrollo de productos de software en un inicio se lo realizaba de una forma empírica, porque solamente se basaban en los requerimientos, lo que ocasionaba en la mayoría de los casos ineficiencia del software, ya que al ser desarrollado sin un orden ni una secuencia, se volvía confuso al punto de ser inentendible incluso por los propios creadores, quienes debido a la gran cantidad de código que manejaban sin un control, se les olvidaba probar la funcionalidad de todos los componentes e incluso olvidaban crear ciertas aplicaciones.

Por estas razones se hizo necesario tener un control de lo que se iba a desarrollar, y se crearon las Metodologías de Desarrollo, las mismas que indicaban con claridad las fases que había que seguir y como ir documentando y probando el producto, lo que facilitó mucho el trabajo de los programadores, pero al ser estas metodologías tan detalladas, implicaban una *pérdida de tiempo*, y a la larga terminaron por ser relegadas, ya que cada desarrollador usaba la metodología que más le gustaba, la aplicaba a su manera, o hacía una mixtura entre varias metodologías.

2.1.1 Inicios:

La informática en sus inicios, es decir, en los años 40, era todavía muy limitada, la programación de computadores se realizaba por secuencias numéricas las mismas que eran ingresadas al computador por medio de *switchs* que se debían prender o apagar, y dependiendo de su estado estos representaban un uno o un cero (*bits*); usando estas secuencias se podía realizar cálculos complejos, pero en sí la forma de programar estos cálculos era compleja.

Tiempo después aparecen en escena los *lenguajes de bajo nivel* los mismos que trabajaban con un conjunto de instrucciones que estaban previamente codificadas (*instrucciones nemotécnicas*) esta fue una etapa crucial en la programación de computadores que permitió desarrollar lenguajes más sencillos de usar.

Para finales de los años 50 se hace popular el uso de *lenguajes de alto nivel*, los mismos que fueron considerados un gran logro, ya que usaban palabras completas para expresar una instrucción, lo que facilitaba el trabajo de los programadores, porque era mucho más *natural* escribir en este tipo de lenguaje por su semejanza al lenguaje usado por los humanos para comunicarse. Los lenguajes de alto nivel más populares en los inicios fueron Fortran y Cobol.

La aparición de una amplia variedad de lenguajes de programación de alto nivel y del popular sistema IBM S/360, planteó la necesidad de que el

software debía ser durable, portable y permanente en el tiempo, además la continua disminución de los costos del hardware implicó que el costo del software podía exceder el costo del hardware sobre el cual se ejecutaba. Estos antecedentes llevaron a plantear un nuevo conjunto de criterios para medir el éxito del software, los cuales se mantienen hasta hoy, estos criterios son:

- Desarrollo inicial con costo relativamente bajo.
- Fácilmente mantenible.
- Portable hacia un nuevo hardware.
- Cumplimiento de requisitos del cliente.
- Satisfacer requisitos de calidad, seguridad, fiabilidad, etc.

A pesar de todos los avances anteriormente citados, nadie se había preocupado aún por establecer reglas que guiaran el trabajo de los programadores a la hora de escribir código. La programación era considerada como un arte, un oficio que normalmente se aprendía por prueba y error, porque no tenía ninguna normativa a la cual apegarse.

2.1.2 El cambio: *Programación Estructurada*

El profesor Edsger Dijkstra en el año 1968 publicó un artículo muy controvertido, llamado *Go to Statement Considered Harmful* (La sentencia Go To se considera perjudicial), en este artículo concluía que el hecho de tener una instrucción que salte entre las líneas del código hacía confuso y

desordenado el programa, y muchas veces esto se traducía en programas incorrectos y a la hora de hacer modificaciones resultaba imposible.

Dijkstra creía que los programadores de computadores al igual que otros ingenieros, deberían aplicar métodos formales para crear programas efectivos que debían ser sometidos a pruebas formales que garantizaran su calidad y cumplimiento de requisitos. Estas ideas formaron las bases de la *programación estructurada*, la misma que es considerada como el primer método de desarrollo de software.

En esos tiempos se aseguraba que siguiendo esta metodología se garantizaba satisfacer los criterios de éxito del software, así empiezan a surgir un sinnúmero de conceptos que pretendían facilitar la creación de software, entre esos conceptos tenemos: *Programación Modular, Refinamiento Sucesivo, Information Hiding*, entre otros.

Así mismo surgieron varias notaciones que permitían modelar la estructura de los programas recogiendo los criterios de la programación estructurada, entre las más conocidas tenemos: *Diagramas de Flujo, Diagramas N –S, Pseudocódigo, Tablas de Decisión, Árboles de Decisión.*

Para el año 1971 el profesor Niklaus Wirth lanzó el lenguaje de programación Pascal el mismo que no permitía el uso de la sentencia *go to* y trabajaba con las estructuras de control que planteaba la programación

estructurada de Dijsktra. Las ideas de Wirth y Dijsktra crean una nueva corriente, que buscaba crear programas bien estructurados y fáciles de leer.

2.1.3 Diseño y Análisis Estructurado

Las ideas de la programación estructurada fueron llevadas al ámbito del diseño y del análisis, entonces nació una nueva disciplina llamada Ingeniería de Software. La meta básica del análisis estructurado era lograr obtener un mayor control sobre el creciente aumento de la complejidad de los sistemas. Surgen entonces varios métodos llamados genéricamente como métodos SA/SD (*Structured Analysis / Structured Design*), que incluían una variedad de notaciones para la especificación formal de software.

Durante la fase de análisis muchas notaciones, entre ellas diagramas de flujo de datos, especificación de procesos, diccionario de datos, diagramas de transición de estados y diagramas de entidad relación, entre otros son usados para describir lógicamente el sistema, los método planteados en esas épocas se denominaban métodos funcionales, debido a que eran inspirados por la arquitectura de los computadores, así se llego a separar los datos y el código, tal como se lo hacía físicamente en el hardware.

Las exigencias de la defensa nacional y las aplicaciones aero-espaciales demandan de productos de software mucho más complejos y de misión crítica, por lo que los programadores responden creando una variedad de

enfoques a través de la adaptación de control estructurado para soportar las especificaciones que los nuevos requerimientos demandaban.

Tiempo después surge la necesidad de *diseñar el software*, dado que se había vuelto habitual modelar la estructura de la solución que se había de proponer para un determinado problema, entonces se añade una fase al proceso, la fase del diseño, para esto se agregan detalles a los modelos de análisis y los diagramas de flujo de datos son convertido en *cartas de estructura* y descripciones de código de lenguajes de programación.

Para entonces el desarrollo de software ya era visto como un proceso industrial, por lo tanto se desarrollaron un gran número de métodos, los mismos que proporcionaban una serie de guías, reglas detalladas y regulaciones para el proceso de desarrollo de software.

2.1.4 Una nueva medida de éxito: el factor humano

Para el año de 1975 se había desatado una verdadera *guerra* de los métodos de Ingeniería de Software, dado que se propusieron una enorme cantidad de métodos con diferente notación y procedimiento, tratando de imponerse unos sobre otros.

Fred Brooks publicó entonces un controversial libro, llamado *The Mytical Man-Month*, en alusión a las unidades de medida de esfuerzo (horas-hombre, hombre-mes, etc.) usadas en el desarrollo de Software, a diferencia de

muchos de los autores de Métodos, Brooks no era un académico, por el contrario aprendía acerca del desarrollo de software cuando él era el administrador del primer proyecto de software a gran escala conocido, el desarrollo del sistema operativo IBM/360 (OS/360).

Brooks proponía un nuevo y revolucionario enfoque, sostenía que el proceso de desarrollo de software debía estar centrado en el ser humano y no ser una rígida disciplina de Ingeniería, estas ideas dieron paso a la primera metodología de desarrollo de software útil (ciclo de vida), porque el método presionaba por administrar procesos de personas y no procesos de ingeniería.

Aunque las ideas de Brooks eran interesantes no tuvieron relevancia en el medio y ningún *gurú* del desarrollo de software las tomó en cuenta en los años 70 y 80, y no fue hasta los años 90 que éstas sirvieron de base para proponer los denominados métodos ágiles.

2.1.5 Orientación a Objetos.

A principios de los años 80 el análisis y diseño estructurado seguían dominando el medio, el modelo de ciclo de vida *Modelo cascada* era el dominante porque planteaba una desarrollo lineal, pero desgraciadamente, el análisis estructurado estaba faltando a su promesa de reducir costos y aumentar confiabilidad debido a que la complejidad de los sistemas iba en ascenso y estaban orientados principalmente a los lenguajes de tercera generación.

Muy pronto varios detractores del modelo en cascada atacaban desarrollando nuevas metodologías que prometían ser mejores, pero siempre enmarcados en el ciclo de vida, basándose en las etapas de este. Así nacen metodologías como:

- Prototipos
- RAD (Rapid Application Development)
- Espiral

A pesar de la disputa surgen dos ideas interesantes acerca del desarrollo de software:

La primera es el uso de herramientas CASE (*Computer Aided Software Engineering*), las cuales otorgaban a los desarrolladores la posibilidad de crear sus diseños (diagramas, cartas de estructura, diccionario de datos) en el computador, lo que hacía más fácil y más corto el desarrollo de software. Esto dio paso a la creación de nuevas notaciones y nuevas formas de modelar soluciones, todas en busca de mayor confiabilidad y mejor desempeño.

La segunda idea interesante y útil es la idea del desarrollo de software Orientado a Objetos (OO). Alan Kay uno de los padres de la OO dijo: “la mejor forma de predecir el futuro es inventarlo”. En 1971 él empezó a desarrollar estas ideas detrás del lenguaje de programación Smalltalk, y el objetivo principal de Kay era hacer que el mundo del software sea mucho más cercano al mundo real.

En el mundo real los objetos se comunican enviando mensajes en diferentes direcciones, cuando un objeto interactúa con otro, este no tiene información acerca del funcionamiento interno del otro objeto, cada objeto conoce los protocolos de comunicación e interacción con los otros objetos. Sistemas muy complejos pueden ser construidos combinando varios objetos y permitiéndoles que interactúen naturalmente.

En los años 80 el Departamento de Defensa de los Estados Unidos decidió que podría reducir millones de dólares y asegurar la confiabilidad del software, si todos los productos de software eran desarrollados en lenguajes de Orientados a Objetos, así que desarrolló una nueva generación de Pascal con aspectos de OO, que se llamó ADA, en el mismo tiempo Bajrne Stroustrup, de Bell Labs creó una nueva generación del Lenguaje C con aspectos de OO llamado C++ pero ninguno de estos lenguajes implementó completamente el poder de *SmallTalk*. Estos avances permitieron la creación de la *Web*, lo que llevó a la explosión en la adopción de los lenguajes y técnicas OO, y aunque Smalltalk nunca tuvo mucha relevancia, sus sucesores Java y Phyton han sido extremadamente exitosos.

2.1.6 Llega un estándar

El nuevo enfoque del desarrollo de software Orientado a Objetos forzó a hacer cambios en la forma de modelar, este cambio iba a ser difícil y doloroso dado que ya se habían establecido algunas reglas para el modelado también, el número de método orientados a objetos se incrementó de 10 a más de 50 en solo cinco años, y los usuarios de estos novedosos métodos no

encontraban un lenguaje de moldeamiento que satisficiera completamente sus necesidades.

En un esfuerzo conjunto Grady Booch (Rational Software Corporation) y James Rumbaugh (General Electric) empezaron a adoptar ideas cada uno de los otros métodos, más tarde iniciaron un proyecto que pretendía unificar los métodos de Booch, este proyecto se lo llamo *Método Unificado.* El proyecto poco a poco se fue expandiendo y abarcando más y más métodos, lo que le permitió crecer y ser confiable, el proyecto a un año de su lanzamiento fue fortalecido con retroalimentación de los ingenieros de software, el proyecto tuvo tal aceptación que varias empresas decidieron colaborar para llegar a una definición más completa de lo que ahora conocemos como *UML*.

Esta colaboración, resulto en *UML 1.0*, un lenguaje de modelamiento que estaba bien definido, explícito y poderoso y era aplicable a una amplia variedad de dominios de problemas, inclusive fue ofrecido para estandarización en enero de 1997, por pedido de Object Management Group, de esta forma se logró llegar a una notación estandarizada para el modelamiento de sistemas de software.

Si bien es cierto que *UML* planteó un estándar para el modelamiento, no así para el desarrollo de software, lo que no detuvo la *guerra* de los métodos, inclusive con una notación unificada, hay nuevos métodos alternativos que proponen su propia notación para modelar y a veces incluyen *UML*. A pesar

de esto *UML* es un gran logro, aunque solo sea una notación y no una metodología.

2.1.7 Una nueva alternativa

Una de las principales críticas realizadas a los métodos propuestos hasta ahora es que son *burocráticos*, es decir que hay tantas cosas que hacer que el desarrollo de software se vuelve lento, por esta razón estos métodos han sido llamados *Heavy Methodologies (Métodos Pesados) o Monumental Methodologies (Métodos Monumentales).*

Como una reacción a estas metodologías, a finales de los 90, surge un nuevo grupo de metodologías sustentadas en las antiguas ideas de Brooks, las que fueron conocidas por un tiempo como *Lightweigt Methodologies (Métodos Livianos)*, pero ahora el término aceptado es *Agiles Methodologies (Métodos Ágiles).* En este sentido la *guerra* de los métodos de desarrollo de software ha venido a completar un círculo vicioso

Mientras el manifiesto de Dijsktra llamaba por *más* disciplina en el desarrollo de software, los principales proponentes de las metodologías ágiles han lanzado un manifiesto que llama por *menos* llamado el *Manifiesto for Agile Software Development* (Manifiesto por el Desarrollo de Software Ágil).

Mientras unos escriben libros acerca del estilo ágil, otros *gurú* ágiles han creado su propia empresa de consultoría y entrenamiento orientando sus

principales esfuerzos hacia tomar ventaja de la *nueva economía*, es decir, el desarrollo sobre Internet, para de esta forma demostrar que su *filosofía* es correcta.

Es destacable, sin duda, que la principal contribución de los métodos ágiles es que ellos están recogiendo ampliamente el enfoque centrado en las personas, propuesto por Brooks, y lo mejor es que están agregando elementos adicionales para el entendimiento de la problemática humana detrás del desarrollo de software en general. Muchas personas apelan a estas metodologías ágiles como reacción a las metodologías burocráticas debido a que estos nuevos métodos plantean un justo equilibrio entre *sin proceso* y *demasiado proceso,* proporcionando solo el proceso indispensable para obtener buenos resultados.

2.2 Modelos de Desarrollo de Software

Los llamados Modelos de Desarrollo de Software Tradicionales, son aquellos que surgieron principalmente de la corriente de la Programación Orientada a Objetos y de la Programación Estructurada.

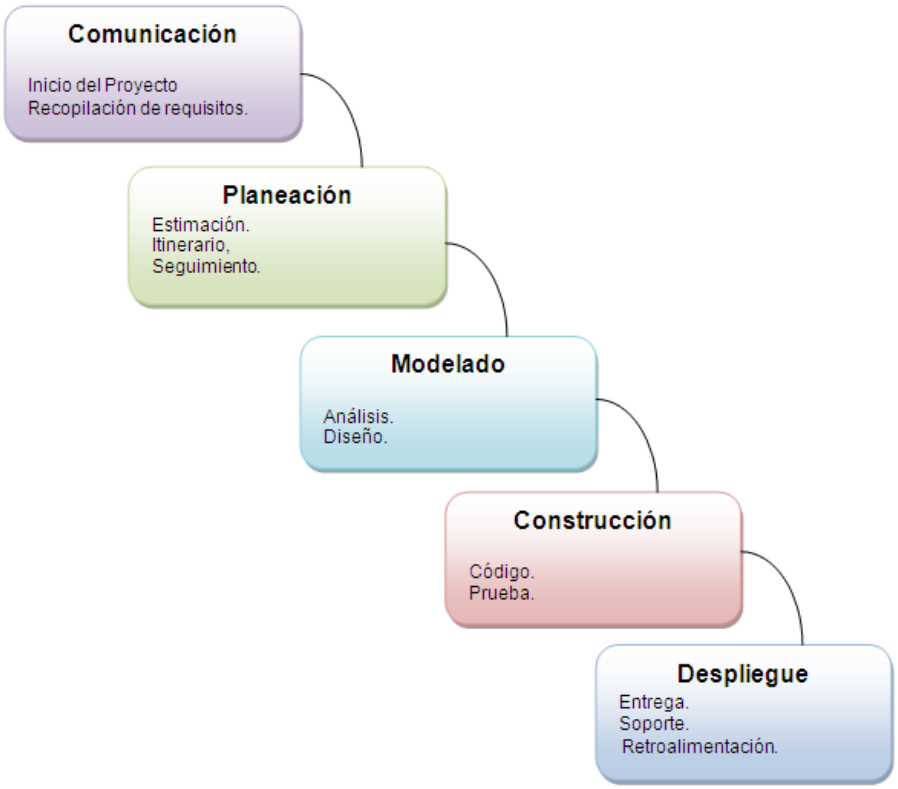
Un modelo de procesos es una representación del mundo real, que captura el estado de actual de las actividades para guiar, reforzar o automatizar partes de la producción de los procesos. Estos modelos son la base en la cual se han sustentado varias metodologías.

Veamos los diferentes modelos de procesos que nos podemos encontrar dentro del mundo del desarrollo del software:

2.2.1 El Modelo en Cascada

El Modelo en cascada es un enfoque de desarrollo secuencial y lineal, en el que el desarrollo es visto como un flujo constante hacia abajo, como una cascada a través de las diferentes fases que son: análisis de requerimientos, diseño, implementación, pruebas (validación), integración y mantenimiento.

Figura 1. Modelo Cascada resumido



Fuente: Ingeniería del Software. Un enfoque práctico. Roger S Pressman (2006).
Elaborado por: Verónica Noriega

Este modelo no tiene una *repetición*, por lo que exige un riguroso cumplimiento de cada una de las etapas para poder iniciar la siguiente, por su carácter lineal.

Los principios básicos del enfoque de cascada son:

- Proyecto se divide en fases secuenciales, con cierta superposición y un lapso aceptable entre fases.
- Se hace hincapié en la planificación, los horarios, fechas, presupuestos y la aplicación de un sistema completo de una vez.
- El estricto control se mantiene durante la vida del proyecto a través del uso de documentación escrita, así como a través de revisiones formales y la aprobación o visto bueno por parte del usuario y la gestión de la tecnología de información que tienen lugar al final de la mayoría de las fases antes de comenzar la siguiente fase.

> "El modelo en cascada es el paradigma más antiguo para la ingeniería de software. Sin embargo en las décadas pasadas, las críticas a este modelo de procesos han ocasionado que aun sus más fervientes practicantes hayan cuestionado su eficacia." (Pressman, 2006)

Los problemas más comunes que se han identificado son:

- Los proyectos *reales* no siguen un flujo secuencial, como el que propone el modelo y aunque este tenga iteraciones son indirectas,

situación que produce una confusión en los cambios mientras el equipo de proyecto trabaja.

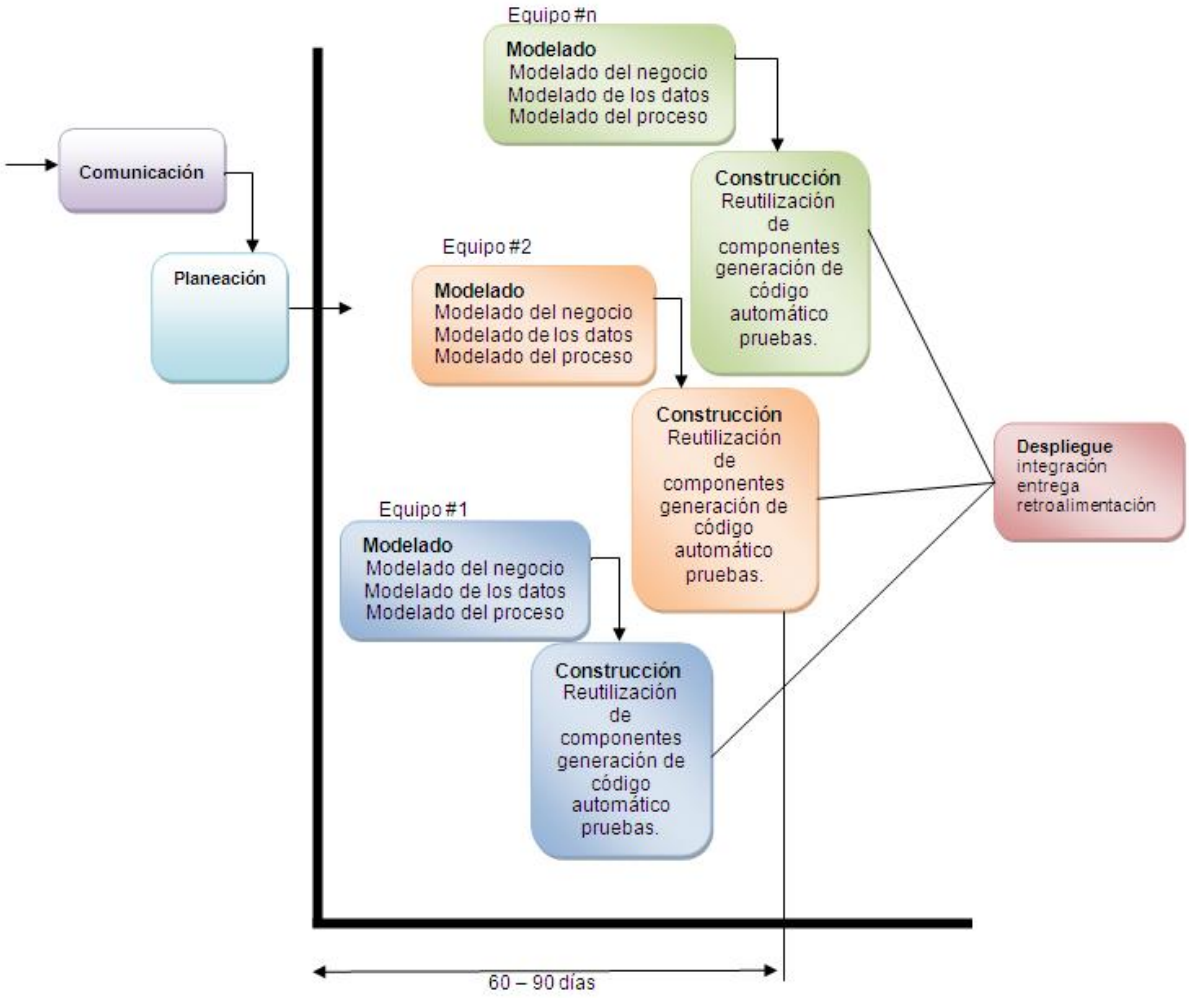
- Los clientes por lo general no logran establecer los requerimientos de forma explícita, y el modelo lo requiere.
- El cliente debe tener paciencia, porque no podrá ver ninguna versión del programa hasta que el proyecto está bastante avanzado, por lo tanto los errores pueden ser graves si no se detectan antes de la revisión.

En la actualidad, son varias las metodologías de desarrollo de software basadas en este modelo, y quizá es este el modelo más conocido y más divulgado, dado que su estructura es simple y eficiente.

2.2.2 El Modelo D.R.A.

Desarrollo Rápido de Aplicaciones (*D.R.A.*) es un modelo de desarrollo de Software que implica el desarrollo iterativo y la construcción de prototipos. Este es un modelo secuencial, al igual que el modelo cascada, pero enfatizado en la rapidez, para lograr esto, la construcción se basa en el uso de componentes previamente creados, lo que disminuye tiempo de ejecución, que oscila entre los 60 y 90 días.

Figura 2. Modelo D.R.A.



Fuente: Ingeniería del Software. Un enfoque práctico. Roger S Pressman (2006).
Elaborado por: Verónica Noriega

Este modelo es una de las bases de las metodologías ágiles y fue sin dudarlo fue el precursor del ahorro de tiempo mediante la administración del recurso humano, y al igual que otros modelos, el *D.R.A.* tiene etapas, las cuales describiremos a continuación:

- **Comunicación** ayuda a entender el problema del negocio y las características de la información que el software debe incluir.

- **Planeación** permite planificar las actividades, es esencial en este modelo porque existen varios equipos de desarrollo trabajando en paralelo.

- **Modelado** que está conformado por tres fases:
    - Modelado de negocios
    - Modelado de datos
    - Modelado de procesos

  De esta fase se obtienen los diseños base para la posterior construcción.
- **Construcción** resalta el empleo de componentes de software existente y la aplicación de generación automática de código.
- **Despliegue** establece una base para las iteraciones subsecuentes si fueran necesarias.

Como todos los modelos, *D.R.A.* tiene ciertas limitaciones para su aplicación como por ejemplo: el proyecto no debe ser muy grande porque esto requeriría un enorme número de personal; debe existir un compromiso por parte de clientes y desarrolladores para poder llevar a cabo las actividades rápidas que propone el modelo de lo contrario el este fallará.

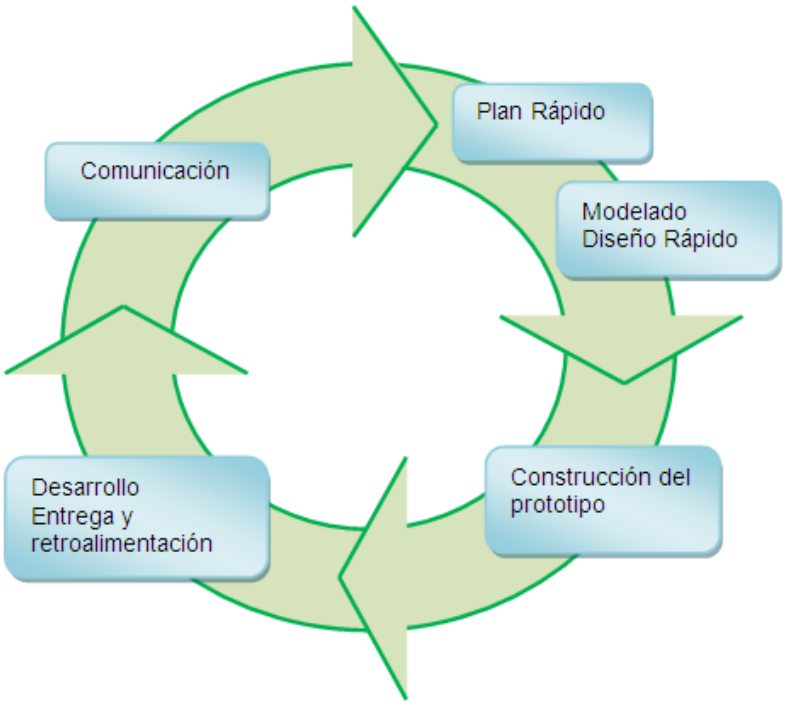
2.2.3 Modelo de Construcción de Prototipos

El método de construcción de prototipos, como su nombre lo indica se basa en la construcción de prototipos no funcionales o semifuncionales de un sistema, los mismos que ayudan tanto al cliente a definir sus requerimientos como al equipo de desarrollo a definir los algoritmos y procesos internos que más se ajusten a las necesidades del cliente.

Este procesos garantiza que los requerimientos del cliente sean totalmente satisfechos, sin embargo es sensible a "malentendidos" ya que al mostrar periódicamente prototipos sin funcionalidad, el cliente puede desilusionarse y terminar por cancelar el proyecto, otro de los riesgos es que por presentar prototipos a tiempo, no se cumpla con las funcionalidades lo que puede devenir en fallas en lo posterior, consta de varias etapas:

- **Comunicación:** el cliente y el ingeniero de software definen los requerimientos y las funcionalidades necesarias y posibles.
- **Plan rápido:** se plantea una iteración rápida de lo definido en los requerimientos.
- **Diseño Rápido:** se centra en representar los aspectos del producto que son visibles al usuario, como por ejemplo las interfaces de uso, los despliegues de información, etc.
- **Construcción del Prototipo:** es la etapa en la que se combina el diseño rápido con ciertos avances del sistema, para poder entregar al cliente a que lo evalúe y saber si se está logrando los propósitos planteados.

Figura 3. Modelo Basado en prototipos



Fuente: Ingeniería del Software. Un enfoque práctico. Roger S Pressman (2006).
Elaborado por: Verónica Noriega

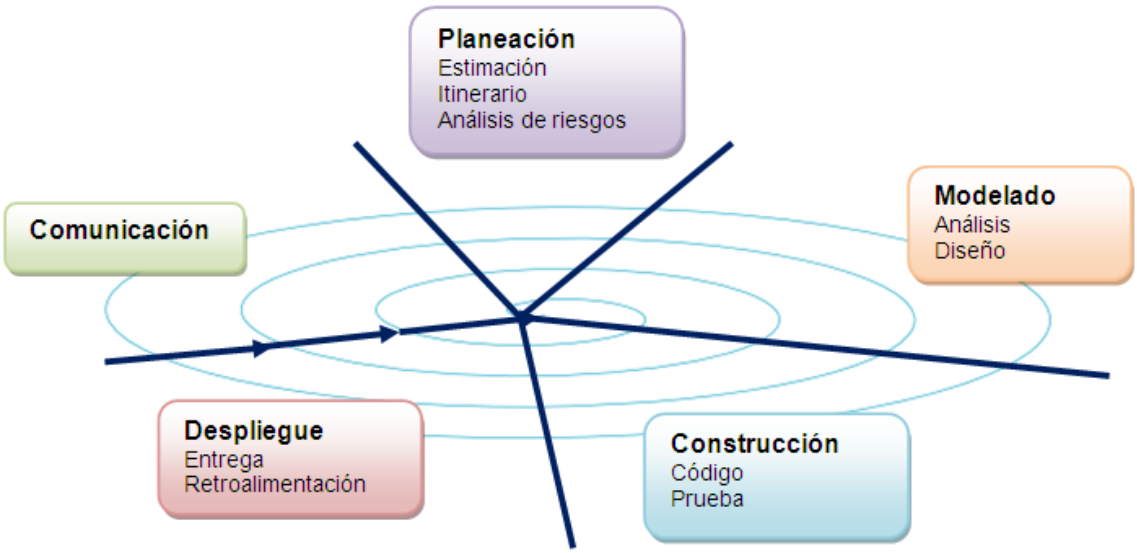
- **Desarrollo:** después de haber sido evaluado y con una retroalimentación por parte del cliente, se procede perfeccionar los requisititos, en busca de la satisfacción del cliente.

"A pesar de que tal vez surjan problemas, la construcción de prototipos puede ser un paradigma efectivo para la ingeniería del software. La clave es definir las reglas del juego desde el principio; es decir, el cliente y el desarrollador se deben poner de acuerdo en que el prototipo de construya y sirva como un mecanismo para la definición de requisitos." (Pressman, 2006).

2.2.4 El Modelo en Espiral

Este es un modelo de proceso de software evolutivo, que combina la naturaleza iterativa de la construcción de prototipos con las propiedades del modelo en cascada por lo tanto proporciona el potencial del desarrollo rápido de versiones de software.

Figura 4. Modelo en Espiral



Fuente: Ingeniería del Software. Un enfoque práctico. Roger S Pressman (2006).
Elaborado por: Verónica Noriega

En este modelo el software se lo realiza en una serie de desarrollos incrementales, las primeras iteraciones podrían ser incluso un boceto en papel o un prototipo, a medida que van avanzando las iteraciones se van produciendo las versiones más completas del software hasta llegar a su fin.

El modelo se divide en actividades estructurales que son:

- **Comunicación:** tarea que permite establecer comunicación entre el desarrollador y el cliente para definir requerimientos.
- **Planificación:** se establecen los recursos para el proyecto, se determinan los objetivos, alcances y restricciones, se revisa todo lo hecho y se evalúa para decidir si se continúa con las fases siguientes.
- **Modelado:** aquí se realiza un análisis de riesgos técnicos y de gestión, se identifican y se diseña en base a los problemas, tratando de dar solución.
- **Construcción:** es el desarrollo en sí de la aplicación donde se puede usar prototipos para el avance de cada iteración, también se aplica pruebas.
- **Despliegue:** consiste en la entrega al usuario, es decir instalación y soporte al usuario, de aquí se desprende también la retroalimentación, que indicara si hace falta alguna modificación.

CAPÍTULO III

CAPITULO III

METODOLOGÍAS DE DESARROLLO ÁGIL

3.1 ¿Qué es una metodología?

La definición de la palabra metodología dice que es un conjunto de métodos y procesos pre-establecidos que se deben seguir para poder lograr un fin.

En materia de Ingeniería de Software, una metodología de desarrollo de software es un conjunto de reglas, procedimientos y técnicas que sirven para llevar a cabo un proyecto de desarrollo. Todos estos elementos describen la forma de realizar la codificación, pruebas, corrección de errores, etc., dentro del entorno de desarrollo del producto de software.

En pocas palabras, la metodología es como una receta, que detalla paso a paso que se debe hacer, como se lo debe hacer, quién lo debe hacer, en qué momento lo debe hacer y con qué artefactos, para obtener resultados de éxito.

3.2 Necesidades de una metodología

Larry Trussell propone los siguientes puntos primordiales que debe tener una metodología:

1. **Visión del producto.** Todo el mundo debe conocer lo que el equipo está tratando de hacer, como debe de ser el producto final, las bases de la estrategia del producto y cuando el producto será entregado.

2. **Vinculación con el cliente.** La metodología se debe encargar de indicar la manera de gestionar el vínculo entre clientes, desarrolladores, especificación de requisitos y el personal de soporte.
3. **Establecer un modelo de ciclo de vida.** Un modelo de ciclo de vida como pueden ser el iterativo, secuencial o cascada, etc. De esta manera se establecen los pasos en el proceso de desarrollo y se pueden ubicar los recursos adecuadamente.

4. **Gestión de los requisitos.** Nivel de detalle que deben tener los requisitos del producto, siendo recomendable cuanto más alto mejor.

5. **Plan de desarrollo.** Es un documento con un plan para organizar los requisitos y cuestiones relacionadas con la calidad. Los ítems de este plan deben ser lo suficientemente detallados para que los programadores puedan desarrollar sus tareas de codificación de un modo no ambiguo. Como mínimo se debe especificar un proceso para añadir y modificar el documento, y mantener un histórico de los cambios.

6. **Integración del proyecto.** Una metodología de desarrollo debe conducir a una organización a determinar cómo se integrará el producto fabricado con los existentes y futuros productos de la compañía.

7. **Medidas de progreso del proyecto.** Se considera un aspecto crítico en una metodología. Desarrolladores, manager y los altos cargos de la organización deben entender el progreso del desarrollo del equipo de desarrollo. Ellos deben conocer el estado actual del producto, sí como una buena estimación del tiempo que resta para la finalización del proyecto.

8. **Métricas para evaluar la calidad.** El responsable de las *release* del producto depende de un proceso de para la medida de la calidad, el cual suele empezar en las primeras etapas de la planificación. Este proceso no es tan solo para encontrar fallos, produce indicadores de la robustez del producto y cuanto se aproxima el producto a las especificaciones iniciales.

9. **Maneras de medir el riesgo.** Un plan debe tener en cuenta los posibles problemas que pueden ocurrir durante el proceso de desarrollo, el impacto de estos problemas y que acciones deberían ser llevadas a cabo para solucionar o prevenir estos problemas. La gestión de los riesgos es la responsabilidad diaria de *los project managers* y desarrolladores.

10. **Como gestionar los cambios.** Nuevas ideas y problemas desembocan en cambios de diseño y especificación aunque ya hayamos empezado a implementar. Un plan debe contemplar estas sugerencias para introducirlas en el proyecto, debatirlas e implementarlas.

11. **Establecer una línea de meta.** La metodología de desarrollo debe forzar a una organización a especificar exactamente que está siendo construido y que

constituye el producto final. Todo el mundo en la organización debe tener una visión clara del producto final y entender qué significa *terminado.*

Estos puntos son solamente una referencia, no se los debería tomar como una ley, dado que en muchos casos las metodologías deben ser flexibles y adaptativas para responder a los retos que se presenten, sin embargo es una guía que nos permitirá establecer con mayor claridad a qué debemos y a qué no debemos considerar una metodología.

3.3 Definición de Desarrollo Ágil de Aplicaciones

En un proyecto de desarrollo de software es importante cumplir con las expectativas que tiene el cliente, que principalmente son tres:

- Que el producto final sea de calidad.
- Que el producto se entregue dentro de los plazos de tiempo estipulados.
- Que el producto no sobrepase el costo inicialmente presupuestado.

A través del tiempo, las metodologías tradicionales han buscado con *desesperación* cumplir con estos parámetros de éxito, sin lograrlo, ya que si lograban reducir uno, como consecuencia aumentaban los otros, lo que no permitía encontrar el balance justo entre rapidez, bajo costo y calidad.

Pero el principal problema era el tiempo que tomaba aplicar una metodología, por esa razón, muchos proyectos, para ahorrar tiempo, dejaban de usarlas a la mitad del desarrollo, lo que representaba un riesgo a futuro de fallas no reconocidas.

El dinamismo de las operaciones informáticas derivado principalmente por el rápido intercambio de información por medio de la internet ocasiona que "los sistemas basados en computadores y los productos de software estén acelerados y en continuo cambio" (Pressman, 2006), lo que ha generado una nueva necesidad entre los desarrolladores, la agilidad.

Pero en Ingeniería de Software ¿qué es agilidad?, ¿qué se puede y qué no se puede considerar ágil? A continuación alguna definiciones interesantes, dadas por expertos en la rama que aclaran algunas dudas:

Según Ivar Jacobson:

Agilidad se ha convertido actualmente en la palabra de moda en cuanto se describe un moderno proceso de software. Cualquiera es ágil. Un equipo ágil es un equipo rápido que responde de manera apropiada a los cambios. Éstos son, en gran parte, la materia del desarrollo de software. Cambios en el software que se va a construir, cambios entre los miembros del equipo, cambios debidos a las nuevas tecnologías, cambios de todo tipo que pueden incidir en el producto que se construye o en el proyecto que crea el producto. En cualquier actividad de software se debe incluir un soporte para los cambios, esto es algo que adoptamos porque es el alma y el corazón del software. Un equipo ágil reconoce que el software lo desarrollan individuos que

trabajan en equipo y que las aptitudes de esta gente, y su capacidad para colaborar, son esenciales para el éxito del proyecto.

Según Steven Goldman: La agilidad es dinámica, con contenido específico, ajustable al cambio de manera dinámica y orientada al crecimiento.

Según Roger Pressman (Pressman, 2006):

La agilidad es más que una respuesta efectiva al cambio. Estimula las estructuras y actitudes de los equipos para que la comunicación (entre los miembros del equipo, entre los técnicos y la gente de negocios, entre los ingenieros de software y sus gerentes) sea más fácil. Resalta la entrega rápida del software operativo y le resta importancia a los productos de trabajo intermedio; adopta al cliente como una parte del equipo de desarrollo y elimina la actitud del tipo *nosotros y ustedes* que aun perjudica a muchos proyectos de software; reconoce que la planeación tiene sus límites en un mundo incierto y que el plan de proyecto debe ser flexible.

Según Eduardo Díaz: La agilidad se define por la siguiente ecuación vectorial:

$$\text{Agilidad} = \text{Flexibilidad} + \text{Rapidez}$$

Por lo tanto, como conclusión de lo anteriormente expuesto, se puede decir que ahora es necesario desarrollar software que sea completamente adaptable (*maleable*), que sea susceptible de cambio, escalable y que sea eficiente en su desempeño como en el uso de recursos para su desarrollo, todo esto enmarcado obviamente en la rapidez de desarrollo.

3.1.1 Manifiesto Ágil

En febrero de 2001 se llevó a cabo una reunión en la que participó un grupo de diecisiete expertos de la industria del software, incluyendo algunos de los creadores o impulsores de metodologías de software. Su objetivo fue esbozar los valores y principios que deberían permitir a los equipos desarrollar software rápidamente y respondiendo a los cambios que puedan surgir a lo largo del proyecto.

Se pretendía ofrecer una alternativa a los procesos de desarrollo de software tradicionales, caracterizados por ser rígidos y dirigidos por la documentación que se genera en cada una de las actividades desarrolladas.

Después de esta reunión se creó *The Agile Alliance*, una organización, sin fines de lucro dedicada a promover los conceptos relacionados con el desarrollo ágil de software y ayudar a las organizaciones para que adopten dichos conceptos. El punto de partida fue el Manifiesto Ágil, un documento que resume la *filosofía ágil.*

El Manifiesto para el Desarrollo Ágil de Software es de suma importancia dentro del movimiento de las metodologías ágiles. El mismo representa una iniciativa conjunta entre los principales responsables de los procesos ágiles mencionados anteriormente para lograr unificar principios compartidos por las diversas metodologías de manera de crear un *framework* de trabajo que contribuya al mejoramiento del desarrollo ágil.

Uno de los principales objetivos del encuentro en que se generó el Manifiesto fue el de extraer un factor común de los principios esenciales que servirían de guía para cualquier metodología que se identifique como ágil. Esto concluyó en la declaración de lo que podríamos denominar el prólogo del Manifiesto:

"Estamos descubriendo mejores maneras de desarrollar software mediante su construcción y ayudando a que otras personas lo construyan." (Agile Alliance, 2001). Esto llevo a que se declararan ciertos principios que rigen a las metodologías ágiles, los mismos que valoran:

- **Al individuo y las interacciones del equipo de desarrollo sobre el proceso y las herramientas**. La gente es el principal factor de éxito de un proyecto de software. Es más importante construir un buen equipo que construir el entorno. Muchas veces se comete el error de construir primero el entorno y esperar que el equipo se adapte automáticamente. Es mejor crear el equipo y que éste configure su propio entorno de desarrollo en base a sus necesidades.

- **Desarrollar software que funciona más que conseguir una buena documentación.** La regla a seguir es no producir documentos a menos que sean necesarios de forma inmediata para tomar una decisión importante. Estos documentos deben ser cortos y centrarse en lo fundamental. La colaboración con el cliente más que la negociación de un contrato. Se propone que exista una interacción constante entre el cliente y el equipo de desarrollo. Esta colaboración entre ambos será la que marque la marcha del proyecto y asegure su éxito.

- **Valorar más la colaboración con el cliente que la negociación de contratos.** Las prácticas ágiles están especialmente indicadas para productos difíciles de definir con detalle en el principio, o que si se definieran así tendrían al final menos valor que si se van enriqueciendo con retroalimentación continua durante el desarrollo. También para los casos en los que los requisitos van a ser muy inestables por la velocidad del entorno de negocio. En el desarrollo ágil, el cliente es un miembro más del equipo, que se integra y colabora en el grupo de trabajo. Los modelos de contrato por obra no encajan.

- **Responder a los cambios más que seguir estrictamente un plan.** La habilidad de responder a los cambios que puedan surgir a los largo del proyecto (cambios en los requisitos, en la tecnología, en el equipo, etc.) determina también el éxito o fracaso del mismo. Por lo tanto, la planificación no debe ser estricta sino flexible y abierta.

Los valores anteriores inspiran los doce principios del manifiesto. Son características que diferencian un proceso ágil de uno tradicional. Los dos primeros principios son generales y resumen gran parte del espíritu ágil. El resto tienen que ver con el proceso a seguir y con el equipo de desarrollo, en cuanto metas a seguir y organización del mismo. Los principios son:

I. La prioridad es satisfacer al cliente mediante tempranas y continuas entregas de software que le aporten un valor.

II. Dar la bienvenida a los cambios. Se capturan los cambios para que el cliente tenga una ventaja competitiva.

III. Entregar frecuentemente software que funcione desde un par de semanas a un par de meses, con el menor intervalo de tiempo posible entre entregas.

IV. La gente del negocio y los desarrolladores deben trabajar juntos a lo largo del proyecto.

V. Construir el proyecto en torno a individuos motivados. Darles el entorno y el apoyo que necesitan y confiar en ellos para conseguir finalizar el trabajo.

VI. El diálogo cara a cara es el método más eficiente y efectivo para comunicar información dentro de un equipo de desarrollo.

VII. El software que funciona es la medida principal de progreso.

VIII. Los procesos ágiles promueven un desarrollo sostenible. Los promotores, desarrolladores y usuarios deberían ser capaces de mantener una paz constante.

IX. La atención continua a la calidad técnica y al buen diseño mejora la agilidad.

X. La simplicidad es esencial.

XI. Las mejores arquitecturas, requisitos y diseños surgen de los equipos organizados por sí mismos.

XII. En intervalos regulares, el equipo reflexiona respecto a cómo llegar a ser más efectivo, y según esto ajusta su comportamiento.

3.2 Metodologías de Desarrollo Ágil de Aplicaciones

3.2.1 *XP* (*eXtreme Programming*)

XP es una metodología ágil que se centra en las personas mas no en los procesos, actualmente *XP* se proyecta a ser un modelo de desarrollo común, sencillo y adaptable a las características cambiantes y exigentes de empresas y clientes.

Se basa principalmente en cuatro principios: simplicidad, comunicación, retroalimentación y coraje. Además, está orientada por pruebas y refactorización, se diseña e implementan las pruebas antes de programar la funcionalidad, el programador crea sus propios *tests de unidad* lo que garantiza la calidad del software. Este método fue creado por Kent Beck, el principal objetivo de *XP* son grupos pequeños y medianos de desarrollo de software en donde los requisitos aún son muy ambigüos, cambian rápidamente o son de alto riesgo.(Jeffries, 2006)

- **Actividades de *Extremme Programming***

XP busca la satisfacción del cliente tratando de mantener durante todo el tiempo su confianza en el producto, también sugiere que el lugar de trabajo sea en una sala amplia, si es posible sin divisiones (en el centro los programadores, en la periferia los equipos individuales). Una ventaja del espacio abierto es el incremento en la comunicación y el proporcionar una

agenda dinámica en el entorno de cada proyecto. *XP* basa su funcionamiento en cuatros actividades fundamentales que se detallan a continuación:

Codificar. Comprende la etapa donde a través del código se va definiendo las soluciones para los requerimientos del usuario, es decir se va haciendo al software de a poco.

Hacer pruebas. Las pruebas permiten saber si lo implementado es lo que en realidad se tenía en mente, además indican si nuestro trabajo funciona, cuando no podemos pensar en ninguna prueba que pudiese originar un fallo en nuestro sistema, entonces habremos acabado por completo.

Escuchar. Para realizar pruebas se debe preguntar a la persona que necesita la información si lo obtenido es lo deseado. Es esencial escuchar al cliente, definir sus requerimientos y alcanzar un nivel de entendimiento entre las partes que permita avanzar con el proyecto, mejorándolo a través del proceso.

Diseñar. El diseño crea una estructura que organiza la lógica del sistema, un buen diseño permite que el sistema crezca con cambios en un solo lugar. Los diseños deben de ser sencillos, si alguna parte del sistema es de desarrollo complejo, lo apropiado es dividirla en varias. Si hay fallos en el diseño o malos diseños, estos deben de ser corregidos cuanto antes.

- **Principios de *Extremme Programming***

Los principios originales de la programación extrema son: simplicidad, comunicación, retroalimentación (*feedback*) y coraje. Un quinto principio,

respeto, fue añadido en la segunda edición de *Extreme Programming Explained.* Los cinco principios se detallan a continuación:

- ✓ **Simplicidad.** La simplicidad es la base de la programación extrema. Se simplifica el diseño para agilizar el desarrollo y facilitar el mantenimiento. Un diseño complejo del código junto a sucesivas modificaciones por parte de diferentes desarrolladores hace que la complejidad aumente exponencialmente. Para mantener la simplicidad es necesaria la refactorización del código, ésta es la manera de mantener el código simple a medida que crece. También se aplica la simplicidad en la documentación, de esta manera el código debe comentarse en su justa medida, intentando eso sí que el código esté autodocumentado. Para ello se deben elegir adecuadamente los nombres de las variables, métodos y clases. Los nombres largos no decrementan la eficiencia del código ni el tiempo de desarrollo gracias a las herramientas de autocompletado y refactorización que existen actualmente. Aplicando la simplicidad junto con la autoría colectiva del código y la programación por parejas se asegura que cuanto más grande se haga el proyecto, todo el equipo conocerá más y mejor el sistema completo.

- ✓ **Comunicación.** La comunicación se realiza de diferentes formas. Para los programadores el código comunica mejor cuanto más simple sea. Si el código es complejo hay que esforzarse para hacerlo inteligible. El código autodocumentado es más fiable que los comentarios ya que éstos últimos pronto quedan desfasados con el código a medida que es modificado.

Debe comentarse sólo aquello que no va a variar, por ejemplo el objetivo de una clase o la funcionalidad de un método. Las pruebas unitarias son otra forma de comunicación ya que describen el diseño de las clases y los métodos al mostrar ejemplos concretos de cómo utilizar su funcionalidad. Los programadores se comunican constantemente gracias a la programación por parejas. La comunicación con el cliente es fluida ya que el cliente forma parte del equipo de desarrollo. El cliente decide qué características tienen prioridad y siempre debe estar disponible para solucionar dudas.

- ✓ **Retroalimentación (*feedback*).** Al estar el cliente integrado en el proyecto, su opinión sobre el estado del proyecto se conoce en tiempo real. Al realizarse ciclos muy cortos tras los cuales se muestran resultados, se minimiza el tener que rehacer partes que no cumplen con los requisitos y ayuda a los programadores a centrarse en lo que es más importante. Considérense los problemas que derivan de tener ciclos muy largos. Meses de trabajo pueden tirarse por la borda debido a cambios en los criterios del cliente o malentendidos por parte del equipo de desarrollo. El código también es una fuente de retroalimentación gracias a las herramientas de desarrollo. Por ejemplo, las pruebas unitarias informan sobre el estado de salud del código. Ejecutar las pruebas unitarias frecuentemente permite descubrir fallos debidos a cambios recientes en el código.

- ✓ **Coraje o valentía.** Los puntos anteriores parecen tener sentido común, entonces, *¿por qué coraje?* Para los gerentes la programación en parejas puede ser difícil de aceptar porque les parece como si la productividad se fuese a reducir a la mitad ya que solo la mitad de los programadores está escribiendo código. Hay que ser valiente para confiar en que la programación por parejas beneficia la calidad del código sin repercutir negativamente en la productividad. Se requiere coraje para implementar las características que el cliente quiere ahora sin caer en la tentación de optar por un enfoque más flexible que permita futuras modificaciones. No se debe emprender el desarrollo de grandes marcos de trabajo (*frameworks*) mientras el cliente espera. En ese tiempo el cliente no recibe noticias sobre los avances del proyecto y el equipo de desarrollo no recibe retroalimentación para saber si va en la dirección correcta.

- ✓ **Respeto.** El respeto se manifiesta de varias formas. Los miembros del equipo se respetan los unos a otros, porque los programadores no pueden realizar cambios que hacen que las pruebas existentes fallen o que demore el trabajo de sus compañeros. Los miembros se respetan su trabajo porque siempre están luchando por la alta calidad en el producto y buscando el diseño óptimo o más eficiente para la solución a través de la refactorización del código.

- **Prácticas de *Extremme Programming***

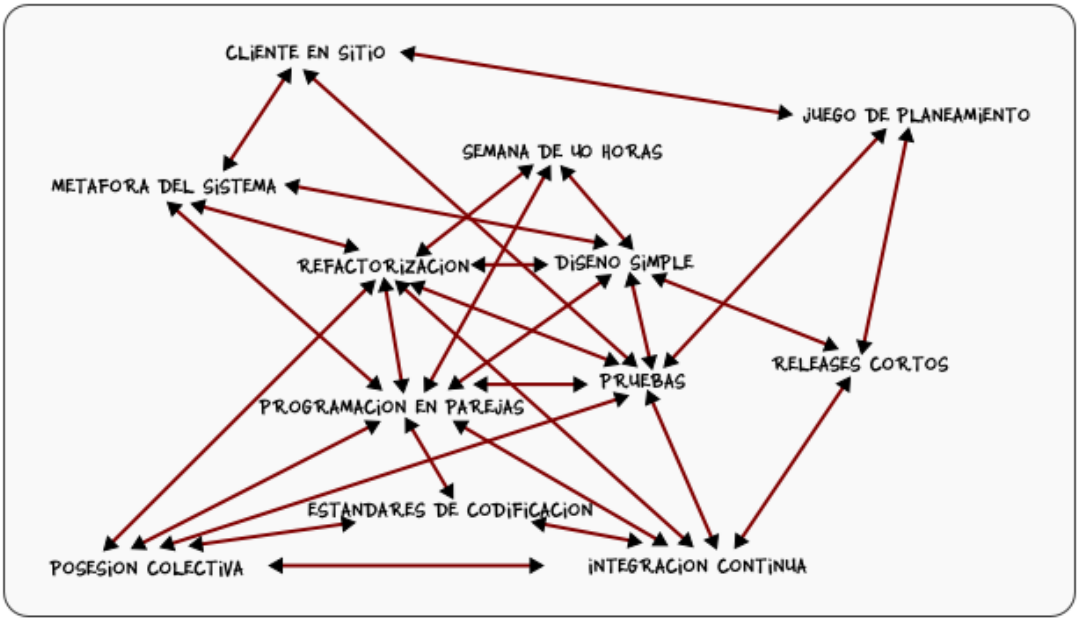
XP puede ser visto como un método demasiado informal, ya que sus procesos no se asemejan a una metodología, pero cabe decir que en forma aislada

cualquier práctica individual de *XP* tiene poco sentido, pero en conjunto, unas compensan las carencias que las otras puedan tener.

En este sentido, se han establecido ciertas prácticas que ayudan al proceso y que garantizan un buen funcionamiento del método, a continuación las describiremos:

El juego de la Planificación (*Planning Game*). El alcance de la siguiente versión está definido por las consideraciones de negocios (prioridad de los módulos, fechas de entrega) y estimaciones técnicas (estimaciones de funciones, consecuencias). El objetivo del juego es maximizar el valor del software producido. La estrategia es poner en producción las características más importantes lo antes posible, las piezas clave son las *story cards*, los jugadores son los desarrolladores y el cliente, y las movidas son Exploración, Selección y Actualización.

Figura 5. Prácticas de *XP* y sus interrelaciones.



Fuente: http://www.extremeprogramming.host56.com/ARTICULO4.php

- **Versiones Pequeñas (*Short Releases*).** Un sistema simple se pone rápidamente en producción. Periódicamente, se producen nuevas versiones agregando en cada iteración aquellas funciones consideradas valiosas para el cliente.

- **Metáfora del Sistema (*Metaphor*).** Cada Proyecto es guiado por una historia simple de cómo funciona el sistema en general, reemplaza a la arquitectura y debe estar en lenguaje común, entendible para todos (Cliente y Desarrolladores), esta puede cambiar permanentemente.

- **Diseño Simple (*Simple Designs*).** El sistema se diseña con la máxima simplicidad posible. Se plasma el diseño en *tarjetas CRC* (Clase-Responsabilidad-Colaboración), no se implementan características que no son necesarias, con esta técnica, las clases descubiertas durante el análisis pueden ser filtradas para determinar qué clases son realmente necesarias para el sistema.

- **Pruebas Continuas (*Testing*).** Los casos de prueba se escriben antes que el código. Los desarrolladores escriben pruebas unitarias y los clientes especifican pruebas funcionales.

- **Refactorización (*Refactoring*).** Es posible reestructurar el sistema sin cambiar su comportamiento, por ejemplo eliminando código duplicado, simplificando funciones, mejorando el código constantemente, si el

código se está volviendo complicado se debería modificar el diseño y volver a uno más simple.

- **Programación por parejas (*Pair Programming*).** El código es escrito por dos personas trabajando en el mismo computador. “Una sola maquina con un teclado y un mouse”

- **Posesión Colectiva del Código (*Collective Code Ownership*).** Nadie es dueño de un módulo. Cualquier programador puede cambiar cualquier parte del sistema en cualquier momento, siempre se utilizan estándares y se excluyen los comentarios, los test siempre deben funcionar al 100% para realizar integraciones con todo el código permanentemente.

- **Integración continua (*Continuous Integration*).** Los cambios se integran en el código base varias veces por día. Todos los casos de prueba se deben pasar antes y después de la integración, se dispone de una máquina para la integración y se realizan test funcionales en donde participa el cliente.

- **Semana laboral de 40 horas (*40-Hour Week*).** Cada colaborador trabaja no más de 40 Horas por semana. Si fuera necesario hacer horas extra, esto no debería hacerse dos semanas consecutivas. Esto hace que se reduzca la rotación del personal y mejora la calidad del producto.

- **Cliente en el Sitio (*On Site Customer*).** El equipo de desarrollo tiene acceso todo el tiempo al cliente, el cual está disponible para responder

preguntas, fijar prioridades, etc. Esto no siempre se consigue, un cliente sin experiencia no sirve y un cliente con la experiencia requerida no es disponible, lo ideal es un cliente Analista. (Jeffries, 2007)

- **Estándares de Codificación (*Coding Standard*).** Todo el código debe estar escrito de acuerdo a un estándar de codificación previamente fijado.

- **Actores y Responsabilidades de *Extremme Programming***

  Existen diferentes roles (*actores*) y responsabilidades en *XP* para diferentes tareas y propósitos durante el proceso:

  - **Programador (*Programmer*)**

    Responsable de decisiones técnicas

    Responsable de construir el sistema

    Sin distinción entre analistas, diseñadores o codificadores

    En *XP*, los programadores diseñan, programan y realizan las pruebas

  - **Cliente (*Customer*)**

    Es parte del equipo

    Determina qué construir y cuándo

    Escribe *tests funcionales* para determinar cuándo está completo un determinado aspecto

- **Entrenador (*Coach*)**

  El líder del equipo - toma las decisiones importantes

  Principal responsable del proceso

  Tiende a estar en un segundo plano a medida que el equipo madura

- **Rastreador (*Tracker*)**

  Observa sin molestar

  Conserva datos históricos

- **Probador (*Tester*)**

  Ayuda al cliente con las pruebas funcionales

  Se asegura de que los *tests funcionales* se ejecutan

- **Fases de *Extreme Programming***

  El ciclo de vida de *XP* se enfatiza en el carácter iterativo e incremental del desarrollo, una iteración de desarrollo es un período de tiempo en el que se realiza un conjunto de funcionalidades determinadas que en el caso de *XP* corresponden a un conjunto de historias de usuarios.

  Las iteraciones son relativamente cortas ya que se piensa que entre más rápido se le entreguen desarrollos al cliente, más retroalimentación se va a obtener y esto va a representar una mejor calidad del producto a largo plazo. Existe una fase de análisis inicial orientada a programar las iteraciones de

desarrollo y cada iteración incluye diseño, codificación y pruebas, fases superpuestas de tal manera que no se separen en el tiempo.

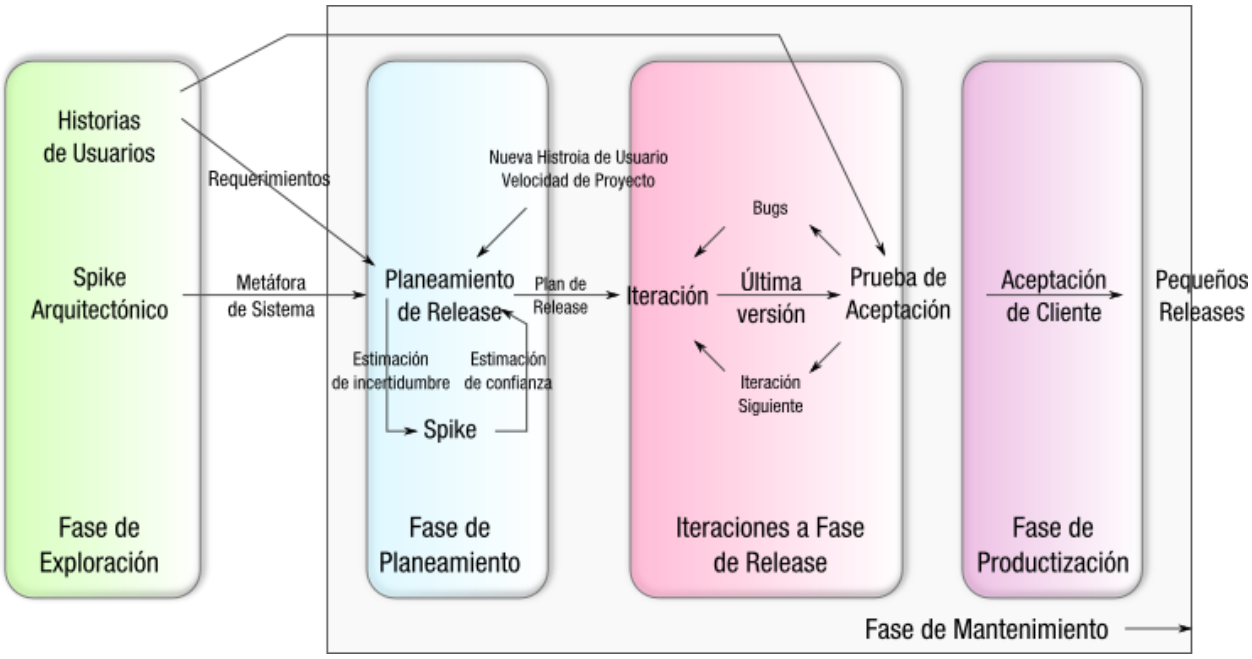
- ✓ **Fase de la exploración:** En esta fase los clientes plantean a grandes rasgos las historias de usuario que son de interés para la primera entrega del producto. Al mismo tiempo el equipo de desarrollo se familiariza con las herramientas, tecnologías y prácticas que se utilizarán en el proyecto.

  Se prueba la tecnología y se exploran las posibilidades de la arquitectura del sistema construyendo un prototipo. La fase de exploración toma de pocas semanas a pocos meses, dependiendo del tamaño y familiaridad que tengan los programadores con la tecnología.

- ✓ **Fase del planeamiento:** se priorizan las historias de usuario y se acuerda el alcance del *release*. Los programadores estiman cuánto esfuerzo requiere cada historia y a partir de allí se define el cronograma. La duración del cronograma del primer *release* no excede normalmente dos meses. La fase de planeamiento toma un par de días. Se deben incluir varias iteraciones para lograr un release. El cronograma fijado en la etapa de planeamiento se realiza a un número de iteraciones, cada una toma de una a cuatro semanas en ejecución. La primera iteración crea un sistema con la arquitectura del sistema completo. Esto es alcanzado seleccionando las historias que harán cumplir la construcción de la estructura para el sistema completo. El cliente decide las historias que se seleccionarán para cada iteración. Las pruebas funcionales creadas por el cliente se ejecutan

al final de cada iteración. Al final de la última iteración el sistema está listo para producción.

Figura 6. Prácticas de *XP* y sus interrelaciones



Fuente: http://www.extremeprogramming.host56.com/ARTICULO4.php

- ✓ **Fase de producción:** requiere prueba y comprobación extra del funcionamiento del sistema antes de que éste se pueda liberar al cliente. En esta fase, los nuevos cambios pueden todavía ser encontrados y debe tomarse la decisión de si se incluyen o no en el *release* actual. Durante esta fase, las iteraciones pueden ser aceleradas de una a tres semanas. Las ideas y las sugerencias pospuestas se documentan para una puesta en práctica posterior por ejemplo en la fase de mantenimiento. Después de que se realice el primer *release* productivo para uso del cliente, el

proyecto de *XP* debe mantener el funcionamiento del sistema mientras que realiza nuevas iteraciones.

- ✓ **Fase de mantenimiento:** requiere de un mayor esfuerzo para satisfacer también las tareas del cliente. Así, la velocidad del desarrollo puede desacelerar después de que el sistema esté en la producción. La fase de mantenimiento puede requerir la incorporación de nueva gente y cambiar la estructura del equipo.

- ✓ **Fase de muerte:** Es cuando el cliente no tiene más historias para ser incluidas en el sistema. Esto requiere que se satisfagan las necesidades del cliente en otros aspectos como rendimiento y confiabilidad del sistema. Se genera la documentación final del sistema y no se realizan más cambios en la arquitectura. La muerte del proyecto también ocurre cuando el sistema no genera los beneficios esperados por el cliente o cuando no hay presupuesto para mantenerlo.

- **Artefactos de Extreme Programming**

  A continuación se describen los artefactos de *XP*, su utilidad dentro del proceso y como se debe usarlos, entre otros se encuentran: *Historias de Usuario, Tareas de Ingeniería, Tarjetas CRC y Spike.*

Historias de Usuario. Representan una breve descripción del comportamiento del sistema, emplea terminología del cliente sin lenguaje técnico, se realiza una por cada característica principal del sistema, se emplean para hacer estimaciones de tiempo y para el plan de lanzamientos, reemplazan un gran documento de requisitos y presiden la creación de las pruebas de aceptación.

Cuadro 4. Modelo Propuesto de Tarjeta para recolectar historias de usuario.

	Historia de Usuario
Número:	**Nombre Historia de Usuario:**
Modificación (o extensión) de Historia de Usuario (Nro. y Nombre):	
Usuario:	**Iteración Asignada:**
Prioridad en Negocio: (Alta / Media / Baja)	**Puntos Estimados:**
Riesgo en Desarrollo: (Alto / Medio / Bajo)	**Puntos Reales:**
Descripción:	
Observaciones:	

Fuente: http://www.monografias.com/trabajos51/programacion-extrema/programacion-extrema2.shtml

Estas deben proporcionar sólo el detalle suficiente como para poder hacer razonable la estimación de cuánto tiempo requiere la implementación de la historia, difiere de los casos de uso porque son escritos por el cliente, no por los programadores, empleando terminología del cliente. Las historias de usuario son más *amigables* que los casos de uso formales.(Wake, 2004)

Las Historias de Usuario tienen tres aspectos:

- **Tarjeta:** en ella se almacena suficiente información para identificar y detallar la historia.

- **Conversación:** cliente y programadores discuten la historia para ampliar los detalles (verbalmente cuando sea posible, pero documentada cuando se requiera confirmación)

***Pruebas de Aceptación*:** permite confirmar que la historia ha sido implementada correctamente.

Cuadro 5. Modelo Propuesto para prueba de aceptación.

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

Fuente: http://www.monografias.com/trabajos51/programacion-extrema/programacion-extrema2.shtml

Tareas de Ingeniería. Aunque *XP* es un método que reconoce al equipo entero como el responsable de las acciones que se realicen dentro del entorno de desarrollo, es necesario fijar ciertas *responsabilidades* específicas a personas que están capacitadas para ello, para eso existen las tareas de

ingeniería que permiten designar a una persona que se hará cargo de una tarea específica, esto facilita el control de avance y cambios, además que mejora la productividad del equipo de desarrollo.

Cuadro 6. Modelo Propuesto de Tarjeta para asignación de tareas de ingeniería.

	Tarea de Ingeniería
Número Tarea:	**Historia de Usuario (Nro. y Nombre):**
Nombre Tarea:	
Tipo de Tarea : Desarrollo / Corrección / Mejora / Otra (especificar)	**Puntos Estimados:**
Fecha Inicio:	**Fecha Fin:**
Programador Responsable:	
Descripción:	

Fuente: http://www.monografias.com/trabajos51/programacion-extrema/programacion-extrema2.shtml

***Tarjetas CRC* (Clase - Responsabilidad – Colaborador).** Estas tarjetas se dividen en tres secciones que contienen la información del nombre de la clase, sus responsabilidades y sus colaboradores.

Una clase es cualquier persona, cosa, evento, concepto, pantalla o reporte. Las responsabilidades de una clase son las cosas que conoce y las que realizan, sus atributos y métodos. Los colaboradores de una clase son las demás clases con las que trabaja en conjunto para llevar a cabo sus responsabilidades.

Cuadro 7. Modelo Propuesto de Tarjeta CRC.

Nombre de la Clase	
Responsabilidades:	**Colaboradores:**

Fuente: http://www.monografias.com/trabajos51/programacion-extrema/programacion-extrema2.shtml

En la práctica conviene tener pequeñas tarjetas de cartón, que se llenarán y que son mostradas al cliente, de manera que se pueda llegar a un acuerdo sobre la validez de las abstracciones propuestas.

Los pasos a seguir para llenar las tarjetas son los siguientes:

- Encontrar clases

- Encontrar responsabilidades
- Definir colaboradores
- Disponer las tarjetas

Para encontrar las clases debemos pensar qué cosas interactúan con el sistema (en nuestro caso el usuario), y qué cosas son parte del sistema, así como las pantallas útiles a la aplicación (un despliegue de datos, una entrada de parámetros y una pantalla general, entre otros). Una vez que las clases principales han sido encontradas se procede a buscar los atributos y las responsabilidades, para esto se puede formular la pregunta ¿Qué sabe la clase? y ¿Qué hace la clase? Finalmente se buscan los colaboradores dentro de la lista de clases que se tenga.

Spike. Hay situaciones en las que los desarrolladores simplemente no saben lo suficiente acerca de una historia para poder prever y estimar. En ese caso, se llevaría a cabo lo que se conoce como *Spike.* El *Spike* simplemente es una rápida y desechable prueba de concepto para permitir a los desarrolladores tener una idea de lo que es realmente necesario para aplicar una historia. El resultado es que los desarrolladores pueden proporcionar una estimación adecuada de la historia. Tenga en cuenta, sin embargo, que un *spike* se debe limitarse a no más de una semana de duración, y lo ideal sería tan sólo un día o dos.

3.2.2 *AUP* (*Agile Unified Process*)

AUP cuyas siglas quieren decir *Agile Unified Process*, es una versión simplificada de *RUP* (*Rational Unified Process*). *AUP* describe un enfoque simple y fácil de usar para desarrollar software de aplicación empresarial, usando técnicas ágiles y conceptos que aún se mantienen fieles a *RUP*, también aplica técnicas ágiles tales como: *Test Driven Development* (TDD), *Agile Model Driven Development* (AMDD), gestión del cambio ágil y *database refactoring* para mejorar su productividad.

La gestión de riesgos desempeña un papel importante en los proyectos de *AUP*, debido a que *AUP* hace hincapié en que los elementos de alto riesgo tengan prioridad en el desarrollo temprano de la aplicación, para este fin se suele crear una lista de riesgos desde el principio y durante todo el proceso de desarrollo. Este método también toma en cuenta el desarrollo temprano de una base arquitectónica ejecutable. Este núcleo arquitectónico se desarrolla durante la fase de elaboración para validar los requisitos clave, hipótesis, y frente a los riesgos técnicos.

- **Prácticas de *Agile Unified Process***

A diferencia de *RUP*, el *AUP* solamente se basa en siete disciplinas:

1. **Modelamiento.** Consiste en entender el negocio de la organización, el dominio del problema que está dirigido por el proyecto e identificar una solución viable al dominio del problema.

2. **Aplicación.** Aquí se transforma el modelo resultado de la fase de modelamiento en código ejecutable y se realizan pruebas de nivel básico, tales como las pruebas unitarias.
3. **Pruebas.** Realizar una evaluación objetiva para asegurar la calidad. Esto incluye encontrar defectos, validar que el sistema funciona como se diseñó y verificar que se cumplan los requisitos establecidos.
4. **Implementación.** Plan para la entrega del sistema y ejecutar el plan para dejar el sistema a disposición de los usuarios finales.
5. ***Configuration Management.*** Administrar el acceso a los artefactos del proyecto. Esto incluye no sólo el seguimiento de versiones de los artefactos a través del tiempo, sino también el control y gestión del cambio para ellos.
6. ***Project Management.*** Dirigir las actividades que se llevan a cabo dentro del proyecto. Esto incluye la gestión de riesgos, la dirección de personas (la asignación de tareas, seguimiento de los progresos, etc), y coordinar con la gente y los sistemas que están fuera del marco del proyecto para asegurarse de que el producto llegue a tiempo y dentro del presupuesto.
7. **Entorno.** Apoyo al resto de los esfuerzos para garantizar que la orientación (normas y directrices), las herramientas (*hardware, software,* etc.) y un proceso adecuado estén disponibles para el equipo según sea necesario.

- **Fases de *Agile Unified Process***

El Proceso Unificado (UP) es un proceso que está caracterizado de desarrollo iterativo e incremental. La elaboración, construcción y fases de transición se dividen en una serie de iteraciones de tiempo determinado. (La fase inicial también se puede dividir en iteraciones si es un proyecto grande.) Los resultados de cada iteración conforman un incremento, que a su vez es una versión nueva del sistema que contiene más y mejor funcionalidad en comparación con la versión anterior.

AUP trabaja de la misma manera, desarrollando el producto en etapas, es decir en su propio ciclo de vida, las etapas o fases de *AUP* a través del proyecto son las mismas que las de *RUP*, y son cuatro:

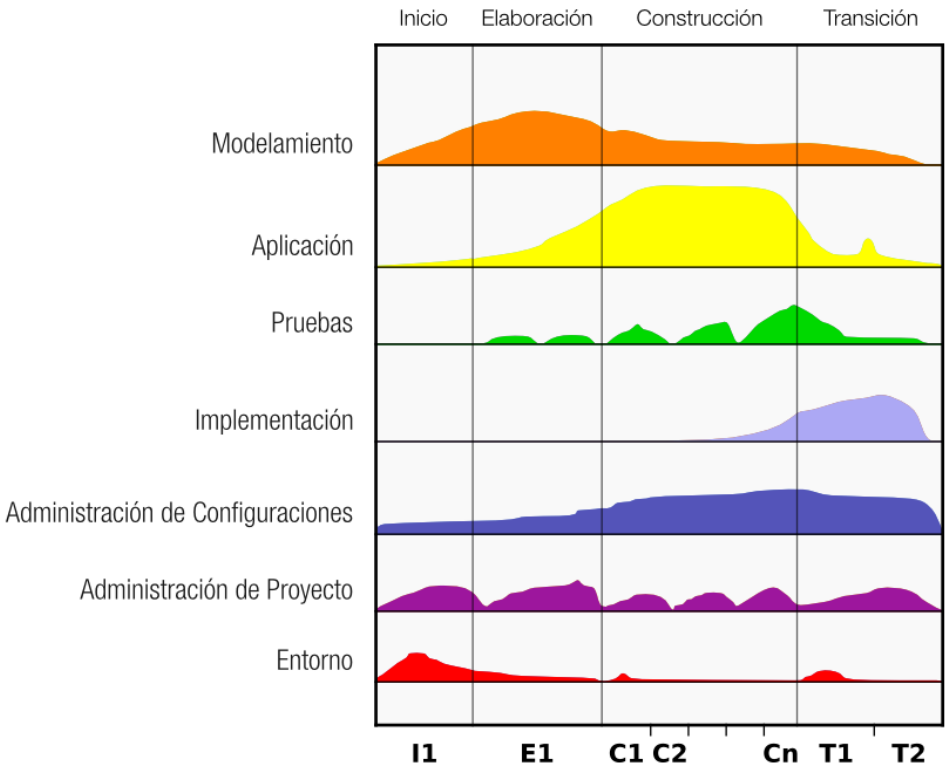
- ✓ **Inicio.** El objetivo de esta etapa es identificar el alcance inicial del proyecto, establecer una arquitectura potencial del sistema, establecer un presupuesto y buscar financiamiento, además de la aceptación de las partes interesadas.

- ✓ **Elaboración.** Durante la fase de elaboración el equipo del proyecto espera captar la mayoría de requisitos útiles para el sistema. Sin embargo, los objetivos primarios de elaboración son: hacer frente a factores de riesgo conocidos, establecer y validar la arquitectura del sistema. Algunos procesos comunes son realizados en esta fase tales como: la creación de *diagramas de casos de uso, diagramas*

conceptuales (*diagramas de clase* sólo con la notación básica) y los diagramas de paquetes (*diagramas de arquitectura*).

El final de la fase de Elaboración es la entrega de un plan que incluye costos y tiempo estimados para la fase de construcción. El plan debe ser preciso y confiable ya que debe basarse en los datos obtenidos durante toda la fase de elaboración.

Figura 7. Ciclo de vida de AUP (*Agile Unified Process*).



Fuente: http://www.ambysoft.com/unifiedprocess/agileUP.html

- ✓ **Construcción.** Esta es considerada la fase más larga en el proyecto, aquí se construye el sistema sobre las bases establecidas en la fase de Elaboración. Las características del sistema se implementan en una serie de iteraciones cortas, de tiempos definidos. Los resultados de

cada iteración son una versión ejecutable del *software*. Se acostumbra a elaborar casos de uso durante la fase de construcción y cada uno se convierte luego en el inicio de una nueva iteración.

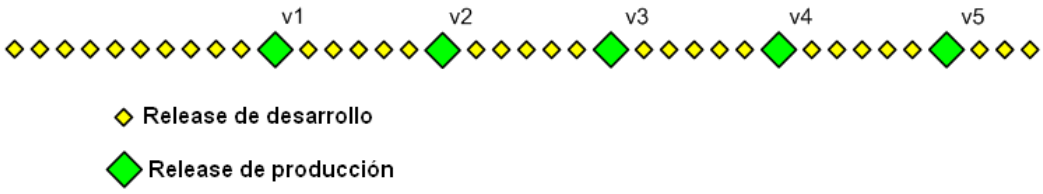
En esta fase se usan algunos diagramas *UML* comunes tales como: *Diagramas de actividad, de secuencia, de colaboración, de estado de transición y diagramas de interacción en general.*

- ✓ **Transición.** Esta es la fase final del proyecto, aquí se muestra el sistema a los usuarios finales, se reciben comentarios y se realiza un *Feedback* de la versión o versiones iniciales, este proceso puede dar lugar a ajustes adicionales que se incorporarán a lo largo de varias iteraciones. La fase de transición también incluye conversiones de sistema y capacitación de los usuarios.

  El final de la Fase de Transición está marcado por la entrega de una versión semifuncional del producto que va de a poco delineando cómo será el producto terminado.

  AUP es una metodología que garantiza la entrega de una nueva versión cada cierto tiempo, ya que todo su proceso trabaja en función de eso. Las versiones pueden variar su tiempo entre 2 y 3 semanas entre ellas, pero la primera versión o *release* siempre demora un poco más que las demás.

Figura 8. Tiempo entre versiones de AUP



Fuente: http://www.ambysoft.com/unifiedprocess/agileUP.html

- **Principios de *Agile Unified Process***

Existen algunos principios de *AUP*, que se deben tomar encuentra para que el proceso no tenga contratiempos, a continuación se enumeran:

1. **El personal necesita saber lo que está haciendo.** La gente no va a leer la documentación de los procesos en detalle, sino que quieren una orientación de alto nivel y/o formación de vez en cuando. El producto *AUP* proporciona enlaces a muchos de los detalles si uno está interesado pero no obliga seguir los detalles.
2. **Simplicidad.** Todo se describe concisamente usando unas páginas, no miles de páginas.
3. **Agilidad.** El *AUP* se ajusta a los valores y principios de la Alianza Ágil.
4. **Centrarse en las actividades importantes.** La atención se centra en las actividades que realmente cuentan.

5. **Herramienta de la independencia.** Poder usar cualquier herramienta que desee utilizar con la *AUP*. Es recomendable utilizar herramientas que mejor se adapten para el trabajo, que a menudo son herramientas simples o incluso herramientas de código abierto.

6. **Querer adaptar este producto para satisfacer sus propias necesidades.** El producto *AUP* es fácil de manejar a través de cualquier herramienta de edición de *HTML*. No se necesita comprar una. herramienta especial, o tomar un curso, para adaptar el *AUP*.

3.2.3 Crystal Clear

Crystal es una metodología de desarrollo de *Software* ágil, más que una metodología se la considera una familia de metodologías, debido a que se subdivide en varias categorías de acuerdo a la cantidad de personas que participan en el proyecto. Es una metodología creada por Alistair Cockburn quien la creó en base al análisis de distintos proyectos de desarrollo de software y su propia experiencia, componentes que dieron como resultado una metodología robusta, la cual es interesante conocer.

Como se explicó antes *Crystal* es una familia de metodologías con un *código genético* en común, se clasifica por el número de participantes en el equipo de desarrollo, y de acuerdo a su clasificación se define el método y los procesos a seguir:

Cuadro 8. Clasificación de la familia Crystal

Clear	**Equipos de hasta 8 personas o menos.**
Amarillo	Equipos entre 10 a 20 personas.
Naranja	Equipos entre 20 a 50 personas.
Rojo	Equipos entre 50 a 100 personas.
Azul	Equipos entre 100 a 200 personas.

Fuente: http://alistair.cockburn.us/Crystal+methodologies
Elaborado por: Verónica Noriega

Los métodos se llaman *Crystal* evocando las facetas de una gema: cada faceta es otra versión del proceso, y todas se sitúan en torno a un núcleo idéntico.

Crystal propone un modelo donde haya menos énfasis en la documentación exhaustiva y más en versiones funcionales que puedan ser probadas. Lo primero son promesas, lo segundo son hechos. Cada proyecto necesita sus propios métodos, los cuales deben adaptarse a los requerimientos.

Tiene una orientación humana, que se centra en las personas. Cockburn considera que las personas encuentran difícil seguir un proceso disciplinado, así que la alternativa que propone es seguir una metodología menos disciplinada que asegure una mayor probabilidad de éxito, intercambiando conscientemente productividad por facilidad de ejecución, es decir que aunque *Crystal* es menos productivo en comparación con otras metodologías, más personas serán capaces de seguirlo.

Las revisiones al final de cada iteración son de vital importancia, ya que esto incita al proceso a aplicar técnicas de mejoramiento continuo en forma automática, afirmando así la idea de que el desarrollo iterativo está para encontrar los problemas temprano y poder corregirlos a tiempo. Esto exige que cada uno de los miembros del equipo ponga más énfasis en su proceso, afinándolo conforme avanza el desarrollo.

Según Watson (Watson, 2006), cuando se usa *Crystal*, se debe considerar a las personas como dispositivos activos que tienen modos de éxito y modos de fallo y que el proyecto es susceptible al fracaso, es una posibilidad que hay que tomar en cuenta. A continuación se describen algunas situaciones que deben ser consideradas durante el proceso de desarrollo, para asegurar una ejecución sin contratiempos:

- Cuando el número de personas aumenta, también aumenta la necesidad de coordinar.
- Cuando el potencial de daños se incrementa, la tolerancia a variaciones se ve afectada.
- La sensibilidad del tiempo en que se debe estar en el mercado varía: a veces este tiempo debe acortarse al máximo y se toleran defectos, otras se enfatiza la auditoria, confiabilidad, protección legal, entre otros.
- Las personas se comunican mejor cara a cara, con la pregunta y la respuesta en el mismo espacio de tiempo.

- El factor más significativo es *comunicación*.

Aunque existen varias clasificaciones, la más exhaustivamente documentada es *Crystal Clear* (CC), que puede ser usada en proyectos pequeños con tiempos cortos de ejecución. El otro método elaborado en profundidad es *Crystal Orange,* apto para proyectos más exigentes de duración estimada en 2 años. Los demás aún se están desarrollando por lo tanto no tienen definidas políticas, ni procesos.

Para este estudio vamos a tomar en cuenta a *Crystal Clear* (CC), debido a que se dispone de mayor información y es el más usado.

- **Principios de *Crystal Clear***

Crystal Clear consiste en valores, técnicas y procesos, los cuales en conjunto aseguran la ejecución exitosa de proyecto de desarrollo de software. Se rige por seis valores o propiedades, que se describen a continuación:

1. **Entrega frecuente.** Consiste en entregar software a los clientes con frecuencia, no solamente en compilar el código. La frecuencia dependerá del proyecto, pero puede ser diaria, semanal o mensual.
2. **Comunicación osmótica.** Todos juntos en el mismo cuarto. Una variante especial es disponer en la sala de un experto diseñador sénior y discutir respecto del tema que se trate.

3. **Mejora reflexiva.** Tomarse un pequeño tiempo (unas pocas horas cada semana o una vez al mes) para pensar bien qué se está haciendo, cotejar notas, reflexionar, discutir.
4. **Seguridad personal.** Hablar con los compañeros cuando algo molesta dentro del grupo.
5. **Enfoque.** Saber lo que se está haciendo y tener la tranquilidad y el tiempo para hacerlo.
6. **Fácil acceso a usuarios expertos.** Tener alguna comunicación con expertos desarrolladores.
7. **Ambiente de desarrollo con Pruebas automáticas, Administración de la configuración e integración frecuente.** Poder dejar corriendo las pruebas de nuestro desarrollo hasta el final sin estar físicamente presente nos ahorra el tiempo que no tenemos, sin mencionar las ventajas de depuración inmediata y de probar el código indiscriminadamente. Todos los desarrolladores deberían ingresar el código en el que trabajan en un sistema de administración de la configuración, de manera que este se encargue de llevar el control de versiones, documentos, etc. y escribir una nota útil sobre el código cuando lo ingresan.

- **Técnicas de Colaboración en el Proceso.**

Para usar *Crystal Clear*, es necesario que exista un conocimiento amplio por parte de los miembros del equipo acerca de las técnicas y procesos a seguir,

ya que al ser esta una metodología ágil el desconocimiento de su aplicación podría entorpecer el desarrollo. Las técnicas usadas son:

a) **Entrevistas de proyectos.** Se suele entrevistar a más de un responsable para tener visiones más amplias acerca de los requerimientos.

b) **Talleres de reflexión.** El equipo debe detenerse treinta minutos o una hora para reflexionar sobre sus convenciones de trabajo, discutir inconvenientes, mejoras y planear siguiente período.

c) **Planeamiento *Blitz*.** Una técnica puede ser el Juego de Planeamiento de *XP*. En este juego, se ponen tarjetas indexadas en una mesa, con una historia de usuario o función visible en cada una. El grupo finge que no hay dependencias entre tarjetas, y las alinea en secuencias de desarrollo preferidas. Los programadores escriben en cada tarjeta el tiempo estimado para desarrollar cada función. El patrocinador del usuario escribe la secuencia de prioridades, teniendo en cuenta los tiempos referidos y el valor de negocio de cada función. Las tarjetas se agrupan en períodos de tres semanas llamados iteraciones que se agrupan en entregas, usualmente no más largas de tres meses.

d) ***Estimación Delphi* con estimaciones de pericia.** Al igual que en el *método Delphi*, se propone que se reúnan los expertos responsables y procedan como en un remate a estimar el tamaño del sistema, su

tiempo de ejecución, la fecha de las entregas según dependencias técnicas y de negocios y para equilibrar las entregas en paquetes de igual tamaño.

e) **Encuentros diarios de pie.** La palabra clave es *brevedad*, cinco a diez minutos como máximo. No se trata de discutir problemas, sino de identificarlos.

f) **Miniatura de procesos.** Una forma de presentar *Crystal Clear* puede consumir entre 90 minutos y un día. La idea es que la gente pueda *degustar* la nueva metodología.

g) ***Gráficos de quemado.*** Su nombre viene de los gráficos de quemado de calorías de los regímenes dietéticos; se usan también en *Scrum*. Se trata de una técnica de graficación para descubrir demoras y problemas tempranamente en el proceso, evitando que se descubra demasiado tarde cuando ya no se puede remediar. Para ello se hace una estimación del tiempo faltante para programar lo que resta al ritmo actual, lo cual sirve para tener dominio de proyectos en los cuales las prioridades cambian bruscamente y con frecuencia. Esta técnica se asocia con algunos recursos como la Lista Témpano, llamada así porque permite agregar ítems con alta prioridad en el tope de las listas de trabajos pendientes, esperando que los demás elementos se hundan bajo la línea de flotación; los elementos que están sobre la línea se entregarán en la iteración siguiente, los que

están por debajo en las posteriores. En otras Metodologías Ágiles la Lista Témpano no es otra cosa que un gráfico de retraso. Los gráficos de quemado ilustran la velocidad del proceso, analizando la diferencia entre las líneas proyectadas y efectivas de cada entrega.

h) ***Programación lado a lado.*** Mucha gente siente que la programación en pares de *XP* involucra una presión excesiva; la versión de *Crystal Clear* establece proximidad, pero cada quien se enfoca a su trabajo asignado, prestando un ojo a lo que hace su compañero, quien tiene su propia máquina. Esta es una ampliación de la Comunicación Osmótica al contexto de la programación.

- **Roles y Responsabilidades de *Crystal Clear***

En la metodología *Crystal Clear* cada miembro debe cumplir con una función específica, lo que permite que las personas se enfoquen en su parte. A continuación se describen las actividades que desempañan cada uno de los responsables de los ocho roles existentes:

Patrocinador. Produce la declaración de misión con prioridades de compromiso *tradeoff*. Consigue los recursos y define la totalidad del proyecto.

Usuario Experto. Junto con el experto en negocios produce la lista de actores objetivos y el archivo de casos de uso y requerimientos. Debe familiarizarse con el uso del sistema, sugerir atajos de teclado, modos de operación, información a visualizar simultáneamente, navegación.

Diseñador Principal. Produce la descripción arquitectónica. Se supone que debe ser al menos un profesional de nivel 3. En metodologías ágiles se definen tres niveles de experiencia:

- **Nivel 1.** es capaz de s*eguir los procedimientos*.
- **Nivel 2.** es capaz de *apartarse de los procedimientos específicos* y encontrar otros distintos.
- **Nivel 3.** es capaz de manejar con fluidez, mezclar e inventar procedimientos. El Diseñador Principal tiene roles de coordinador, arquitecto, mentor y programador más experto.

Diseñador Programador. Produce, junto con el diseñador principal, los borradores de pantallas, el modelo común de dominio, las notas y diagramas de diseño, el código fuente, el código de migración, las pruebas y el sistema empaquetado. Un programa en *CC* es *diseño y programa*; sus programadores son diseñadores programadores. En *CC* un diseñador que no programe no tiene cabida.

Experto en Negocios. Junto con el usuario experto produce la lista de actores objetivos y el archivo de casos de uso y requerimientos. Debe conocer las reglas y políticas del negocio.

Coordinador. Con la ayuda del equipo, produce el mapa de proyecto, el plan de entrega, el estado del proyecto, la lista de riesgos, el plan y estado de iteración y la agenda de visualización.

Verificador. Produce el reporte de errores y bugs. Puede ser un programador en tiempo parcial, o un equipo de varias personas.

Escritor. Produce el manual de usuario. El equipo como grupo es responsable de producir la estructura y convenciones del equipo y los resultados del taller de reflexión.

3.2.4 SCRUM

Scrum es una metodología de desarrollo ágil que Jeff Sutherland inventó en 1993. Jeff trabajado junto con Ken Schwaber formalizaron *Scrum*, lo fueron ampliando y mejorado para que pueda ser usado en empresas de *software* y fue de gran ayuda al momento de escribir el Manifiesto Ágil.

La palabra *Scrum*, procede de la terminología del juego de rugby, donde designa al acto de preparar el avance del equipo en unidad pasando la pelota a uno y otro jugador. Igual que el juego, *Scrum* es adaptativo, ágil, auto-organizante y con pocos tiempos muertos. (Schwaber, 2006)

Es una de las metodologías ágiles más usadas, aunque tenga algunos limitantes, es conocida por ser la metodología que usa *Google* para desarrollar sus proyectos.

Scrum por sus características no es válida para cualquier proyecto, ni para cualquier persona o equipo de personas, es más según muchos especialistas dicen que esta metodología es óptima para equipos de trabajo de hasta ocho personas, aunque hay empresas que han utilizado *Scrum* con éxito con equipos más grandes. Es una metodología válida para el 90% de los proyectos y empresas, pero no es válida en un 100%, aunque para ser sinceros, ninguna metodología existente lo es.

Al igual que otras metodologías ágiles, *Scrum* toma en cuenta los principios básicos del desarrollo ágil, es decir, desarrollar software con rapidez, calidad y reducción de costos, sin dejar de lado la agilidad y flexibilidad, para esto propone ciclos de desarrollo cortos en medida posible, reuniones periódicas que garanticen calidad y *feedbacks* que minimicen la probabilidad de errores. Esto no obedece a una tendencia o *moda*, sino estrictamente a las necesidades que realmente son demandadas.

Scrum no es ni la mejor metodología ni la única, pero si es una metodología que está haciéndose popular por la facilidad de implantación y por su agilidad en cuanto a cambios. Por otro lado podemos destacar que *Scrum* evita la *burocracia* y la generación documental, debido a que no exige documentar nada para iniciar un proyecto, sin que esto quiera decir que no se

pueda o no se deba documentar, sino que es opcional y queda a decisión de los miembros del equipo, situación que en otras metodologías es impensable.

La idea principal de *Scrum* es trabajar prácticamente desde el primer momento y tratar de sacar el mayor provecho al tiempo, obteniendo frutos del trabajo para que el cliente sea testigo de los avances y esté siempre satisfecho durante el tiempo de ejecución del proyecto, además que esto le brinda la confianza de que se está desarrollando un producto de calidad, porque conoce exactamente lo que se está haciendo y como se lo está haciendo.(Serrano, 2007)

Al ser *Scrum* una metodología ágil, está centrada en las personas, y sus interacciones, las cuales ahorran recursos, una de las fortalezas de *Scrum* es la realización periódica de reuniones, las cuales deben contar con la presencia de todos los involucrados. Por lo tanto lo que determina el éxito de un proyecto es el compromiso de todos quienes participan en el proceso.

- **Roles y Responsabilidades de *SCRUM***

Existen varios roles que los miembros de un equipo deben desempeñar dentro de *Scrum*, y son los siguientes:

- ***Product Owner.*** Conoce y marca las prioridades del proyecto o producto.

- ***Scrum Master.*** Es la persona que asegura el seguimiento de la metodología guiando las reuniones y ayudando al equipo ante cualquier problema que pueda suscitarse. Su responsabilidad, entre otras, es ser mediador ante las presiones externas.

- ***Scrum Team.*** Son las personas responsables de implementar las funcionalidades descritas por el *Product Owner*.

- **Usuarios o Clientes.** Son los beneficiarios finales del producto, y son quienes en base al progreso del proyecto puede aportar ideas, sugerencias, necesidades, etc.

- **Artefactos de *SCRUM***

A continuación se describirán los elementos que intervienen en el proceso de desarrollo de *Scrum* y las funciones que cumplen dentro del marco de desarrollo:

- ***Product Backlog*.** Es una lista de los requerimientos del cliente, los mismos que se documentan para en base a ellos construir el software. Aquí el *Product Owner* describe quién realiza cada operación, que es lo que realiza y cómo lo realiza (*who, what, why*). Esta lista debe estar correctamente

organizada, es decir los requerimientos más importantes al inicio y los de menor importancia al final.

- ***Sprint Backlog.*** Es una lista de tareas para convertir una *Product Backlog* en un *Sprint*, es decir en una iteración más del producto entregable.

- ***Burndown.*** Es una medida de retraso en el tiempo restante.

- ***Release Burndown.*** Son los requerimientos del *Producto Backlog* que aún no han sido atendidos antes de cada lanzamiento.

- ***Sprint Burndown.*** Son las tareas del *Sprint Backlog* que aún no han sido atendidas en el momento que inicia un *Sprint.*

- ***Daily Standup Meeting.*** Es esencial organizar reuniones de *Scrum* diarias, estas reuniones son llamadas *standup* (de pie en español), porque hacen referencia a que son reuniones rápidas, máximo de 15 minutos, para las cuales no es necesario sentarse. Tiene como objetivo informar a todos los involucrados acerca del progreso de diseños, desarrollo, pruebas y despliegue según el plan, durante el diseño de las iteraciones normalmente se integran diseñadores, desarrolladores así como equipos de prueba y desarrollo. Además estas reuniones sirven para realizar monitoreos generales del estado del proyecto.

Figura 9. Modelo de procesos de *SCRUM*.

Fuente: Introduction to *Scrum* in just 8 minutes! (http://www.youtube.com/watch?v=_QfFu-YQfK4)

- ***Sprint Planning Meeting*.** Es una reunión que tiene por objetivo planificar el *Sprint* a partir del *Product Backlog*, es decir mover las tareas del *Product Backlog* al *Sprint Backlog*. En esta reunión participan: el *Product Owner* quien prioriza las tareas, el *Scrum Master* quien coordina las acciones y el *Scrum Team* quienes ejecutan las tareas.

- ***Sprint Goal.*** Es un pequeño documento resultante del *Sprint Planning Meeting*, donde describe las metas que se intentarán alcanzar en el próximo *Sprint.*

- ***Sprint Review.*** Es una reunión de máximo 2 horas, que se realiza al finalizar un *Sprint*, aquí se le muestra al cliente una versión semifuncional del producto con los avances realizados. En esta reunión participan el *Product Owner*, el *Scrum Master*, el *Scrum Team* y los usuarios o clientes y se definen los errores, aciertos y cambios necesarios.

- ***Sprint Restrospective.*** En esta etapa el *Product Owner* revisará junto con el equipo los objetivos marcados inicialmente en el *Sprint Backlog* concluido, se aplicarán los cambios y ajustes si son necesarios, y se marcará los aspectos positivos (para repetirlos) y los aspectos negativos (para evitar que se repitan) del *Sprint.*

- **Técnicas de Colaboración en el Proceso.**

Si bien es cierto que *Scrum* es una metodología sencilla de usar y fácil de asimilar, es necesario que se adopten ciertas prácticas que aseguren el éxito del proyecto, entre las que más destacan tenemos:

Uso de herramientas de colaboración tales como *Wiki.* Es un práctica general usar herramientas de colaboración como *Wiki* para documentar y compartir historias de usuarios, diseños, documentos, *diagramas UML*, etc., esto facilita el intercambio de información almacenada en un lugar.

Colaboración. La colaboración incrementa la comunicación entre los equipos, resultando en una toma de decisiones más rápida.

Uso de ayudas audiovisuales. En caso de equipos esparcidos en varias localidades, es una buena ayuda utilizar este tipo de herramientas, como cámaras, grabaciones, presentaciones o envío de información.

Retroalimentación y toma de decisiones. La participación de todas las partes interesadas durante el ciclo de vida asegura la correcta toma de decisiones en el tiempo establecido y también facilita una temprana retroalimentación permitiendo hacer el ciclo de vida realmente ágil.

Elementos reutilizables. Es una buena práctica la utilización de elementos reutilizables como plantillas de diseño, desarrollo y pruebas, *plantillas Wiki*, plantillas de progreso de desarrollo, plantillas en general.

Grupos de revisión. Al final de cada iteración es una buena práctica tener un grupo de revisión rápido, para que no exista ninguna discrepancia que se haya perdido en alguna parte del proyecto.

Mejora continua y lecciones aprendidas. Es recomendable realizar sesiones regulares que sirven de retroalimentación al final de cada iteración, para observar qué funcionó de manera correcta y qué no, con la finalidad de que en el futuro se trate de evitar cometer los mismos errores. Esto fomenta una mejora continua en el proceso, a través de una entrega ágil.

3.3 Métodos que colaboran al desarrollo ágil de aplicaciones.

3.3.1 Agile Modeling

Agile Modeling (*AM*) fue propuesto por Scott Ambler no como un método ágil, sino como complemento de otras metodologías, sean éstas ágiles o convencionales. En el caso de *XP* los practicantes podrían definir mejor los procesos de modelado que en ellos faltan, y en el caso de *AUP* el modelado ágil permite hacer más ligeros los procesos que ya usan. *AM* es una estrategia de modelado (de clases, de datos, de procesos) pensada para contrarrestar la sospecha de que los métodos ágiles no modelan y no documentan. Se lo podría definir como un proceso de software basado en prácticas cuyo objetivo es orientar el modelado de una manera efectiva y ágil. (Metologías ágiles, 2006)

Los principales objetivos de *AM* son:

- Definir y mostrar de qué manera se debe poner en práctica una colección de valores, principios y prácticas que conducen al modelado de peso ligero.
- Enfrentar el problema de la aplicación de técnicas de modelado en procesos de desarrollo ágiles.
- Enfrentar el problema de la aplicación de las técnicas de modelado independientemente del proceso de software que se utilice.

Los valores de *AM* incluyen a los de *XP*: comunicación, simplicidad, *feedback* y coraje, añadiendo humildad. Una de las mejores caracterizaciones de los principios subyacentes a *AM* está en la definición de sus alcances:

1. Es una actitud, no un proceso prescriptivo. Comprende una colección de valores a los que los modeladores ágiles se adhieren, principios en los que creen y prácticas que aplican. Describe un estilo de modelado; no es un recetario de cocina.
2. Es suplemento de otros métodos. El primer foco es el modelado y el segundo la documentación.
3. Es una tarea de conjunto de los participantes. No hay *yo* en *AM.*
4. La prioridad es la efectividad. *AM* ayuda a crear un modelo o proceso cuando se tiene un propósito claro y se comprenden las necesidades de la audiencia; contribuye a aplicar los artefactos correctos para afrontar la situación inmediata y a crear los modelos más simples que sea posible.
5. Es algo que funciona en la práctica, no una teoría académica. Las prácticas han sido discutidas desde 2001 en comunidad.
6. Es para el programador promedio, pero no reemplaza a la gente competente.
7. No es un ataque a la documentación. La documentación debe ser mínima y relevante.
8. No es un ataque a las herramientas *CASE*.

Los principios de *AM* especificados por Scott Ambler (Ambler. 2006) incluyen:

- **Presuponer simplicidad.** La solución más simple es la mejor.
- **El contenido es más importante que la representación.** Pueden ser notas, pizarras o documentos formales. Lo que importa no es el soporte físico o la técnica de representación, sino el contenido.
- **Abrazar el cambio.** Aceptar que los requerimientos cambian.
- **Habilitar el esfuerzo siguiente.** Garantizar que el sistema es suficientemente robusto para admitir mejoras ulteriores; debe ser un objetivo, pero no el primordial.
- **Todo el mundo puede aprender de algún otro.** Reconocer que nunca se domina realmente algo.
- **Cambio incremental.** No esperar hacerlo bien la primera vez.
- **Conocer tus modelos.** Saber cuáles son sus fuerzas y sus debilidades.
- **Adaptación local.** Producir sólo el modelo que resulte suficiente para el propósito.
- **Maximizar la inversión del cliente.** Modelar con un propósito. Si no se puede identificar para qué se está haciendo algo ¿para qué molestarse?
- **Modelos múltiples.** Múltiples paradigmas en convivencia, según se requiera.
- **Comunicación abierta y honesta.**
- **Trabajo de calidad.**

- **Retroalimentación rápida.** No esperar que sea demasiado tarde.
- **El software es el objetivo primario.** Debe ser de alta calidad y coincidir con lo que el usuario espera.
- **Viajar ligero de equipaje.** No crear más modelos de los necesarios.
- **Trabajar con los instintos de la gente.**

Lo más concreto de *AM* es su rico conjunto de prácticas, cada una de las cuales se asocia a lineamientos decididamente narrativos, articulados con minuciosidad, pero muy lejos de los rigores cuantitativos:

1. Colaboración activa de los participantes.
2. Aplicación de estándares de modelado.
3. Aplicación adecuada de patrones de modelado.
4. Aplicación de los artefactos correctos.
5. Propiedad colectiva de todos los elementos.
6. Considerar la verificabilidad.
7. Crear diversos modelos en paralelo.
8. Crear contenido simple.
9. Diseñar modelos de manera simple.
10. Descartar los modelos temporarios.
11. Exhibir públicamente los modelos.
12. Formalizar modelos de contrato.
13. Iterar sobre otro artefacto.
14. Modelo en incrementos pequeños.

15. Modelar para comunicar.
16. Modelar para comprender.
17. Modelar con otros.
18. Poner a prueba con código.
19. Reutilizar los recursos existentes.
20. Actualizar sólo cuando duele.
21. Utilizar las herramientas más simples (*CASE*, pizarras, tarjetas, *post-its*).

Como *Agile Modeling* se debe usar como complemento de otras metodologías, nada se especifica sobre métodos de desarrollo, tamaño del equipo, roles, duración de iteraciones, trabajo distribuido y criticidad, todo lo cual dependerá del método que se utilice.

3.3.2 Programación Pragmática (*Pragmatic Programming*)

El título en realidad no es totalmente cierto, ya que no hay un método de programación pragmática, sino que existe un conjunto muy interesante de las mejores prácticas de programación publicadas en el libro The Pragmatic Programmer, pero por practicidad se lo llama *Programación Pragmática* (*PP*).

La *PP* no tiene proceso, fases, roles distintos o productos de trabajo sin embargo cubre, la mayoría de las mejores prácticas de programación. Hay un

total de 70 recomendaciones que se enfocan en los problemas del día a día, pero la filosofía por detrás de esto es que son universales y pueden ser aplicadas a cualquier fase del desarrollo de software. La filosofía está definida en 6 puntos:

1. Tome las responsabilidades para usted, piense en soluciones en lugar de estar pensando en excusas.
2. No diseñe o codifique mal, arregle las inconsistencias o planee arreglarlas tan pronto como sea posible.
3. Tome un rol activo para introducir cambios donde usted vea que son necesarios.
4. Haga software que satisfaga a su cliente, pero sepa cuándo parar.
5. Constantemente amplíe su conocimiento.
6. Mejore sus habilidades de comunicación.

Desde un punto de vista ágil, la mayoría de las prácticas más interesantes se enfocan sobre el desarrollo iterativo, incremental, *testing* riguroso y diseño centrado en el usuario. El enfoque tiene un punto de vista muy práctico, y la mayoría de las prácticas son ilustradas con ejemplos positivos y negativos, a menudo complementados con preguntas y trocitos de código. Es un esfuerzo considerable explicar cómo diseñar e implementar software tal que pueda resistir cambios. Una de las soluciones que se plantean para mantener software a través de los cambios es la refactorización.

El enfoque de la Programación Pragmática respecto del *testing* consiste en testear el código que está siendo implementado sobre el código real, y todos los testeos deben ser hechos en forma automática. La idea es que si cada bug

corregido no es agregado dentro de la biblioteca del *test* y si los *tests* de regresión no se corren periódicamente, el tiempo y el esfuerzo se gastan en encontrar los mismos *bugs* repetidamente, y los efectos adversos de cambios en el código no pueden detectarse suficientemente temprano. La automatización se encuentra en la *PP* en algunas otras tareas también. De hecho llega a sugerir: *No use procedimientos manuales*. Por ejemplo, en la creación de la primera documentación de código fuente y en la creación de código de las definiciones de la base de datos.

Recomienda conservar las especificaciones a un nivel razonable de detalle. La *PP* demuestra prácticas de software simples, responsables y disciplinadas. Las prácticas sugeridas en *PP* son escritas desde el punto de vista de un programador, independientemente de los métodos o procesos que se utilicen, para evitar errores típicos en la codificación y en el diseño y errores de comunicación en el grupo de trabajo.

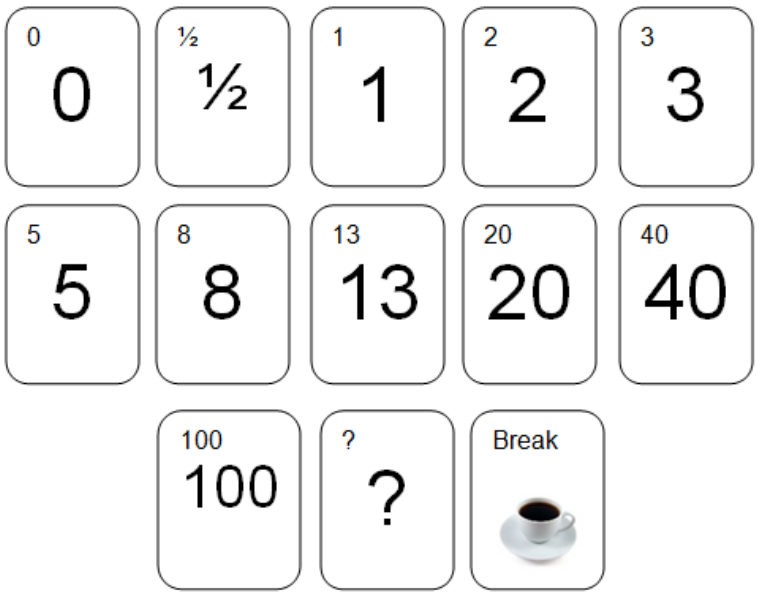
3.3.3 Planning Poker

Esta es una técnica para calcular una estimación basada en el consenso, se utiliza en mayor parte para estimar el esfuerzo o el tamaño relativo de las tareas de desarrollo de software. Es una variación del método *Wideband Delphi* y es utilizado comúnmente en el desarrollo ágil de software. El

método fue descrito por primera vez por James Grenning en 2002 y más tarde fue popularizado por Mike Cohn.

El *planning poker* consiste en una lista de requerimientos que se deben desarrollar y una baraja de cartas.

Figura 10. Cartas para jugar Planning Poker.



Fuente: Time estimating using planning poker (http://agileplusplus.blogspot.com/2008/05/planning-poker-time-estimations.html)

La lista de requerimientos, por lo general es una lista de historias de usuario, donde se describe en detalle las funcionalidades que necesita el software a ser desarrollado. Las cartas en el mazo están numeradas. Un mazo típico contiene tarjetas mostrando la *secuencia de Fibonacci* incluyendo un cero: 0, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89. Otros mazos utilizan progresiones similares.

La razón de utilizar la *secuencia de Fibonacci* es reflejar la incertidumbre inherente en la estimación. Existe en el mercado un mazo que utiliza la siguiente secuencia: 0, ½, 1, 2, 3, 5, 8, 13, 20, 40, 100, y, opcionalmente, una tarjeta con "?" (Inseguro) y una taza de café (necesito un descanso).

El procedimiento de planeación, se realiza de la siguiente manera:

- Se lleva a cabo una reunión de estimación donde a cada estimador se le da un conjunto de cartas (mazo). Un moderador, que no jugará, preside la reunión, apoyado y asesorado por el Gestor del Proyecto.
- El desarrollador con más conocimiento de una determinada característica proporciona una breve introducción sobre la misma. El equipo tiene la oportunidad de hacer preguntas y discutir para aclarar los supuestos y riesgos. Un resumen de la discusión se registra por el Gestor del Proyecto.
- Cada persona coloca una tarjeta boca abajo que representa su estimación. Las unidades utilizadas pueden ser variadas y definidas previamente. Pueden ser días de duración, días ideales (8 horas de trabajo) o puntos de la historia (referencias anteriores de duración de un requerimiento parecido). Durante el debate, los números no debe ser mencionados en absoluto.
- Todo el mundo muestra sus tarjetas de forma simultánea.
- Si las estimaciones están todas dentro de un rango aceptable, se calcula un promedio y ese será el tiempo estimado.

- Si existen estimaciones muy altas o muy bajas, se les da a las personas que las presentaron un tiempo para ofrecer su justificación para tal estimación y la discusión continúa.
- Se repite el proceso de cálculo hasta que se alcance un consenso.
- Se puede utilizar un reloj para asegurar que el debate sea estructurado, si la discusión se está estancando el moderador o el Gestor del Proyecto podrán en cualquier punto detener el reloj, entonces toda discusión debe cesar y se jugará otra ronda de *póquer*.
- Las cartas están numeradas de esta forma para explicar el hecho de que, cuando una estimación es mayor, existe mayor incertidumbre. Así, si un desarrollador quiere jugar un 6 se ve obligado a reconsiderar y aceptar que parte de la incertidumbre percibida no existe y jugar un 5, o por el contrario aceptar una estimación más conservadora de la incertidumbre y jugar un 8.

CAPÍTULO IV

CAPITULO IV

ANÁLISIS COMPARATIVO

4.1 Evaluación de las Metodologías.

4.1.1 Fortalezas y Debilidades de *XP*

Si bien es cierto que *XP* es uno de los métodos más simples de usar y más fáciles de aplicar, como todos tiene algunos puntos a favor y otros en contra, que deben ser evaluados antes de emitir un juicio crítico acerca de la utilidad o no. A continuación se analizaran y sus debilidades:

Fortalezas

- ✓ Es un método que se centra en las personas y no en el proceso, esto es sumamente beneficioso debido a que los resultados esperados están sujetos a la falla humana y por lo tanto es más fácil de realizar estimaciones exactas.

- ✓ Se ajusta con facilidad a proyectos cuyos requisitos son ambiguos, cambian rápidamente o son de alto riesgo, esta es una fortaleza dado que los proyectos de software en la

actualidad cambian de requerimientos muy rápido y muchas veces.

- ✓ El trabajo en equipo que propone *XP* es muy beneficioso, ya que la comunicación permite la ágil solución de problemas evitando el estancamiento en una labor, cosa que puede demorar todo el proceso.

- ✓ El continuo *feedback* o retroalimentación por parte del usuario o cliente, minimiza los errores y garantiza que las funcionalidades están acorde a los requerimientos.

- ✓ Las pruebas continuas y la refactorización de código son una gran fortaleza ya que van depurando el software a medida que avanza, esto hace que se avance más rápido y minimiza los errores al final, que son muy difíciles de manejar.

Debilidades

- ✓ Es un proceso demasiado informal debido a que no tiene una arquitectura del proyecto, y su desarrollo se basa solo en historias de usuario (*story cards*), lo que hace que a veces los requerimientos de funcionalidades no estén bien definidos para los miembros del equipo y provoquen confusión.

- ✓ Se necesita de un cliente in situ, situación que en la vida real es imposible o muy difícil de cumplir dado que el cliente no está disponible la mayoría del tiempo. La falta de un cliente disponible a todo momento puede desequilibrar el normal funcionamiento de este método ocasionando retrasos en la entrega o peor aún el fracaso del proyecto.

- ✓ Este método tiene una escasa documentación, ya que la mayoría de veces se maneja la comunicación únicamente de forma verbal, esta situación hace difícil la posterior modificación del software.

4.1.2 Fortalezas y Debilidades de *AUP*

Siendo *AUP* (*Agile Unified Process*) perteneciente a la familia de Procesos Unificados (*UP*), es un método robusto y que maneja de una forma muy metódica los procesos que se deben seguir, sin embargo esto no garantiza su eficacia al 100%, a continuación se detallan sus fortalezas y debilidades:

Fortalezas

- ✓ Establece una base arquitectónica desde el inicio del proyecto, que ayuda a valorar requisitos, hipótesis y gestionar riesgos, de una forma fácil.

- ✓ Toma muy en cuenta los elementos de alto riesgo y les da prioridad, lo que garantiza que el proceso de desarrollo de software tenga el mínimo de inconvenientes.

- ✓ El desarrollador tiene la libertad de escoger entre las diferentes disciplinas disponibles.

- ✓ Este método maneja un concepto de entorno que facilita la aplicación del mismo.
- ✓ La aplicación de documentación hace que el software desarrollado sea fácilmente escalable.

Debilidades

- ✓ No se interesa por las interacciones entre los miembros del equipo, lo que puede traducirse en problemas de comunicación inmediata, la no socialización de problemas e incluso la deserción de miembros del equipo.

- ✓ Algunos procedimientos, como la documentación, son demasiado engorrosos, lo que le resta agilidad, sobre todo en proyectos pequeños.

4.1.3 Fortalezas y Debilidades de *CRYSTAL*

Crystal es una metodología escalable, es decir, se acopla con facilidad al número de miembros del equipo de desarrollo, esto le permite tener un amplio espectro de acción. Como se explicó en su momento, para este trabajo se tomará en cuenta la metodología *Crystal Clear* y a continuación analizaremos sus fortalezas y debilidades:

Fortalezas

- ✓ Tiene una orientación humana, es decir que más personas serán capaces de seguir el proceso de *Crystal* dado que es fácil de asimilarlo.

- ✓ La comunicación que plantea este método está encaminada a crear un ambiente de trabajo tranquilo que mejorará el desempeño de los miembros del equipo y propiciará el compañerismo, situación que beneficiará el trabajo de todo el equipo.

- ✓ Establece una variedad de técnicas que sirven de soporte al proceso de desarrollo de software, entre otras tenemos: entrevistas a los usuarios, *planeamiento de Blitz*, *estimación Delphi*, talleres de reflexión, encuentros diarios en pie, programación lado a lado. Al estar estas técnicas descritas

dentro del método se garantiza que el proceso de desarrollo de software será exitoso.

Debilidades

- ✓ Si el número de miembros aumenta se deberá escalar en la clasificación de colores de *Crystal*, cada color establece políticas y métodos nuevos, por lo tanto si el equipo crece el proceso puede verse afectado por el cambio de las reglas de juego.

- ✓ La existencia de roles específicos limita las labores de los miembros del equipo, es decir no se aprovecha el máximo de sus conocimientos, (esto podría verse como un desperdicio de recursos). Además el hecho de cumplir una función específica limita su campo de visión acerca del proyecto.

4.1.4 Fortalezas y Debilidades de *SCRUM*

SCRUM es hoy por hoy el método más popular, y está siendo usado en empresas de toda índole, incluso en aquellas que no se dedican al desarrollo de software, si aunque parezca inverosímil, este método es tan flexible y eficaz que está siendo usado para administrar procesos de variada naturaleza.

Google usa *SCRUM* para administrar sus proyectos de desarrollo de software, es por eso quizá que tiene tantos adeptos, y es que *Google*

es una empresa seria que ha logrado ubicar en el mercado varios productos de software con éxito.

Pero muy aparte de todos los elogios que *SCRUM* pueda recibir, no es la panacea de los desarrolladores, ya que al igual que todas las metodologías existentes, tiene sus limitaciones y defectos, a continuación se realiza un análisis sobre eso:

Fortalezas

- ✓ Sirve para equipos de trabajo relativamente pequeños, aunque existen empresas que lo han aplicado con equipos de 90 personas, esto denota que es escalable.

- ✓ Es de fácil implantación además es muy ágil en la gestión de cambios, evita la burocracia y deja la elaboración de documentación a decisión del equipo, en resumen es ágil en todo sentido.

- ✓ Se centra en las personas y sus interacciones, lo cual ahorra recursos. Las reuniones periódicas que propone este método garantizan el compromiso de todos los participantes. Además el esquema de auto organización que maneja es bastante eficaz.

- ✓ Existen en el mercado una gran variedad de herramientas *CASE* que facilitan la aplicación, gestión y control de avances de un proyecto que use *SCRUM*, esta puede ser en gran medida la razón por la cual es el más popular de los métodos ágiles.

Debilidades

- ✓ No especifica las técnicas de soporte para el proceso de desarrollo, es decir, que deja abierta una amplia variedad de posibilidades, lo que genera retrasos en el avance de todo el proyecto.

- ✓ Puede tomar algún tiempo definir las reglas de negocio dado que el *Product Backlog* (requerimientos) puede ser modificado por el cliente en cada iteración.

- ✓ La falta de documentación formal hace difícil la escalabilidad del producto de software o la modificación a posteriori del mismo.

4.2 Tablas Comparativas

Una vez que se ha realizado un análisis preliminar de las fortalezas y debilidades de cada una de las metodologías estudiadas en el presente documento, es momento

entonces de iniciar un análisis más a fondo, de cada una de ellas, donde se podrá tener una visión más amplia de éstas.

Para esta labor se utilizarán tablas comparativas, que reflejan de una forma directa y en resumen los aspectos más relevantes a comparar entre las distintas metodologías.

La incógnita que surge ahora es: que parámetros son susceptibles de comparación y cuáles no, para esto realizaremos un análisis retrospectivo hacia el inicio del capítulo 3, donde se describe qué *requisitos* necesita una metodología para que sea considerada como tal. Por lo tanto se tomó como base esas características para realizar un análisis adecuado.

Además de estas características se tomó en cuenta ciertos parámetros que de acuerdo al criterio de expertos, influyen mucho en el entorno del proyecto de desarrollo de software, entonces se llegó a la conclusión de hacer seis comparaciones que permitirán visualizar en perspectiva a las metodologías:

- Comparativa del ciclo de vida.
- Comparativa del tipo de proceso. (ajustable o prescriptivo)
- Comparativa de tamaño del equipo.
- Comparativa de existencia de herramientas colaborativas dentro de la descripción de la Metodología.
- Comparativa de requisitos iniciales.
- Comparativa del Estado Actual de la Metodología.

4.2.1 Comparativa del Ciclo de vida.

De acuerdo a las fases de un ciclo de vida estándar de un proceso de desarrollo de Software, vamos a realizar un análisis acerca de cómo cada una de las metodologías cumple o no con estas etapas y con qué rigurosidad lo hace.

En un análisis preliminar podemos constatar, que las metodologías ágiles por su naturaleza se limitan a cumplir estrictamente con las etapas que son concernientes al proceso de desarrollo en sí, dejando de lado las labores de mantenimiento y de pruebas con el cliente.

Se pudo observar además que algunas metodologías incluyen dentro de sus procesos las etapas del ciclo vida de una manera bastante explícita, mientras otras solo hacen referencia a dichas etapas, se puede considerar entonces que la inclusión de etapas del ciclo de vida dentro del proceso es una ventaja para el desarrollo de software.

También se pudo constatar que las diferentes metodología hacen énfasis en diferentes etapas del ciclo, por ejemplo *Crystal Clear* se centra más en la etapa de diseño, codificación y pruebas, mientras que *SCRUM* no hace mucho énfasis en ello. Esto deja la puerta abierta para integrar varias metodologías y de esta forma obtener un mejor resultado.

El siguiente cuadro muestra como cada una de las metodologías incluye entre sus procesos fases del ciclo de vida, para esto existen 3 equivalencias:

- ✓ define la fase de una manera explícita dentro del proceso.
- o define la fase de una manera trivial dentro del proceso,
- \- No define la fase de ninguna manera.

Tabla Comparativa 1. Comparativa del Ciclo de vida.

Metodología	Especificación de Requisitos	Diseño	Codificación	Pruebas Unitarias	Pruebas de Integración	Pruebas del Sistema	Pruebas de Aplicación	Mantenimiento
Extremme Programming	✓	✓	✓	✓	✓	✓	✓	-
Agile Unified Process	✓	✓	✓	✓	○	✓	-	-
Crystal Clear	-	✓	✓	✓	✓	-	-	-
SCRUM	✓	○	○	○	✓	-	-	-

Elaborado por: Verónica Noriega.

Al término de esta comparativa se puede observar que las metodologías ágiles, han tomado el camino de la simplicidad omitiendo algunas etapas del ciclo de vida, no solo para ahorrar tiempo sino también para dar énfasis a los procesos más críticos dentro del entorno de desarrollo.

4.2.2 Comparativa del tipo de proceso (ajustable o prescriptivo).

Cuando se habla de metodologías ágiles es difícil asociar el término prescriptivo, tan relacionado a las metodologías tradicionales. Cuando se dice que una metodología ofrece una guía prescriptiva, se está declarando que las prácticas, métodos y procesos que enuncia la metodología son estrictas y deben seguirse tal y como están especificadas.

Este es un parámetro puede ser determinante a la hora de decidir la adopción de una metodología u otra. Una guía prescriptiva asegura resultados a nivel de proyecto en cambio una guía no prescriptiva esta asegurando resultados a nivel individual y de proyecto, tan solo si se utiliza como si fuese prescriptiva.

Obviamente la no prescriptiva es más ajustable a diferentes tipos de proyectos, pero es necesario tener experiencia, esto quiere decir que la metodología funcionará solo si se ejecuta completamente, las diferentes combinaciones que se hagan con otros métodos, corren a cuenta y riesgo de los miembros del equipo, nadie puede asegurar que vayan a funcionar.

Una vez que se conoce en lo que consiste cada uno de estos dos tipos de proceso, procedemos a realizar la comparación.

Tabla Comparativa 2. Comparativa del tipo de proceso.

Metodología	Guía Prescriptiva
Extremme Programming	No
Agile Unified Process	No
Crystal Clear	Si
SCRUM	No

Elaborado por: Verónica Noriega.

Como conclusión de esta comparativa podemos decir que los procesos entre menos prescriptivos son, son más fáciles de adaptar, esto se traduce en una ventaja indiscutible a la hora de usar una metodología.

4.2.3 Comparativa de tamaño del equipo.

Aunque parecería irrelevante el número de miembros de un equipo para usar una metodología, no es así, pues dependiendo del tamaño del proyecto se necesitará de más o menos personas.

Este parámetro también puede marcar el éxito del proyecto, ya que al utilizar una metodología que no se ajusta al tamaño del equipo podrían presentarse problemas de coordinación y asignación de tareas, además que se corre el riesgo de sobrecargar a los miembros del equipo situación que puede derivar en un bajo rendimiento. Algunas metodologías contemplan la posibilidad de

que existan varios equipos dentro de un mismo proyecto, esto también puede marcar diferencias a la hora del desempeño de los miembros de los equipos.

Tabla Comparativa 3. Comparativa del tamaño del equipo.

Metodología	Número de equipos	Mínimo de Integrantes por Equipo	Máximo de Integrantes por Equipo
Extremme Programming	solo 1 por proyecto	3	16
Agile Unified Process	De 1 a 4 por proyecto	2	8
Crystal Clear	de 1 a 3 por proyecto	1	8
SCRUM	hasta 4, si es necesario más	5	9

Elaborado por: Verónica Noriega.

4.2.4 Comparativa de existencia de herramientas colaborativas dentro de la descripción de la Metodología.

Las metodologías ágiles en sus descripciones, especifican los procesos a seguir, los miembros del equipo que se deben encargar y las actividades involucradas, pero algo que muchas olvidan, es la especificación de herramientas que faciliten las tareas.

El término herramientas se refiere a la técnicas que se pueden usar para colaborar en la realización de determinadas tareas dentro de los diferentes procesos que plantea una metodología, estas herramientas muchas veces

pueden ser recursos informáticos, como *herramientas CASE* o también pueden ser ayudas investigativas como encuestas y entrevistas.

Las *Metodologías Monumentales* acostumbraban a describir paso a paso cada una de las tareas que se debían llevar cabo, y detallaban también que herramientas debían ser usadas, es tanto así que algunas describían incluso una nomenclatura propia para llenar formularios de control, una forma específica de medir e incluso se valían de técnicas como *UML* para realizar estas actividades. Estas prácticas justamente volvían al proceso engorroso, lento e inadaptable.

En cambio las metodologías ágiles, no se toman la molestia de describir a profundidad las herramientas, dejando así abierta la posibilidad para que cada uno de los participantes del proyecto use la técnica que mejor considere para realizar su trabajo, esto es de cierta forma es un gran paso, ya que se puede escoger de una infinidad de herramientas disponibles, la otra ventaja es que este esquema permite usar las herramientas que son estrictamente necesarias, ni más, ni menos; situación que permite aprovechar el tiempo y los recursos al máximo porque no se deben cumplir con tareas que son mera formalidad.

Algunos detractores exponen que el hecho de no existir técnicas determinadas, reduce la productividad, puesto que dicen que se gasta más tiempo en escogerlas además argumentan que se ve comprometida la calidad del software.

Entre las tareas que necesitan de técnicas de colaboración para su ejecución tenemos:

- ✓ tareas de estimación de costos.
- ✓ tareas de estimación de tiempos.
- ✓ tareas de recolección de datos.
- ✓ tareas de modelado.
- ✓ tareas de diseño.
- ✓ tareas de programación y codificación.
- ✓ tareas de prueba.
- ✓ tareas de control de avances del proyecto.

A continuación se elabora un cuadro donde especifica si la metodología describe o no las herramientas que se deben usar en cada una de estas etapas:

Tabla Comparativa 4. Comparativa de la existencia de herramientas colaborativas.

Tareas	*Extremme Programming*	*Agile Unified Process*	*Crystal Clear*	*SCRUM*
Tareas de estimación de costos.	X	X	X	X
Tareas de estimación de tiempos.	✓	X	✓	X
Tareas de recolección de datos.	✓	X	✓	✓
Tareas de modelado.	X	X	X	X
Tareas de diseño.	X	X	X	X
Tareas de programación y codificación.	✓	X	✓	X
Tareas de prueba.	X	X	X	X
Tareas de control del proyecto.	✓	X	✓	✓

Elaborado por: Verónica Noriega.

En la Tabla anterior se puede efectivamente constatar que la mayoría de tareas no están delimitadas por una herramienta específica para ser llevada a cabo, inclusive podemos ver el caso de *AUP* que no define ninguna herramienta, dado que dentro de sus principios está estipulado que podrá usarse la herramienta que se crea conveniente o que sea más fácil de usar para los desarrolladores.

Esta *libertad* que brindan las metodologías ágiles las han convertido en una opción a tomar se cuenta, porque hacen que el proceso sea flexible y adaptable a cualquier situación.

4.2.5 Comparativa de requisitos iniciales.

Para la mayoría de la metodologías *monumentales* la definición de requisitos al inicio del proyecto era primordial, ya que el resto del proceso se basaba en ello, y dadas estas circunstancias el proceso de análisis de requerimientos debía llevarse de una forma muy meticulosa para evitar errores y tratando en lo posible de no olvidar nada, porque una vez que se empezaba con el desarrollo, era muy difícil volver atrás.

Sin embargo no era imposible agregar funcionalidades y realizar cambios en el software durante el proceso, pero esto era muy complicado de llevar a cabo, ya que en la mayoría de casos se debía esperar a un raleases funcional para poder realizar cambios, esto impedía a los desarrolladores hacer un mejoramiento continuo del software que significaba una inadaptabilidad, que

muchas veces dejaba al cliente con un sinsabor, dado que muy rara vez se permitía integrar nuevas funcionalidades.

Este problema limitaba mucho a los desarrolladores, por eso fue una de las cosas que se tomó en cuenta para que las metodologías ágiles mejoraran, por esta razón uno de los objetivos del desarrollo ágil fue brindar una mayor flexibilidad en este aspecto.

Sin embargo esto no quiere decir que no se necesite un análisis inicial de requerimientos, pero hay metodologías que solicitan a los miembros del equipo una explicación más metódica, mientras otros no. A continuación en la tabla se podrá observar cuales son las exigencias de cada método:

Tabla Comparativa 5. Comparativa de requisitos iniciales.

Metodología	Exigencia	Nivel de rigurosidad
Extremme Programming	Solicita requisitos iniciales que permitan delimitar las acciones a tomar, esto se realiza con la ayuda de las historias de usuario. Es flexible al cambio.	Baja
Agile Unified Process	Tiene una amplia capacidad de adaptarse, sin embargo por su política de manejo de riesgos solicita en un principio los requisitos críticos para darles prioridad.	Media
Crystal Clear	Es muy flexible y menciona que puede gestionar los cambios sin problemas, aunque si plantea un manejo de requisitos iniciales para empezar con el proceso.	Baja
SCRUM	Recoge todos los datos que puede al inicio, sin que esto signifique que no permita cambios después, es más en cada sprint acepta nuevos requerimientos.	Media

Elaborado por: Verónica Noriega.

Es necesario tener en cuenta que rigurosidad en los datos iniciales exige el método seleccionado para ser usado, dado que de esto depende la forma en que se recojan los datos.

También se debe tomar en cuenta según la metodología que datos son los más relevantes para poder iniciar el proceso de desarrollo.

4.2.6 Comparativa del Estado Actual de la Metodología.

En este punto, las características que conciernen en sí a la metodología han dejado de ser lo único importante para escoger una, es también indispensable además conocer cómo se maneja la metodología, es decir, si tiene una documentación adecuada que permita a los miembros del equipo saber que, quién, cómo y cuándo deben hacer.

Al ser las metodologías ágiles, una opción relativamente nueva, hay algunas que aún no tienen la suficiente documentación, y por lo tanto no son muy confiables para usar. Por lo tanto se realizó una clasificación, donde una metodología puede tener uno de los siguientes estados:

- **Recién nacida.** Es aquella metodología que tiene un año o menos y de la cual no tenemos evidencias empíricas, ni estudios.

- **En construcción.** Aquellas metodologías con más de un año de existencia, pero que no disponen de experiencias documentadas y/o estudios.
- **Activa.** Son aquellas metodologías que llevan varios en el desarrollo del software y de las cuales podemos encontrar experiencias y estudios que corroboren su efectividad.
- **Olvidada.** Aquellas metodologías que llevan el suficiente tiempo sin ser utilizadas y de las cuales no se encuentran experiencias actuales.

A continuación una breve descripción de cada metodología y el estado de la misma:

Tabla Comparativa 6. Comparativa del Estado de la Metodología.

Metodología	Descripción	Estado
Extremme Programming	Desde 1999 se ha convertido en la punta del iceberg de las metodologías ágiles, es la primera en internet, en adopciones y experiencias y la primera en estudios.	Activa
Agile Unified Process	Nace a mediados del 2000 como una respuesta ágil de Unified Process, aun no existe documentación explícita y se integran cada vez nuevas características.	En construcción
Crystal Clear	*Crystal Clear* es una de las metodologías de la Familia *Crystal* que tiene una amplia documentación y ha sido aplicada con éxito.	Activa
SCRUM	Una de las metodologías más antiguas y que más de moda esta últimamente, actualmente podemos encontrar muchos estudios y experiencias con *SCRUM*	Activa

Elaborado por: Verónica Noriega.

Con esta comparativa se puede observar que algunas metodologías tienen un mayor grado de madurez, alcanzada por aplicaciones exitosas, esto es un punto a favor dado que mientras más haya sido aplicada una metodología, es más confiable.

Como acotación al término de este análisis, se elaboró un cuadro, donde con una frase o palabra se intenta resumir el propósito de cada una de las metodologías.

Cuadro 9. Resumen de las Metodologías

Metodología	Resumen
Extremme Programming	Simplicidad.
Agile Unified Process	Eficaz manejo de riesgos.
Crystal Clear	Manejo óptimo de tamaño y criticidad.
SCRUM	Prioridad a las reglas de negocio.

Elaborado por: Verónica Noriega.
Fuente: Tablas comparativas del presente trabajo.

CAPÍTULO V

CAPITULO V

GUÍA DE APLICACIÓN

Para poder realizar eficientemente una guía de aplicación que sea útil para el desarrollador, es necesario en primer lugar hacer una clasificación de los proyectos o sistemas a desarrollar para que de este modo sea fácil ubicar a un proyecto en una determinada categoría.

Esta clasificación permitirá identificar de una mejor manera el tipo de proyecto y en base a esto se podrá emitir una recomendación acerca de la metodología que mejor se acopla a los requerimientos y a los recursos de los que se dispone.

En base a los conocimientos adquiridos a lo largo de la carrera de Ingeniería Informática y Ciencias de la Computación, y a la experiencia resultante de la aplicación de estos conocimientos, se plantea una clasificación generalizada, que se explica a detalle en lo sucesivo**.**

5.1 Clasificación de Proyectos de Desarrollo de Software

Como se explicó antes la clasificación se la realizará bajo conceptos objetivos, es así que se realiza la siguiente clasificación:

5.1.1 Por el entorno de Aplicación.

El entorno de aplicación se refiere al ambiente sobre el cual se va a ejecutar una aplicación, basados en este parámetro se clasifican en:

- **Escritorio.-** se refiere a que el software trabajará localmente en una máquina, es decir necesitará ser instalado y configurado en un computador para que pueda funcionar. Este software no tiene la capacidad de trabajar en red, solo puede compartir datos a través de este medio.

- **Web.-** se refiere a que el software trabajará remotamente, más concretamente en la nube, es decir que los usuarios pueden acceder desde cualquier lugar por medio de internet a través de un navegador y la aplicación podrá trabajar enviando y recibiendo datos. En este caso la aplicación está alojada en un servidor.

- **Híbrido.-** se refiere a que el software puede trabajar en entornos de escritorio y en web, un ejemplo claro de esto son las presentaciones multimedia que pueden ser ejecutadas en una máquina localmente y también pueden ser agregadas en un sitio web.

5.1.2 Por el Tamaño del Proyecto

Los proyectos de desarrollo de software obviamente tienen un espectro diferente de alcance, es decir algunos abarcan más cosas que otros, por este motivo, es importante hacer una clasificación de los proyectos de acuerdo a su tamaño, para esto se propone clasificarlos de acuerdo al número de

módulos que el sistema necesita para funcionar. Se entiende por módulo a un conjunto de funcionalidades que sirven para realizar una acción en específico dentro del software, por ejemplo, en un sistema para administrar una empresa de ventas, un módulo sería aquel que maneja la contabilidad y las finanzas, otro módulo sería el que maneja inventario, otro el que maneja el personal y otro el que maneja las adquisiciones.

Con esta introducción se puede ahora realizar la clasificación, que sería la siguiente:

- **Grande:** Se puede considerar a un proyecto grande si este necesita la implementación de 16 módulos o más.

- **Mediano:** Un proyecto es mediano si la cantidad de módulos necesarios para su implementación está entre los 6 y los 15.

- **Pequeño:** Un proyecto es pequeño si necesita implementar entre 1 y 5 módulos.

5.1.3 Por la Criticidad

Como se indica al inicio de este documento, el software ha venido a constituir una parte cada vez más importante en el acontecer humano debido a la automatización de procesos, es por este motivo que existen varias actividades que se le *encargan* hacer al software, estas actividades pueden ser de diversa índole, y dependiendo de los datos y de la información que

maneje, puede ser un software crítico, la criticidad de un proyecto de desarrollo de software se puede definir como la dificultad para corregir errores, si estos sucedieran, es decir el costo de un error, el nivel de criticidad puede variar de una sistema a otro dependiendo del tipo de información que se maneja h la complejidad de los procesos que realiza. Para esta clasificación se proponen tres niveles:

- **Criticidad Alta:** Cuando el software a construir va a manejar información delicada, es decir que pueda involucrar pérdida de vidas humanas, de dinero o atenten contra la seguridad de los ciudadanos o de una nación, y cuyo fallo pueda representar pérdidas significativas. Por ejemplo, un sistema de manejo de tráfico aéreo.

- **Criticidad Media:** Cuándo el software a desarrollar va a manejar información importante, pero que sin embargo su pérdida y/o corrupción no afecta en gran medida a personas o instituciones. Por ejemplo, un sistema de control de inventario.

- **Criticidad Baja:** Este nivel es el más seguro, ya que aquí no se maneja información de importancia, es decir, que si existiera una perdida y/o corrupción de la información, esto afectarían en poca o ninguna medida a las personas y/o instituciones involucradas. Por ejemplo, un juego.

5.1.4 Por la Calidad Requerida

Si bien es cierto que todo el software desarrollado debe ser de calidad, esto no siempre es verdad. En el mundo real para ahorrar recursos y de esta forma llevar a cabo un proceso eficiente y eficaz, es necesario aprovechar al máximo las situaciones que se presenten, por ejemplo un software con mucha calidad, es decir con un desempeño de 10/10 puede tomar más tiempo de desarrollo o mayor enfoque en la codificación de la aplicación, esto deriva claramente en un uso mayor de recursos, la calidad de un software es necesaria cuando el software tiene cierto nivel de criticidad, debido a que un fallo de este sería muy costoso.

Por tales razones, no todos los proyectos de software necesitan el mismo nivel de rigurosidad en su desempeño, ya que dependiendo de las funciones que hagan pueden o no requerir de calidad extrema, por tal razón se realiza la siguiente clasificación:

- **Excelente Calidad:** Se necesita que el software sea de excelente calidad si es un software que maneja información relevante, ya que exige que sus procesos sean exactos, oportunos y confiables. Por ejemplo, un sistema de seguridad para una entidad Bancaria.

- **Buena Calidad:** Un software necesita buena calidad cuando maneja información importante pero no muy relevante, debido a que no realiza procesos que puedan desencadenar eventos indeseables. Por ejemplo, un sistema de registro de asistencias.

- **Calidad regular:** Un software no necesita de mucha calidad si un fallo en sus procesos no significa pérdidas irreparables, es decir que un error de este tipo de software no tendrá mayores impactos en su entorno, debido a que los procedimientos que maneja son críticos. Por ejemplo, una presentación multimedia.

Cabe aclarar que todo software necesita calidad, ya que esta es una medida de éxito, pero hay que tomar en cuenta que una excelente calidad implica un mayor uso de recursos, por lo que en esta parte es muy recomendable analizar el entorno y tomar una decisión que permita un balance entre costos y calidad.

5.1.5 Por el tiempo Disponible para el Desarrollo del Proyecto.

Es muy importante también tomar en cuenta el tiempo del que se dispone para desarrollar un software, debido a que este tiempo puede marcar en gran medida diferencias entre unos y otros. A continuación se plantea una clasificación del tiempo disponible para realizar un software, bajo criterios de plazos estándar en el medio:

- **Muy Corto:** Cuando el tiempo que se dispone para el desarrollo del software es de 1 mes o menos.

- **Corto:** Cuando el plazo para terminar el software está entre 2 y 4 meses.

- **Medio:** Cuando el plazo para desarrollar el software oscila entre 5 y 11 meses.

- **Largo:** Cuando el plazo para desarrollar el software es de 12 meses o más.

5.1.6 Por la definición de Requerimientos Iniciales

Para el efecto de analizar la situación inicial de un proyecto de desarrollo de software, se ha contemplado también tomar en cuenta los datos iniciales con los que se cuenta para empezar con el desarrollo, es decir, evaluar si se tiene la suficiente información acerca de las funciones que va a desempeñar el software, esto permitirá iniciar con el proyecto de desarrollo. Esta clasificación es muy importante debido a que no todos los proyectos pueden iniciarse con un 100% de conocimiento de los requerimientos, es más se puede decir que la mayoría de proyectos exitosos ha tenido que integrar cambios a lo largo del proceso de desarrollo, el caso ideal sería tener un proyecto completamente definido desde el inicio, caso que es utópico. Dados estos antecedentes, la clasificación de los proyectos de software por la definición de requerimientos iniciales, está dada de la siguiente manera:

- **Definidos:** Son aquellos proyectos que han sido perfectamente definidos antes del inicio del desarrollo, es decir que poseen la suficiente información para empezar; además que tiene perfectamente delineados los alcances del proyecto.

- **No definidos:** Son aquellos proyectos en los cuales se deja abierta la posibilidad de cambios durante el desarrollo debido a que los requerimientos iniciales no han sido definidos en su totalidad, esto principalmente se debe a una falta de visión por parte del cliente, ya que no sabe con exactitud lo que quiere, por lo tanto el desarrollador se ve obligado a tener siempre una opción de integrar cambios.

Una vez que se han definido los parámetros en los que se va a basar la guía de aplicación el siguiente paso es justamente delinear las condiciones favorables para usar tal o cuál metodología de desarrollo, en pocas palabras es ahora cuando se va a guiar al desarrollador para escoger la metodología que más se ajuste a sus necesidades.

5.2 Guía de aplicación

A continuación a través de cuadros de resumen de las características anteriores, se hará la guía de aplicación, que permitirá ubicar a un proyecto de desarrollo de software dentro de una clasificación concreta, esto hará más fácil la labor de escoger la metodología adecuada.

Los cuadros tienen valores correspondientes a cada una de las clasificaciones anteriores, por lo tanto se señalará con una X cuando la metodología sea recomendable para esa característica.

5.2.1 Cuándo Usar Extremme Programming

Cuadro 10. Cuándo usar Extremme Programming

• **Si el entorno de aplicación es:**		
Web	Escritorio	Híbrido
X		X
• **Si el Tamaño del Proyecto es:**		
Grande	Mediano	Pequeño
	X	X
• **Si la Criticidad es:**		
Alta	Media	Baja
X	X	
• **Si la Calidad Requerida es:**		
Excelente	Buena	Baja
	X	X

• **Si el tiempo Disponible para el Desarrollo del Proyecto es:**			
Muy Corto	Corto	Medio	Largo
X	X		

• **Si los Requerimientos Iniciales están:**	
Definidos	No Definidos
	X

Elaborado por: Verónica Noriega.

5.2.2 Cuándo Usar Agile Unified Process

Cuadro 11. Cuándo usar Agile Unified Process

• **Si el entorno de aplicación es:**		
Web	Escritorio	Híbrido
X	X	
D		
• **Si el Tamaño del Proyecto es:**		
Grande	Mediano	Pequeño
X	X	
• **Si la Criticidad es:**		
Criticidad Alta	Criticidad Media	Criticidad Baja
X	X	
• **Si la Calidad Requerida es:**		
Excelente Calidad	Buena Calidad	Calidad Baja
X		
• **Si el tiempo Disponible para el Desarrollo del Proyecto es:**		
Muy Corto	Corto	Medio	Largo
	X	X	
• **Si los Requerimientos Iniciales están:**		
Definidos	No Definidos	
X		

Elaborado por: Verónica Noriega.

5.2.3 Cuándo Usar Crystal Clear

Cuadro 12. Cuándo usar Crystal Clear

• **Si el entorno de aplicación es:**		
Web	Escritorio	Híbrido
X		
• **Si el Tamaño del Proyecto es:**		
Grande	Mediano	Pequeño
		X
• **Si la Criticidad es:**		
Criticidad Alta	Criticidad Media	Criticidad Baja
	X	X
• **Si la Calidad Requerida es:**		
Excelente Calidad	Buena Calidad	Calidad Baja
	X	

• **Si el tiempo Disponible para el Desarrollo del Proyecto es:**			
Muy Corto	Corto	Medio	Largo
X	X		

• **Si los Requerimientos Iniciales están:**	
Definidos	No Definidos
	X

Elaborado por: Verónica Noriega.

5.2.4 Cuándo Usar SCRUM

Cuadro 13. Cuándo usar SCRUM

• **Si el entorno de aplicación es:**		
Web	Escritorio	Híbrido
X	X	
• **Si el Tamaño del Proyecto es:**		
Grande	Mediano	Pequeño
X	X	
• **Si la Criticidad es:**		
Criticidad Alta	Criticidad Media	Criticidad Baja
	X	
• **Si la Calidad Requerida es:**		
Excelente Calidad	Buena Calidad	Calidad Baja
X	X	

• **Si el tiempo Disponible para el Desarrollo del Proyecto es:**			
Muy Corto	Corto	Medio	Largo
X	X	X	X

• **Si los Requerimientos Iniciales están:**	
Definidos	No Definidos
	X

Elaborado por: Verónica Noriega.

5.3 Análisis por casos de aplicación.

Una vez que se ha realizado el análisis comparativo de las metodologías y que se ha determinado una clasificación que ha servido de base para la guía de aplicación, es necesario entonces realizar un análisis de la factibilidad de aplicar estas metodologías en diferentes escenarios de la vida real, es decir, que se realizará un análisis donde dependiendo de la situación que se presente se recomendará el uso de cierta metodología, a manera de ejemplos didácticos.

Caso uno.- Una empresa nacional requiere fabricar un sistema de control de inventarios, para un cliente específico, no requiere de implementación modular pues no está previsto incrementar funcionalidades críticas a través del tiempo, se requiere que el desarrollo de la aplicación se la realice en el menor tiempo posible tomando en cuenta que mientras más demore el proceso de creación del software el procesamiento de datos acumulados incrementaría de manera exponencial.

Solución: Al ser éste un sistema de tamaño medio que no maneja datos críticos, pero sí importantes, el cual no será modular y no requerirá actualizaciones periódicas a no ser que el marco legal del país cambie, entonces, en este caso según lo anteriormente analizado sería recomendable usar *SCRUM*, que es una metodología ligera que permite emprender un proyecto con pocas personas y con la ayuda del cliente se puede avanzar en un tiempo sumamente corto, además tiene la flexibilidad de que se pueden añadir funciones dentro del proceso si estas ni fueron especificadas en un inicio sin demorar el proceso.

Caso dos.- Se requiere construir una aplicación para *MS Office* que implemente a éste las funcionalidades de: Abrir, Visualizar, Modificar y Guardar archivos en formato *PDF*, el tiempo que se tiene es corto debido a que la competencia está desarrollando el mismo producto y debemos salir al mercado antes que ellos.

Solución: Para este caso se nos solicita un sistema pequeño, que sin embargo tiene un número considerable de funciones y módulos, los mismos que están relacionados entre sí, pero algunos módulos tienen un papel más significativo que otros dentro del ámbito del producto, también es necesario que se cumpla con un proceso riguroso que asegure calidad, además que el tiempo es corto, dada esta situación y debido a que existen procesos más críticos que otros, sería recomendable usar *AUP*, porque esta metodología se enfoca en los procesos críticos, además que su política de libre uso herramientas permitiría que los desarrolladores puedan usar módulos ya creados e incluso herramientas libres para complementar el producto, esto permite un ahorro significativo de tiempo y recursos, sin descuidar la calidad, la misma que está garantizada por el hecho de haber dado prioridad a los procesos críticos y usar herramientas que ya han sido probadas y usadas con éxito anteriormente.

Caso tres.- Se necesita desarrollar un sistema que permita automatizar el manejo de un consultorio médico, el mismo que por su naturaleza debe manejar procesos propios, esto quiere decir que tiene reglas de negocio específicas demanda además que sea realizado en corto tiempo y que los requerimientos puedan variar durante el proceso, el consultorio cuenta con 10 computadores con Sistema Operativo *Windows*, en diferentes versiones.

Solución: Se requiere un sistema mediano, que tiene un número considerable de funciones y módulos, además exige que procesos que son únicos del negocio sean elaborados de una forma muy meticulosa, ya que son procesos críticos dentro de la organización, para este caso sería muy recomendable usar *Extremme Programming*, porque puede manejar un equipo mediano en condiciones extremas de tiempo y recursos, con altos estándares de calidad dado a las sucesivas pruebas, además el hecho de que tenga un cliente en el sitio, permite que haya una supervisión constante de lo que se hace, de este modo se asegura que se cumplirán con los requerimientos del usuario con calidad y rigurosidad.

Caso cuatro.- Una empresa multinacional que compite en el mercado de los juegos para *PC,* requiere desarrollar un *juego FPS* (*First Person Shooter*, tipo *Doom*) en *3D*, no se requiere que sea multiplataforma y utilizará las librerías *OpenGL*. La empresa requiere empezar a comercializar el juego en máximo seis meses.

Solución: Para un sistema como este, es decir un juego, se necesita desarrollar de forma rápida y por su naturaleza no es necesario ahorra recursos de sistema en la programación ni tampoco un nivel de calidad muy exigente. Por lo tanto es recomendable usar *Extremme Programming*, dado que esta metodología se enfoca más en el desarrollo rápido de versiones funcionales, antes que en la calidad, situación que para este caso es muy beneficiosa, puesto que al imponer menores estándares de revisión de calidad ahorra tiempo para éste tipo de desarrollo específico.

Caso cinco.- Una empresa nacional necesita crear un sistema que recepte mensajes de teléfonos celulares, y que automáticamente envíe una respuesta de acuerdo al texto recibido, además el sistema debe tener una conexión al web server de la correspondiente operadora telefónica para solicitar el cobro acordado del envío del mensaje.

Solución: Este es un sistema muy pequeño que no necesita de seguridades extremas, los estándares de calidad para este tipo de desarrollo son casi nulas, lo que se necesita es que el sistema trabaje de acuerdo con los requerimientos y con el contrato que se hizo con la operadora, por lo tanto es recomendable usar *Crystal Clear*, que permite manejar sistemas sumamente pequeños que pueden ser desarrollados por una sola persona, además *Crystal* permite un desarrollo ágil con énfasis en versiones probadas y funcionales, además es muy recomendable para este caso ya que esta metodología no exige mucha disciplina y es precisamente lo que el sistema requiere. Debido a que no se manejan procesos críticos como el descuento de los valores de cobro (puesto que esto lo hace la operadora), no se necesita poner demasiado énfasis en la calidad.

Caso seis.- La Importadora "SUSANITA" tiene un software de contabilidad que ha venido usando durante varios años, pero debido a cambios en el marco tributario del País se han visto en la obligación de hacer cambios en el módulo de declaración de impuestos. Debido a la constante actividad comercial que maneja la empresa se necesita que los cambios se los realice en el menor tiempo posible y tratando de no

afectar el funcionamiento del resto de módulos del sistema. Cabe recalcar que se tiene acceso al código fuente de este sistema.

Solución: Lo que se requiere no es un sistema en sí, sino solamente la modificación de un sistema existente, este es un proceso sumamente pequeño pero muy crítico dado que es de mucha importancia dentro del ámbito de la organización, al ser este un proceso que necesita de altos estándares de calidad y rapidez sería recomendable usar *AUP*, debido a que maneja de una manera eficiente los riesgos desde el inicio del proceso asegurando calidad, además que por el hecho de dejar al desarrollador usar las herramientas que considere convenientes para el desarrollo, se puede ahorrar tiempo.

Caso siete.- Se necesita desarrollar una página web que tenga la funcionalidad de un foro, es decir se necesita crear una aplicación que dé soporte a discusiones u opiniones en línea. Se dispone de poco tiempo para su realización y de pocos recursos (humanos y materiales).

Solución: Al ser este un sistema web relativamente pequeño, dado que no maneja muchas funcionalidades y no requiere de procesos muy específicos, puede ser desarrollado por pocas personas, incluso por una sola, además por lo descrito antes el sistema no requiere de mucha calidad, para este caso sería recomendable usar *Crystal Clear*, ya que es un proceso que no exige mucha disciplina y puede ser llevada a cabo por una persona, además que *Crystal* permite llegar de una forma rápida a la versión final del software debido a que se desarrolla basándose en

versiones funcionales continuas, esto permite que se corrijan errores conforme se avanza y hace que el proceso además de ágil sea eficiente, es decir que tenga el menor número de errores.

Caso ocho.- El consorcio del nuevo aeropuerto de Quito, necesita crear el sistema de administración del aeropuerto, el cual debe tener a su vez conexión con los sistemas individuales de cada aerolínea, lo que le permitirá obtener la información de éstas para la labor de administración. El sistema debe ser seguro, robusto, escalable y adaptable. El sistema debe ser realizado en el menor tiempo posible, debido a que el aeropuerto iniciará sus operaciones en dos años.

Solución: El sistema que se requiere es sumamente grande, debido a que la administración de un aeropuerto tiene una gran cantidad de actores y funciones, situación que vuelve al sistema complicado por las interrelaciones que existen entre éstos. Debido a las reglas de negocio, los procesos son bastante críticos, es decir que deben ser manejados con suma precaución, y lo más importante que el sistema debe ser eficaz, oportuno y con cero errores, porque un error en este sistema podría significar vidas humanas, por lo tanto este proyecto necesita de una metodología robusta pero flexible, que garantice procesos 100% seguros. En este caso se recomendaría usar *SCRUM*, que es una metodología altamente escalable, esto quiere decir que puede manejar proyectos grandes como éste debido a que se ha logrado implementar con éxito en otros proyectos de similares proporciones, además que es una metodología robusta dado que recoge información importante en un principio que le permite manejar de una forma eficiente los procesos críticos del software, las

reuniones continuas, el *feedback* (retroalimentación) y sobre todo la creación de versiones funcionales al final de cada *Sprint* asegura una calidad extrema, porque se minimiza la existencia de errores en el funcionamiento del sistema, además que no hay que olvidar que *SCRUM* es muy flexible a la hora de insertar cambios y/o nuevas funcionalidades, situación frecuente en sistemas grandes dado que es difícil definir en el inicio todas las funciones necesarias.

CAPÍTULO VI

CAPITULO VI

CAPACITACIÓN

Al inicio del presente documentos se planteó como avance del proyecto, que luego de realizar un análisis de las metodologías, se realizaría también una capacitación que permitiera y a los estudiantes de la Carrera de Ingeniería Informática y Ciencias de la Computación de la Universidad Tecnológica Equinoccial, conocer acerca de estas nuevas tendencias en el desarrollo de software.

El motivo de realizar la capacitación fue la de compartir los conocimientos adquiridos a través de la consecución del presente trabajo con las personas que están dentro del ámbito del desarrollo, y de esta forma consolidar conocimientos que les serán de ayuda en un futuro.

Para poder llevar a cabo la capacitación, en primer lugar se deberá elaborar una Guía de Capacitación, que describa los objetivos de la capacitación y los temas que se van a tratar, así también una evaluación que será una herramienta que servirá para saber que tanto fue retenido por las personas a quienes les impartió la capacitación.

A continuación se desarrolla una guía de capacitación que servirá de base para impartir los conceptos que fueron objeto de investigación del presente trabajo de Tesis de Grado.

6.1 Guía de Capacitación.

1. **Tema:** Metodologías de Desarrollo Ágil de Aplicaciones: sus ventajas y desventajas. Un Análisis Comparativo.

2. **Objetivos:**

- **Objetivo General**

  Brindar a los participantes una alternativa para el desarrollo de aplicaciones de software, que está acorde a los requerimientos actuales.

- **Objetivos Específicos**
  - Enseñar acerca de las metodologías ágiles, sus ventajas y desventajas, para que en el futuro sean tomadas en cuenta a la hora de desarrollar.

  - Otorgar a los participantes una herramienta de distinción, que les permita escoger la metodología adecuada.

  - Dar a conocer el enfoque de desarrollo ágil, que es relativamente nuevo, para que sea usado y aprovechado al máximo.

3. **Requisitos mínimos:**

La persona que quiera participar en la capacitación debe tener los siguientes conocimientos previos:

- ✓ Metodología de la Investigación.
- ✓ Programación (estructurada u orientada a objetos).
- ✓ Conocer que es un proyecto.
- ✓ Herramientas de colaboración de la programación es decir, herramientas de diseño, modelado, pruebas.

4. **Temario de la Capacitación**

Sesión 1.- Antecedentes Históricos y Modelos Base

Propósito.

En esta sesión se tratará el inicio de las metodologías y su evolución a través del tiempo, además se abordará el tema de los modelos de desarrollo de software *tradicionales,* que son la base de las metodologías.

Objetivos.

Al finalizar esta sesión los participantes conocerán:

- Cómo nacieron las metodologías
- Como están estructuradas desde sus bases las metodologías de desarrollo de software.

Contenido.

1. Breve Historia de las Metodologías.
2. Definición del Modelo.
3. Modelos de Desarrollo de Software.
    a) Modelo en Cascada.
    b) Modelo *DRA.*
    c) Modelos de Construcción de Prototipos.
    d) Modelo en Espiral.

Sesión 2.- Metodologías y Agilidad

Propósito.

Se planteará el significado de *Metodología,* es decir ¿qué es?, ¿para qué sirve? y que características debería cumplir.

Aquí también se tratará el tema de la agilidad, ¿Qué es considerado ágil? y como cumplirlo, además se analizará el manifiesto ágil y sus principios

Objetivos.

Al término de esta sesión, los participantes conocerán:

- Qué es una metodología y para qué sirve
- La diferencia entre agilidad y rapidez
- Los principios del desarrollo ágil y su aplicación

Contenido.

1. Denominación de Metodología.
2. Definición de Agilidad
3. El Manifiesto Ágil y sus principios.

Sesión 3.- Metodologías Ágiles

Propósito.

Aquí se describirán las cuatro metodologías ágiles más representativas (*XP*, *SCRUM*, *Crystal Clear* y AUP), detallando los procesos, artefactos, fases y herramientas de cada una de ellas.

Objetivos.

Al finalizar esta sesión los participantes conocerán:

- Las metodologías ágiles.
- La forma en que se manejan cada una de las metodologías.
- Cómo se lleva a cabo un proyecto usando las metodologías.
- Los elementos que interactúan en una metodología ágil.

Contenido.

1. *Extremme Programming* (*XP*)
    a. Generalidades.
    b. Actividades.
    c. Principios.
    d. Prácticas.

e. Actores y Responsabilidades.

f. Fases.

g. Artefactos.

2. *Agile Unified Process* (AUP)

a. Generalidades.

b. Prácticas.

c. Fases.

d. Principios.

3. *Crystal Clear* (CC)

a. Generalidades

b. Principios.

c. Técnicas de Colaboración en el Proceso.

d. Roles y Responsabilidades.

4. *SCRUM*

a. Generalidades.

b. Roles y Responsabilidades.

c. Artefactos.

d. Técnicas de Colaboración en el Proceso.

Sesión 4.- Comparativas de metodologías ágiles

Propósito.

En esta sesión se tratarán las fortalezas y debilidades de cada una de las metodologías, además se realizará el análisis de los aspectos más importantes y se realizará un análisis comparativo, así entonces se podrá concluir qué metodologías usar en determinadas situaciones.

Objetivos.

Al término de esta sesión los participantes podrán:

- Reconocer las fortalezas y debilidades de las metodologías ágiles.
- Emitir un juicio acerca del uso o no de una metodología para un caso específico.
- Reconocer las diferencias existentes entre las metodologías ágiles.
- Escoger la metodología que más se ajuste a los requerimientos de un producto.

Contenido.

1. Fortalezas y Debilidades
2. Comparativa del Ciclo de vida
3. Comparativa del tipo de proceso (ajustable o prescriptivo)
4. Comparativa de tamaño del equipo
5. Comparativa de existencia de herramientas colaborativas dentro de la descripción de la Metodología.

6. Comparativa de requisitos iniciales.
7. Comparativa del Estado Actual de la Metodología

Sesión 5.- Casos de Éxito

Propósito.

Aquí se expondrán algunos casos de éxito a nivel empresarial y se emitirán recomendaciones acerca del uso de las Metodologías.

Objetivos.

Al finalizar esta sesión los participantes sabrán:

- Como aplicar exitosamente una metodología.
- Manejar a su favor los factores de éxito de otros proyectos.
- Qué metodologías son eficientes y pueden ser aplicadas en sus proyectos.

Contenido.

1. Análisis de aplicación.
2. Factibilidad de uso.
3. Casos de empresas que los han aplicado.
4. Resultados.
5. Conclusiones y Recomendaciones.

5. Evaluación.

Como corolario de la capacitación realizada, se hace necesaria una evaluación que permita observar el real alcance de la capacitación y que tanto captaron los alumnos acerca del tema.

La evaluación además es una herramienta que nos dice que tan efectiva fue la clase y si los conocimientos impartidos han sido asimilados y por ende podrán ser aplicados en el futuro.

CAPÍTULO VII

CAPITULO VII

CONCLUSIONES Y RECOMENDACIONES

7.1 Conclusiones

Aunque el enfoque *ágil* de las Metodologías de Desarrollo de software, parezca ser completo y pensado para abarcar todo el espectro de posibilidades, y a simple vista podría decirse que son la panacea para quienes siempre vivieron buscando una metodología que se ajuste a los requerimientos, esto dista mucho de la verdad.

Es sabido que la tecnología cambia a un ritmo bastante acelerado, y que cada vez las exigencias del cliente son diferentes, es por esto que aunque hoy las metodologías ágiles estén en su apogeo, gocen de buena fama y sean consideradas como la solución definitiva, a muy corto plazo existirá un nuevo enfoque que hará que estas metodologías queden obsoletas.

Es por esto que no se puede aseverar que la real solución al desarrollo de software son las metodologías ágiles, ya que a través de la historia, siempre han aparecido nuevas y mejoradas tendencias, que han logrado cerrar un círculo vicioso en busaca de una mejor alternativa para desarrollar, es probable que en un futuro próximo las metodologías ágiles sean desplazadas y relegadas.

Es también interesante ver como las metodologías ágiles han desplazado por completo a la concepción de *Ingeniería de Software*, debido a que rompen con el paradigma de que el desarrollo debe estar guiado por procesos estrictos propios de la ingeniería, y proponen algo interesante: **centrarse en las personas**, esto es un cambio trascendental en la corriente del desarrollo de software, y además ha permitido que los procesos sean más fáciles de llevar.

Uno de los problemas de las *metodologías monumentales* era, que por el hecho de ser tan rigurosas, los desarrolladores a veces las seguían pero no con el suficiente rigor, las cumplían a medias o simplemente no las cumplían, debido a que demandaban demasiado tiempo y esfuerzo en tareas que solo eran apoyo del proceso de desarrollo de software.

Por su parte las metodologías ágiles proponen manejar procedimientos más flexibles, y no exigen que se cumplan con tanto rigor las actividades de apoyo al proceso de desarrollo, esto permite a los desarrolladores concentrarse en lo que es importante y no desperdiciar tiempo valioso, esta diferencia ha logrado que los desarrolladores ahora opten por usar metodologías porque saben que es una forma de garantizar la calidad de su trabajo y de llevar un proceso ordenado y óptimo.

Otra conclusión importante es que estas metodologías a pesar de haber nacido para un objetivo específico, el desarrollo de software, hoy están siendo usadas por organizaciones de distinta índole para regular, controlar y planear las labores de sus empresas, dado que es muy interesante la opción que plantean de enfocarse en las personas más que en los procesos en sí, esta tendencia ha logrado crear personas más

activas y participativas, hecho que a la larga desemboca en eficiencia y calidad, factores que son de suma importancia para cualquier organización.

7.2 Recomendaciones

Al momento de iniciar un proyecto de desarrollo de software, es muy importante tomar en cuenta las variables del entorno que intervienen en este proyecto, como tiempo, recursos, nivel de dificultad y otras exigencias externas, debido a que si no se lo hace, podría elegirse de una forma errada la metodología que se va a usar. Aunque las metodologías ágiles son bastante adaptables, pueden generar resultados indeseados si se usan de forma incorrecta. Por esta razón es muy importante elegir la metodología adecuada, por lo que se recomienda que se realice un análisis preliminar del proyecto, para saber las exigencias y de acuerdo a eso elegir una.

Es muy común que la gente en general crea que si un método funcionó para llevar a cabo un plan A, funcionará también para llevar a cabo un plan B. Esto no es así, en desarrollo de software no se debe cometer ese error, porque como se mencionó antes, podría obtenerse resultados no deseados.

Que una metodología haya funcionado para un proyecto, no quiere decir que va a funcionar para todos los proyectos, es un error de concepción bastante usual. Es necesario tener en cuenta que cada proyecto es diferente, y por tal razón es necesario que se lo analice para poder elegir una metodología que se ajuste a los requerimientos.

Después de haber realizado este estudio una de las recomendaciones más importantes es la de seguir una metodología para desarrollar un software, esto garantiza que el proceso se realizará aprovechando al máximo los recursos disponibles y se obtendrá los resultados deseados.

Es recomendable no tomar como una panacea a las metodologías ágiles, ya que al igual que métodos anteriores estas son solo un enfoque, que permite ciertas ventajas, pero que no siempre puede ser la solución a todos los problemas.

Si un equipo de desarrollo de software quiere adaptar estas nuevas metodologías orientadas al desarrollo ágil de aplicaciones, es recomendable que se vaya haciendo un cambio paulatino si es que antes se usaba otra metodología, ya que esto permite adaptarse al cambio más fácilmente. Además esta es una de las ventajas de las metodologías ágiles, que son tan flexibles que permiten integrar varias técnicas para conseguir el resultado deseado.

El uso de metodologías ágiles es mucho más recomendable que el uso de metodologías tradicionales, debido a que el enfoque de las metodologías ágiles permite obtener resultados visibles en menor tiempo, centrándose en las interacciones de las personas y no en la documentación.

Las término *Metología Ágil* no se refiere a que este tipo de metodologías solo sirven para aplicar en proyectos de desarrollo donde se dispone de poco tiempo para la consecución del proyecto, sino por el contrario se refiere a que cualquier proyecto,

de la naturaleza que sea, puede verse radicalmente agilizado por el uso correcto de estos métodos, es decir que un proyecto no debe ser ágil para aplica una de estas metodologías, sino que un proyecto por grande, se convertirá en ágil por el uso de estas metodologías.

Si se necesita desarrollar un software para un entorno web o híbrido, que tenga un tamaño entre mediano y pequeño, cuya criticidad sea alta, la calidad requerida sea buena, para el cuál el tiempo disponible sea corto, y los requisitos no estén bien definidos, entonces es recomendable usar Extremme Programming,

Si se necesita un software cuyo entorno de aplicación sea escritorio, su tamaño es grande, con criticidad alta y exige excelente calidad en un tiempo de corto a medio, teniendo requerimientos definidos, es recomendable usar AUP.

Si se requiere desarrollar un software en un entorno web, que sea pequeño y cuyos requerimientos de calidad y criticidad sean poco exigentes, y teniendo muy poco tiempo para desarrollarlo tomando en cuento que los requisitos no están definidos, es recomendable usar Crystal Clear.

Sin embargo, pueden existir situaciones en las cuales sea demasiado difícil clasificar a un proyecto de desarrollo de software, debido a varios factores que pueden haber sido obviados en la clasificación presentada a través de este trabajo investigativo, en ese caso es muy recomendable usar SCRUM, debido a que es una metodología bastante versátil que se puede adaptar a cualquier tipo de requerimientos, sin que esto represente un proceso de adaptación complejo.

SCRUM es recomendable si se tiene un tiempo indeterminado, es decir si no se conoce bien el límite de tiempo, ya que con un manejo adecuado se puede usar muy poco tiempo, sirve tanto en entornos web como escritorio, cuyo tamaño sea mediano y los requerimiento de calidad y criticidad sean exigentes, además puede trabajar muy bien aún si los requerimientos iniciales aún no han sido definidos.

GLOSARIO

Desarrollador.- es un informático que programa aplicaciones en distintos lenguajes de programación informáticos.

Herramientas CASE.- son diversas aplicaciones informáticas destinadas a aumentar la productividad en el desarrollo de software reduciendo el costo de las mismas en términos de tiempo y de dinero. Estas herramientas nos pueden ayudar en todos los aspectos del ciclo de vida de desarrollo del software en tareas como el proceso de realizar un diseño del proyecto, cálculo de costes, implementación de parte del código automáticamente con el diseño dado, compilación automática, documentación o detección de errores entre otras.

Método Delphi.- es una metodología de investigación multidisciplinaria para la realización de pronósticos y predicciones.

Proceso de Desarrollo de Software.- Un proceso de desarrollo de software es una estructura impuesta al desarrollo de un producto de software. También se le conoce como *ciclo de vida del software* y *proceso de software.*

Programación.- es el proceso de diseñar, escribir, probar, depurar y mantener el código fuente de programas computacionales. El código fuente es escrito en un

lenguaje de programación. El propósito de la programación es crear programas que exhiban un comportamiento deseado.

Requerimientos.- es una necesidad documentada sobre el contenido, forma o funcionalidad de un producto o servicio. Se usa en un sentido formal en la ingeniería de sistemas o la ingeniería de software.

UML.- es el lenguaje de modelado de sistemas de software más conocido y utilizado en la actualidad; está respaldado por el OMG (Object Management Group). Es un lenguaje gráfico para visualizar, especificar, construir y documentar un sistema.

BIBLIOGRAFÍA

1. Beck, Kent, Extreme Programming Explained: Embrace Change. Addison-Wesley, Chicago, 2005.

2. Cockburn, Allistair, Agile Software Development. Addison-Wesley, Chicago, 2002.

3. Highsmith, Jim, Agile Software Development Ecosystems. Addison-Wesley, 2006.

4. Jacobson, Ivar, A Resounding 'Yes' to Agile Processes – But Also More. Cutter IT Journal, (Revista), enero, 2004.

5. Jeffries Ron, Anderson Ann, Hendrickson Chet, Extreme Programming Installed, Addison-Wesley, The XP Series, 2007.

6. Kniberg, Henrik, XP and Scrum from the Trenches. How we do Scrum. InfoQ Magazine.(Revista) 2007.

7. Pressman S., Roger, Ingeniería del Software. Un enfoque práctico. Mac Graw Hill, 2006.

8. Schwaber, K. y Beedle, M. Agile Software Development with Scrum. Prentice Hall, 2006.

9. Wake, W, Extreme Programming Explored, Addison-Wesley, 2004.

10. Watson, M, Managing Smaller Projects: A practical Guide. Project Manager Today (Revista), octubre 2006.

LINKS DE PÁGINAS WEB CONSULTADAS

1. http://www.agilemanifesto.org/principles.html, Principles behind the Agile Manifesto.

2. http://www.extremeprogramming.org/

3. htpp://www.conductitlan.net , Asociación Oaxaqueña de Psicología, Elaboración de programas de capacitación.

4. http://xprogramming.com/index.php

5. http://www.methodsandtools.com/archive/archive.php?id=21, The Agile Unified Process (AUP).

6. http://www.agilealliance.org/

7. http://www.objectmentor.com/

8. http://www.scribd.com/doc/38230062/Wide-Band, Método Wideband Delphi.

9. http://www.agile-spain.com/

10. http://es.wikipedia.org/wiki/*Scrum*

11. http://www.lnds.net/blog/2009/09/¿que-es-agilidad.html, "¿Qué es Agilidad?".

12. http://scrumtraininginstitute.com

13. http://www.controlchaos.com/

14. http://www.c2.com/cgi/wiki?*Crystal*ClearMethodology

15. http://www.otssolutions.com/blog/?p=99

16. http://www.informatizate.net/articulos/metodologias_de_desarrollo_de_software_076204.html

17. http://geeks.ms/blogs/jorge/archive/2007/05/09/explicando-scrum-a-mi-abuela.aspx

18. http://www.willydev.net/descargas/prev/TodoAgil.pdf

VIDEOS DE AYUDA CONSULTADOS

1. Agile Styles: Feature Driven Development and the *Crystal* Methodologies. Visto en: http://www.infoq.com/presentations/fdd-crystal-agile-overview

2. Ingeniería del Software. Visto en:
   http://www.youtube.com/watch?v=owKZgt4k6Ns&feature=grec_index

3. XP - Decisión de Negocio. Visto en:
4. http://www.youtube.com/watch?v=-wFZjQ27hbo

5. Modelos de Desarrollo de Software. Visto en:
   http://www.youtube.com/watch?v=axrN3zTRKxI

6. Introduction to *Scrum* in just 8 minutes! Visto en:
   http://www.youtube.com/watch?v=_QfFu-YQfK4

7. Modelo Desarrollo de Software (*SCRUM*). Visto en :
   http://www.youtube.com/watch?v=N5gcr8B-xoc

8. Lecture 24: eXtreme Programming - Richard Buckland. Visto en :
   http://www.youtube.com/watch?v=*XP*4o0ArkP4s

ANEXOS

ANEXOS

Anexo 1. Video de la Capacitación realizada.

A propósito de la Capacitación que se realizó en la cual participaron alumnos de la Universidad Tecnológica Equinoccial, de la carrera de Ingeniería en Informática y Ciencias de la Computación, se realizó un video que evidencia la realización de esta actividad.

Este video está subido en un servidor gratuito de video, y puede ser accedido través de la siguiente dirección:

http://www.megavideo.com/?v=VJCALZ25

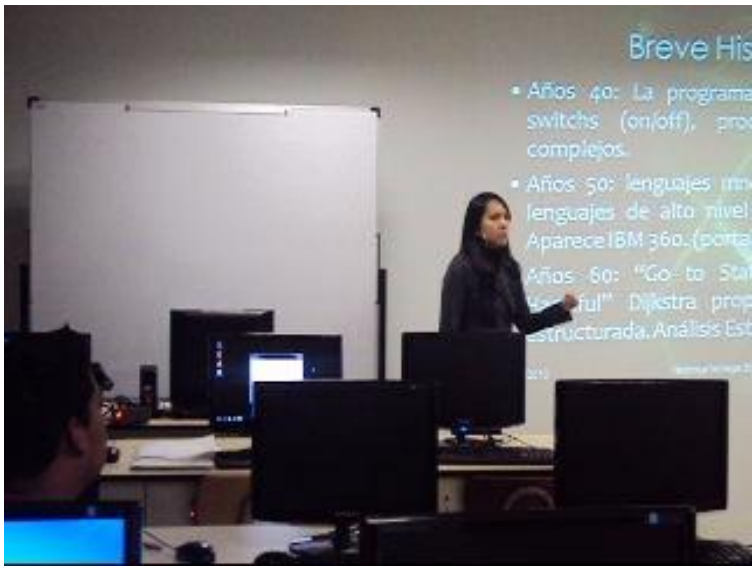
Anexo 2. Fotografías de la Capacitación realizada.

De la misma manera que el video, las fotografías fueron hechas con el propósito de evidenciar el trabajo de capacitación impartida a los alumnos, a continuación una muestra:

Fotografía 1. Capacitación.

Fuente: Tomada el 1 de diciembre de 2010 en el laboratorio 206 IDIC-UTE

Fotografía 2. Capacitación.



Fuente: Tomada el 1 de diciembre de 2010 en el laboratorio 206 IDIC-UTE

Fotografía 3. Capacitación.

Fuente: Tomada el 1 de diciembre de 2010 en el laboratorio 206 IDIC-UTE

Fotografía 4. Capacitación.

Fuente: Tomada el 1 de diciembre de 2010 en el laboratorio 206 IDIC-UTE

Fotografía 5. Capacitación.

Fuente: Tomada el 1 de diciembre de 2010 en el laboratorio 206 IDIC-UTE

.

Fotografía 6. Capacitación.

Fuente: Tomada el 1 de diciembre de 2010 en el laboratorio 206 IDIC-UTE

Anexo 3. Evaluación Aplicada.

Evaluación de Metodologías de Desarrollo Ágil.

Nombre:…… Fecha:……..……..

1. ¿Una metodología y un modelo de desarrollo de software son lo mismo? si, no, por qué?

   ……
   ……

2. Indique la diferencia entre el modelo en espiral y el modelo en cascada

   ……
   ……

3. ¿Cuál es la diferencia entre agilidad y rapidez en el ámbito del desarrollo de software?

   ……
   ……

4. ¿Cuál es la principal orientación de las metodologías ágiles?

   ……
   ……

5. ¿Se necesita desarrollar un juego para iPhone, el tiempo es corto debido a la alta competitividad de este mercado, que metodología recomendaría usted que se aplique en este caso y porque?

   ……
   ……
   ……

Evaluación de Metodologías de Desarrollo Ágil.

Al finalizar la capacitación se hacía necesaria una evaluación que permita conocer el nivel de captación por parte de los alumnos, es decir saber que tanto comprendieron.

La evaluación se la realizó de forma física y arrojó los siguientes resultados:

Evaluación de Metodologías de Desarrollo Ágil.

8.5

Nombre: Erik Marroquín Fecha: 08-12-2010

1. ¿Una metodología y un modelo de desarrollo de software son lo mismo? si, no, por qué?

   No son lo mismo, un modelo es parte de una metodología, y una metodología es un conjunto de requerimientos para desarrollar un software. 2.0

2. Indique la diferencia entre el modelo en espiral y el modelo en cascada

   El modelo en espiral se lo realiza por datos y el modelo de cascada se lo realiza secuencialmente y sobre todo se diferencian en que el modelo en espiral tiene análisis de riesgos y el de cascada no. 2.0

3. ¿Cuál es la diferencia entre agilidad y rapidez en el ámbito del desarrollo de software?

   Agilidad es la capacidad de adaptarse a los cambios, necesidades y requerimientos mientras que rapidez es hacer un desarrollo velozmente pero no necesariamente es o resulta eficaz el software, es mejor ser ágil a ser rápido. 2.0

4. ¿Cuál es la principal orientación de las metodologías ágiles?

   buscar nuevas maneras y formas ágiles de construir proyectos. Sin estar enfocandose en una sola forma sino utilizar todos los modelos y herramientas para que el cliente quede satisfecho. 0.5

5. Se necesita desarrollar un juego para Iphone, el tiempo es corto debido a la alta competitividad de este mercado, que metodología recomendaría usted que se aplique en este caso y porque?

   XP porque para algo tan simple como un juego de iphone la simplicidad que posee el Extremme Programing es la idónea para tal requerimiento. 2.0

Evaluación de Metodologías de Desarrollo Ágil.

5.5

Nombre: Pablo Rocha Fecha: 08 - XII - 2010

1. ¿Una metodología y un modelo de desarrollo de software son lo mismo? si, no, por qué?

No porque una metodología es un plan o un esquema de como se va realizar la creación del Software y un modelo son las reglas por las cuales tiene q' pasar el software para garantizar su calidad

2.0

2. Indique la diferencia entre el modelo en espiral y el modelo en cascada

El espiral es un modelo q' nos sirve en el caso de q' el software sea demasiado grande y más q' todo este este planeado para ser mejorado continuamen y el de cascada es optimo en el caso de un desarrollo pequeño, y sin mayores cambios

0.5

3. ¿Cuál es la diferencia entre agilidad y rapidez en el ámbito del desarrollo de software?

la agilidad tiene q' ver con el talento para poder solucionar los problemas q' se presente y la rapidez el tiempo q' me toma llegar a una resolución o una respuesta para ese problema

2.0

4. ¿Cuál es la principal orientación de las metodologías ágiles?

la retroalimentación

1.5

5. Se necesita desarrollar un juego para Iphone, el tiempo es corto debido a la alta competitividad de este mercado, que metodología recomendaría usted que se aplique en este caso y porque?

Scrum por q' esta metodología ya tiene o contempla muchas aplicaciones q' podemos utilizar sin tener q' probar a profundidad ya q' fueron probadas anteriormente y eso nos ahorraría demasiado tiempo

0.5

Evaluación de Metodologías de Desarrollo Ágil.

9.0

Nombre: Carlos Montenegro Fecha: 8/12/2010

1. ¿Una metodología y un modelo de desarrollo de software son lo mismo? si, no, por qué?

No porque la metodología es como o los pasos a seguir para un proyecto y el modelo es la meta o lo q se quiere llegar hacer. 2.0

2. Indique la diferencia entre el modelo en espiral y el modelo en cascada

Que en el modelo en cascada no se puede regresar una vez q se termino el proyecto y en el espiral se lo va mejorando por ciclos. 2.0

3. ¿Cuál es la diferencia entre agilidad y rapidez en el ámbito del desarrollo de software?

(Que rápidez es el tiempo de respuesta del sistema) la agilidad se refiere a la capacidad del equipo en desarrollar y rapidez el tiempo q se demoran 1.5

4. ¿Cuál es la principal orientación de las metodologías ágiles?

la función de retroalimentación 1.5

5. Se necesita desarrollar un juego para Iphone, el tiempo es corto debido a la alta competitividad de este mercado, que metodología recomendaría usted que se aplique en este caso y porque?

Aplicaría el XP ya q es pequeño y se lo podría realizar en corto tiempo porque no pide muchas pruebas 2.0

Evaluación de Metodologías de Desarrollo Ágil.

6.5

Nombre: Hector Caluache. Fecha: 8-12-2010

1. ¿Una metodología y un modelo de desarrollo de software son lo mismo? si, no, por qué?

no por que en una metodologia contiene procesos, metodos ~~para~~ ~~en proyecto el cual el modelo es esto para~~ modelos para desarrollar un proyecto el cual se pueden aplicar dependiendo del programador 1.0

2. Indique la diferencia entre el modelo en espiral y el modelo en cascada

En que el modelo en espiral tiene analisis de riesgo a comparación del modelo de cascada 1.0

3. ¿Cuál es la diferencia entre agilidad y rapidez en el ámbito del desarrollo de software?

En q' agilidad ~~es para adaptarse a cambios~~ para un proyecto es adaptarse a cambios, procesos, etc el cual este debe estar bien realizado y con todas las posibilidades a comparacion rapidez es para q' un proyecto sea rapido sin necesidad de q' este bien echo. 2.0

4. ¿Cuál es la principal orientación de las metodologías ágiles?

La orientacion de las metodologias es hacer a un proyecto lo mas posibles agiles. 0.5

5. Se necesita desarrollar un juego para Iphone, el tiempo es corto debido a la alta competitividad de este mercado, que metodología recomendaría usted que se aplique en este caso y porque?

~~el de XP porque en este~~ XP por que es igual a simplicidad 2.0

Evaluación de Metodologías de Desarrollo Ágil.

6.0

Nombre: David Vallejo Fecha: 22/12/2010

1. ¿Una metodología y un modelo de desarrollo de software son lo mismo? si, no, por qué?

No, porque metodologia son mas flexible y estan enfocada al cumplimiento, mientras que los modelos son cerrados y estrictos 0.5

2. Indique la diferencia entre el modelo en espiral y el modelo en cascada

Que en que el modelo espiral cada que finaliza una vuelta se realiza un riesgo 2.0

3. ¿Cuál es la diferencia entre agilidad y rapidez en el ámbito del desarrollo de software?

La rapidez es la realizar en poco tiempo un determinado proyecto La agilidad es la flexibilidad con la que se maneja un proyecto 1.0

4. ¿Cuál es la principal orientación de las metodologías ágiles?

Estan enfocadas al cumplimiento de los proyectos 0.5

5. Se necesita desarrollar un juego para Iphone, el tiempo es corto debido a la alta competitividad de este mercado, que metodología recomendaría usted que se aplique en este caso y porque?

La metodologia XP ya que esta se enfoca en la simplicidad de los proyectos y ya que no se realizan controles y se puede utilizar nuevas ya existentes 2.0

8.5

Evaluación de Metodologías de Desarrollo Ágil.

Nombre: Daniela Mazón Fecha: 2010/12/08

1. ¿Una metodología y un modelo de desarrollo de software son lo mismo? si, no, por qué?

   No, porque un modelo es algo que quiera seguir o sea una meta a alcanzar mientras que una metodología son los pasos que voy a tomar para hacer el modelo 2.0

2. Indique la diferencia entre el modelo en espiral y el modelo en cascada

   El modelo en cascada no tiene retroalimentación y si en algún paso nos equivocamos no podemos ir para atras así como en el espiral que cada vez vamos 2.0

3. ¿Cuál es la diferencia entre agilidad y rapidez en el ámbito del desarrollo de software?

   Agilidad és la rapidez ~~que~~ con la que se toman los cambios 2.0

4. ¿Cuál es la principal orientación de las metodologías ágiles?

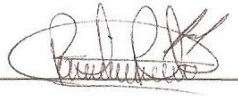
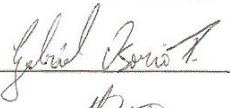
   La orientación a Objetos 0.5

5. Se necesita desarrollar un juego para Iphone, el tiempo es corto debido a la alta competitividad de este mercado, que metodología recomendaría usted que se aplique en este caso y porque?

   Creo que recomendaría XP (Extreme Programing) ya que es un método simple, sencillo y no necesita de mucha calidad ~~MF~~ ni Flexibilidad 2.0

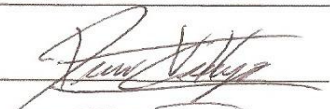
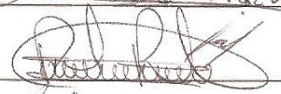
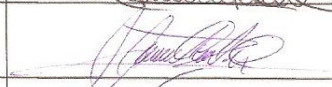
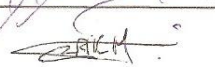
REGISTRO DE ASISTENCIA

(Miércoles 1 de diciembre de 2010)

Nombre	Firma	Correo Electrónico
Daniela Mazón S.		danny-jm2002@yahoo.es
Pablo Rocha		pablorocha_123gg1@hotmail.com.
Gabriel Osorio Poveda		gd_osorio@hotmail.com
Carlos Montenegro		carlosF_20@hotmail.es
Erik Marroquín		erik_marroquin@hotmail.com
Hector Coluche		mangix114@hotmail.com.

REGISTRO DE ASISTENCIA

(Miércoles 8 de diciembre de 2010)

Nombre	Firma
David A. Vallejo M.	
Daniela Mazón	
Pablo Rocha	
Héctor Calvache	
Erik Marroquín	
Gabriel Osorio	
Carlos Montenegro