UNIVERSIDAD NACIONAL DE SANTIAGO DEL ESTERO

FACULTAD DE CIENCIAS EXACTAS Y TECNOLOGÍAS

LICENCIATURA EN SISTEMAS DE INFORMACIÓN

TRABAJO FINAL DE GRADUACIÓN

PROPUESTA METODOLÓGICA PARA EL DESARROLLO ÁGIL DE SOFTWARE

Autor:

ADRIÁN NICOLÁS BELLUOMINI

Profesora Guía**:**

DIANA PALLIOTTO

Octubre de 2006

UNIVERSIDAD NACIONAL DE SANTIAGO DEL ESTERO
FACULTAD DE CIENCIAS EXACTAS Y TECNOLOGÍAS

TRABAJO FINAL DE GRADUACIÓN DE LA LICENCIATURA EN SISTEMAS DE INFORMACIÓN

PROPUESTA METODOLÓGICA PARA EL DESARROLLO ÁGIL DE SOFTWARE

Autor:

.....

Adrián N. Belluomini

Profesor Guía:

.....

Diana Palliotto

* ——————————————————— * ——————————————————— *

Aprobado el día ..... del mes de ..... del año 20.....

por el Tribunal integrado por

....

....

Santiago del Estero – Argentina

A mi familia

y todas las personas

que aprecio

Adrián Nicolás Belluomini

Agradecimientos

En primer lugar quiero darle gracias a Dios, por haberme dado inteligencia, constancia y fuerzas necesarias para crecer como profesional y persona.

Gracias, a mis padres por su amor, apoyo incondicional y por haberme educado y enseñado los valores éticos y morales que rigen mi vida. Gracias por que están siempre conmigo y por darme todas aquellas cosas que me ayudaron y ayudan a ser quien soy.

Gracias a mis hermanos por todo su aprecio, apoyo, comprensión y por acompañarme siempre.

Gracias a mi profesora guía, Ing. Diana Palliotto, por haberme brindado su tiempo, interés, y atinados consejos.

Quiero agradecer además a mis parientes y amigos que me alentaron y desearon lo mejor para mi en este emprendimiento, gracias por su apoyo y por estar a mi lado.

Por último quisiera agradecer a todas las personas que me apoyaron y guiaron durante mis estudios universitarios.

Adrián Nicolás Belluomini
Santiago del Estero, Argentina
Octubre de 2006

CONTENIDO

RESUMEN xiii

INTRODUCCIÓN 1

Capítulo I. PROBLEMA, HIPÓTESIS Y OBJETIVOS

I.1. INTRODUCCIÓN 3

I.2. DELIMITACIÓN DEL PROBLEMA 3

I.3. ANTECEDENTES 4

I.4. JUSTIFICACIÓN E IMPORTANCIA DE LA INVESTIGACIÓN 7

I.5. OBJETIVOS 8

I.6. CARACTERÍSTICAS DE LA INVESTIGACIÓN 8

I.7. ALCANCES 9

I.8. HIPÓTESIS Y VARIABLES 10

Capítulo II. MARCOS REFERENCIALES

II.1. INTRODUCCIÓN 11

II.2. MARCO TEÓRICO 11

II.2.1. DEFINICIÓN DE SOFTWARE 11

II.2.2. APLICACIONES SOFTWARE 12

II.2.3. PROCESO SOFTWARE 13

II.2.4. SOBRE LOS MÉTODOS ÁGILES 14

II.2.4.1. Desarrollo Guiado por Rasgos (FDD) 14

II.2.4.2. Programación Extrema (XP) 19

II.2.4.3. Administración Evolutiva de Proyectos (EVO) 23

II.2.4.4. Desarrollo Adaptativo de Software (ASD) 27
II.2.4.6. Desarrollo Liviano (LD) y Desarrollo Liviano de Software (LSD) ... 28
II.2.4.7. Scrum 30
II.2.4.8. Proceso Unificado Racional (RUP) 32
II.3. MARCO EMPÍRICO 34
II.4. MARCO METODOLÓGICO 35
II.3.1. ESTÁNDAR ISO/IEC 15504 35
II.3.2. MODELO CONSTRUCTIVO DE COSTOS (COCOMO) 36

Capítulo III. METODO MIXTO DE DESARROLLO DE SOFTWARE – MMDS

III.1. INTRODUCCIÓN 39
III.2. ROLES INTERVINIENTES 40
III.3. CICLO DE VIDA 40
III.3.1. ANÁLISIS DE REQUISITOS 42
III.3.1.1. Historias de Usuario 43
III.3.1.2. Diagramas de casos de uso (DCU) 44
III.3.2. DESARROLLO 45
III.3.3. MANTENIMIENTO 51
III.3.4. MUERTE Y REEMPLAZO 53
III.4. DESARROLLO DE UN PROYECTO SOFTWARE CON MMDS 54
III.5. APLICACIÓN DE MMDS 58

Capítulo IV. EVALUACIÓN DE RESULTADOS

IV.1. INTRODUCCIÓN 67
IV.2. COMPARACIÓN ENTRE DESARROLLO CON MMDS Y TRADICIONAL 67
IV.3. EVALUACIÓN DE CALIDAD DEL PRODUCTO OBTENIDO 68
VI.3.1. FACILIDAD DE MANTENIMIENTO 69
VI.3.2. REUSABILIDAD O CAPACIDAD DE REUTILIZACIÓN 70
VI.3.3. CORRECIÓN 71

VI.3.4. FACILIDAD DE USO 71
IV.4. DISCUSIÓN 76
IV.5. CONCLUSIÓN 76
CONCLUSIONES 79
BIBLIOGRAFÍA Y REFERENCIAS 81

RESUMEN

En el entorno económico-social de Santiago del Estero, el mercado es relativamente pequeño, y son escasas las empresas clientes que pueden absorber los costos del desarrollo de software con un proceso altamente evolucionado como, por ejemplo, RUP, o pagar licencias por el uso de metodologías como DSDM. Por lo cual, está latente la necesidad de generar propuestas metodológicas adaptables a los requerimientos y las necesidades de los clientes y usuarios de nuestro medio.

El presente trabajo propone un método mixto para el desarrollo ágil de software (Método Mixto de Desarrollo de Software - MMDS), sin descuidar la calidad, buscando ser, para desarrolladores y empresas, una alternativa ágil y económica para la generación de software de gestión de mediana a pequeña envergadura. Este método se sustenta principalmente en principios, ideas y patrones de los métodos RUP, XP, Scrum, EVO, FDD, ASD y LD, sin embargo, utiliza prácticas tradicionales como complemento. El método presentado es iterativo, interactivo e incremental. MMDS fue probado en el desarrollo de un producto software para un estudio de arquitectura de nuestro medio. Se evaluaron duración, esfuerzo y número de desarrolladores respecto de los valores estimados por el modelo COCOMO, arrojando resultados muy satisfactorios y posicionando a MMDS en un buen nivel, con tiempo y esfuerzo de desarrollo menor que los estimados por COCOMO para un método tradicional. La evaluación de la calidad del producto desarrollado con MMDS, arrojó también resultados alentadores, superando con holgura todas las expectativas y logrando la satisfacción de los clientes y usuarios. Se consiguió un producto fácil de mantener y con un nivel de reusabilidad aceptable. Además, el producto es correcto, lo que se ve respaldado por la muy buena aceptación del cliente en lo que respecta a conformidad con los requisitos. Los resultados que obtuvieron respecto de la facilidad de uso, alcanzaron valores muy altos en una escala de usabilidad de cero a cien.

Palabras claves: desarrollo de software, método mixto, software de gestión, métodos ágiles, prácticas tradicionales.

INTRODUCCIÓN

El desarrollo de software es una tarea compleja para la que existen diversas propuestas metodológicas que inciden de diferente forma en su construcción. En las últimas décadas, las notaciones de modelado, y posteriormente las herramientas, han sido muy importantes para el éxito en el desarrollo de software.

El proceso de desarrollo, en general, lleva asociada una marcada tendencia hacia el control mediante una rigurosa definición de actividades, artefactos y roles. Este es el esquema que se usa en las metodologías tradicionales, las cuales han demostrado efectividad en algunos proyectos y numerosos problemas en otros; no obstante, se trataron de mejorar incluyendo más actividades, más artefactos y más restricciones basadas en los puntos débiles que se detectaron. El resultado de estas acciones fue un proceso complejo que incide negativamente en el propio equipo encargado de llevar adelante el proyecto.

Se han descrito otras formas de solución centrando la atención en otras dimensiones como ser el factor humano o el producto software en sí. Esta es la filosofía que siguen los **Métodos Ágiles** (MA) dando mayor valor al individuo, a la interacción con el cliente y al desarrollo incremental del software.

Los MA son sin duda uno de los temas emergentes de la ingeniería del software y, en la actualidad, están mostrando efectividad en proyectos con requisitos muy cambiantes y cuando es necesario reducir el tiempo de desarrollo manteniendo la calidad.

Los MA constituyen una solución a medida para proyectos pequeños con una elevada simplificación y, a pesar de ello, aseguran la calidad del producto [6].

Actualmente, la gran expectativa generada por los MA, hacen prever una fuerte proyección en los ámbitos industrial y comercial de los mismos. Las características de los proyectos para los cuales se han pensado especialmente los MA, son representativas de un amplio rango de proyectos de desarrollo de software: aquellos en los cuales los

equipos de desarrollo son pequeños, con plazos reducidos y requisitos volátiles, donde se emplean nuevas tecnologías, etc.

El presente trabajo busca generar un método mixto para el desarrollo ágil de software, y colaborar con los desarrolladores y empresas del entorno de nuestra provincia brindándoles una alternativa ágil y económica para la generación de software de gestión.

CAPÍTULO I

PROBLEMA, HIPÓTESIS Y OBJETIVOS

I.1. INTRODUCCIÓN

En el presente capítulo se realizará la delimitación y el planteo del problema que propiciaron el inicio de este trabajo. Además, se realizará una exposición de los antecedentes, el alcance, la finalidad y los objetivos que se plantearon al comenzar el estudio. Por último, otro aspecto relevante de este capítulo es la presentación de la hipótesis que conduce al proceso de investigación.

I.2. DELIMITACIÓN DEL PROBLEMA

El proceso de desarrollo de software implica riesgos y, muchas veces, se torna difícil de controlar; por esta razón, a lo largo de los años se crearon y fueron perfeccionándose metodologías que indicaban que los proyectos exitosos son aquellos que se administran siguiendo una serie de procesos que permiten organizar y luego controlar al proyecto [16].

En la mayoría de los casos, los métodos tradicionales hacen uso de la experiencia de otras áreas de la ingeniería, cuya tradición en cuanto a proyectos exitosos es mucho más antigua; no obstante, el software posee características especiales que lo hacen particularmente distinto a otros proyectos [31].

Si bien existen proyectos software con especificaciones fijas para los cuales los métodos tradicionales se ajustan perfectamente, hay un importante volumen de proyectos software que no concuerdan con tales parámetros y, en estos casos, la escuela de gestión de proyectos software no es la más adecuada [31].

En el mundo actual, el cambio permanente en los negocios está a la orden del día; debido a esto, en los últimos años, ha surgido una corriente en la industria del software

que considera que las necesidades de los clientes son muy cambiantes, por lo que es imprescindible una rápida adaptación para no correr el riesgo de estar resolviendo el problema equivocado. Debido a que la programación libre y sin dirección no es una alternativa, especialmente cuando se trata de proyectos grandes, se propusieron nuevos métodos que en apariencia contradicen la visión tradicional de los proyectos; dichos métodos se conocen como livianos o ágiles.

A pesar de que la mayoría de los métodos ágiles (MA) permiten un desarrollo adaptable y veloz, pagan un alto precio al reducir la documentación prácticamente a cero, y es durante las fases posteriores al desarrollo donde esta carencia se hace evidente, porque la falta de documentación dificulta el mantenimiento y condiciona la futura evolución del software. En este marco surge la necesidad de plantear una combinación de métodos que busque un equilibrio entre el desarrollo tradicional y el ágil.

I.3. ANTECEDENTES

El desarrollo de software tiene más de cuatro décadas durante las cuales se realizaron diversas investigaciones sobre la mejor forma de construirlo. Tal es así que, a fines del siglo XX, había un abanico de metodologías, procesos y modelos disponibles para la construcción de software:

- el modelo en cascada o lineal-secuencial, que es el más convencional;
- el modelo en "V", desarrollado en 1992 por el Ministerio de Defensa de Alemania, el cual consiste básicamente en coordinar el modelo en cascada con iteraciones;
- el modelo de desarrollo rápido o RAD, que se caracteriza por un modelo lineal secuencial con ciclos de desarrollo breves, presentado en 1991 por J. Martin, con sus respectivas mejoras en 1994 realizadas por Kerr/Hunter y en 1996 por McConnell;
- el modelo incremental (*staged delivery*) introducido en 1996 por McConnell, se caracteriza por tener sus fases tempranas en cascada y sus fases ulteriores descompuestas en etapas;
- el modelo en espiral básico presentado por Barry Boehm en 1988, y mejorado en 1998 también por Barry Boehm bajo el nombre de espiral win-win, que

consiste en un modelo iterativo de desarrollo incremental en el cual se aplica teoría-W[1] a cada etapa;

- el modelo de desarrollo concurrente cuya versión de origen se da a conocer en 1994 por Davis y Sitaram, se trata de un modelo cíclico con análisis de estado.

No se deben olvidar otros modelos importantes como el modelo iterado con prototipado presentado por Brooks en 1975, y un conjunto de modelos iterativos o evolutivos (por ejemplo, los modelos basados en componentes, etc.) que en un primer momento convivían apaciblemente con los restantes, aunque cada uno de ellos se originaba en la crítica o en la percepción de las limitaciones de algún otro. Algunos eran prescriptivos, otros descriptivos. Algunos de los modelos incluían iteraciones, incrementos, recursiones o bucles de retroalimentación; y algunos se preciaban también de rápidos, adaptables y expeditivos.

A comienzos del siglo XXI varios de estos modelos ya estaban desacreditados, o empezaban a desacreditarse, y se dieron las condiciones para que irrumpieran los MA. Kent Beck, el líder de XP, comentó que sería difícil encontrar una reunión de anarquistas organizacionales más grande que la de los diecisiete proponentes de los MA que se reunieron, en marzo de 2001 en Salt Lake City, para discutir los nuevos métodos. Lo que surgió de esta reunión fue el "Manifiesto para el Desarrollo Ágil de Software" [5]. Fue todo un acontecimiento y, aunque es muy reciente, tiene la marca de los sucesos que quedan en la historia. Bob Charette y Ken Orr, independientemente, han comparado el sentido histórico del Manifiesto con las Tesis de Lutero; ambos desencadenaron guerras dogmáticas [33].

En el encuentro hubo representantes de modelos muy distintos, pero todos compartían la idea de que era necesaria una alternativa a los métodos consagrados basados en una gestión burocrática, una documentación exhaustiva y procesos de peso pesado. Resumían su punto de vista argumentando que el movimiento ágil no es una "anti-metodología":

> "*De hecho, muchos de nosotros deseamos devolver credibilidad a la palabra metodología. Queremos recuperar un equilibrio. Promovemos el modelado, pero no con el fin de archivar algún diagrama en un polvoriento repositorio corporativo. Promovemos la documentación, pero no cientos de*

[1] Teoría-W proporciona un entorno de trabajo para la gestión de proyectos orientado a la reconciliación de intereses opuestos (Boehm y Ross, 1989).

páginas en tomos que nunca se mantienen y rara vez se usan. Planificamos, pero reconocemos los límites de la planificación en un entorno turbulento" [5].

Aunque los creadores e impulsores de los MA más populares han suscrito el manifiesto ágil y declarado sus doce principios que diferencian a un proceso ágil de uno tradicional, cada método tiene características propias y hace hincapié en algunos aspectos más específicos. La mayoría de ellos ya se estaban utilizando con éxito en proyectos reales antes del mencionado Manifiesto, pero les faltaba una mayor difusión y reconocimiento. Dichos métodos son:

- **SCRUM**. Desarrollado por Ken Schwaber, Jeff Sutherland y Mike Beedle. Define un marco para la gestión de proyectos que se ha utilizado con éxito durante los últimos 10 años.
- **Programación Extrema** (XP - eXtreme Programming). Presentada por Kent Beck en 1999. Es un método ágil centrado en potenciar las relaciones interpersonales como clave para el éxito en desarrollo de software. XP se define como especialmente adecuada para proyectos con requisitos imprecisos y muy cambiantes, y donde existe un alto riesgo técnico.
- **Crystal Methodologies**. Es un conjunto de metodologías para el desarrollo de software, que se caracterizan por estar centradas en las personas que componen el equipo y la reducción al máximo del número de artefactos producidos. Han sido desarrolladas por Alistair Cockburn.
- **Dynamic Systems Development Method** (**DSDM**). Define el marco para desarrollar un proceso de producción de software. Nace en 1994 de la mano de Jennifer Stapleton. Sus principales características son: es un proceso iterativo e incremental, y el equipo de desarrollo y el usuario trabajan juntos. Fue aplicado exitosamente por Shell, Loyds Bank Insurance Services, British Telecom, British Airways, Deutsche Bank, Hewlett-Packard, Renault (en Los Ángeles), y otras 500 compañías inglesas.
- **Adaptive Software Development** (**ASD**). Su impulsor es Jim Highsmith. Sus principales características son: iterativo, orientado a los componentes software más que a las tareas y tolerante a los cambios. Entre las empresas que han aplicado ASD se encuentran AS Bank de Nueva Zelanda, CNET,

GlaxoSmithKline, Landmark, Nextel, Nike, Phoenix International Health, Thoughworks y Microsoft.

- **Feature-Driven Development** (**FDD**). Se centra en las fases de diseño e implementación del sistema partiendo de una lista de características que debe reunir el software. Sus impulsores son Jeff De Luca y Peter Coad. Actualmente FDD es una marca registrada de Nebulon CO.
- **Lean Development** (**LD**). Definida por Bob Charette. Su principal característica es introducir un mecanismo para implementar los cambios que surgen en el sistema. Existen abundantes casos de éxito documentados empleando LD, la mayoría en Canadá. Algunos de ellos son los de Canadian Pacific Railway, Autodesk y PowerEx Corporation.

I.4. JUSTIFICACIÓN E IMPORTANCIA DE LA INVESTIGACIÓN

No hace falta estar completamente de acuerdo con las premisas más extremas de los MA para encontrar aspectos cuestionables en todos y cada uno de los modelos clásicos. Incluso los textos que sirven de introducción a la Ingeniería de Software incluyen la lista de limitaciones bien conocidas de la mayor parte de ellos, siendo el modelo en cascada el más castigado. Se le objeta su rigidez, su insistencia por exigir una estipulación previa y completa de los requerimientos, su modelo inapropiado de correcciones, su progresivo distanciamiento de la visión del cliente y su lentitud para liberar piezas ejecutables. En los últimos dos o tres años de la década de 1990, las críticas a los métodos convencionales aumentaron en intensidad y detonaron por todas partes.

Es así que los MA no surgieron porque sí, sino que estuvieron motivados por una conciencia particularmente aguda de la crisis del software, por la responsabilidad que se imputa a las grandes metodologías en la gestación de esa crisis, y por el propósito de articular soluciones [38].

Por otra parte, es importante analizar como es el entorno económico-social de la provincia de Santiago del Estero, y otras con características similares, en donde el mercado de desarrollo de software tiene sus particularidades que inciden fuertemente en la forma en que se desarrolla el mismo. En nuestra provincia, este mercado es relativamente pequeño, y las empresas clientes que pueden absorber los costos del

desarrollo de software con un proceso altamente evolucionado como, por ejemplo, RUP (Rational Unified Process), o pagar licencias por el uso de metodologías como DSDM, son sólo unas pocas. También se debe considerar que cualquiera sea el cliente que opte por el desarrollo de software a medida, tiene en mente tres objetivos para el software: que organice y mejore el tratamiento de la información; que su desarrollo sea rápido; y que cueste lo menos posible.

Bajo estas premisas, surge la necesidad de generar propuestas metodológicas adaptables a los requerimientos y las necesidades de los clientes y usuarios de nuestro medio, de manera que su impacto incentive el desarrollo rápido de software, con una buena calidad y a un bajo costo, y se generalice en el entorno comercial de nuestra provincia.

I.5. OBJETIVOS

La finalidad de este trabajo es ofrecer una propuesta metodológica para la construcción ágil de software.

- ***Objetivos Generales***
  - Brindar a los programadores y analistas una guía metodológica para la construcción de software.
  - Proporcionar a las organizaciones un método que les permita desarrollar software a bajo costo, rápidamente, y sin descuidar la calidad.
- ***Objetivos Específicos***
  - Reconocer y destacar la importancia de contar con una metodología ágil.
  - Explorar las diversas alternativas ágiles.
  - Contrastar las prácticas ágiles y tradicionales.
  - Presentar a las organizaciones una nueva forma de percibir la construcción del software, incrementando la interacción entre clientes y desarrolladores.

I.6. CARACTERÍSTICAS DE LA INVESTIGACIÓN

La presente investigación es de carácter exploratorio, descriptivo y explicativo.

En la etapa inicial del proyecto, la investigación será de carácter exploratorio. Se orientará a comprender y a lograr familiaridad con los MA existentes, con el objetivo de ampliar la esfera de alternativas, y con la esperanza de identificar e incluir las más adecuadas a la hora de comenzar con cualquier descripción metodológica.

Superada la etapa inicial, la investigación tomará un carácter descriptivo, ya que el marco teórico y la descripción del problema se construirán, registrarán e interpretarán tomando como referencia la información que aportan diferentes autores que han tratado el desarrollo ágil y tradicional de software, para sentar las bases que darán solvencia a la presente investigación. Existe una amplia bibliografía que describe el abanico de prácticas necesarias para la construcción ágil de software; sin embargo, esta investigación se centrará en aquellas que se consideren adecuadas a nuestro entorno.

La investigación tendrá además aspectos explicativos ya que se pretende desarrollar una propuesta metodológica para la construcción ágil de software, en donde la interacción cliente-desarrollador juegue un papel importante.

I.7. ALCANCES

El desarrollo de software posee un conjunto de actividades que pueden ser contempladas desde dos puntos de vista: por un lado, desde el punto de vista de las entidades u organizaciones en donde se trata de crear y gestionar software enfatizando la calidad, la agilidad y el bajo costo; y, por otro lado, desde el punto de vista de los desarrolladores en general, donde es necesaria una guía para las distintas actividades de desarrollo y mantenimiento del software, adaptándolas a las características concretas de cada proyecto y su entorno para que se apliquen en la práctica.

En el transcurso de este trabajo se realizará una propuesta metodológica para el desarrollo de software de gestión, que intente cubrir las necesidades de las organizaciones y los desarrolladores de nuestro medio en lo referente al desarrollo de software, pretendiendo brindarles una herramienta útil con características ágiles y de bajo costo. El método propuesto se probará en el desarrollo del software para un estudio de arquitectura de la ciudad de Santiago del Estero, buscando alcanzar la etapa de implantación del mismo; no obstante, se pretende dirigir la aplicación del método al software de gestión en general.

I.9. HIPOTESIS

"El método propuesto permitirá reducir tiempo y costos en el desarrollo de software de gestión, asegurando la calidad del mismo".

Operacionalización de Variables

Variable independiente:	X: método propuesto
Variable dependiente:	Y: reducción de tiempo y costos en el desarrollo de software de gestión, asegurando la calidad del mismo.

Para realizar la comprobación de la hipótesis se procederá a enmarcar el método propuesto dentro de la normativa establecida en ISO/IEC 15504 (1998), que es un estándar para la evaluación y la mejora de procesos software, y provee un modelo conceptual y un marco para evaluación, validación, optimización y certificación del proceso de desarrollo o construcción de software. Por otro lado, la estimación de tiempo y costos se realizará con el método COCOMO.

Para evaluar la calidad del producto software se usarán como atributos la corrección, la facilidad de mantenimiento y la facilidad de uso. El atributo corrección se medirá según la formula: ***defectos / KLDC***, donde *defecto* es una falta verificada de conformidad con los requisitos y *KLDC* es una medida que representa miles de líneas de código. La facilidad de mantenimiento se medirá usando como medida de referencia el tiempo medio de cambio (*TMC*), que es el promedio del tiempo que se tarda en analizar la petición del cambio, diseñar e implementar una modificación adecuada, probar dicha modificación y distribuir el cambio a todos los usuarios [37]. Finalmente, la facilidad de uso se cuantificará en forma subjetiva sobre la disposición del usuario hacia el sistema, para lo cual se implementaran y evaluaran cuestionarios.

CAPÍTULO II

MARCOS REFERENCIALES

II.1. INTRODUCCIÓN

En el presente capítulo se ofrece una breve reseña de los marcos referenciales (teórico, metodológico y empírico). Tiene como fin introducir el cuerpo de teorías, conceptos, referencias y supuestos donde se inscribe el problema. Se presentará el universo de estudio y el caso de aplicación sobre el que se implementará el método propuesto; también se introducirán los procesos y las técnicas que se usarán posteriormente para la medición de las variables, lo que permitirá la corroboración de la hipótesis de la investigación.

II.2. MARCO TEÓRICO

II.2.1. DEFINICIÓN DE SOFTWARE

Existe una gran cantidad de definiciones de *software*, pero probablemente la definición más formal es la que se atribuye al IEEE en su estándar 729: "*la suma total de los programas de cómputo, procedimientos, reglas, documentación y datos asociados que forman parte de las operaciones de un sistema de cómputo*" [24]. Al considerar esta definición, el concepto de software va más allá de los programas de cómputo en sus distintas versiones: código fuente, binario o código ejecutable. Pressman, en su obra "*Ingeniería del Software*", expresa que a pesar de cuán completa sea cualquier definición es necesario algo más que una definición formal para entender el concepto de software; por dicha razón afirma que para su comprensión es importante examinar sus características, las cuales lo diferencian de otras cosas que pueden ser construidas por los hombres [36].

Estas características son:

- *El software se desarrolla, no se fabrica en un sentido clásico*. Ambas actividades, construcción de software y hardware, requieren la construcción de un producto, pero los métodos difieren.
- *El software no se estropea*. Un producto software bien construido no se estropea, cada fallo de software indica un error en el diseño o en el proceso mediante el cual se tradujo el diseño a código de máquina, no hay piezas de repuesto para esto, por lo tanto el mantenimiento de software es considerablemente más complejo que el de hardware.
- *La mayoría del software se construye a medida*. Hasta hace algunos años esta era una característica muy fuerte, la mayoría del software se construía a medida de algún sistema (organizaciones, equipos de hardware, etc.) en particular, y a diferencia del hardware que hasta ahora puede comprarse y ensamblarse por piezas, a menudo el software sólo podía comprarse como un todo ya desarrollado, una unidad completa y no como componentes para ensamblarse. Con el correr de los años esta característica fue perdiendo fuerza y, aunque si bien no ha desaparecido, los desarrolladores de software ya han comprendido el papel crítico de la reutilización de componentes software, ya que esta es una característica de calidad deseada y necesaria en nuestros días.

II.2.2. APLICACIONES SOFTWARE

Conforme aumenta la complejidad del software, es más difícil establecer categorías genéricas significativas para sus aplicaciones, las siguientes áreas del software indican la amplitud de las aplicaciones potenciales [36]:

- *Software de sistemas*. Es un conjunto de programas que se han escrito para servir a otros programas. Este tipo de software se caracteriza por una fuerte interacción con el hardware de la computadora, una gran utilización por múltiples usuarios, una operación concurrente que requiere una avanzada planificación, compartición de recursos y gestión de procesos, unas estructuras de datos complejas y múltiples interfaces externas.

- *Software de tiempo real*. Este software mide, analiza y controla sucesos del mundo real conforme ocurren. Su característica más importante es el tiempo de respuesta, ya que éste se encuentra dentro de estrictos límites.
- *Software de gestión*. El procesamiento de información comercial constituye el área más grande de aplicación del software. Este tipo de software accede a una o más bases de datos que contienen información comercial. Las aplicaciones en esta área reestructuran los datos existentes para facilitar las operaciones comerciales o gestionar la toma de decisiones, realizando también cálculos interactivos.
- *Software de ingeniería y científico*. Se caracteriza por los algoritmos de manejo de números; caen dentro de esta categoría de software los sistemas CAD, los programas de simulación, etc.
- *Software empotrado*. Se denomina de esta manera al software incorporado a la memoria de algunos productos denominados "inteligentes". Este tipo de software está íntimamente ligado al software de tiempo real, de hecho, una considerable parte de éste es empotrado.
- *Software de computadoras personales*. Dentro de esta categoría están los procesadores de texto, editores gráficos, planillas de cálculo, software de entretenimientos, etc.
- *Software de inteligencia artificial*. Este software hace uso de algoritmos no numéricos para resolver problemas complejos para los cuales no son adecuados los cálculos y el análisis directo.

II.2.3. PROCESO SOFTWARE

Jacobson [26] define al proceso de la Ingeniería de Software como “un conjunto de etapas parcialmente ordenadas con la intención de lograr un objetivo, en este caso, la obtención de un producto de software de calidad”. Booch [27], por su parte, expresa que el proceso de desarrollo de software es “el conjunto de actividades necesarias para transformar los requisitos de un usuario en un sistema software”.

El proceso de desarrollo de software requiere un conjunto de conceptos, una metodología y un lenguaje propio. A este proceso también se le llama "*ciclo de vida del*

software" que comprende cuatro grandes fases: concepción, elaboración, construcción y transición. La concepción define el alcance del proyecto y desarrolla un caso de negocio. La elaboración establece un plan del proyecto, especifica las características y fundamenta la arquitectura. La construcción crea el producto y la transición transfiere el producto a los usuarios.

La definición de un ciclo de vida facilita el control sobre los tiempos en que es necesario aplicar recursos de todo tipo (personal, equipos, suministros, etc.) al proyecto. Si el proyecto incluye subcontratación de partes a otras organizaciones, el control del trabajo subcontratado se facilita en la medida en que esas partes encajen bien en la estructura de las fases. El control de calidad también se ve facilitado si la separación entre fases se hace corresponder con puntos en los que ésta deba verificarse (mediante comprobaciones sobre los productos parciales obtenidos).

II.2.4. SOBRE LOS MÉTODOS ÁGILES

Actualmente existe una amplia diversidad de MA, algunos de ellos llevados a la práctica exitosamente; otros, sin embargo, no corrieron la misma suerte. Algunos contemplan todo el proceso software, otros sólo parte de él; sin embargo, es importante mencionar que a pesar de que ciertos métodos no son completos, permiten su combinación con otros, sean ágiles o no, logrando así métodos híbridos.

A continuación se describen algunos MA de interés para el trabajo, ya que el método que se propondrá utiliza algunas de sus prácticas, métodos, ideas y filosofías.

II.2.4.1. Desarrollo Guiado por Rasgos (Feature Driven Development – FDD)

FDD, es un MA iterativo y adaptativo. A diferencia de otros MA, no cubre todo el ciclo de vida sino sólo las fases de diseño y construcción, y se considera adecuado para proyectos mayores y de misión crítica. FDD es, además, marca registrada de una empresa, Nebulon Pty. Aunque hay coincidencias entre la programación orientada por rasgos (Feature Oriented Programming[1] - FOP) y el desarrollo guiado por rasgos, FDD no necesariamente implementa FOP.

[1] **Feature Oriented Programming** (FOP) es una técnica de programación guiada por rasgos o características y centrada en el usuario, no en el programador. Los rasgos son pequeños ítems escritos en un lenguaje sencillo para que pueda ser comprendido por cualquiera. Esta técnica tiene como objetivo sintetizar un programa conforme a los rasgos requeridos. En un desarrollo en términos de FOP, los objetos se organizan en módulos o capas de acuerdo a rasgos [38].

FDD no requiere un modelo específico de proceso y se complementa con otras metodologías. Enfatiza cuestiones de calidad y define claramente entregas tangibles y formas de evaluación del progreso. Fue implementado por primera vez en el Singapore Project por Jeff DeLuca quien había sido contratado para salvar un sistema muy complicado para el cual el contratista anterior había producido, luego de dos años, 3500 páginas de documentación y ninguna línea de código. Naturalmente, el proyecto basado en FDD fue todo un éxito, y permitió fundar el método en un caso real de misión crítica.

FDD cuenta con seis principios simples [21]:

- Se requiere un sistema para construir sistemas si se pretende escalar a proyectos grandes.
- Un proceso simple y bien definido trabaja mejor.
- Los pasos de un proceso deben ser lógicos y su mérito inmediatamente obvio para cada miembro del equipo.
- Vanagloriarse del proceso puede impedir el trabajo real.
- Los buenos procesos van hasta el fondo del asunto, de modo que los miembros del equipo se puedan concentrar en los resultados.
- Los ciclos cortos, iterativos, orientados por rasgos son mejores.

Existen tres categorías de roles en FDD, estas son: roles claves, roles de soporte y roles adicionales.

Hay seis roles claves de un proyecto y son [38]:

(1) Administrador del proyecto, quien tiene la última palabra en materia de visión, cronograma y asignación del personal.

(2) Arquitecto jefe, puede dividirse en arquitecto de dominio y arquitecto técnico.

(3) Administrador del desarrollo, que puede combinarse con arquitecto jefe o administrador del proyecto.

(4) Programador jefe, que participa en el análisis de requerimientos y selecciona rasgos del conjunto a desarrollar en la siguiente iteración.

(5) Propietarios de clases, que trabajan bajo la guía del programador jefe en diseño, codificación, prueba y documentación, repartidos por rasgos.

(6) Experto de dominio, que puede ser un cliente, patrocinador, analista de negocios o una mezcla de todo eso.

Es importante destacar que un miembro de un equipo puede tener varios roles a cargo, y un solo rol puede ser compartido por varias personas.

FDD cuenta con cinco son los roles de soporte, ellos son [38]:

(1) Administrador de entrega, que controla el progreso del proceso revisando los reportes del programador jefe y manteniendo reuniones breves con él; reporta al administrador del proyecto.

(2) Abogado/gurú de lenguaje, que conoce a la perfección el lenguaje y la tecnología.

(3) Ingeniero de construcción, que se encarga del control de versiones de los ensamblados (builds) y publica la documentación.

(4) Herramientista, que construye herramientas ad-hoc o mantiene bases de datos y sitios Web.

(5) Administrador del sistema, que controla el ambiente de trabajo y la productividad el sistema cuando se lo entrega.

Los tres roles adicionales serían los de verificadores, encargados del despliegue y escritores técnicos.

Es importante destacar que un miembro de un equipo puede tener varios roles a cargo, y un solo rol puede ser compartido por varias personas.

El proceso FDD consta de cuatro etapas secuenciales (figura 2.1) durante los cuales se diseña y construye el sistema, no obstante, la última etapa puede ser considerada como dos etapas separadas.

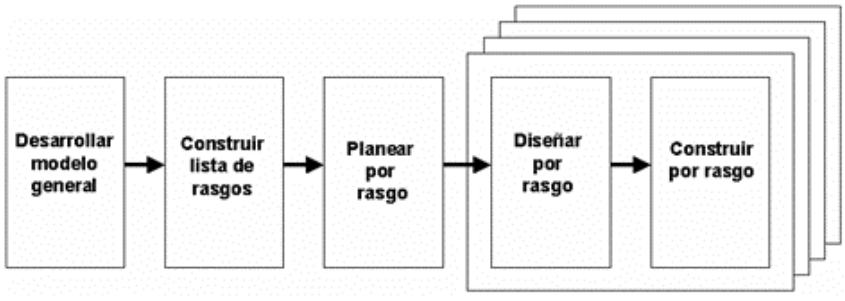


Figura 2.1. Proceso FDD

La parte iterativa soporta desarrollo ágil con rápidas adaptaciones a cambios en los requerimientos y las necesidades del negocio. Cada fase del proceso tiene un criterio de entrada, tareas, pruebas y un criterio de salida. Típicamente, la iteración de un rasgo insume de una a tres semanas. Las fases son [34]:

(1) *Desarrollo de un modelo general.* Este desarrollo requiere como punto de partida el contexto y los requerimientos del sistema a construir. Los expertos de dominio presentan un ensayo (walkthrough) en el que los miembros del equipo y el arquitecto principal se informan de la descripción de alto nivel del sistema. El dominio general se subdivide en áreas más específicas y se define un ensayo más detallado para cada uno de los miembros del dominio. Luego de cada ensayo, un equipo de desarrollo trabaja en pequeños grupos para producir modelos de objetos de cada área de dominio. Simultáneamente, se construye un gran modelo general para todo el sistema.

(2) *Construcción de la lista de rasgos*. Los ensayos, modelos de objetos y documentación de requerimientos proporcionan la base para construir una amplia lista de rasgos. Los rasgos son pequeños ítems escritos en un lenguaje sencillo para que pueda ser comprendido por cualquiera. Las funciones se agrupan conforme a diversas actividades en áreas de dominio específicas, descomponiendo aquellos rasgos que requieran más de diez días en otros más pequeños. Por último, la lista de rasgos es revisada por los usuarios y patrocinadores para asegurar su validez y exhaustividad.

(3) *Planeamiento por rasgo*. En este proceso se crea un plan de alto nivel, en el que los conjuntos de rasgos se ponen en secuencia, conforme a su prioridad y dependencia, y se asigna a los programadores jefes. Las listas se priorizan en secciones que se llaman paquetes de diseño. Por ultimo, se asignan las clases definidas en la selección del modelo general a programadores individuales, o sea propietarios de clases, y se pone fecha para la entrega de los conjuntos de rasgos.

(4) *Diseño por rasgo y Construcción por rasgo*. Se selecciona un pequeño conjunto de rasgos del conjunto y los propietarios de clases seleccionan los correspondientes equipos dispuestos por rasgos. Luego, se procede iterativamente hasta que se producen los rasgos seleccionados. Una iteración

puede tomar de unos pocos días a un máximo de dos semanas. Puede haber varios grupos trabajando en paralelo. El proceso iterativo (figura 2.2) incluye inspección de diseño, codificación, prueba de unidad, integración e inspección de código. Luego de una iteración exitosa, los rasgos completos se promueven al ensamblado principal. Este proceso puede demorar una o dos semanas en implementarse.

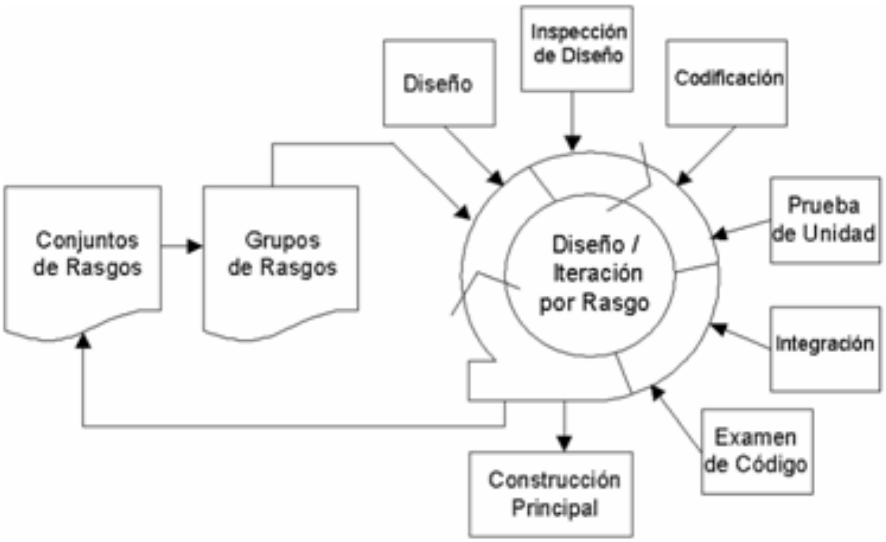


Figura 2.2. Proceso iterativo de FDD [34]

FDD tiene un conjunto de prácticas canónicas, las cuales distan de ser nuevas, pero se combinan de una manera original; ellas son [38]:

- Modelado de objetos del dominio, resultante en un framework[2] cuando se agregan los rasgos. Esta forma de modelado descompone un problema mayor en otros menores; el diseño y la implementación de cada clase u objeto es un problema pequeño a resolver. Cuando se combinan las clases completas, constituyen la solución al problema mayor.
- Desarrollo por rasgo. Hacer simplemente que las clases y los objetos funcionen no refleja lo que el cliente pide. El seguimiento del progreso se realiza mediante el examen de pequeñas funcionalidades descompuestas y funciones valoradas por el cliente.

[2] **Framework**: representa una arquitectura de software que modela las relaciones generales de las entidades del dominio; provee una estructura y una metodología de trabajo la cual extiende o utiliza las aplicaciones del dominio [45].

- Propiedad individual de clases (código). Cada clase tiene una sola persona nominada como responsable por su consistencia, performance e integridad conceptual.
- Equipos de rasgos, pequeños y dinámicamente formados. La existencia de un equipo garantiza que cada decisión se tome teniendo en cuenta múltiples alternativas.
- Inspección. Se refiere al uso de los mejores mecanismos conocidos para la detección de defectos.
- Ensamblados regulares. Los ensamblados forman la base a partir de la cual se van agregando nuevos rasgos.
- Administración de configuración. Permite realizar el seguimiento histórico de las últimas versiones completas de código fuente.
- Reporte de progreso. Se comunica el progreso del proyecto a todos los niveles organizacionales necesarios.

Algunos agilistas afirman que FDD es demasiado jerárquico para ser un método ágil; otros críticos expresan que la ausencia de procedimientos detallados de prueba en FDD es llamativa e impropia. Los promotores del método aducen que las empresas ya tienen implementadas sus herramientas de prueba, pero subsiste el problema de adecuar las mismas para su uso en FDD.

Un rasgo notable de FDD es que no exige la presencia del cliente.

II.2.4.2. Programación Extrema (Extreme Programming XP)

La programación extrema es un método reciente en el desarrollo de software. La filosofía de XP es satisfacer totalmente las necesidades del cliente y, por eso, lo integra como una parte más del equipo de desarrollo.

Inicialmente XP se creó para el desarrollo de aplicaciones en las que el cliente no sabe muy bien lo que quiere, lo que provoca un cambio constante en los requisitos que debe cumplir la aplicación.

XP está diseñada para el desarrollo de aplicaciones que requieren un grupo pequeño de programadores, donde la comunicación es más fácil que en grupos de

desarrollo grandes, ya que la misma es un punto importante y debe realizarse entre los programadores, los jefes de proyecto y los clientes.

Las características esenciales de este método son las siguientes [2]:

- *Comunicación*. Los programadores están en constante comunicación con los clientes para satisfacer sus requisitos y responder rápidamente a los cambios de los mismos. Muchos problemas que surgen en los proyectos se deben a que, después de concretar los requisitos que debe cumplir el programa, no hay una revisión de los mismos, pudiendo dejar olvidados puntos importantes.
- *Simplicidad*. Codificación y diseños simples y claros. Muchos diseños son tan complicados que cuando se quieren ampliar resulta imposible hacerlo y se tienen que desechar y partir de cero.
- *Realimentación (Feedback)*. Mediante la realimentación se ofrece al cliente la posibilidad de conseguir un sistema apto para sus necesidades, ya que se le va mostrando el proyecto a tiempo para poder cambiarlo y retroceder a una fase anterior para rediseñarlo a su gusto.
- *Coraje*. Se debe tener coraje o valentía para cumplir los tres puntos anteriores. Hay que tener valor para comunicarse con el cliente y enfatizar algunos puntos, a pesar de que esto pueda dar sensación de ignorancia por parte del programador, hay que tener coraje para mantener un diseño simple y no optar por el camino más fácil y, por último, hay que tener valor y confiar en que la realimentación sea efectiva.

XP consta de cuatro fases; ellas son [21]:

(1) *Planificación del proyecto*

- Historias de usuario. El primer paso de cualquier proyecto que siga la metodología XP es definir las historias de usuario con el cliente. Las historias de usuario tienen la misma finalidad que los casos de uso pero con algunas diferencias. Cuando llega la hora de implementar una historia de usuario, el cliente y los desarrolladores se reúnen para concretar y detallar lo que tiene que hacer dicha historia. El tiempo de desarrollo ideal para una historia de usuario es entre una y tres semanas.

- Planificación de entregas (Release planning). Después de tener ya definidas las historias de usuario es necesario crear un plan de entregas, donde se indiquen las historias de usuario que se crearán para cada versión del programa y las fechas en las que se entregarán estas versiones. En esta planificación, los desarrolladores y clientes establecen los tiempos de implementación ideales de las historias de usuario, la prioridad con la que se implementarán y las historias que se implementarán en cada versión del programa.
- Iteraciones. Todo proyecto de XP se ha de dividir en iteraciones de aproximadamente tres semanas de duración.
- Velocidad del proyecto. La velocidad del proyecto es una medida que representa la rapidez con la que se desarrolla el proyecto; estimarla es muy sencillo, basta con contar el número de historias de usuario que se pueden implementar en una iteración. De esta forma, se sabrá el cupo de historias que se pueden desarrollar en las distintas iteraciones. Esto se usa para controlar que todas las tareas se puedan desarrollar en el tiempo del que dispone la iteración.
- Programación en pareja. XP aconseja la programación en parejas pues incrementa la productividad y la calidad del software desarrollado.
- Reuniones diarias. Es necesario que los desarrolladores se reúnan diariamente y expongan sus problemas, soluciones e ideas de forma conjunta. Las reuniones tienen que ser fluidas y todo el mundo tiene que tener voz y voto.

(2) *Diseño*

- Diseños simples. XP sugiere que hay que conseguir diseños simples y sencillos ya que, a la larga, costará menos tiempo y esfuerzo desarrollar.
- Glosarios de términos. Usar glosarios de términos y una correcta especificación de los nombres de métodos y clases ayudará a comprender el diseño y facilitará sus posteriores ampliaciones y la reusabilidad del código.

- Riesgos. Si surgen problemas potenciales durante el diseño, XP sugiere utilizar una pareja de desarrolladores para que investiguen y reduzcan al máximo el riesgo que supone ese problema.
- Funcionalidad extra. Nunca se debe añadir funcionalidad extra al programa aunque se piense que en un futuro se utilizará.
- Refactorizar. Refactorizar es mejorar y modificar la estructura y la codificación de códigos ya creados sin alterar su funcionalidad.
- Tarjetas CRC. El uso de las tarjetas CRC (Class, Responsabilities and Collaboration) permiten al programador centrarse y apreciar el desarrollo orientado a objetos olvidándose de los malos hábitos de la programación procedimental clásica.

(3) *Codificación*

Al codificar una historia de usuario la presencia del cliente es muy importante, ya que éste es el que crea las historias de usuario y negocia los tiempos en los que se implementarán. Antes del desarrollo de cada historia de usuario, el cliente debe especificar detalladamente lo que ésta hará y también tendrá que estar presente cuando se realicen las pruebas que verifiquen que la historia implementada cumple la funcionalidad especificada.

La codificación debe hacerse atendiendo a estándares de codificación ya creados. Programar bajo estándares mantiene el código consistente y facilita su comprensión y escalabilidad[3].

XP opta por la programación en parejas ya que permite un código más eficiente y con una gran calidad.

XP sugiere un modelo de trabajo usando repositorios de código, donde las parejas de programadores publican cada pocas horas sus códigos implementados y corregidos junto a las pruebas que deben pasar. De esta forma el resto de los programadores que necesiten códigos ajenos trabajarán siempre con las últimas versiones. Para mantener un código consistente, la

[3] **Escalabilidad**: es la capacidad de un sistema informático de adaptarse a un número de usuarios cada vez mayor, sin perder calidad en los servicios. En general, se podría definir como la capacidad del sistema informático de cambiar su tamaño o configuración para adaptarse a las circunstancias cambiantes [45].

publicación de código en un repositorio es una acción exclusiva para cada pareja de programadores.

La optimización del código siempre se debe dejar para el final. Hay que hacer que funcione y que sea correcto, más tarde se puede optimizar.

(4) *Pruebas*

Uno de los pilares de la metodología XP es el uso de pruebas para comprobar el funcionamiento del código que se vaya implementando.

El uso de las pruebas en XP es el siguiente:

- Se deben crear las aplicaciones que realizarán las pruebas con un entorno de desarrollo específico para pruebas.
- Hay que someter a pruebas las distintas clases del sistema omitiendo los métodos más triviales.
- Se deben crear las pruebas que pasarán los códigos antes de implementarlos; es importante crear las pruebas antes que el código.

Un punto importante es crear pruebas que no tengan ninguna dependencia del código que en un futuro evaluará. Hay que crear las pruebas abstrayéndose del futuro código, de esta forma aseguraremos la independencia de las pruebas respecto al código que evalúa.

El uso de las pruebas es adecuado para observar la refactorización. Las pruebas permiten verificar que un cambio en la estructura de un código no tiene porqué cambiar su funcionamiento.

II.2.4.3 Administración Evolutiva de Proyectos (Evolutionary Proyect Managment-EVO)

EVO fue creado por Tom Gilb y es el método iterativo ágil más antiguo. Consta de diez principios fundamentales que son [38]:

(1) Se entregarán temprano, y con frecuencia, resultados verdaderos, de valor para los participantes reales.

(2) El siguiente paso de entrega de EVO será el que proporcione el mayor valor para el participante en ese momento.

(3) Los pasos de EVO entregan los requerimientos especificados de manera evolutiva.

(4) No se puede saber cuáles son los requerimientos por anticipado, pero se puede descubrirlos más rápidamente intentando proporcionar valor real a participantes reales.

(5) EVO es ingeniería de sistemas holística (todos los aspectos necesarios del sistema deben ser completos y correctos) y con entrega a un ambiente de participantes reales (no es sólo sobre programación, es sobre satisfacción del cliente).

(6) Los proyectos de EVO requieren una arquitectura abierta, porque las ideas del proyectocambiarán tan a menudo como se necesite hacerlo, para entregar realmente valor a los participantes.

(7) El equipo de proyecto de EVO concentrará su energía como equipo hacia el éxito del paso actual. En este paso tendrán éxito o fracasarán todos juntos. No gastarán energías en pasos futuros hasta que hayan dominado los pasos actuales satisfactoriamente.

(8) EVO tiene que ver con aprendizaje a partir de la dura experiencia, tan rápido como se pueda: qué es lo que verdaderamente funciona, qué es lo que realmente entrega valor. EVO es una disciplina que hace confrontar los problemas tempranamente, pero que permite progresar rápido cuando probadamente se lo ha hecho bien.

(9) EVO conduce a una entrega temprana, a tiempo, tanto porque se lo ha priorizado así desde el inicio, y porque se aprende desde el principio a hacer las cosas bien.

(10) EVO debería permitir poner a prueba nuevos procesos de trabajo y a deshacer tempranamente de los que funcionan mal.

Kai Gilb [18] establece en su publicación "Evolutionary Project Management & Product Development", cinco elementos mayores:

- *Metas, Valores y Costos*. Son los objetivos, las metas estratégicas, los requerimientos, los propósitos, los fines, las ambiciones, las cualidades y las intenciones.
- *Soluciones*. El banco de ideas sobre la forma de alcanzar metas y valores dentro de un rango de costos.

- *Estimación de Impacto*. Se trata de averiguar si se tienen ideas adecuadas para lograr las metas dentro de los costos.
- *Plan Evolutivo*. Inicialmente se plantea una idea general de la secuencia a desarrollar y de la evolución hacia las metas. Los detalles necesarios evolucionan junto con el resto del plan a medida que se desarrolla el producto/servicio.
- *Funciones*. Describen qué hace el sistema, son extremadamente secundarias y deben mantenerse al mínimo.

A diferencia de lo que es el caso en IEEE 1471, donde todos, clientes y técnicos, son participantes (stakeholders), en EVO se llama participante sólo al cliente. Cuando se inicia el ciclo, primero se definen los Valores y Metas del Participante; esta es una lista tradicional de recursos tales como dinero, tiempo y personal. Una vez que se comprende hacia dónde se quiere ir y cuándo se podría llegar allí, se definen las Soluciones para lograrlo.

Conforme se desenvuelve el proyecto, se obtiene retroalimentación en tiempo real sobre las mejoras que implica la entrega evolutiva sobre las metas y los valores del participante, así como sobre el consumo de recursos. Esta información se usa para establecer qué es lo que está bien y lo que no, cuáles son los desafíos y qué es lo que no se sabía desde un principio; también se aprende sobre las nuevas tecnologías y técnicas que no estaban disponibles cuando el proyecto empezó. Se ajusta luego todo según se necesite, pero sin detallar las soluciones o las entregas evolutivas hasta que se aproxime la entrega. Por último, vienen las funciones y subfunciones, de las que se razona teniendo en cuenta que en rigor, en un nivel puro, son de hecho soluciones a metas.

Un recurso fundamental de EVO es el Planguage, que es un lenguaje estructurado de especificación que sirve para formalizar el requerimiento. El lenguaje es simple pero rico.

Planguage integra todas las disciplinas de solución de problemas. El método consiste en un lenguaje para planificación y un conjunto de procesos de trabajo, el lenguaje de procesos. Proporciona un conjunto rico de formas de expresar propósito, restricciones, estrategia, diseño, bondad, inadecuación, riesgo, logro y credibilidad.

En el tratamiento metodológico de EVO, es esencial la concepción que liga la evolución con el aprendizaje. Los ciclos de aprendizaje de EVO son exactamente lo

mismo que el ciclo Planear-Hacer-Estudiar-Actuar. Los principios que orientan la idea de la Entrega Evolutiva del Proyecto son [38]:

(1) *Aprendizaje*. La Entrega Evolutiva es un ciclo de aprendizaje. Se aprende de la realidad y la experiencia; se aprende lo que funciona y lo que no.

(2) *Temprano.* Aprender lo suficiente para cambiar lo que necesita cambiarse antes que sea tarde.

(3) *Pequeño*. Pequeños pasos acarrean pequeños riesgos. Manteniendo el ciclo de entrega breve, es poco lo que se pierde cuando algo anda mal.

(4) *Más simple*. Lo complejo se hace más simple y más fácil de manejar cuando se lo descompone en pequeñas partes.

La evaluación que se realiza en el estudio debe apoyarse en una herramienta objetiva de inspección o control de calidad. Para ello, EVO implementa la especificación para el control de calidad (Specification Quality Control – SQC), un método probado durante décadas, capaz de descubrir fallas que otros métodos pasan por alto.

Tom Gilb distingue claramente entre los pasos de la entrega evolutiva y las iteraciones propias de los modelos iterativos tradicionales, o de algún MA como RUP. Las iteraciones siguen un modelo de flujo, tienen un diseño preestablecido y son una secuencia de construcciones definidas desde el principio; los pasos evolutivos se definen primero de una manera genérica y luego se van refinando en cada ciclo, adoptando un carácter cada vez más formal. Las iteraciones no se realizan sólo para corregir los errores de código mediante refinamiento convencional, sino en función de la aplicación del control de la calidad del software (SQC) u otras herramientas con capacidades métricas.

En proyectos evolutivos, las metas se desarrollan tratando de comprender de quiénes vienen, qué es lo que son y cómo expresarlas, procurándose pasar el menor tiempo posible en tareas de documentación.

Paralelamente al método de Planguage, EVO incluye lineamientos de métricas, procedimientos formales y orientaciones para descomponer procesos grandes en pasos, políticas de planeamiento y un conjunto de plantillas y tablas de Estimación de Impacto.

A diferencia de otros MA, que son más experimentales y que no tienen mucho respaldo de casos sistemáticamente documentados, EVO es un método probado desde

hace mucho tiempo en numerosos clientes corporativos como ser la NASA, Lockheed Martin, Hewlett Packard, etc.

II.2.4.4. Desarrollo Adaptativo de Software (Adaptive Software Development – ASD)

ASD fue desarrollado en el año 2000 por James Highsmith con la intención de ofrecer una alternativa a la idea, propia de CMM[4] Nivel 5, de que la optimización es la única solución para problemas de complejidad creciente. Este MA pretende abrirse un camino entre el desarrollo monumental de software y el desarrollo accidental o, dicho de otra forma, entre la burocracia y la adhocracia. Highsmith afirma que debe buscarse el rigor estrictamente necesario, para lo cual hay que situarse apenas un poco fuera del caos y ejercer menos control que el que se cree necesario [38].

La estrategia de ASD se basa en el concepto de emergencia, una propiedad de los sistemas adaptativos complejos, que describe la forma en que la interacción de las partes genera una propiedad que no puede ser explicada en función de los componentes individuales.

ASD presupone que las necesidades del cliente son siempre cambiantes. La iniciación de un proyecto involucra definir una misión, determinar las características y las fechas, y descomponer el proyecto en una serie de pasos individuales, cada uno de los cuales puede abarcar entre cuatro y ocho semanas. Las fases de ASD se muestran en la figura 2.3.

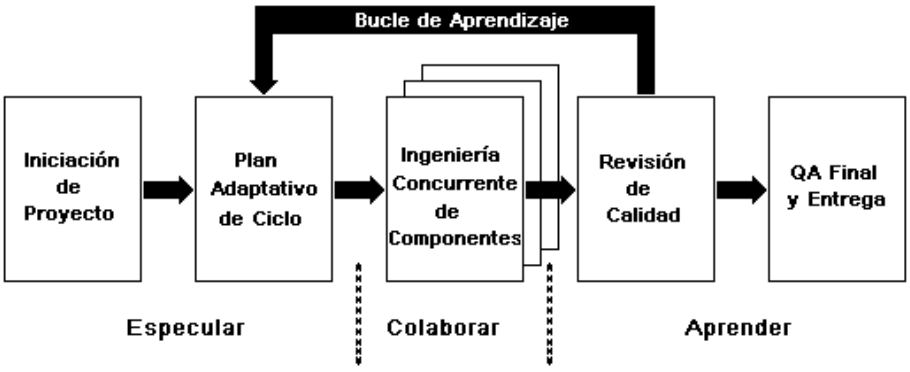


Figura 2.3. Fases del ciclo de vida de ASD [38]

[4] El **Modelo de Madurez y Capacidad**, o **CMM** (Capability Maturity Model), es un método para definir y gestionar los procesos a realizar por una organización. Fue desarrollado inicialmente para los procesos relativos al software por el SEI (Software Engineering Institute) de la Universidad Carnegie-Mellon. [8]

ASD posee tres aspectos claves [21]:

(1) Un conjunto no estándar de "artefactos de misión", incluyendo una visión del proyecto, una hoja de datos, un perfil de misión del producto y un esquema de su especificación.

(2) Un ciclo de vida, inherentemente iterativo.

(3) Cajas de tiempo, con ciclos cortos de entrega orientados por riesgo.

ASD no proporciona un método para el desarrollo de software sino que más bien suministra la forma de implementar una cultura adaptativa en la empresa, con capacidad para reconocer que la incertidumbre y el cambio son el estado natural.

ASD se concentra más en los componentes que en las tareas; en la práctica, esto se traduce en ocuparse más de la calidad que en los procesos que se usan para producir un resultado. En los ciclos adaptativos de la fase de colaboración, el planeamiento es parte del proceso iterativo, y las definiciones de los componentes se refinan continuamente. La base para los ciclos posteriores radica en el bucle de aprendizaje que se obtiene a través de repetidas revisiones de calidad con la presencia del cliente como experto. Esto ocurre solamente al final de las fases, por lo que la presencia del cliente se suplementa con sesiones de desarrollo conjunto de aplicaciones.

El modelo de Highsmith es complementario a cualquier concepción dinámica del método; no podría ser otra cosa que adaptable, después de todo, y por ello admite y promueve integración con otros modelos y marcos.

II.2.4.6. Desarrollo Liviano (Lean Development - LD) y Desarrollo Liviano de Software (Lean Software Development - LSD)

Lean Development (LD) es el método menos divulgado entre los reconocidamente importantes. Fue presentado por Bob Charette, y se inspira en el éxito del proceso industrial Lean Manufacturing, proceso que tiene como precepto la eliminación de residuos a través de la mejora constante, haciendo que el producto fluya a instancias del cliente para hacerlo lo más perfecto posible.

LD se inspira en doce valores centrados en estrategias de gestión [21]; a saber:

(1) Satisfacer al cliente es la máxima prioridad.

(2) Proporcionar siempre el mejor valor para la inversión.

(3) El éxito depende de la activa participación del cliente.

(4) Cada proyecto LD es un esfuerzo de equipo.

(5) Todo se puede cambiar.

(6) Soluciones de dominio, no puntuales.

(7) Completar, no construir.

(8) Una solución al 80% hoy, en vez de una al 100% mañana.

(9) El minimalismo es esencial.

(10) La necesidad determina la tecnología.

(11) El crecimiento del producto es el incremento de sus prestaciones, no de su tamaño.

(12) Nunca se debe empujar a LD más allá de sus límites.

Dado que LD es más una filosofía de administración que un proceso de desarrollo no hay mucho que decir del tamaño del equipo, la duración de las iteraciones, los roles o la naturaleza de sus etapas. Recientemente LD ha evolucionado como Lean Software Development (LSD), método que tiene como figura de referencia a Mary Poppendieck.

LSD se inspira en los siguientes principios [38]:

(1) *Eliminar basura*. Basura es todo lo que no agregue valor a un producto, desde la óptica del sistema de valores del cliente.

(2) *Amplificar el conocimiento*. El desarrollo se considera un ejercicio de descubrimiento.

(3) *Decidir tan tarde como sea posible*. Las prácticas de desarrollo que promueven la toma de decisiones tardías son efectivas en todos los dominios que involucran incertidumbre, porque brindan una estrategia basada en opciones fundadas en la realidad, no en especulaciones.

(4) *Entregar tan rápido como sea posible*. El cliente recibe lo que necesita hoy, no lo que necesitaba ayer. Esto se logra favoreciendo ciclos cortos de diseño, implementación, etc.

(5) *Otorgar poder al equipo*. Los desarrolladores que mejor conocen los elementos de juicio son los que pueden tomar las decisiones más adecuadas.

(6) *Integridad incorporada*. La integridad conceptual significa que los conceptos del sistema trabajan como una totalidad armónica de arquitectura coherente.

(7) *Ver la totalidad*. Uno de los problemas más intratables del desarrollo de software convencional es que los expertos en áreas específicas maximizan la corrección de la parte que les interesa, sin percibir la totalidad.

Aunque la formulación del método es relativamente reciente, la familiaridad de muchas empresas con los principios de Lean Production & Lean Manufacturing ha facilitado la penetración en el mercado de su análogo en ingeniería de software. LD se encuentra hoy en América del Norte en una situación similar a la de DSDM en Gran Bretaña, llegando al 7% respecto del total de los MA usados a nivel mundial.

II.2.4.7. Scrum

Scrum es un método adaptativo, ágil, auto-organizante y con pocos tiempos muertos; fue aplicado por Jeff Sutherland y elaborado más formalizadamente por Ken Schwaber.

Scrum no está concebido como método independiente, sino que se promueve como complemento de otros métodos, incluyendo XP, MSF o RUP. Como método, Scrum enfatiza valores y prácticas de gestión, sin pronunciarse sobre requerimientos, implementación y demás cuestiones técnicas; de allí su deliberada insuficiencia y su complementariedad. Scrum se define como un proceso de administración y control que implementa técnicas de control de procesos, y se lo puede considerar un conjunto de patrones organizacionales.

Los valores de Scrum son:

- Equipos auto-dirigidos y auto-organizados. No hay directores que decidan, ni otros títulos que “miembros del equipo”; la excepción es el Scrum Master que debe ser 50% programador y que resuelve problemas, pero no manda.
- Una vez elegida una tarea, no se agrega trabajo extra. En caso que se agregue algo, se recomienda quitar alguna otra cosa.
- Encuentros diarios con las tres preguntas indicadas en la figura 2.5. Se realizan siempre en el mismo lugar, en círculo.
- Iteraciones de treinta días; se admite que sean más frecuentes.

- Demostración a participantes externos al fin de cada iteración.
- Al principio de cada iteración, planeamiento adaptativo guiado por el cliente.

La dimensión del equipo total de Scrum no debería ser superior a diez ingenieros, si hay más, lo más recomendable es formar varios equipos.

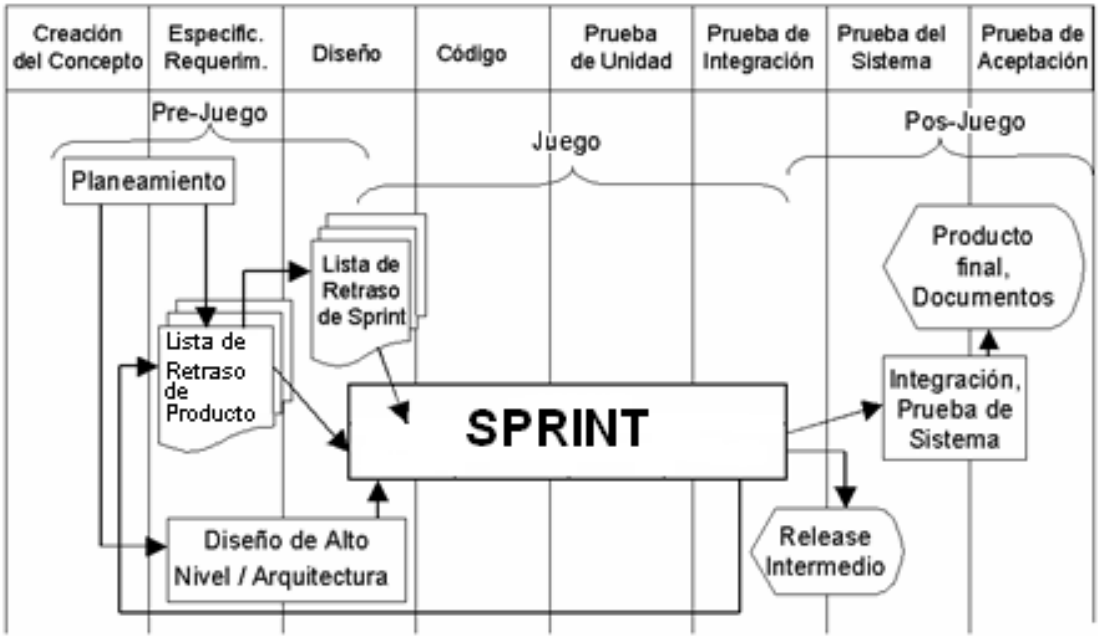


Figura 2.4. Ciclo de Scrum [34]

Scrum plantea su ciclo de vida de la siguiente manera [34]:

(1) *Pre-Juego*: *Planeamiento*. El propósito es establecer la visión, definir las expectativas y asegurarse la financiación.

(2) *Pre-Juego*: *Montaje (Staging)*. El propósito es identificar más requerimientos y priorizar las tareas para la primera iteración.

(3) *Juego*: *Desarrollo*. El propósito es implementar un sistema listo para la entrega en una serie de iteraciones de treinta días llamadas "carreras cortas" o "sprints". El ciclo de sprint se muestra en la figura 2.5.

(4) *Pos-Juego*: *Liberación*. El propósito es el despliegue operacional.

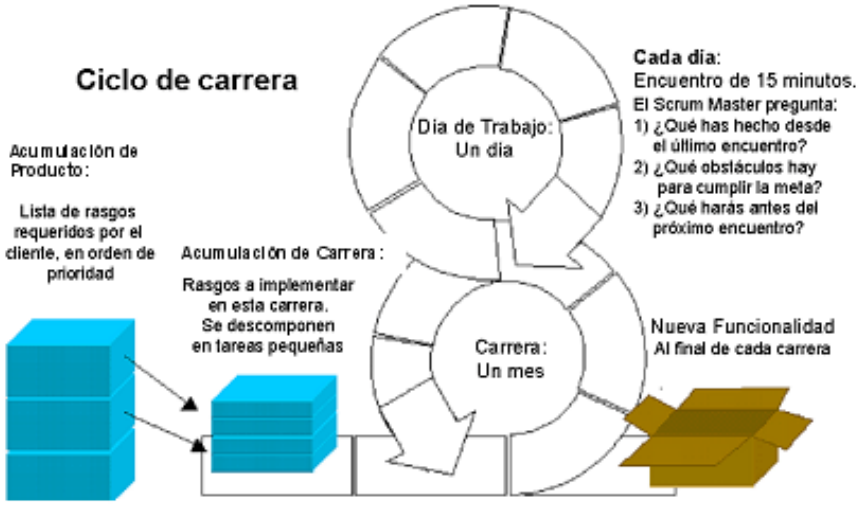


Figura 2.5. Ciclo de Sprint [44]

Es habitual que Scrum se complemente con XP; en estos casos, Scrum suministra un marco de administración basado en patrones organizacionales, mientras XP constituye la práctica de programación, usualmente orientada a objetos y con fuerte uso de patrones de diseño. Uno de los nombres que se utiliza para esta alianza es XP@Scrum. También son viables los híbridos con otros MA.

II.2.4.8. Proceso Unificado Racional (Rational Unified Process – RUP)

El proceso de ciclo de vida de RUP se divide en cuatro fases bien conocidas llamadas Incepción, Elaboración, Construcción y Transición. Esas fases se dividen en iteraciones, cada una de las cuales produce una pieza de software demostrable. La duración de cada iteración puede extenderse desde dos semanas hasta seis meses. Las fases son [27]:

(1) *Incepción*. Significa "comienzo". Se especifican los objetivos del ciclo de vida del proyecto y las necesidades de cada participante; se identifican los casos de uso que orientarán la funcionalidad; y, finalmente, se diseñan las arquitecturas candidatas y se estima la agenda y el presupuesto de todo el proyecto, en particular para la siguiente fase de elaboración. Típicamente es una fase breve que puede durar unos pocos días o unas pocas semanas.

(2) *Elaboración*. Se analiza el dominio del problema y se define el plan del proyecto. Se describen en detalle la infraestructura y el ambiente de desarrollo, así como el soporte de herramientas de automatización. Al cabo de esta fase,

debe estar identificada la mayoría de los casos de uso y los actores, debe quedar descripta la arquitectura de software y se debe crear un prototipo de ella. Al final de la fase se realiza un análisis para determinar los riesgos y se evalúan los gastos hechos contra los originalmente planeados.

(3) *Construcción*. Se desarrollan, integran y verifican todos los componentes y rasgos de la aplicación. Los resultados de esta fase (las versiones alfa, beta y otras versiones de prueba) se crean tan rápido como sea posible. Se debe compilar también una versión de entrega. Es la fase más prolongada de todas.

(4) *Transición*. Comienza cuando el producto está suficientemente maduro para ser entregado. Se corrigen los últimos errores y se agregan los rasgos pospuestos. Se produce también la documentación. Se llama transición porque se transfiere a las manos del usuario, pasando del entorno de desarrollo al de producción.

A través de las fases se desarrollan en paralelo nueve flujos de trabajo: modelado de negocios, requerimientos, análisis y diseño, implementación, prueba, gestión de configuración y cambio, gestión del proyecto y entorno. Además de estos workflows, RUP define algunas prácticas comunes [26]:

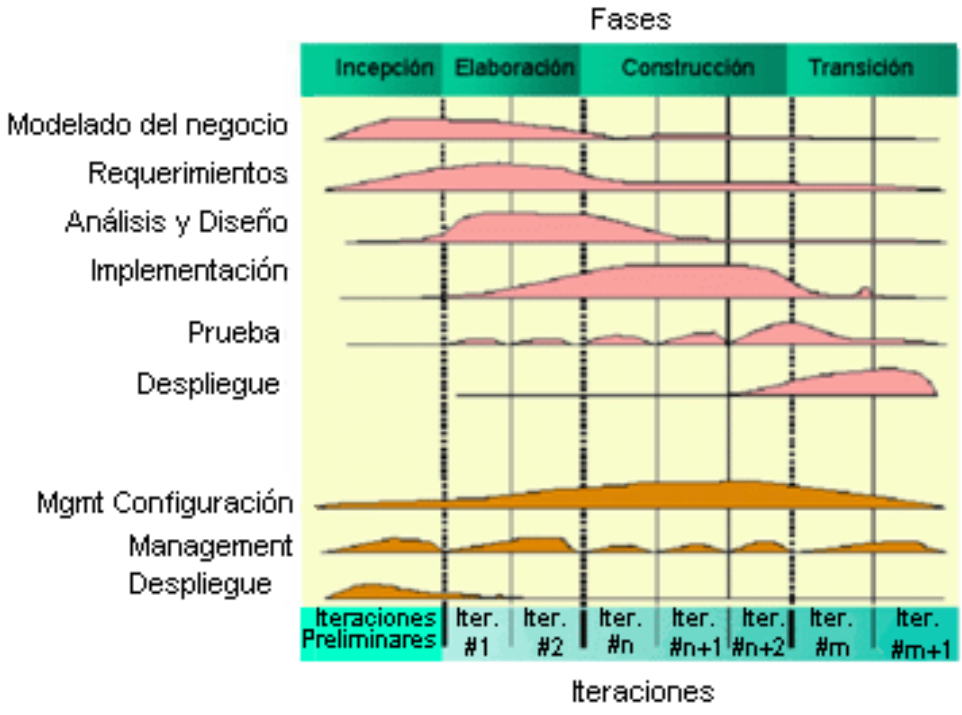


Figura 4. Fases y flujos de trabajo de RUP [27]

- *Desarrollo iterativo de software*. Las iteraciones deben ser breves y proceder por incrementos pequeños. Esto permite identificar riesgos y problemas tempranamente y reaccionar frente a ellos en consecuencia.
- *Administración de requerimientos*. Identifica requerimientos cambiantes y postula una estrategia disciplinada para administrarlos.
- *Uso de arquitecturas basadas en componentes*. La reutilización de componentes permite asimismo ahorros sustanciales en tiempo, recursos y esfuerzo.
- *Modelado visual del software*. Se deben construir modelos visuales, porque los sistemas complejos no podrían comprenderse de otra manera. Utilizando una herramienta como UML, la arquitectura y el diseño se pueden especificar sin ambigüedad y comunicar a todas las partes involucradas.
- *Prueba de calidad del software*. RUP pone gran énfasis en la calidad del producto entregado.
- *Control de cambios y trazabilidad*. La madurez del software se puede medir por la frecuencia y tipos de cambios realizados.

II.3. MARCO EMPÍRICO

El presente trabajo toma como base de análisis de investigación tres sistemas de software de gestión que se desarrollaron en nuestra provincia con diferentes finalidades de usos. Todos ellos con distintas características, tanto desde el punto de vista de la arquitectura, como del tipo de base de datos, y el uso que se le da a los mismos; pero, al mismo tiempo, con características similares en el concepto de desarrollo, es decir, todos se desarrollaron en poco tiempo y con equipos pequeños, conformados por dos a cuatro personas.

Los sistemas de referencia son:

a) Un sistema para la administración de un correo convencional.
b) Un sistema de control académico para la asignatura de Fundamentos de la Programación.
c) Un sistema de ventas y control de inventario para las farmacias de SMAUNSE.

Se pretende volcar la experiencia obtenida en el desarrollo de estos sistemas y probar el método propuesto en el desarrollo de un sistema de gestión para un estudio de arquitectura, el cual consta de las siguientes funciones: emitir presupuestos; hacer un seguimiento de las obras, tanto los ingresos como los gastos de materiales y personal en cada obra; realizar liquidaciones de sueldo; emitir informes con propósitos contables; y administrar las cuentas de caja chica de los usuarios.

II.4. MARCO METODOLÓGICO

La finalidad de este trabajo es proponer un método de desarrollo de software mixto. La generación de este método se basará principalmente en principios, ideas y patrones de los métodos RUP, XP, Scrum, EVO, FDD, ASD y LD, descriptos anteriormente.

II.4.1. ESTÁNDAR ISO/IEC 15504

El estándar ISO/IEC 15504 (1998) es un estándar internacional para la evaluación y la mejora de procesos software. En este estándar se desarrolla un conjunto de medidas de capacidad estructuradas con el objetivo de evaluar el proceso de ciclo de vida del software [23].

Este estándar provee un modelo conceptual y el marco para la evaluación, la validación, la optimización y la certificación del proceso de desarrollo o la construcción de software. Su primera publicación tiene un carácter de reporte técnico y se realizó entre julio de 1998 y mayo de 1999. Este marco puede ser utilizado por organizaciones que están involucradas en las diferentes etapas del proceso de construcción y/o selección de software, o proveedores del mismo, así como en el planeamiento, la gestión, el monitoreo, el control y las mejoras en la adquisición, el desarrollo, la operación y el soporte.

ISO/IEC 15504 se enmarca bajo el nombre "*Tecnologías de la Información: Proceso de Evaluación*", y consiste de las siguientes partes [25]:

1) *Conceptos y vocabulario*. Introduce los conceptos y el vocabulario de términos relacionados con la evaluación de procesos.

2) *Realización de la evaluación*. Define las bases para evaluar los procesos.

3) *Guía para la realización de la evaluación*. Proporciona una guía para la interpretación de los requerimientos para la realización de una evaluación.

4) *Guía para usar en la determinación de la capacidad del proceso y la mejora de procesos*.

5) *Un ejemplo de un modelo de evaluación de procesos*. Contiene un ejemplo de un modelo de evaluación de proceso que está basado en el modelo de proceso de referencia ISO/IEC 12207:1995/Amd 1:2002.

II.4.2. MODELO CONSTRUCTIVO DE COSTOS (CONSTRUCTIVE COST MODEL – COCOMO)

El modelo COCOMO fue desarrollado por B. W. Boehm a finales de la década de 1970 y comienzos de 1980, exponiéndolo detalladamente en su libro "Software Engineering Economics", publicado por Prentice-Hall en 1981.

COCOMO es una jerarquía de modelos de estimación de costos de software que incluye los submodelos *básico*, *intermedio* y *avanzado*, de acuerdo con el tamaño del proyecto respectivamente.

El modelo *básico* calcula el esfuerzo y el costo de desarrollo en función del tamaño del programa estimado en miles de líneas de código.

El modelo *intermedio* calcula el esfuerzo de desarrollo en función del tamaño del programa y un conjunto de conductores del costo que incluyen la evaluación subjetiva del producto, del hardware, del personal y de atributos del proyecto.

El modelo *avanzado* incorpora las características del modelo intermedio y, además, lleva a cabo una evaluación del impacto de los conductores del costo en cada fase (análisis, desarrollo, etc.) del proceso.

Los modelos de COCOMO están definidos para tres tipos de proyectos software:

- *Orgánico*. Proyectos pequeños y sencillos, con equipos chicos con experiencia en la aplicación, y con requisitos poco rígidos.
- *Semiacoplado*. Proyectos de tamaño y complejidad intermedios, con equipos con variados niveles de experiencia, y con requisitos poco o medio rígidos.
- *Empotrado*. Incluye proyectos que deben ser desarrollados con un conjunto de requisitos muy restringidos.

Si bien el modelo en sí no estima directamente el costo, brinda mediante ecuaciones tres factores por demás influyentes en el mismo; ellos son: el esfuerzo (que se mide en meses/hombres), la duración del proyecto y la cantidad de personas necesarias para la ejecución del mismo.

CAPÍTULO III

MÉTODO MIXTO DE DESARROLLO DE SOFTWARE - MMDS

III.1. INTRODUCCIÓN

El desarrollo del software tiene más de cuatro décadas, en las que surgió un amplio abanico de métodos y técnicas para afrontarlo, entre los cuales existen métodos centrados en el proceso, en la arquitectura, y en el usuario. En general, todos estos métodos pueden reunirse en dos grandes grupos: los métodos tradicionales (MT) y los métodos ágiles (MA). Los MT realizan el control de proyectos mediante una rigurosa definición de actividades, artefactos y roles; mientras que los MA valorizan más al individuo, a la interacción con el cliente y al desarrollo incremental del software.

En el entorno económico-social de Santiago del Estero, el mercado es relativamente pequeño, y las empresas clientes que pueden absorber los costos del desarrollo de software con un proceso altamente evolucionado como, por ejemplo, RUP, o pagar licencias por el uso de metodologías como DSDM, son escasas. Por lo cual, está latente la necesidad de generar propuestas metodológicas adaptables a los requerimientos y las necesidades de los clientes y usuarios de nuestro medio.

En este capítulo se presenta, como propuesta, un método nuevo para la construcción de software, denominado "Método Mixto de Desarrollo de Software" (MMDS).

Básicamente, este método propone una integración de métodos clásicos con otros más heterodoxos. Para la propuesta, se tomaron cada una de las etapas del ciclo de vida del software y se las integró con patrones metodológicos, tanto clásicos como ágiles, buscando un equilibrio entre ambos paradigmas de desarrollos, tratando de adaptarla al entorno de nuestra provincia.

III.2. ROLES INTERVINIENTES

MMDS está dirigido a la construcción de software de tamaño pequeño o mediano. Por este motivo, de forma similar a lo propuesto por Scrum, los equipos de desarrolladores son auto-dirigidos y auto-organizados, y de tamaño pequeño (con no más de 10 integrantes).

Los roles intervinientes, son los siguientes:

- Cliente: Es quien contrata al equipo de desarrollo (puede o no ser usuario del sistema), toma decisiones estratégicas importantes y define las políticas marco para el sistema software.
- Usuarios: Son las personas que usarán una o más partes del sistema software y, durante el desarrollo, son quienes definen los detalles sobre cómo deberían comportarse determinadas partes del sistema.
- Equipo de desarrollo: Son las personas encargadas del desarrollo, la prueba, la implantación y el mantenimiento del sistema. Si bien este método no requiere establecer una jerarquía dentro del equipo, dependiendo del número de personas que lo integren, puede ser necesario nombrar uno o dos representantes para tratar con el cliente.

III.3. CICLO DE VIDA

Existen muchas formas de afrontar un proyecto de software pequeño. Es así que, muchas veces, los desarrolladores dejan de lado el uso de metodologías, y la opción que resulta aparentemente más sencilla es separar rápidamente el sistema en procesos, cada proceso en funciones y comenzar a programar sin más. Sin embargo, si se piensa en el mantenimiento del producto, la programación libre y sin dirección no es una buena alternativa, sobre todo si éste necesita crecer.

En la siguiente sección se presentará un método ajustado a un ciclo de vida incremental, iterativo e interactivo. Es incremental, ya que se pretende construir el producto a través de la evolución de prototipos agregando funcionalidad en cada iteración. Es de carácter iterativo, por que el proceso de evolución se plantea a partir de una serie cíclica de repeticiones hasta obtener el producto deseado. Y es de carácter

interactivo, puesto que en el mismo estarán en constante relación todos los roles definidos en el apartado anterior.

Se opta por un ciclo con estas características debido a que su carácter iterativo e incremental permite reducir costos y riesgos, como así también acelerar el ritmo de desarrollo. Además, una interacción alta ayuda a una mejor comprensión de los requisitos y la funcionalidad que debe tener el software.

Para su estudio, se procederá a dividir al ciclo de vida en cuatro fases o etapas significativas (figura 3.1). Las fases son:

- Análisis de requisitos.
- Desarrollo.
- Mantenimiento.
- Muerte y reemplazo.

Sin embargo, es importante mencionar que estas fases no se encuentran bien diferenciadas, sino que se entremezclan, como consecuencia de las características inherentes al desarrollo del software.

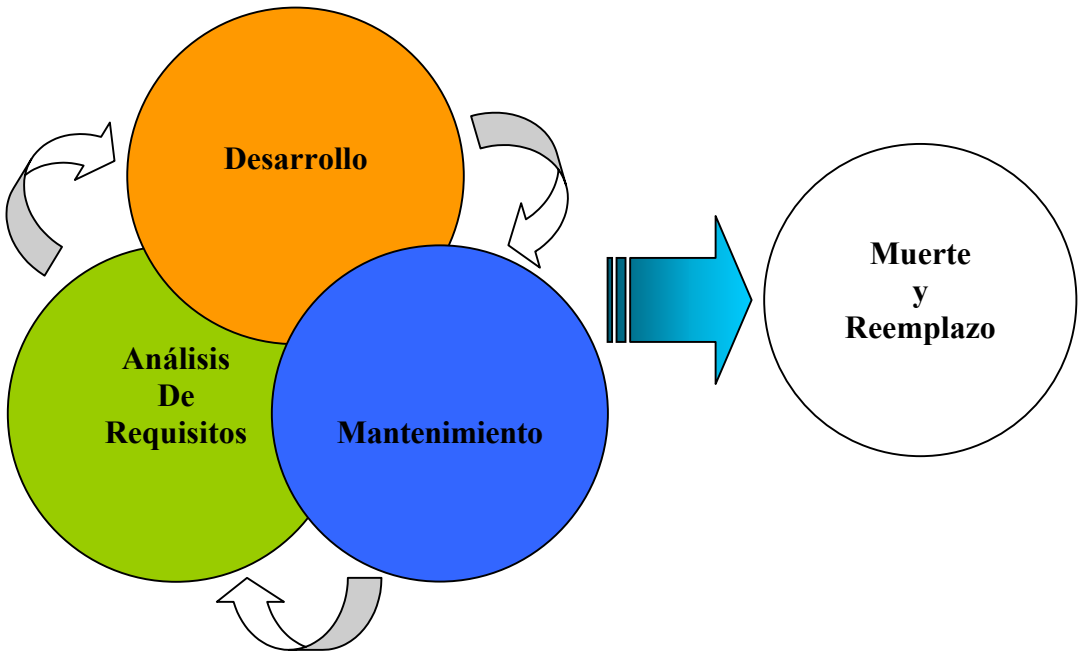


Figura 3.1. Fases del ciclo de vida de MMDS

III.3.1. ANÁLISIS DE REQUISITOS

Cualquier proyecto comienza necesariamente por la búsqueda de requisitos del sistema. Un requisito es una característica de diseño, una propiedad o un comportamiento de un sistema [28].

En el desarrollo tradicional de proyectos software se usa la expresión "capturar los requisitos", como si se estuviera a la caza de algunas bestias salvajes que acechan en la selva; sin embargo, lo que realmente se quiere expresar con esta frase, es una actividad menos excitante que involucra la producción de documentos que especifican un sistema en detalle suficiente para que el desarrollo del software puede proceder exclusivamente por la referencia a estos [10]. El resultado es una gran cantidad de documentos de requisitos en los que se invirtió demasiado tiempo.

El enfoque tradicional es documentar los requisitos completos para todo el sistema y luego congelarlos para mantener una base estable para la implementación del proceso. Es decir, se asume que se puede llegar a una comprensión completa del sistema antes de proceder con el desarrollo. Quizás esto es posible en dominios simples o muy conocidos de algunas aplicaciones, pero es arriesgado para constituir proyectos de desarrollo de software complejos o con entornos muy variables.

Otra desventaja del enfoque tradicional es que los documentos son un medio de comunicación unidireccional; la información fluye del autor de la documentación al desarrollador. No hay ninguna oportunidad para que los demás desarrolladores puedan hacer preguntas, ofrecer ideas y visiones. El autor puede intentar anticiparse a las preguntas, pero no se puede esperar que se anticipe a todo. Y en el esfuerzo por cubrirlo todo, una abstracción concisa puede perderse en un bosque de páginas. Escribe desde su perspectiva personal al sistema en desarrollo y, naturalmente, hace suposiciones sobre el conocimiento del entorno del sistema que tienen los desarrolladores.

El núcleo del desarrollo ágil es la construcción incremental del sistema, ya que los MA se afirman en la idea de que se continúa aprendiendo sobre el ambiente operativo del sistema durante el proyecto. Al reconocer esto, se hace evidente que congelar los requisitos en las fases tempranas de desarrollo puede ocasionar que se vuelva a trabajar sobre lo mismo o que el sistema que se entrega no satisfaga las necesidades del cliente. Por este motivo, MMDS no buscará una comprensión absoluta de todo el sistema en la etapa inicial, sino que la misma será de carácter global y no muy detallada, con el fin de

introducir al desarrollador al ámbito del proyecto, dándole así una idea general de los tiempos y costos del desarrollo. El detalle de la funcionalidad, necesario para el desarrollo del sistema, se buscará a lo largo de los diferentes incrementos. Esta forma de trabajar permitirá al cliente, o a los usuarios, probar el software durante su desarrollo en lugar de tener que esperar hasta el final, lo que originará una retroalimentación que puede usarse para refinar el comportamiento del sistema.

MMDS sugiere usar diagramas de casos de uso y tarjetas de historias de usuario para la documentación de los requisitos. Propone dos enfoques para trabajar con estas técnicas, los mismos se diferencian en cuanto a formalidad, agilidad y tamaños de proyectos en los que se pueden aplicar.

El primero de estos enfoques indica documentar todo con historias de usuario. Sin embargo, las historias de usuario pueden llegar a tener problemas con entrevistas largas sobre una parte compleja o grande del sistema a desarrollarse, por esta razón se aconseja en esos casos apoyar a las historias con los diagramas de casos de uso. Este enfoque se plantea como la alternativa más ágil en lo que respecta a la documentación de requisitos de sistemas software pequeños.

El segundo enfoque apunta a una documentación más formal de los requisitos, y puede usarse tanto en sistemas pequeños como en sistemas medianos. En primera instancia se utilizan las historias para documentar las entrevistas y los encuentros con los usuarios, para luego volcarlos en los diagramas de caso de uso. En este enfoque, las tarjetas de historias de usuario no son las protagonistas de la documentación, por lo que se simplifican en cuanto a forma y contenidos.

III.3.1.1. Historias de Usuario

Una *Historia de Usuario* (HU) describe un rasgo de un sistema en un lenguaje comprensible al usuario y al desarrollador. Consta de tres o cuatro líneas escritas por el cliente en un lenguaje no técnico, sin hacer hincapié en detalles; no se debe hablar ni de posibles algoritmos para su implementación, ni de diseños de base de datos adecuados, etc.

Para favorecer el trabajo colaborativo, las HU se documentan en tarjetas indexadas, que funcionan como medio de comunicación y también como un mecanismo de almacenamiento, evitando de esta manera que las conversaciones y entrevistas a los usuarios se pierdan en la memoria de los desarrolladores. No existe un método o

estándar para escribir estas tarjetas; se pueden escribir de cualquier forma que se considere útil para expresar lo comentado en las reuniones y entrevistas. Es importante destacar que el desarrollo ágil no se asienta sobre una serie de cascadas pequeñas, por lo que las tarjetas indexadas no son una división de los requisitos en una serie de partes, sino que representan a modelos de procesos diferentes.

El tratamiento de las HU es muy dinámico y flexible. En cualquier momento, HU pueden romperse, reemplazarse por otras más específicas o generales, añadirse nuevas o modificarse.

No existe un consenso respecto a la información que debe contener una HU. Beck, en su libro "Extreme Programming Explained. Embrace Change" [4], presenta un ejemplo de ficha a la que llama "customer story and task card", y contiene, entre otra información, fecha, tipo de actividad (nueva, corrección, mejora), prueba funcional, número de historia, prioridad técnica y del cliente, referencia a otra historia previa, riesgo, estimación técnica, descripción, notas y una lista de seguimiento con fecha, cosas por terminar y comentarios. Sin embargo, MMDS propone incluir, en las tarjetas de HU, solamente la fecha, el número de HU (para construir un índice de las mismas), una descripción de requisitos especificados por el cliente, y un espacio de observaciones (en donde puede hacerse referencia a otras HU, incluir el tipo de actividad, etc.).

III.3.1.2. Diagramas de Casos de Uso

Es una técnica propia de UML (Lenguaje Unificado de Modelado) que se usa para modelar el comportamiento de un sistema (aquellos servicios visibles externamente que proporciona el sistema en el contexto de su entorno). Usualmente, los *Diagramas de Casos de Uso* (DCU) se usan para modelar el contexto o los requisitos de un sistema. El modelado de los requisitos de un sistema implica especificar qué debería hacer el sistema independientemente de cómo lo haga.

Los elementos de un DCU son los siguientes:

- Caso de Uso: es una descripción de una secuencia de acciones que ejecuta un sistema y que produce un resultado observable de interés para un actor particular. Un caso de uso se utiliza para estructurar los aspectos de comportamiento en un modelo [28]. Gráficamente se representa como una elipse de borde continuo que incluye sólo su nombre.

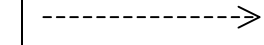
Figura 3.2. Caso de Uso

- Actor: representa un conjunto coherente de roles que juegan los usuarios de los casos de uso al interactuar con éstos. Los actores pueden ser personas, sistemas, mecanismos, etc. [28]. Se los representa gráficamente como se muestra en la figura a continuación.

Figura 3.3. Actor

- Relaciones: en UML existen cuatro tipos de relaciones, y se presentan en la siguiente tabla.

Tabla 3.1. Relaciones existentes en UML

Nombre	Descripción	Representación Gráfica
Dependencia	Es una relación en la cual un cambio a un elemento puede afectar a la semántica del otro elemento.	
Asociación	Es una relación estructural que define un conjunto de enlaces que son conexiones entre objetos.	
Generalización	Es una relación en la cual los objetos del elemento especializado (hijo) pueden sustituir a los objetos del elemento general (padre).	
Realización	Es una relación semántica entre dos calificadores, en donde un calificador especifica un contrato que otro calificador garantiza que cumplirá.	

III.3.2. DESARROLLO

La fase de desarrollo se centra en el cómo. Durante esta fase se llevan a cabo una serie de actividades bien conocidas por quienes se dedican a la construcción del software.

Estas actividades comprenden el diseño de las bases de datos, las pruebas, la codificación y el diseño de software propiamente dicho. De todas ellas, en la concepción de diseño del software es donde la diferencia entre los MA y los MT es más marcada.

El enfoque tradicional utiliza el diseño planeado. Esta perspectiva del diseño contiene nociones tomadas de otras ramas de la ingeniería. En el diseño planeado, los diseñadores piensan los grandes problemas con anticipación. No necesitan programar porque no están construyendo el software, sólo lo están planeando. Así que pueden usar técnicas de diseño como UML, que deja de lado algunos de los detalles de la programación y permite a los diseñadores trabajar a un nivel más abstracto. Una vez hecho el diseño, pueden pasarlo a otro grupo (o a otra compañía) que lo construya. Debido a que los diseñadores están pensando en una escala mayor, pueden evitar las decisiones tácticas que llevan a la entropía del software. Los programadores pueden seguir la dirección del diseño y, dado que siguen el diseño al pie de la letra, tener un sistema bien construido.

El diseño planeado existe desde la década de 1970, y mucha gente lo ha usado pero tiene algunas fallas. La primera es que es imposible pensar en todos los problemas que se presentan cuando se programa. Por la naturaleza misma del desarrollo de software, es inevitable que al programar se encuentren cosas que ponen en duda el diseño y, si los diseñadores ya terminaron y se encuentran trabajando en otro proyecto, los programadores empiezan a codificar haciendo suposiciones con respecto al diseño, con lo que la entropía aparece. Aún si el diseñador está, lleva tiempo organizar los problemas de diseño, cambiar los dibujos y alterar el código, por lo que usualmente se hace una corrección rápida debido a la presión del tiempo, ocasionando nuevamente la aparición de entropía [15].

Frecuentemente, también existe un problema cultural, que tiene su origen en el hecho de que los diseñadores se han formado por su habilidad y experiencia, pero están tan ocupados trabajando en diseños que ya no tienen tiempo para programar. Sin embargo, las herramientas y los materiales de desarrollo de software cambian rápidamente, por lo que cuando se deja de programar, no sólo se pierden los cambios que ocurren en este flujo tecnológico, sino que también se genera un distanciamiento respecto de quienes programan [15].

Si bien puede darse solución a estos problemas consiguiendo diseñadores tan hábiles para tratar con la mayoría de los problemas y tener un proceso suficientemente disciplinado para cambiar los dibujos, queda todavía el problema de los requerimientos cambiantes. Una manera de controlar este problema es añadir flexibilidad en el diseño de modo que se pueda cambiar fácilmente conforme cambian los requerimientos, pero esto no ayuda con los cambios imprevistos que surjan en el ambiente de negocios.

Algunos de los problemas de requerimientos se deben a la falta de un entendimiento claro de los mismos y a cambios en el ambiente de negocios. Por esta razón, la mayoría de los MA plantean un diseño evolutivo.

En este enfoque, el diseño del sistema crece conforme se implanta el sistema. Es decir, el diseño es parte del proceso de programación y conforme el programa evoluciona el diseño cambia. Esta forma de trabajar se radica en la afirmación de que se continúa aprendiendo sobre los requisitos y el entorno del sistema durante todo su desarrollo.

En su uso común, el diseño evolutivo puede terminar en desastre, deteriorándose conforme se agregan parches de código para cambiar o agregar nuevos controles y funcionalidad al software. Esto no sólo hace el software más difícil de cambiar, sino que también facilita la generación de defectos (bugs) y dificulta el encontrarlos y eliminarlos con seguridad. Ante esta situación, los MA proponen diversas formas y filosofías para afrontar el diseño evolutivo.

MMDS sugiere un diseño simple, que forme parte del proceso de programación, integrando la documentación referida al diseño en el código fuente, pero sólo lo necesario para facilitar su comprensión y mejorar su mantenibilidad. Esta idea parece ser simple, pero en realidad implica una elevada complejidad, ya que el proceso de programación debe ser ordenado y guiado por cierta estandarización; de otra manera resulta imposible de controlar, llevando a un caos de versiones, comentarios y redundancia de código. Con el fin de mantener un cierto control sobre el desarrollo y el crecimiento del proceso de programación, MMDS utiliza una serie de pautas y prácticas que orientan a los desarrolladores en las tareas de programación, con el fin de obtener código con la menor cantidad de errores posibles, como así también facilitar su comprensión, apuntando siempre a la calidad del producto final.

Es importante mencionar que MMDS se centra en el usuario y, en este sentido, un concepto de gran importancia es el de usabilidad. La usabilidad mide que tan fácil de

usar resulta un sistema para sus usuarios. En este contexto, la interfaz del sistema con el usuario adquiere una gran importancia. En general, puede entenderse que la interfaz de usuario es lo que ve el usuario del sistema; por lo tanto, puede decirse que incluye todas las interacciones que el usuario tiene con el producto.

El objetivo principal de la interfaz es hacer usable el software desde la perspectiva del usuario; por esta razón, no se debe menospreciar a la interfaz del sistema. Se sugiere que la interfaz se diseñe cuidadosamente, con una amplia participación del usuario, buscando la conformidad del mismo en ese aspecto, ya que según las características del diseño de esta interfaz se pueden facilitar o impedir la interacción entre el usuario y el sistema desarrollado.

Otro aspecto importante, que sirve para validar el código y la funcionalidad de los programas, son las pruebas. Actualmente, existen muchas alternativas para probar los programas y el código fuente; por ejemplo, pruebas tradicionales como las técnicas de caja blanca y de caja negra, pruebas orientadas a objetos, pruebas automáticas como las pruebas de unidad sugeridas por XP, etc. Sin importar la técnica que se escoja, MMDS recomienda que las pruebas sean incrementales y acompañen todo el proceso de desarrollo. De esta manera, a medida que se desarrolla se prueban las diferentes funciones, los procedimientos y las interfaces de usuario que se generen. Esto facilita su reutilización de manera segura en las próximas iteraciones.

A continuación, en la tabla 3.2, se presenta una lista de comprobación de código para ayudar tanto a las tareas de programación como a las de pruebas. La misma consta de cinco partes, cada una de las cuales está referida a un aspecto diferente del programa.

Tabla 3.2. Reglas para guiar el proceso de desarrollo

Aspecto	Reglas	Descripción
Referente a los datos: se busca la correcta definición de variables y evitar posibles errores con los datos.	Para la prevención de errores de declaraciones	- Todas las variables deben tener nombres claros y descriptivos, y se debe respetar una forma estándar para definir las variables; por ejemplo, se podrían definir las variables usando la notación húngara o la notación camelCase, etc. - Las variables y las estructuras de datos deben estar correctamente inicializadas.

Tabla 3.2. Reglas para guiar el proceso de desarrollo (continuación)

Aspecto	Reglas	Descripción
Referente a los datos (continuación)	Para la prevención de errores de declaraciones	-La inicialización debe ser consistente con los tipos de datos de las variables y las estructuras. -No se deben declarar variables que no se utilizarán. -Debe evitarse definir variables y estructuras de datos con nombres similares. -Si se programa con lenguajes orientados a objetos, se debe tener especial cuidado con el alcance de las variables, como así también si su declaración es privada, pública o protegida.
	Para la prevención de errores de referencia	-Asignar los valores correctos a todas las variables y las estructuras definidas. -No asignar valores a una constante después de su inicialización. -Referenciar correctamente las variables tipo puntero y objeto.
	Para la prevención de errores de cálculo	-Controlar si puede producirse división por cero. -Controlar si los cálculos aritméticos se aplican sobre las variables del tipo indicado. -Las asignaciones deben ser sobre variables del tipo correcto. -Controlar que los valores de las variables no superen los rangos definidos por su tipo (eliminar la presencia de overflow o underflow). -Si se utiliza más de un operador, controlar la precedencia.
Referente al flujo de programa: Se busca un adecuado flujo del programa a través una lógica correcta. Consta de tres tipos de controles que a puntan a resolver errores de comparación, errores de flujo de control y errores de entrada-salida.	Para la prevención de errores de comparación	-Las comparaciones deben ser entre variables de un mismo tipo. -En caso de condiciones compuestas se debe controlar que la precedencia sea adecuada.
	Para la prevención de errores de flujo de control	-Controlar que no existan ciclos indefinidos. -Controlar que todos los lugares del programa sean alcanzables desde el principio y que no existan ramas innecesarias. -Evitar el uso de saltos incondicionales (sentencias del tipo “goto”). -En el caso de existir ciclos anidados, controlar que los mismos cierren correctamente. -Controlar si es posible reemplazar múltiples “if” por sentencias del tipo “case”. -En ciclos con múltiples salidas, controlar que todas ellas funcionen y sean necesarias. -Todos los procedimientos y las funciones deben terminar correctamente. -Las funciones siempre deben retornar valores, los que deben coincidir con el tipo de dato definido para la función. -Debe contemplarse la aparición de posibles excepciones.

Tabla 3.2. Reglas para guiar el proceso de desarrollo (continuación)

Aspecto	Reglas	Descripción
Referente al flujo de programa (continuación)	Para la prevención de errores de entrada-salida	- Si se utilizan archivos de entrada y salida, deben considerase las situaciones de que el archivo no exista o no esté disponible. - Todos los archivos de entrada y salida deben abrirse y cerrarse. - En el uso de bases de datos, debe contemplarse la posibilidad de accesos concurrentes a un mismo registro ya sea para lectura o escritura. - Debe contemplarse la posibilidad de una excepción que obligue a deshacer la operación de escritura sobre un archivo o base de datos.
Referente al entorno: se busca minimizar los errores de interfaz con los diferentes procedimientos y funciones.	Para la prevención de errores de interfaz entre procedimientos y funciones	- Los parámetros deben pasarse correctamente a las funciones y los procedimientos, estos deben coincidir en cantidad y tipo con los de su definición. - Si se programa con lenguajes orientados a objetos, se debe tener especial cuidado con el alcance de los procedimientos o funciones, como así también si su declaración es privada, pública o protegida.
Referente a la interfaz con el usuario: se busca un programa fácil de usar para el usuario	Para obtener un programa fácil de usar	- El diseño de las ventanas y pantallas debe respetarse en todo el software; debe ser uniforme, claro y sencillo. - Definir los modos de interacción de manera que no obligue al usuario a realizar acciones innecesarias y no deseadas. - El formato visual de la interfaz debe basarse en una metáfora del mundo real, para facilitar la comprensión y el aprendizaje del usuario. - La interfaz debe ser flexible y permitir al usuario interrumpir y deshacer. - Debe controlarse el ingreso correcto de datos por parte del usuario. - Las consultas a bases de datos o archivos deben devolver la información justa que necesita el usuario, y presentarla de forma clara y comprensible. - Las salidas impresas deben mantener un diseño uniforme y expresar la información en forma clara.
Referente a la documentación: se presta especial atención a la documentación en el código, debido a que se utiliza para mejorar la mantenibilidad	Para realizar la documentación	- En cada archivo de código debe comentarse a qué requisito, o conjunto de éstos, apunta a resolver el código contenido en el mismo; por ejemplo, si se trabaja en Delphi, al iniciar una unidad se puede comentar los requisitos a los que apunta dicha unidad. - Debe incluirse un comentario al inicio de los métodos (procedimientos y funciones) en donde se incluya información sobre cuál es la función de los mismos, qué parámetros reciben, qué datos devuelven. - Los comentarios que se hacen, a lo largo de los diferentes bloques de programa, deben ser claros y completos para ayudar a comprender el bloque al que hacen referencia.

III.3.3. MANTENIMIENTO

El mantenimiento del software puede definirse como el conjunto de actividades de ingeniería del software que se producen después de que el software se haya entregado al cliente y esté en funcionamiento [37].

La fase de mantenimiento se centra en el cambio que se produce porque se han encontrado errores, porque el software debe adaptarse para acoplarse a los modificaciones de su entorno externo (por ejemplo, se requiere un cambio debido a un sistema operativo o dispositivo periférico nuevo), o porque el cliente requiere mejoras funcionales o de rendimiento.

Pressman [37] reconoce cuatro tipos de cambios que pueden producirse durante la fase de mantenimiento:

- *Corrección*. El mantenimiento correctivo cambia el software para corregir defectos, ya que incluso llevando a cabo las mejores actividades de garantía de calidad, es muy probable que el cliente encuentre defectos en el software.
- *Adaptación*. El mantenimiento adaptativo produce modificaciones en el software para que se adecue a los cambios de su entorno externo.
- *Mejora*. El mantenimiento perfectivo lleva al software más allá de sus requisitos funcionales originales, ya que, conforme el usuario utilice el software, puede descubrir funciones adicionales que van a producir beneficios.
- *Prevención*. El software de computadora se deteriora debido al cambio y, por esto, el mantenimiento preventivo, también llamado reingeniería del software, hace cambios en programas de computadora a fin de que se puedan corregir, adaptar y mejorar con mayor facilidad; en otras palabras, su objetivo es mejorar la matenibilidad del software.

El software indudablemente sufrirá cambios después de ser entregado al cliente. El cambio es algo inevitable cuando se construyen sistemas basados en computadoras. De acuerdo con las estadísticas [37] sólo el 20% de los esfuerzos de mantenimiento se invierten en corregir errores, lo que quiere decir que el 80% restante se dedica a adaptar los sistemas existentes a los cambios de su entorno, efectuar las mejoras solicitadas por los usuarios y a rehacer la ingeniería de las aplicaciones para su posterior reutilización.

De lo expuesto, se puede deducir la importancia de desarrollar mecanismos para evaluar, controlar y realizar modificaciones.

MMDS presta especial atención al mantenimiento preventivo, ya que éste eleva la calidad del producto y facilita los demás tipos de mantenimiento.

MMDS utiliza tres principios para guiar el mantenimiento preventivo:

- Antes de empezar cualquier actividad de mantenimiento preventivo examinar el software para ver si se justifica o no la aplicación de dicha actividad.
- Si se opta por la reestructuración del software, procurar utilizar las técnicas y herramientas más actuales, ya que ayudarán al mantenimiento posterior y disminuirán su costo.
- Afrontar cualquier forma de mantenimiento preventivo con una actitud disciplinada, y utilizar prácticas que deriven en un producto de calidad.

En MMDS se realiza el mantenimiento preventivo mediante el uso de "refactorización". Este concepto nuevo, se ha introducido en la terminología del mundo de desarrollo por medio de los MA, más precisamente por XP. Sin embargo, la idea de refactorización no es nueva; lo que resulta novedoso es la forma en que se lleva a cabo, una forma ágil y agresiva, pero al tiempo ordenada y segura. La refactorización es esencial si se quiere llevar a cabo un diseño evolutivo y un desarrollo ágil de una aplicación.

Pero, ¿qué significa refactorizar? La respuesta a esta pregunta se obtiene de Martin Fowler, uno de los padres de esta técnica, quien dijo que [14]: "*refactorizar es realizar modificaciones en el código con el objetivo de mejorar su estructura interna, sin alterar su comportamiento externo*". Refactorizar no es una técnica para encontrar y corregir errores en una aplicación, puesto que su objetivo, y uno de los pilares de cualquiera de las prácticas que forman parte de la técnica, es no es alterar su comportamiento externo. La esencia de esta técnica consiste en aplicar una serie de pequeños cambios en el código manteniendo su comportamiento. Cada uno de estos cambios debe ser tan pequeño que pueda ser completamente controlado por los desarrolladores sin miedo a equivocarse. Es el efecto acumulativo de todas estas modificaciones lo que hace de la refactorización una potente técnica. La meta final, que se persigue al refactorizar, es mantener al código sencillo y bien estructurado.

La refactorización del código no se reduce a una cuestión estética, pero no siempre es justificable llevarla a cabo. Sólo se debe refactorizar cuando se identifique código mal estructurado o diseños que supongan un riesgo para la futura evolución del sistema que se esté desarrollando. Aplicar la refactorización es conveniente siempre que se detecte que el diseño empieza a ser complicado y difícil de entender, y está conduciendo al proyecto a una situación donde cada cambio empieza a ser muy costoso; de esta manera se reducen los costos de desarrollo y de mantenimiento.

Muchas veces es complicado justificar la refactorización, ya que supone un esfuerzo que no se ve compensado por ninguna nueva funcionalidad; sin embargo, al introducir esta práctica como parte del desarrollo se obtiene código de calidad que, posteriormente, permitirá recuperar con creces el tiempo empleado durante el desarrollo del proyecto. Es evidente que refactorizar facilita en gran medida el mantenimiento de un código siempre correctamente estructurado, lo cual es una ventaja por su mantenibilidad y extensibilidad. Sin embargo, esta técnica que parece tan ventajosa, requiere de su aplicación continua, el empleo de pruebas que proporcionen seguridad a su aplicación, un buen conocimiento de los patrones de diseño que van a permitir al desarrollador saber hacia donde camina y, por último, mantener una clara y amplia comunicación con todos los miembros del equipo.

III.3.4. MUERTE Y REEMPLAZO

Todo sistema software está condenado a muerte antes de nacer, y la principal razón son los cambios en su entorno. A medida que va transcurriendo el tiempo, los sistemas crecen y cambian para adaptarse a las modificaciones que surgen en su entorno y a los problemas no previstos durante su concepción. En la práctica, el mantenimiento no es eterno. Dependiendo de la flexibilidad y la calidad del diseño del sistema, tarde o temprano se llega a un punto de quiebre en el que los costos del mantenimiento dejan de ser factibles, ya que cada nuevo cambio implica un aumento en la complejidad del sistema y, por ende, un mayor esfuerzo y costo en su ejecución. Por analogía, se dice que el sistema aumenta su entropía. Este límite se conoce como el "límite de la mantenibilidad" y se grafica en la siguiente figura [40].

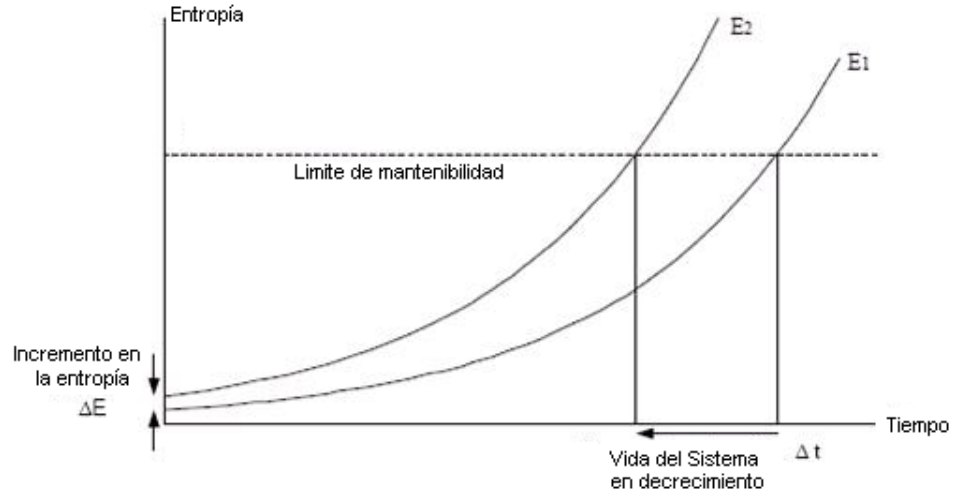


Figura 3.4. Límite de la Mantenibilidad

Una vez llegado el punto en el cual los costos de mantenimiento se vuelven irrealizables, llega el momento de reemplazar al sistema existente.

Si bien el reemplazo tiene mucha importancia, y toda una serie de prácticas ingenieriles para poder llevarlo a cabo exitosamente, este aspecto va más allá de los fines del presente trabajo.

III.4. DESARROLLO DE UN PROYECTO SOFTWARE CON MMDS

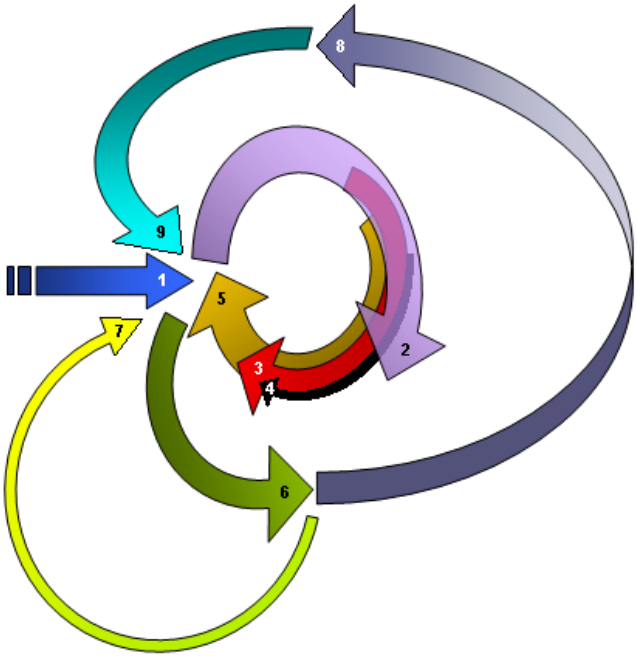
Como se mencionó en el Capítulo II, MMDS se basa principalmente en principios, ideas y patrones de los métodos RUP, XP, Scrum, EVO, FDD, ASD y LD.

MMDS se presenta como un MA, iterativo, similar a los métodos antes mencionados; además es de carácter adaptativo tal cual lo son XP, ASD y EVO; por último, y al igual que LD o XP, es un método centrado en el usuario.

Durante el desarrollo de un proyecto software con MMDS pueden identificarse nueve fases o etapas (figura 3.5.), a saber: análisis inicial, definición de requisitos para la iteración, desarrollo (codificación, documentación), mantenimiento preventivo (al igual que XP, MMDS hace uso de la refactorización), pruebas sobre el software desarrollado, implantación para su uso o prueba piloto, mantenimiento correctivo y perfectivo (refinado), estabilidad y mantenimiento adaptativo. Es importante mencionar que la forma en que se distribuyen las tareas y fases de desarrollo es similar a la propuesta por FDD; no obstante, en cada una de las etapas, aparecen semejanzas con

otros MA, por ejemplo el diseño evolutivo de XP o EVO, la adaptabilidad y apertura a los cambios propuestos por XP, ASD o EVO, y la búsqueda de la satisfacción del cliente sugerida por LD, etc.

A continuación, se describen cada una de las fases o etapas antes mencionadas.



Referencias

1- Análisis inicial (Construcción de un modelo global).
2- Definición de requisitos para la iteración (duración, funcionalidad, etc.).
3- Desarrollo: codificación, documentación.
4- Mantenimiento preventivo (refactorización).
5- Pruebas sobre el software desarrollado.
6- Implantación – Uso/Prueba piloto.
7- Mantenimiento correctivo / perfectivo (refinado).
8- Uso – Estabilidad.
9- Mantenimiento adaptativo.

Figura 3.5. Fases de un proyecto de desarrollo con MMDS

- ***Análisis inicial***

  El desarrollo comienza con la construcción de un modelo general, que no es necesariamente un modelo completo y definitivo del sistema final, sólo es una aproximación inicial que permite identificar cuáles son los rasgos y las características generales del sistema, para poder de esta forma, realizar una primera división del mismo de acuerdo a sus funciones, como así también empezar a pensar qué tipo de base de datos se ajusta más a las necesidades identificadas.

- ***Definición de requisitos para la iteración***

  Una vez que se tiene una idea del contexto del sistema, el desarrollo da inicio con la creación de la base de datos. Conjuntamente se empieza a construir el

software, en una serie de iteraciones en forma de pequeños ciclos de análisis de requisitos, desarrollo e implementación de prototipos. A lo largo de este proceso iterativo, la base de datos, puede y de seguro, cambiará y crecerá; y, si parte del sistema ya está en uso, se deben tomar las medidas necesarias para realizar las adaptaciones manteniendo la integridad de la información del sistema. Por esta razón resulta de gran importancia que la base de datos se encuentre documentada correctamente.

La duración de cada iteración es variable y se recomienda que sea lo más corta posible; se sugiere que dicha duración oscile entre una semana y un mes. La razón de mantener iteraciones breves, es que existe una alta probabilidad de que los requisitos se mantengan estables en un período corto de tiempo.

Además, no es necesario que todas las iteraciones tengan igual duración, puesto no todas las áreas de un sistema informático tienen la misma complejidad y, conforme a lo que evalúe el desarrollador, se puede acordar con el cliente la duración de la iteración.

MMDS centra su atención en el usuario. Si bien no es necesario un cliente in-situ como propone XP, el usuario tiene una participación activa, ya que antes de empezar cada iteración, los desarrolladores se reunirán con él para definir a fondo la funcionalidad de la parte del sistema que se desarrollará en la siguiente iteración, y también cuestiones de diseño como ser forma y colores de ventanas, informes, etc.

- ***Desarrollo / Mantenimiento preventivo / Pruebas***

  Una vez definidas las diversas cuestiones sobre funcionalidad, tiempos e interfaces de usuarios, en lo referente al módulo o subsistema requerido, se comienza con la construcción de un prototipo. Este método no contempla descartar los prototipos sino que, por el contrario, pretende que crezcan, se integren y evolucionen hacia el sistema final con ayuda de los usuarios, ya que dichos prototipos seguramente tendrán ajustes y, por pequeños que sean, contribuirán a mejorar la calidad del producto, como así también su facilidad de uso. La construcción de los prototipos se guiará con la lista de comprobación presentada anteriormente (tabla 3.2).

Simultáneamente, si se detecta que el diseño empieza a ser complicado y difícil de entender, se aplicará refactorización, buscando simplificar el diseño y disminuir los futuros costos de mantenimiento. Las pruebas sobre los prototipos también evolucionan con los mismos, son de carácter constante y prácticamente acompañan a todo el proceso de desarrollo; se prueban las funciones, los procedimientos y las interfaces de usuario, conforme estas van apareciendo, y se realiza una prueba integral antes de instalar un prototipo.

- ***Implantación – Uso/Prueba piloto***

  Superada las pruebas, se procede a implantar los prototipos. Existen dos clases de implantación de acuerdo con las características del sistema: de prueba piloto y de uso. Si el sistema se caracteriza por una fuerte interrelación entre cada una de sus partes (un subsistema necesita del otro para funcionar), la implantación necesariamente será a modo de prueba piloto; en este caso, el usuario solo utilizará al sistema con datos de prueba para verificar si la funcionalidad y el entorno generado por los desarrolladores se ajusta a lo solicitado. Si el sistema se caracteriza por poseer partes independientes las unas de las otras, la implantación puede ser de uso; en este caso, el usuario podrá hacer uso del subsistema implantado, podrá verificar las mismas cosas que con una prueba piloto y, además, empezar a sacar rédito de su inversión.

- ***Mantenimiento correctivo / perfectivo (refinado)***

  Independientemente del tipo de implantación que se realice, el cliente puede objetar cualquier aspecto del prototipo, en cuyo caso se realiza un refinamiento, para ajustar el prototipo a lo que realmente desea el cliente. Este refinamiento tiene la misma forma que el ciclo de desarrollo, se trabaja de la misma manera, pero la duración de las iteraciones no debe superar las dos semanas.

- ***Uso – Estabilidad***

  Si el cliente se encuentra conforme con el prototipo implantado, se puede esperar a tener otros prototipos antes de su uso oficial, o bien, puede empezar a funcionar por sí mismo. A partir de este momento, el sistema atravesará un período de estabilidad, en el cual se lo podrá utilizar sin la necesidad de

mantenimiento alguno. La duración de este período de estabilidad es variable y está estrictamente ligada a las características del ambiente en el que se desenvuelve el sistema.

- ***Mantenimiento adaptativo***

  El mantenimiento adaptativo es inevitable. Transcurrido un tiempo desde su implantación, los cambios en el entorno obligan a realizar diversas adaptaciones al sistema, ya sea de funcionalidad, de seguridad, etc. MMDS enfrenta el desarrollo de estas adaptaciones de la misma manera que al desarrollo inicial, teniendo especial cuidado de mantener siempre actualizada la documentación del sistema.

III.5. APLICACIÓN DE MMDS

MMDS se aplicó exitosamente en el desarrollo de software para un estudio de arquitectura en nuestra provincia, al que denominaremos Sistema de Gestión para Estudios de Arquitectura (SGEA). Logrando un producto de calidad y la satisfacción del cliente.

Los requisitos se capturaron utilizando únicamente historias de usuario. A continuación se presenta, a modo de ejemplo algunas historias.

N°: 1	**Usuario:** NN	**Fecha:**
Descripción	* El programa debe ser sencillo de utilizar. * Tiene que permitir corregir toda información que se ingrese. * El programa tiene dos partes una permite hacer presupuestos, y otra permite hacer un seguimiento de las obras, tanto los ingresos como los gastos de materiales y personal en cada obra. * Tiene que llevar el control de los pagos al personal. * Tiene que emitir informes de los pagos, y gastos por obra y personal. * Se tiene que poder trabajar en varias computadoras simultáneas.	
Observaciones	1) Se agrega una nueva funcionalidad, gestión de caja y cuenta del estudio. Ver tarjeta nro. 25 Fecha: ……..	

N°: 6	**Usuario:** NN	**Fecha:**
Descripción	* Existen 2 tipos de empleados, los de monto fijo y los de pago quincenal. * Se tiene que poder ingresar un empleado y poderlo agrupar en uno de estos dos tipos de empleado, a los de pago quincenal es obligatorio definirle un sueldo base. * Se tiene que poder modificar cualquier dato para corregir errores. * Se tiene que poder registrar adelantos y prestamos al personal.	
Observaciones		

N°: 10	**Usuario:** NN	**Fecha:**
Descripción	* Generar una cuenta de cada empleado de monto fijo. * La cuenta se genera con lo que estos empleados presupuestan por su trabajo en una obra. * Existe una probabilidad aunque baja de otorgar préstamos éste personal.	
Observaciones	1)Relacionado con la tarjeta 6	

N°: 11	**Usuario:** NN	**Fecha:**
Descripción	* Levar cuenta de cuanto se va pagado al empleado de monto fijo. * Al realizar cualquier pago controlar si tiene préstamos pendientes. * Imprimir un recibo. * Actualizar la información de los gastos en este tipo de personal en la/s obra/s correspondiente/s según los montos de pago.	
Observaciones	1) Relacionado con la tarjeta 10 2) No se debe permitir el pago total de lo adeudado al empleado si no se cancelaron los prestamos, o al menos advertir la situación.	

Se utilizó para este proyecto una base de datos con arquitectura cliente servidor, se codificó y diseñaron las ventanas y los informes de acuerdo a lo establecido por la lista de comprobación presentada anteriormente. El resultado fue el código correctamente documentado y fácil de seguir como se muestra a continuación.

```
//*********************************************************************
// Unidad: UPagosMF (Unidad Pagos a personal de Monto Fijo)
// Resumen: Esta unidad esta orientada a emitir los recibos al personal de
//          monto fijo, para lo cual toma la finformación de las asignaciones
//          de las asignaciones a obras (presupuestos de éste tipo de personal)
//          realizadas en la unidad UAsignaciónMontoFijo, y construye una cuenta
//          de lo adeudado y pagado al empleado. Además realiza los descuentos
//          respectivos por préstamos realizados a éstos empleados.
//          Además actualiza la información de los gastos en personal de monto fijo en
//
//Tarjetas de Historias de Usuario Vinculadas: directamente vinculada a esta unidad
//                              es la tarjeta nro 11
//*********************************************************************
unit UPagosMF;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, ..., ZDataset, Math;

type
  TFPagosMF = class(TForm)
    ...
  end;

var
  FPagosMF: TFPagosMF;
  CodigoEmpleado, CodPres:string;
  MontoCuota, totgral, totpagado, MontoPagado:double;
  CuotasPagadas, CantCuotas:integer;

implementation
```

```
uses SQL_U, Funciones, UDataModule, format_mysql, UReciboMF;

{$R *.dfm}

Procedure LimpiarGrillas;
var
  i:integer;
begin
//*********************************************************************
// Descripción: Este procedimiento se encarga de limpiar los datos de las grillas
//              en donde se presenta la información al usuario.
//
// Parametros de entrada: No recibe ningún parámetro
// Datos devueltos: No devuelve datos
//*********************************************************************
 i:= 1;
 while i < FPagosMF.SGObras.RowCount do
     begin
       with FPagosMF do
         begin
           SGObras.Cells[0,i] := '';
           SGObras.Cells[1,i] := '';
           SGObras.Cells[2,i] := '';
           SGCod.Cells[0,i] := '';
           SGCod.Cells[1,i] := '';
           SGCod.Cells[2,i] := '';
         end;
       i:=i+1;
     end;
  FPagosMF.SGObras.Cells[0,0] := 'Obras';
  FPagosMF.SGObras.Cells[1,0] := 'Tot. Cobrado';
  FPagosMF.SGObras.Cells[2,0] := 'Tot. Pagado';
  FPagosMF.SGCod.Cells[0,0] := 'CodEmpleadoObra';
  FPagosMF.SGCod.Cells[1,0] := 'Tot. Cobrado';
  FPagosMF.SGCod.Cells[2,0] := 'Tot. Pagado';
  FPagosMF.SGObras.RowCount := 2;
  FPagosMF.SGCod.RowCount := 2;
end;
//-----------------------------------------------------------------------------------------------------
...
//-----------------------------------------------------------------------------------------------------
procedure TFPagosMF.BConfirmarPagoClick(Sender: TObject);
var
  error, cuotas_restantes, CC:integer;
  PagoFinal, SaldoPrestamo, DebeAlEmp:double;
begin
//*************************************************************************
// Descripción: Este procedimiento se origina al presionar el boton de Confirmar
//              Pago. Este procedimiento es el que se encarga de controlar los
//              datos que se grabarán en la base de datos; controla que el usuario
//              ingrese correctamente el monto a pagar, que el monto no supere lo
//              que se adeuda al personal.
//
// Parametros de entrada:  Sender (puede hacer uso de todos los objetos definidos
//                         en el proyecto)
// Datos devueltos: No devuelve datos
//*************************************************************************
  error :=0;
  if (Trim(EMonPag.Text) <> '') then
    begin
```

```
    PagoFinal := totgral - (totpagado + StrToFloat(EMonPag.Text));
    if PagoFinal = 0 then
      begin
        if (CantCuotas > 0) then
          begin
            if (CuotasPagadas < CantCuotas) then
              begin
                error := 1;  // tiene cuotas pendientes y le esta pagando todo lo que le debe
              end;
          end ;
      end
      else
      begin
        if PagoFinal < 0 then
          begin
            error := 2; // le esta pagando más de lo que le debe
          end  ;
      end;
  end
 else error := 3; //no cargo el monto a pagarse
case error of
 0: begin
      if CantCuotas = 0 then
        begin
          // realiza el pago y no hay prestamo
          FReciboMF.LApeYNom.Caption := SQL_U.DataAsString(QEmpleado,'ApeYNom');
          FReciboMF.LFecha.Caption := DateToStr(FechaPago.Date);
          FReciboMF.MObra.Lines.Clear;
          FReciboMF.MMC.Lines.Clear;
          Pagar(0, true);
          FReciboMF.QR.Preview;
        end
        else
        begin
          cuotas_restantes := CantCuotas - CuotasPagadas;
          SaldoPrestamo := cuotas_restantes * MontoCuota;
          DebeAlEmp := totgral-(totpagado+StrToFloat(EMonPag.Text));
          if DebeAlEmp >= SaldoPrestamo then
            begin
              // realiza el pago y no me importa si hay prestamo
              …
              if (ConfPrestamo.Checked) then
                begin
                  Pagar(1, false);
                end
                else
                begin
                  Pagar(0, true);
                end;
              FReciboMF.QR.Preview;
            end
            else
            begin
              // realiza el pago y reduce las cuotas necesarias del prestamo cuota del prestamo prestamo
              CC := 1;
              while DebeAlEmp < (SaldoPrestamo-MontoCuota*CC) do
                begin
                  CC := CC + 1;
                end;
               ...
```

```
                Pagar(CC,false);
                FReciboMF.QR.Preview;
              end;
          end;
        LimpiarForm(Sender);
      end;
    1: begin
        if ((PagoFinal = 0)and (CantCuotas > 0)) then
          begin
            cuotas_restantes := CantCuotas - CuotasPagadas;
            if cuotas_restantes > 1 then
              begin
                SaldoPrestamo := cuotas_restantes * MontoCuota;
                DebeAlEmp := totgral-totpagado;
                if DebeAlEmp > SaldoPrestamo then
                  begin
                    if ConfPrestamo.Checked then
                      begin
                        if MessageDlg('El empleado debe más de una cuota, si continua se descontará'+
                                      ' del pago el total de su deuda.'+#13+'¿Desea Continuar?'
                                      ,mtWarning, [mbYes, mbNo], 0) = mrYes then
                          begin
                            // realizar el pago y la cancelacion del prestamo
                            CC := 1;
                            while (SaldoPrestamo-MontoCuota*CC)> 0 do
                              begin
                                CC := CC + 1;
                              end;
                             ...
                            Pagar(CC,false);
                            FReciboMF.QR.Preview;
                            LimpiarForm(Sender);
                          end
                      end
                    else
                      begin
                        if MessageDlg('El empleado adeuda varias cuotas.'+#13+'¿Desea
                          Continuar?',mtWarning, [mbYes, mbNo], 0) = mrYes then
                          begin
                            // realizar el pago y la cancelacion del prestamo
                            CC := 1;
                            while (SaldoPrestamo-MontoCuota*CC)> 0 do
                              begin
                                CC := CC + 1;
                              end;
                            ...
                            Pagar(CC,false);
                            FReciboMF.QR.Preview;
                            LimpiarForm(Sender);
                          end;
                      end;
                  end;
              end
            else
              begin
                if ConfPrestamo.Checked then
                  begin
                    // realizar el pago y la cancelacion del prestamo
                    …
                    Pagar(1, false);
```

```
                    FReciboMF.QR.Preview;
                    LimpiarForm(Sender);
                  end
                 else
                  begin
                    if MessageDlg('El empleado adeuda una cuota.'+#13+'¿Desea continuar?',mtWarning,
                       [mbYes, mbNo], 0) = mrYes then
                      begin
                        // realizar el pago y la cancelacion del prestamo
                        …
                        Pagar(1, false);
                        FReciboMF.QR.Preview;
                        LimpiarForm(Sender);
                      end;
                  end;
              end;
          end ;
      end;
    2: begin
         MessageDlg('El monto del pago supera el monto adeudado.', mtError,[mbOk], 0);
         EMonPag.SetFocus;
       end;
    3: begin
         MessageDlg('Debe ingresar el monto a pagar.', mtError,[mbOk], 0);
         EMonPag.SetFocus;
       end;
  end;
end;
//---------------------------------------------------------------------------------------------------
…
```

Con respecto al diseño de ventanas e informes, el mismo es uniforme e intuitivo. Este diseño se pactó con el usuario buscando la facilidad de uso, que era uno de los requisitos fundamentales para el software. A continuación se presentan dos ventanas que se refieren a diferentes funciones del sistema como ejemplificación de este diseño.

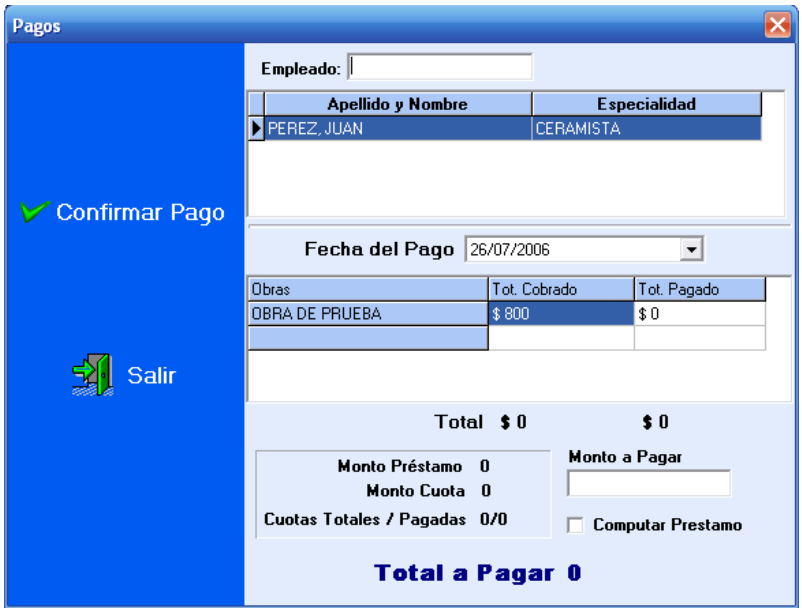


Figura 3.6. Ventana Pagos

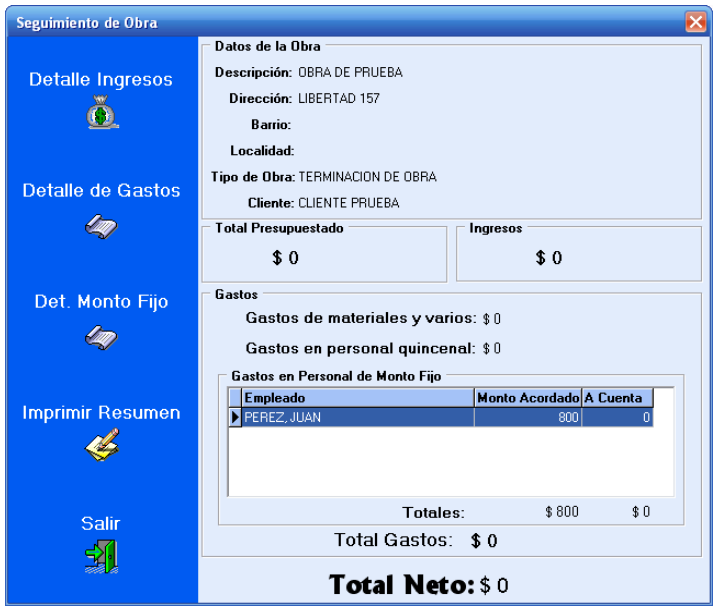


Figura 3.7. Ventana Seguimiento de Obra

Con respecto al mantenimiento preventivo, se aplicó refactorización. La técnica de refactorización más usada fue reemplazar código repetido por procedimientos o funciones según fuese el caso. Un ejemplo claro de este caso es el procedimiento *LimpiarGrillas* presentado anteriormente; el mismo se encuentra repetido en varias unidades, si bien está particularizado para cada una de ellas con los títulos de las columnas y las dimensiones de las grillas que corresponden según las características de la información que se muestra, el contenido y funcionalidad del procedimiento es similar, por lo que una alternativa sería extraer el procedimiento de todas las unidades, dejando sólo su invocación o llamado, y generar una nueva unidad donde ubicar un procedimiento común y general que cumpla con esta funcionalidad.

El procedimiento *LimpiarGrillas* quedaría formulado entonces de la siguiente manera.

```
Procedure LimpiarGrillas(var SG: TStringGrid; SGEncabezados:TStringGrid);
var
  i, j:integer;
begin
//*****************************************************************************
// Descripción: Este procedimiento se encarga de limpiar los datos de las grillas
//                en donde se presenta la información al usuario.
//
// Parámetros de entrada: 1) SG: TStringGrid --  Contiene la grilla de datos que se muestran al usuario,
//                           las modificaciones en este parámetro se reflejan en la grilla del form.
//                        2) SGEncabezados: TStringGrid – Contiene los encabezados de las columnas
//                           de SG
// Datos devueltos: No devuelve datos
//*****************************************************************************
```

```
// Limpio la grilla de datos
 i:= 0;
 while i < SG.RowCount do
     begin
.........j = 0;
        while j < SG.ColCount do
          begin
            SG.Cells[j,i] := '';
.............j := j + 1;
          end;
        i:= i + 1;
     end;

// Cargo los encabezados de las columnas
   SG.ColCount := SGEncabezados.RowCount;
   i=0;
   while i < SG.ColCount do
   begin
     SG.Cells[i,0] := SGEncabezados.Cells[0,i];
     i:= i + 1;
   end;
   SG.RowCount := 2;
end;
```

La aplicación de MMDS en este proyecto ha demostrado poder conseguir un producto de calidad logrando la satisfacción del cliente en lo referente al diseño, la facilidad de uso y la completitud de la información que puede obtener del sistema. El producto se encuentra actualmente en uso atravesando un período de estabilidad.

CAPÍTULO IV

EVALUACIÓN DE RESULTADOS

IV.1. INTRODUCCIÓN

En este capítulo se confrontará el desarrollo de software propuesto por MMDS contra el desarrollo tradicional. Dicha confrontación se realizará tomando las variables tiempo y costo insumidos para el desarrollo, para lo cual se hará uso de estimaciones obtenidas mediante el modelo COCOMO y los datos reales de la aplicación de MMDS en el desarrollo del sistema para un estudio de arquitectura.

Además, debido a que una de las premisas de ésta investigación es lograr un producto de buena calidad, se evaluarán algunos aspectos que hacen a ésta característica, ellos son: corrección, facilidad de mantenimiento, facilidad de uso y reusabilidad.

IV.2. COMPARACIÓN ENTRE DESARROLLO CON MMDS Y TRADICIONAL

El producto SGEA que se desarrolló con MMDS cuenta, en su versión final, con 16.128 líneas de código. Fue desarrollado por un equipo con dos integrantes. El desarrollo tomó tres meses y diecinueve días, período durante el cual se realizaron ocho iteraciones.

Para contrastar estos valores respecto a los que podrían obtenerse con desarrollo tradicional, se realizará una estimación utilizando el modelo COCOMO, introduciéndose como, dato de entrada, el valor real del número de líneas de código, buscando de esta manera mejorar la precisión de la estimación [36].

Al ser SGEA un proyecto de tamaño pequeño, se optó por el modelo básico y el modo orgánico. Las ecuaciones definidas por COCOMO en este modelo se plantean de la siguiente manera:

$E = ab\ (KLDC)^{bb}$

$D = cb\ (E)^{db}$

$P = E / D$

Donde *E* es el esfuerzo aplicado en persona-mes, *D* es el tiempo de desarrollo en meses, *KLDC* es el número de líneas estimadas para el proyecto (en miles) y *P* es el número de personas necesarias.

Los coeficientes *ab*, *bb*, *cb* y *db* para el modo orgánico son los siguientes:

$ab = 2,4$	$cb = 2,5$
$bb = 1,05$	$db = 0,38$

De la aplicación de las ecuaciones existentes surge:

$$E = 2,4\ (16,12)^{1,05} = 44,457 \text{ mes / persona}$$
$$D = 2,5\ (44,457)^{0,38} = 10,57 \text{ meses}$$
$$P = 44,457 / 10,57 = 4,2 \cong 4 \text{ personas}$$

El costo de un producto software es subjetivo y depende de los desarrolladores, ya que está sujeto a la situación y al lugar en donde ellos ejerzan su profesión. No obstante, el costo está estrictamente ligado a los factores de esfuerzo, duración y número de personas que intervinieron en su desarrollo.

IV.3. EVALUACIÓN DE CALIDAD DEL PRODUCTO OBTENIDO

La norma ISO 8402 define calidad como: "*conjunto de propiedades y características de un producto o servicio que le confieren aptitud para satisfacer unas necesidades explícitas o implícitas*".

En la ingeniería del software se puede encontrar definiciones más específicas para calidad de productos software. El IEEE, en su estándar STD 610 – 1990, la define como el grado con el que un sistema, componente o proceso cumple con los requerimientos especificados y las necesidades o expectativas del cliente o usuario. Por su parte, Pressman [36], la define como la concordancia del software producido con los requerimientos explícitamente establecidos, con los estándares de desarrollo prefijado y los requerimientos implícitos que desea el usuario y que no fueron establecidos formalmente.

Existe un gran número de factores que afectan la calidad de un producto software, McCall los divide en dos grupos amplios: los factores que se pueden medir directamente (por ejemplo, defectos por punto de función), y los factores que sólo pueden medirse indirectamente (por ejemplo, la facilidad de uso o mantenimiento) [37].

McCall propone una clasificación útil de los factores que afectan a la calidad del producto concentrándose en tres aspectos importantes del mismo, como ser sus características operativas, su capacidad de cambios y su adaptabilidad a nuevos entornos (figura 4.1).

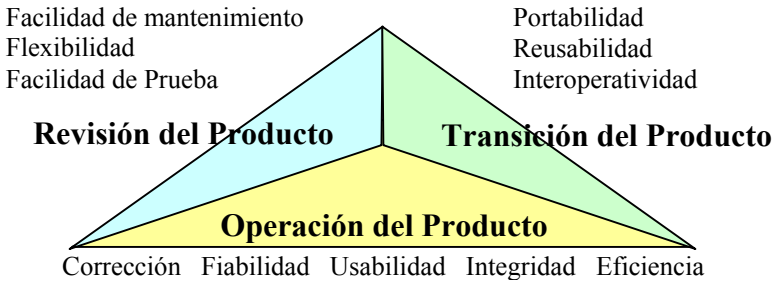


Figura 4.1. Factores de calidad de McCall [37]

Respecto del SGEA se evaluarán, cuatro atributos de calidad: la facilidad de mantenimiento, la reusabilidad, la corrección y, por último, la usabilidad o facilidad de uso. Nótese que dos de estos atributos están dirigidos a la operación del producto, ya que el método MMDS se centra sobre todo en el usuario.

IV.3.1. FACILIDAD DE MANTENIMIENTO

El mantenimiento del software es una de las actividades, dentro de la ingeniería del software, a la que se dedica un gran esfuerzo. La facilidad de mantenimiento es la facilidad con la que se puede corregir un programa si se encuentra un error, se puede adaptar si su entorno cambia, o mejorar si el cliente desea un cambio en los requisitos. Al no existir una forma directa de medir la facilidad de mantenimiento, se deben recurrir a medidas indirectas.

Para medir la facilidad de mantenimiento pueden utilizarse, por ejemplo, métricas orientadas al costo, o métricas orientadas al tiempo [37]. Una métrica orientada al costo es la métrica de desperdicios utilizada por Hitachi, que consiste en el costo de

corregir defectos encontrados después de haber distribuido el software a sus usuarios finales. Una métrica simple orientada al tiempo es el *Tiempo Medio de Cambio* (*TMC*), que consiste en promediar las mediciones del tiempo que se tarda en analizar la petición del cambio, diseñar e implementar una modificación adecuada, probar dicha modificación y, finalmente, distribuir el cambio a todos los usuarios. Es evidente que los programas con un *TMC* bajo son fáciles de mantener.

Para evaluar la facilidad de mantenimiento del SGEA se utilizó como indicador el *TMC*. Debido a la elevada interacción con el cliente se produjo una gran cantidad de cambios (se realizaron 28 cambios) durante el desarrollo; la mayoría de ellos fueron cambios pequeños que no tomaron más de un día en implementarse, probarse y distribuirse, siendo el promedio aproximado de su duración 1,13 días. Cabe mencionar que se trabajó sólo con jornadas diarias de 6 horas aproximadamente, por lo que este valor de *TMC* es bastante alentador.

IV.3.2. REUSABILIDAD O CAPACIDAD DE REUTILIZACIÓN

McCall [37] define la reusabilidad como la capacidad con la que un programa, o parte de él, puede usarse en otras aplicaciones en relación al empaquetamiento y al alcance de las funciones que realiza el programa.

Existen varias formas de medir la reusabilidad. Las métricas para el diseño orientado a objetos (Metrics for Object Oriented Design – MOOD) permiten medir los principales mecanismos, tales como encapsulamiento, herencia, polimorfismo, etc. Estas métricas poseen dos factores que pueden usarse para estimar la reusabilidad, que son el Factor de Herencia de Métodos (MIF – Method Inheritance Factor) y el Factor de Herencia de Atributos (AIF – Attribute Inheritance Factor).

Fuera de las métricas MOOD, una medida mucho más simple para evaluar el nivel de reusabilidad consiste en expresar un valor porcentual del código reusable (*CR*) mediante un cociente entre la cantidad total de líneas reusables (*TLR*) y la cantidad total de líneas del programa (*TLP*).

Para medir la reusabilidad del SGEA, se utilizó la tercera de éstas alternativas arrojando el siguiente valor: $CR = TLR / TLP = 1280 / 16128 = 0{,}079$. El valor de *TLR* es de 1280, este valor representa todas las líneas de código contenidas en las bibliotecas de funciones y clases, que pueden utilizarse tal cual están en el desarrollo de cualquier

software siempre que sean requeridas. El *CR* indica que, aproximadamente el 8 % del código es reutilizable, este resultado es aceptable, considerando que el producto que se desarrolló es un software a medida, y que la excesiva reusabilidad hace a los sistemas más difíciles de comprender y mantener.

IV.3.3. CORRECCIÓN

La *corrección* es el grado con el que el software lleva a cabo su función requerida. Un programa que no opera correctamente proporciona poco valor a sus usuarios.

Una de las medidas más comunes para la corrección es: *defectos* / *KLDC*, donde *defecto* es una falta verificada de conformidad con los requisitos y *KLDC* es una medida que representa miles de líneas de código [17].

En el transcurso del desarrollo y uso del software para el estudio de arquitectura, el cliente manifestó su falta de conformidad con los requisitos iniciales en tres ocasiones, las cuales fueron resueltas a medida que aparecieron.

Entonces, aplicando este valor en la fórmula expresada anteriormente, se tiene lo siguiente:

corrección = 3 / 16128 = 0,000186

La interpretación de este resultado sería que se tiene un software correcto, que satisface a los usuarios y al cliente, puesto que este valor representa menos de un 1%. Esta afirmación se radica en el hecho de que, la fórmula presentada por Gilb [17], muestra a la *corrección* como un valor porcentual de defectos respecto del total de líneas de código, es así que, cuando menor sea el valor arrojado como resultado, mayor será el grado de corrección que posea el software.

IV.3.4. FACILIDAD DE USO

Esta característica tiene especial importancia en el software para el estudio de arquitectura, puesto que la facilidad de uso era un requisito explícito del cliente. Esto no resulta nada extraño ya que, en la actualidad, la frase “amigable con el usuario” tiene un gran peso sobre todos aquellos que desarrollen productos software. Esto es así porque un software que no es amigable está, por lo general, destinado al fracaso, incluso si sus funciones son valiosas para el usuario.

La facilidad de uso es un intento de medir la amigabilidad de un software hacia el usuario. Este atributo de calidad no puede medirse directamente y, por lo general, se mide en función de cuatro características:

- Habilidad intelectual y/o física requerida por el usuario para aprender el sistema.
- El tiempo requerido por el usuario para llegar a ser moderadamente eficiente en el uso del sistema.
- Aumento neto en productividad, intenta medir si el usuario utiliza el sistema eficientemente.
- Valoración subjetiva de la disposición de los usuarios hacia el sistema mediante el uso de un cuestionario.

Para medir la facilidad de uso en el software para el estudio de arquitectura, se utilizaron dos cuestionarios que trabajan con escalas subjetivas para la usabilidad del software (Subjective Usability Scales for Software – *SUS*), uno orientado hacia el uso del software en general y el otro orientado hacia la amigabilidad de la interfaz.

El propósito del *SUS* es proporcionar un cuestionario fácil de completar, fácil de puntuar y que permita establecer comparaciones cruzadas entre productos. Se utiliza después de que un usuario ha tenido la oportunidad de utilizar un sistema. La escala *SUS* es un único número representando una medida compuesta de la usabilidad del sistema global sometido a estudio [29].

- Cuestionario Nº 1

	Completamente en desacuerdo 1	2	3	4	Completamente de acuerdo 5
1. Creo que me gustará usar este sistema frecuentemente.					
2. Encuentro al sistema innecesariamente complejo.					
3. Pienso que el sistema es fácil de usar.					
4. Creo que necesitaría el soporte de personal técnico para poder usar este sistema.					
5. Creo que las diferentes funciones de este sistema están bien integradas.					
6. Creo que existen demasiadas inconsistencias en el sistema.					
7. Creo que la mayoría de las personas pueden aprender muy rápido como usar este sistema.					
8. Me pareció que el sistema es muy lento en su uso.					
9. Me siento seguro usando el sistema.					
10. Necesito aprender muchas cosas antes de poder acceder al sistema.					

SUS arroja un número simple que representa una medida compuesta de la usabilidad de todo el sistema estudiado. Cabe destacar que, por sí solos, los valores registrados individualmente no tienen resultados significativos.

Para calcular la puntuación de este cuestionario *SUS*, primero deben sumarse los puntajes asignados a cada ítem, los cuales pueden variar en un rango de 0 a 4. En Los ítems 1, 3, 5, 7 y 9 el valor con el que contribuyen se calcula restándole 1 a su posición en la escala; en los ítems 2, 4, 6, 8 y 10 el valor con el que contribuyen se obtiene restando a 5 la escala de posición. Finalmente, para obtener el valor global de *SUS*, se multiplica la suma de las contribuciones de los diferentes ítems por el valor 2,5.

Los valores de *SUS* tienen un rango de 0 a 100.

Para ejemplificar el procedimiento de ponderación, se presenta el siguiente cuestionario realizado por uno de los usuarios.

	Completamente en desacuerdo 1	2	3	4	Completamente de acuerdo 5
1. Creo que me gustará usar este sistema frecuentemente.				✓	
Puntos 4 – 1 = 3					
2. Encuentro al sistema innecesariamente complejo.		✓			
Puntos 5 – 2= 3					
3. Pienso que el sistema es fácil de usar.					✓
Puntos 5 – 1 = 4					
4. Creo que necesitaría el soporte de personal técnico para poder usar este sistema.	✓				
Puntos 5 – 1 = 4					
5. Creo que las diferentes funciones de este sistema están bien integradas.				✓	
Puntos = 3					
6. Creo que existen demasiadas inconsistencias en el sistema.	✓				
Puntos = 4					
7. Creo que la mayoría de las personas pueden aprender muy rápido como usar este sistema.					✓
Puntos = 4					
8. Me pareció que el sistema es muy lento en su uso.	✓				
Puntos = 4					
9. Me siento seguro usando el sistema.				✓	
Puntos = 3					
10. Necesito aprender muchas cosas antes de poder acceder al sistema.	✓				
Puntos = 4					

Puntaje aportado por las marcas = 36

Puntaje total de SUS = 36 * 2,5 = 90

El sistema tiene en total cuatro usuarios. SUS arrojó los siguientes valores:

Usuario 1 = 90

Usuario 1 = 85

Usuario 1 = 92

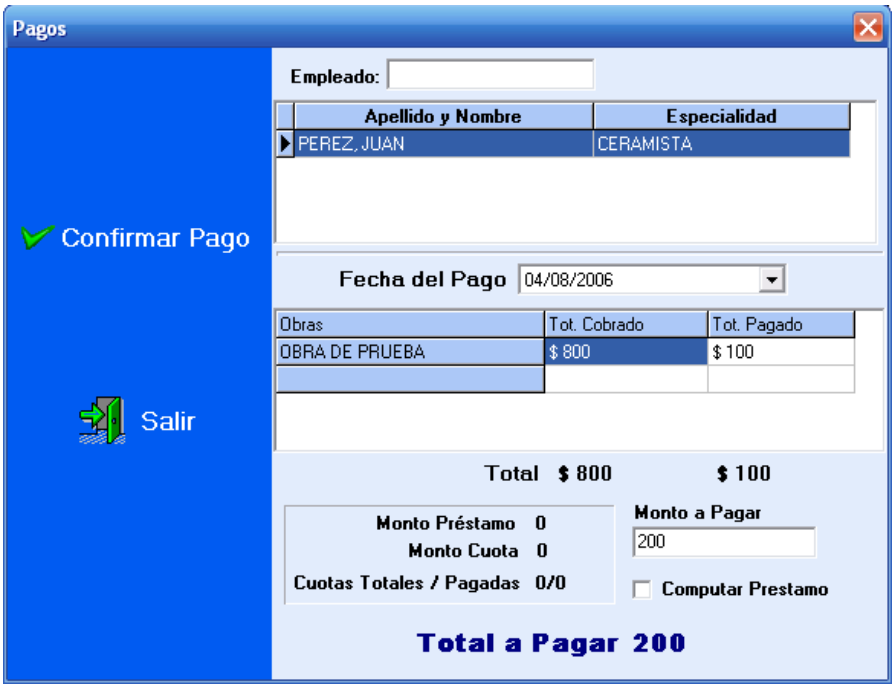
Usuario 1 = 92,5

Como se puede observar, el cuestionario antes presentado, esta dirigido a medir la amigabilidad del sistema. Para medir la facilidad de uso de la interfaz, se utilizó el siguiente cuestionario.

- Cuestionario Nº2:

A continuación se presenta parte de la interfaz de usuario,

Estudie la imagen durante un minuto, y luego responda las siguientes preguntas marcando la respuesta correcta.



1. ¿Le gusta este diseño en particular?

Lo odio											Me encanta
	0		1		2		3		4		5

2. ¿Es atractivo o estéticamente agradable este diseño?

Horrible											**Hermoso**
	0		**1**		**2**		**3**		**4**		**5**

3. ¿Cómo calificaría el diseño gráfico?

Pobre											**Excelente**
	0		**1**		**2**		**3**		**4**		**5**

4. ¿Es fácil de entender e interpretar éste diseño?

Difícil											**Fácil**
	0		**1**		**2**		**3**		**4**		**5**

5. ¿Le resulta sencillo aprender este diseño?

Difícil											**Fácil**
	0		**1**		**2**		**3**		**4**		**5**

Basándose en lo que entendió de la figura anterior, responda las siguientes preguntas.

6. ¿Le resulta fácil encontrar a un empleado?

Imposible											**Extremadamente Fácil**
	0		**1**		**2**		**3**		**4**		**5**

7. ¿Le resulta fácil comprender la información del estado de cuenta del empleado?

Imposible											**Extremadamente Fácil**
	0		**1**		**2**		**3**		**4**		**5**

8. ¿Le resulta fácil comprender la información de préstamos hechos al empleado?

Imposible											**Extremadamente Fácil**
	0		**1**		**2**		**3**		**4**		**5**

9. ¿Cuanto se agiliza su trabajo con respecto a la liquidación de sueldo?

Lento											**Ágil**
	0		**1**		**2**		**3**		**4**		**5**

10. Por último, ¿como calificaría a la facilidad de uso de este diseño?

Muy difícil											**Muy Fácil**
	0		**1**		**2**		**3**		**4**		**5**

Al igual que en el cuestionario anterior, en éste también se construye una escala que puede tomar valores de cero a cien. Para obtener esta escala, se suman los valores obtenidos de cada marca, resultado que posteriormente se multiplica por dos para arribar al resultado final.

Al evaluar los cuestionarios SUS se obtuvieron los siguientes valores en la escala: 90, 92, 82 y 88 correspondientemente, que representan a valores por demás buenos en esta escala.

IV.4. DISCUSIÓN

Del análisis de las evaluaciones realizadas surgen afirmaciones e inferencias interesantes y dignas de destacar.

En lo referente a los valores del costo del desarrollo del producto, se puede afirmar que el costo del desarrollo con MMDS es mas bajo, si bien no se realizó una estimación formal del mismo. Esta afirmación se fundamenta en bases reales, ya que fue desarrollado por menos personas y en menor tiempo de los arrojados por el modelo COCOMO, por lo tanto también insumió menos esfuerzo, factor altamente influyente a la hora de evaluar costos.

En lo referente a la evaluación de la calidad del producto, y a diferencia de otras ramas de las ciencias exactas, la ingeniería del software es relativamente joven, por lo que los métodos de evaluación de calidad están aún en desarrollo constante. A pesar de que se pueden determinar muchos atributos que hacen a la calidad de un producto software, no existen aún indicadores que permitan medir en forma objetiva y precisa la mayoría de ellos, ya que la medición se realiza con indicadores subjetivos.

En el presente trabajo se seleccionaron cuatro atributos, dos de ellos orientados a la perspectiva del cliente (facilidad de uso y corrección), mientras que los otros dos orientados y de gran utilidad para los desarrolladores puesto que les permite inferir a cerca de la facilidad de mantenimiento y de la cantidad de código reusable que se puede obtener en un desarrollo.

IV.5. CONCLUSIÓN

Concluyendo sobre la comparación realizada entre MMDS con los valores estimados por el modelo COCOMO respecto de la duración y esfuerzos del desarrollo de software, se puede afirmar que los resultados obtenidos fueron muy satisfactorios, posicionando a MMDS en un buen nivel, con tiempo y esfuerzo de desarrollo menor que los estimados por COCOMO para un método tradicional de desarrollo.

Con respecto a la evaluación de la calidad del producto desarrollado con MMDS, los resultados obtenidos son bastante alentadores, muy buenos alcanzando con holgura todas las expectativas y logrando la satisfacción de los clientes y usuarios.

Se consiguió un producto fácil de mantener y con un nivel de reusabilidad aceptable. Además, el producto que se logró es correcto, lo que se ve respaldado por una muy buena aceptación del cliente en lo que respecta a conformidad con los requisitos. Por último, se puede afirmar que, como resultado de la evaluación de la facilidad de uso, tanto del software en general como así también de la interfaz gráfica que presenta el mismo, se alcanzaron valores muy altos en una escala de usabilidad de cero a cien.

CONCLUSIONES

Del gran abanico de métodos para el desarrollo de software, existentes en la actualidad, la generación de MMDS se basó fundamentalmente en los MA, para darle capacidad de adaptación y flexibilidad a sus procesos. El desarrollo del presente trabajo se sustenta en la combinación de varios MA, buscando obtener un método de desarrollo de software útil para Santiago del Estero con características de adaptabilidad y flexibilidad.

MMDS demostró ser un método confiable para el desarrollo de software de pequeña y mediana envergadura. Se mostró como un método flexible y adaptable, cualidades que heredó fundamentalmente de los métodos XP, ASD y EVO.

La buena calidad del producto obtenido está sustentada en las características de su ciclo de vida, que lo convierten en una guía útil y digna de considerar a la hora de emprender cualquier desarrollo.

El desarrollo iterativo e incremental, conjuntamente con una activa participación de los usuarios y el cliente, facilitó la obtención de un producto en poco tiempo y bajo costo de desarrollo.

Además, se logró un muy buen nivel de aceptación del usuario hacia el sistema desarrollado.

La evaluación de los resultados arrojó valores altamente satisfactorios. El producto desarrollado con MMDS mostró tiempo y esfuerzo de desarrollo menores que los estimados por COCOMO para un método tradicional. Las evaluaciones de los atributos de calidad arrojaron resultados muy alentadores alcanzando con holgura y superando todas las expectativas planteadas, lográndose también la satisfacción de los clientes y usuarios.

No obstante los buenos resultados alcanzados, no debe olvidarse que MMDS no está pensado para el desarrollo de grandes sistemas software, y no debe ser presionado

más allá de sus límites. Cualquier adaptación de MMDS para este tipo de desarrollo queda fuera del alcance de este trabajo y abierta a futuras investigaciones.

Por último, luego de los argumentos expresados, sólo resta concluir que el fin y los objetivos de este trabajo se alcanzaron satisfactoriamente.

BIBLIOGRAFÍA Y REFERENCIAS

[1] Abián, M. (2003). Promesas Incumplidas: Sobre los Métodos Ágiles. Disponible en: ji.ehu.es/isw/Metodos_Agiles_Promesas_Incumplidas.pdf. Fecha de acceso: septiembre 2005.

[2] Aguilar Sierra, A. (2003). Introducción a la Programación Extrema. Disponible en: www.willydev.net/descargas/articulos/general/IntroXP.PDF. Fecha de acceso: agosto 2005.

[3] Barbero, M. (2005). Trabajando con Programación Extrema – Experiencias Reales. V Jornadas de Software Libre en Asturias; 14 al 16 de Marzo de 2005. Oviedo (España).

[4] Beck, K. (2000). Extreme Programing Explained: Embrace Change. Ed. Addison-Wesley.

[5] Beck, K.; Beedle, M.; Van Bennekum, A.; Cockburn, A.; Cunnimgham, W.; Fowler, M.; Grenning, J.; Highsmith, J.; Hunt, A.; Jeffries, R.; Kern, J.; Marick, B.; Martin, R.; Mellor, S.; Schwaber, K.; Sutherland, J. y Thomas D. (2001). "Agile Manifesto". Disponible en: http://agilemanifesto.org/. Fecha de acceso: octubre de 2005.

[6] Canós, J.; Letelier, P.; Penadés, M. (2003). Metodologías Ágiles en el Desarrollo de Software. VIII Jornadas de Ingeniería del Software y Bases de Datos, 12 al 14 de Noviembre de 2003. Alicante (España).

[7] Cockburn, A. (2002). Aprendiendo del Desarrollo de Software Ágil – Primera Parte. Disponible en: http://www.agile-spain.com. Fecha de acceso: septiembre de 2005.

[8] Cockburn, A. (2002). Aprendiendo del Desarrollo de Software Ágil – Segunda Parte. Disponible en: http://www.agile-spain.com. Fecha de acceso: septiembre de 2005.

[9] Cockburn, A. (2002). Agile Software Development. Ed. Addison Wesley.

[10] Davies, Rachel. Agile Requirements. Disponible de http://www.methodsandtools.com/archive/archive.php?id=27. Fecha de acceso: junio de 2006.

[11] Diez, Eduardo. Calidad del Software. CMM - Capability Maturity Model. Disponible en: www.itba.edu.ar/capis/rtis/articulosdeloscuadernosetapaprevia/DIEZ-CMM.pdf. Fecha de acceso: marzo de 2006.

[12] Esposito, D. (2003). Escalabilidad, Dulce Escalabilidad. Disponible en: http://www.microsoft.com/spanish/msdn/articulos/archivo/180501/voices/data03082001.asp. Fecha de acceso: febrero de 2006.

[13] Fowler, Martin (2003). La Nueva Metodología. Disponible en: www.programacionextrema.org/articulos/newMethodology.es.html. Fecha de acceso: agosto 2005.

[14] Fowler, Martin. (1999). Refactoring: Improving the Desing of Existing Code. Ed. Addison Wesley.

[15] Fowler, Martin (2000). ¿Ha muerto el Diseño? Disponible en: http://www.programacionextrema.org/articulos/designdead.es.html. Fecha de acceso: septiembre 2005.

[16] Gabardini, J.; Campos, L. (2004). Balanceo de Metodologías Orientadas al Plan y Ágiles. Herramientas para la selección y adaptación. PMI Global Congress Proceedings. Buenos Aires, Argentina.

[17] Gilb, T. (1988). Principles of Software Engineering Management. Ed. Addison Wesley.

[18] Gilb, Kai (2003). Evolutionary Project Management & Product Development (or The Whirlwind Manuscript). Disponible en: http://www.gilb.com/ Download/EvoProjectMan.pdf. Fecha de acceso: febrero de 2006.

[19] Gómez García, Manuel A. (2002). Parte 7: Guía Para La Mejora de Procesos Software. Publicado por SPICE.

[20] Gramajo, E., García-Martínez, R., Rossi, B., Claverie, E. Y Britos, P. Combinación de Alternativas para la Estimación de Proyectos Software. Disponible en: www.itba.edu.ar/capis/webcapis/RGMITBA/articulosrgm/R-ITBA-22-estimacion.pdf. Fecha de acceso: marzo de 2006.

[21] Highsmith, Jim (2002). Agile Software Development Ecosystems. Ed. Addison Wesley.

[22] Highsmith, Jim (2002). What Is Agile Software Development?. CROSSTALK The Journal of Defense Software Engineering. Disponible en: www.stsc.hill.af.mil/crosstalk/2002/10/highsmith.html. Fecha de acceso: febrero de 2006.

[23] Hurtado, Julio A.; Bastarrica, Cecilia (2005). Hacia Una Línea de Procesos Ágiles. Proyecto SIMEP-SW, Universidad de Cauca (Colombia).

[24] IEEE Computer Society Press, 1993. IEEE Std 729-1993, IEEE Software Engineering Standard 729-1993: Glossary of Software Engineering Terminology.

[25] ISO.SPICE Home Page (2005). Standard. Disponible en: http://www.isospice.com /standard/tr15504.htm y www.isospice.typepad.com/isospice_is15504/. Fecha de acceso: marzo 2006.

[26] Jacobson, I. (1998). Applying UML in The Unified Process. Presentación. Rational Software. Presentación disponible: en http://www.rational.com/uml como UMLconf.zip. Fecha de acceso: noviembre de 2005.

[27] Jacobson, I.; Booch, G.; Rumbaugh, J. (1999). El proceso unificado de desarrollo de software. Ed. Addison Wesley.

[28] Jacobson, I.; Booch, G.; Rumbaugh, J. (1999). El Lenguaje de Modelado Unificado. Ed. Addison Wesley.

[29] Lozano, María; Montero, Francisco. (2003). Nuevos Mecanismos para el Desarrollo de Sistemas Interactivos de Calidad. Publicado en: XIII Escuela de Verano de Informática, sobre: Tendencias Actuales en la Interacción Persona-Ordenador: Accesibilidad, Adaptabilidad y Nuevos Paradigmas. Universidad de Castilla La Mancha. España. Disponible en: http://www.info-ab.uclm.es/personal/caballer/download/papers/CursoVerano2003.pdf . Fecha de acceso diciembre de 2005.

[30] Maguire, Steve. (1994). Código Sin Errores. Ed. Mc Graw Hill – Microsoft Press.

[31] Manuel, José (2005). ¿Serán la Teoría del Caos y un Enfoque Despreocupado las Claves para un Desarrollo de Proyectos Mejor y Más Rápido? Disponible en: www.wikilearning.com/software_y_la_teoria_del_caos_un_enfoque_despreocupado-wkc-3880.htm. Fecha de acceso: agosto de 2005.

[32] Navas, P. El Ciclo de Vida. Disponible en: http://www.getec.etsit.upm.es/docencia/gproyectos/planificacion/cvida.htm. Fecha de acceso: enero 2006.

[33] Orr, Ken. (2003). "CMM versus Agile Development: Religious wars and software development". Cutter Consortium, Executive Reports, 3(7).

[34] Pekka Abrahamsson; Outi Salo; Jussi Ronkainen y Juhani Warsta. (2002). Agile Software Development Methods. VTT Publications 478, Universidad de Oulu, Suecia.

[35] Pérez Sánchez, J. (2004). Metodologías Ágiles: La ventaja competitiva de estar preparado para tomar decisiones lo más tarde posible y cambiarlas en cualquier momento. Disponible en: http://www.willydev.net/descargas/prev/metodologiasagiles.pdf. Fecha de acceso: septiembre de 2005.

[36] Pressman, Roger (1998). Ingeniería del software – Un enfoque práctico. Cuarta edición. Ed. McGraw Hill.

[37] Pressman, Roger (2002). Ingeniería del software – Un enfoque práctico. Quinta edición. Ed. McGraw Hill.

[38] Reynoso, C. (2004). Métodos Heterodoxos en Desarrollo de Software. Versión 1.0 –Universidad De Buenos Aires. Revisión técnica de Nicolás Kicillof – Universidad de Buenos Aires.

[39] Rizzi, Francisco M. (2001). Sistema Experto Asistente de Requerimientos. Tesis de Magíster en Ingeniería de Software. ITBA.

[40] Saffirio, Mario. Costo Total de Propiedad (TCO) y Administración del Ciclo de Vida (LCM). Disponible en: http://msaffirio.wordpress.com/2006/04/08/costo-total-de-propiedad-tco-y-administracion-del-ciclo-de-vida-lcm/. Fecha de acceso: junio 06

[41] Salvetto, Pedro; Nogueira, Juan C.; Segovia, Javier (2004). Métodos Formales De Estimación De Tiempo Y Esfuerzo Adaptable A Los Cambios En Proyectos Software. IX Jornadas de Informática e Investigación Operativa. 8 al 12 de noviembre de 2004. Montevideo, Uruguay. Disponible en: www.ls.fi.upm.es/jiisic04/papers/66.pdf. Fecha de acceso: marzo de 2006.

[42] Varas, Marcela (2000) Gestión de Proyectos de Desarrollo de Software. Dpto. de Ingeniería Informática y Ciencias de la Computación. Facultad de Ingeniería. Universidad de Concepción.

[43] Voigt, Benjamin J. J. (2004). Dynamic System Development Method. Departamento de Tecnología de la Información. Universidad de Zurich.

[44] What is Scrum? Disponible en: http://www.controlchaos.com/. Fecha de acceso: diciembre de 2005.

[45] WIKIPEDIA. La Enciclopedia Libre. Disponible en: http://es.wikipedia.org/wiki/Portada. Fecha de acceso: febrero de 2006.

[46] Zavala, J. 2000. Ingeniería de Software. Disponible en: www.angelfire.com/scifi/jzavalar/apuntes/IngSoftware.html#Zavala2000. Fecha de acceso: febrero de 2006.