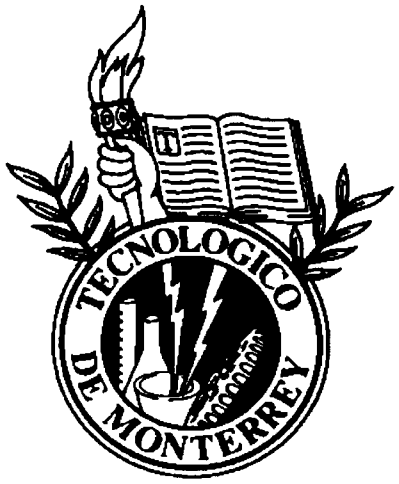
EL PROCESO PERSONAL DE DESARROLLO DE SOFTWARE



TESIS

MAESTRIA EN CIENCIAS
ESPECIALIDAD EN TECNOLOGIA INFORMATICA

INSTITUTO TECNOLOGICO Y DE ESTUDIOS SUPERIORES
DE MONTERREY

POR

LUIS ENRIQUE IBARRA TREVIÑO

DICIEMBRE DE 2000

EL PROCESO PERSONAL DE DESARROLLO DE SOFTWARE

POR

LUIS ENRIQUE IBARRA TREVIÑO

TESIS

Presentada a la División de Graduados en
Computación, Información y Comunicaciones
Este Trabajo es Requisito Parcial
Para Obtener el Título de
Maestro en Ciencias

INSTITUTO TECNOLOGICO Y DE ESTUDIOS SUPERIORES
DE MONTERREY

DICIEMBRE DE 2000

DEDICATORIA

A mi Dios. Fe y esperanza.

A mis padres. Protección y enseñanza.

A mis hermanos. Orgullo y lealtad.

A mis amigos. Amistad y respeto.

A mi novia. Comprensión y atención.

A todos ustedes, porque ustedes confían en mi. Porque ustedes significan AMOR.

AGRADECIMIENTO

Quiero expresar mi agradecimiento al Ing. Gustavo Cervantes Ornelas por ofrecerme todo su apoyo y su experiencia a lo largo de la realización de este trabajo. A mis sinodales: el Ing. Renan Silva y la Lic. Leticia Almaguer por sus ideas, comentarios y contribuciones al mismo. A los tres por su paciencia.

También quiero agradecer a la Srita. Amparo Muñoz por haberme brindado su ayuda. Muchas gracias.

A mis compañeros dentro y fuera de clases; por su amistad y su aliento.

L.E. Ibarra

Diciembre, 2000

RESUMEN

El Proceso Personal de Desarrollo de Software es un modelo de trabajo para los programadores. Este modelo les ayuda a observar el proceso de desarrollo de software en una manera sistémica, describe las interrelaciones básicas entre los diferentes procesos que conforman el desarrollo de software y, es un modelo de trabajo que provee una orientación general de las funciones y naturaleza del programador.

El PPDS busca incrementar la habilidad y la productividad de los individuos que desarrollan soluciones de software. Para lograrlo, el PPDS convierte al desarrollo de software en un proceso de solución del problema en lugar de enfocarse en un detallado proceso de codificación.

El PPDS está fundamentado sobre cinco disciplinas; estas son: modelos conceptuales, enfoque sistémico, desarrollo personal, visión compartida y, aprendizaje en equipo. Las tres primeras disciplinas tienen particular aplicación en el individuo, mientras que las últimas dos, tienen aplicación grupal.

El PPDS ayuda a los programadores a determinar la madurez de sus prácticas de trabajo, da lineamiento a un programa de desarrollo continuo, agrupa prioridades para la acción inmediata y, establece una cultura de excelencia en la ingeniería de software. Además, El PPDS puede ser utilizado por el programador en cualquier tipo de ambiente de desarrollo.

TABLA DE CONTENIDO

LISTA DE FIGURAS

Capítulo 1. Introducción

1.1 Planteamiento del problema

Actualmente, en las organizaciones en las que se desarrollan sistemas de software, se ha puesto un especial interés en el mejoramiento de sus procesos [Herb94]; adquieren tecnología de punta; proceden de acuerdo a estándares; y a pesar de que existen muchos modelos de ciclos de vida de desarrollo [DaBC88] y técnicas en la administración del proceso de desarrollo de software [IEEE94], se ha demostrado que los resultados obtenidos respecto a productividad en el desarrollo de software distan mucho de ser los deseados [Lehm94].

Se han desarrollados y refinados un gran número de propuestas, métodos, técnicas y herramientas en un continuo esfuerzo por mejorar un proceso de desarrollo de software más efectivo [Lehm94]. Lenguajes de alto nivel, programación estructurada, tipos de datos abstractos, métodos formales, paradigmas de programación no-procedimental, tecnologías orientadas a objetos, herramientas CASE, ambientes de desarrollo; estos conceptos y muchos otros más fueron, en su tiempo, la esperanza para resolver los problemas que han frustrado al continuo logro de desarrollar software confiable, funcionalmente satisfactorio y dentro del presupuesto y el calendario inicial.

Muchos autores han señalado que parte de la productividad de los programadores radica en cuestiones que tienen que ver con el uso de las herramientas, métodos y procesos de desarrollo de software [Jone86, Boeh87]. Esta productividad se ha visto rezagada debido a que las herramientas de software, que son usadas por los programadores como apoyo en el proceso de desarrollo, son cada vez más complejas. La cuestión es que, en algunos casos, el programador no tiene un margen de tiempo para ajustarse y aprender a utilizar las herramientas porque estas cambian constantemente; en otros casos, las herramientas de software o los procesos de desarrollo simplemente no se aplican en determinados proyectos. El problema pues, no radica en las herramientas que se utilizan sino en la forma como se emplean.

Los métodos, técnicas, y herramientas aplicados en el desarrollo de software han sido creados para facilitar tanto el desarrollo como el mantenimiento de software [Orli93, Marc94] y para incrementar la productividad de los programadores [DiSt98]. Estas herramientas han sufrido mejoras sustanciales a través del tiempo, pero en algunos casos la capacidad y la productividad de los programadores se han quedado rezagadas [NCBM89, Yell90, Lehm95].

Las proyecciones señalan que debido a este problema, en estos tiempos la demanda de aplicaciones de software sobrepasa a la habilidad de producirlos, y que el margen entre la oferta y la demanda se está incrementando [NgYe90, Diaz97].

En algunos casos, la comprensión de las herramientas y técnicas particulares puede estar incompleta, además estas técnicas usadas pueden no estar construidas sobre alguna teoría formal [Szaj94]. Debido a esta complejidad, La capacidad de adaptación a esta complejidad debe apoyarse en una adecuada metodología de aprendizaje.

Es importante prestar especial atención al factor humano ya que en el desarrollo de software se aplican las estructuras técnicas y administrativas incorporando métodos, herramientas y gente a las tareas del desarrollo de sistemas de información, y es precisamente en la gente donde se establecen las responsabilidades y las tareas específicas.

Existe entonces, una necesidad de establecer nuevos mecanismos y procedimientos para lograr el éxito en los quehaceres del desarrollo de software, formas de trabajo que sean capaces de traer consigo calidad y productividad al proceso utilizado y al producto desarrollado, fundamentados en la aplicación de un proceso ordenado, fácil y rápido en su implantación, que permita a los programadores incrementar su productividad en el proceso de desarrollo de software y les permita aprender de sus propias experiencias [Seng90].

Una solución para mantener la armonía entre la oferta y la demanda es incrementar la habilidad y la productividad de los individuos, para que puedan resolver directamente un amplio rango de problemas asociados al desarrollo de software.

1.2 Marco teórico

En otras disciplinas es bien sabido que un individuo puede desarrollar un mejoramiento continuo en productividad como consecuencia de una creciente acumulación de conocimiento y experiencia ganados mediante la repetida realización de una tarea [Arro62]. Las organizaciones han utilizado esta técnica para tomar decisiones concretas concernientes a la estimación de costos y presupuestos, producción y programación de actividades, etc. [ArBS92]. Y mientras que se ha realizado una considerable investigación sobre este tópico en el sector industrial y de manufactura [Yell79], existe muy poca en el negocio del desarrollo de software. Dando como resultado, la carencia de tal conocimiento en el campo de los procesos de software, carencia que puede estar frenando la creación de procesos más efectivos [Lehm95].

Añadir más gente a un proyecto retrasado fue una de las primeras formas de atacar la efectividad del trabajo de desarrollo, se creía que más gente disminuiría el retraso del proyecto. En uno de los libros más conocidos de esta época, "The Mythical Man-Month", el Dr. Frederick Brooks [Broo75] expone este caso; observó que añadir más gente a un proyecto retrasado de software solo hará que se atrase más.

En otro artículo igualmente conocido, "No Silver Bullet: Essence and Accidents of Software Engineering", el Dr. Brooks hace la siguiente observación:

> *"No existe el desarrollo sencillo, en ninguna tecnología o técnica administrativa, que por sí misma prometa un mejoramiento de igual orden de magnitud en productividad, en integridad, en simplicidad... (pero) un disciplinado y consistente esfuerzo para desarrollar, propagar, y explotar innovaciones debe producir un mejoramiento de igual orden de magnitud. No existe un camino auténtico, pero existe un camino".*

Brooks distingue entre las dificultades intrínsecas: complejidad, concordancia o proporción, modificabilidad e invisibilidad; que son inherentes al desarrollo de software y, las dificultades accidentales, que no son inherentes al desarrollo del mismo, pero que ayudan en la solución del problema: lenguajes orientados a objetos, inteligencia artificial, lenguajes gráficos, etc.

En su lista de posibles soluciones, Brooks destaca dos características cruciales de un proyecto de desarrollo exitoso: la utilización de estrategias y de métodos que ataquen las dificultades inherentes al proceso de desarrollo.

Han pasado ya 30 años desde que se empezaron a usar los términos "crisis del software" e "ingeniería de software". Desde entonces, se han hecho muchos esfuerzos para encontrar las razones detrás de los síntomas y resolver los problemas. Han sido desarrollados y usados nuevos lenguajes de programación, metodologías y herramientas. La idea de que las computadoras y el software pueden ayudar a la ingeniería de software condujo al desarrollo de las herramientas CASE. Hoy, tenemos computadoras más poderosas, lenguajes de programación más avanzados, complejas herramientas para apoyar en las diferentes tareas del desarrollo de software y mucho más conocimiento acerca del desarrollo de software que hace 30 años.

El mensaje que el Dr. Brooks nos deja es que ninguna técnica o tecnología sencilla será perfecta, y que, en lugar de buscar soluciones instantáneas, necesitamos entender que las mejoras vendrán en una manera evolutiva y no radical.

1.3 Hipótesis

Debido a la naturaleza del software [Broo87], el desarrollo del mismo conlleva una relativa complejidad [Rama84], pero la gente que es buena en su trabajo, generalmente hace buenos equipos, y los buenos equipos generalmente elaboran mejores productos.

Los atributos del personal y las actividades del recurso humano proveen con mucho la mayor fuente de oportunidad para incrementar la productividad en el desarrollo de software [Boeh84].

El punto central de cómo mejorar el arte del desarrollo de software se enfoca, como siempre, en la gente [Broo87].

A partir de estas proposiciones se define la siguiente hipótesis:

H_1 Una forma para incrementar la productividad y la calidad del trabajo del programador, es utilizar un proceso personal de desarrollo de software. De tal manera que cuando el programador logre compartir con el resto de su equipo un modelo conceptual acerca del proceso de desarrollo, llegue a generarse un ambiente donde los procedimientos de la ingeniería de software, el cumplimiento de los requerimientos del usuario y las especificaciones de los estándares fluyan juntos en forma natural.

Esto implica la creación de una nueva disciplina con la que los programadores apliquen los conocimientos para que repercutan en una mejora sustancial en la calidad de su trabajo.

Tomando conciencia de sus capacidades para mejorar, el programador tendrá también, un mejor conocimiento acerca de la forma en cómo ejecuta o desempeña el proceso de desarrollo de software. Con esto, podrá afinar sus cálculos respecto a la calidad y las fechas de terminación de los entregables que desarrollará. También será capaz de buscar y encontrar formas para proponer la utilización de herramientas mejoradas y la tecnología de soporte en el proceso de desarrollo de software.

1.4 Propuesta

En esta tesis se propone un modelo de trabajo para los programadores, que le ayuda a observar de una manera sistémica el proceso de desarrollo de software, describe las interrelaciones básicas entre los diferentes procesos que conforman el desarrollo de software y que provee una orientación general de sus funciones y de su naturaleza.

Es necesario que este modelo le permita estructurar una concepción global del proceso de desarrollo de software y le provea un marco conceptual que le permita establecer metas personales como profesional. Un modelo que ayude al programador a tomar conciencia del proceso que usa para hacer su trabajo y a evaluar la productividad del mismo. Herramientas que le ayuden a aprender a desarrollar su productividad personal por me-

dio del mejoramiento y análisis de su trabajo, y a ajustar su técnica de trabajo para lograr esas metas. De esta manera, el programador podrá incrementar su habilidad para evaluar su desempeño y para administrar la calidad del trabajo que realiza.

Este estudio propone un proceso personal de desarrollo de software, el cual está compuesto por una serie de actividades que los programadores podrán desempeñar para desarrollar una estructura definida del método que usan durante el proceso de programación.

El Proceso Personal de Desarrollo de Software (PPDS) es una solución que sustituye al modelo de trabajo tradicional del programador, el cual consiste en un enfoque orientado al código [Rout92].

El PPDS busca incrementar la productividad de los individuos que desarrollan soluciones de software e incrementar su habilidad como programadores. Para lograrlo, el PPDS convierte al desarrollo de software en un proceso de solución del problema en lugar de enfocarse en un detallado proceso de codificación.

1.5 Objetivo

En base a las opiniones y las observaciones de los expertos expresadas anteriormente, se propone diseñar y construir un ambiente de desarrollo basado en el individuo, con el objetivo de resolver el problema de la productividad del programador antes mencionado.

En esta tesis se documenta el PPDS: los fundamentos del modelo, una introducción al mismo, los conceptos básicos, sus etapas y su implementación.

OBJETIVO GENERAL.

El objetivo general de esta tesis es: dar al programador un enfoque progresivo para que aprenda a obtener experiencia de sus actividades par-

ticulares como programador y, para que aprenda a lograr la productividad y la calidad de los sistemas que produce como resultado de su trabajo.

El objetivo del PPDS, es incrementar la productividad del programador para que este, a su vez, tenga la facilidad de incrementar la calidad en el producto que desarrolla, mediante la integración de las prácticas de la ingeniería de software y el concepto del enfoque sistémico [Seng90] dentro de un ambiente de desarrollo de software.

OBJETIVO ESPECÍFICO.

Suministrar al programador las herramientas para que mediante la definición y la especificación de su trabajo, entienda mejor qué es lo que debe hacer para lograr desarrollar software con calidad. Este conocimiento le permitirá tener bases más sólidas para adoptar los métodos que funcionen mejor de acuerdo a las necesidades y naturaleza del proyecto. Además, será capaz de detectar la mejor manera de aplicarlos consistentemente y de cómo mejorarlos. Con todo esto, logrará incrementar su productividad en el proceso de desarrollo de software.

1.6 Delimitaciones

El PPDS está orientado a los programadores que se desempeñan individualmente o en equipos pequeños [KVHS97], considerando exclusivamente los aspectos intrínsecos de su actividad profesional.

Mientras que el aprendizaje de primer orden es el conjunto de conocimientos y experiencias que una persona adquiere mediante la ejecución repetida de una misma tarea, el aprendizaje de segundo orden es el que adquiere debido a la introducción de tecnología por parte de la empresa [AdCl91].

Esto nos prepara para entender que es importante notar que el aprendizaje de segundo orden no debe ser confundido con el término "aprendizaje organizacional" [Isaa93]. Este último término se refiere al aprendizaje acumulado por todos los individuos de una empresa debido al trabajo del equipo. En esta tesis, no se analizará el aprendizaje organiza-

cional sino el aprendizaje individual, el cual está compuesto por elementos de primero y segundo orden. Ese aprendizaje individual eventualmente se traducirá en aprendizaje organizacional, pero este tema queda fuera del alcance de esta tesis.

Este trabajo no propone cambios en los métodos, herramientas o técnicas de la ingeniería de software. Tampoco se enfoca ni sugiere el uso de alguno de los métodos, herramientas o técnicas en especial.

En esta tesis se examinará el paradigma del desarrollo de software desde una perspectiva orientada a la parte humana. No se analizarán las formas para desarrollar software de hoy en día. Más bien se enfocará al trabajo del individuo y a su aprendizaje.

1.7 Justificación

Mediante el uso de un conjunto de prácticas personales ordenadas, consistentemente aplicadas y de calidad, los programadores deberán ser elementos más efectivos en sus equipos de desarrollo y proyectos de software [Hump94b]. Los programadores tendrán definida una estructura del proceso y criterios mensurables para evaluar y aprender de sus propias experiencias [Seng90].

Sabiendo esto, los programadores podrán seleccionar los métodos y prácticas que mejor se ajusten a sus trabajos particulares y a sus habilidades.

El desarrollo de software tiene una fuerte necesidad de dirección en estos tiempos de caos y transición. Requiere de un rendimiento especial en el desarrollo individual; gente que pueda desenvolverse productivamente.

El programador, un recurso crítico en la tarea del desarrollo de software, necesita de una reestructuración en las bases de su disciplina de trabajo. Necesita de una nueva forma de ver la naturaleza de su trabajo que le permita incrementar su productividad.

El programador debe adoptar una actitud que le permita observar al proceso de desarrollo en una forma diferente, misma que le ayudará a entender las interrelaciones que existen entre los diferentes subprocesos que integran al proceso global de desarrollo; necesita desarrollar procesos que le ayuden a construir sistemas de calidad productiva y lucrativamente, por ejemplo, guiar el progreso del proceso de desarrollo cuantitativamente, entender la relación entre procesos y actividades, lograr una visión compartida acerca del producto de software y de los objetivos de cada proceso, funcionalidad y cualidades del producto, etc., además de aprender a evaluar la calidad del producto terminado. Estos modelos de proceso y producto deben ser ajustados basándose en los datos colectados dentro de la organización y deben poder evolucionar continuamente basados sobre las experiencias evolutivas de la organización [Seng90].

Actualmente, se han desarrollado algunos ambientes centrados en los procesos, pero la mayoría de ellos son prototipos de investigación. La adopción de esta nueva tecnología ha sido, de cualquier manera, muy lenta.

1.8 Organización del documento

El resto de esta tesis está estructurado de la siguiente manera:

Capítulo 2. Discute los conceptos relacionados con la ingeniería de software: ciclos de vida de desarrollo, modelación de procesos, la calidad del software, el ambiente de desarrollo, etc. Este capítulo provee al lector de suficientes bases conceptuales para relacionarse con el resto de la tesis.

Capítulo 3. Presenta el modelo conceptual del ambiente PPDS. Primero se discuten los conceptos del ambiente, seguidos por una descripción más detallada de las entidades y relaciones que comprenden al ambiente.

Capítulo 4. Describe la arquitectura del ambiente del PPDS. Primero se hace una revisión general. Después, se describe a detalle cada componente de la arquitectura.

Capítulo 5. Describe el ambiente de trabajo y las actitudes necesarias del programador para llevar a la práctica la disciplina del PPDS.

Capítulo 6. Se describen las cuatro etapas de la implantación del PPDS, así como las tareas a realizar y las consideraciones a tomar en cuenta por el programador para tener éxito en la implantación del PPDS

Capítulo 7. Evalúa el trabajo, se discuten los futuros planes del PPDS y se hacen algunas recomendaciones finales.

Capítulo 2. Fundamentos

2.1 El software y la Ingeniería de Software

Un sistema de software puede verse como parte de la solución de un problema; este sistema es una serie de instrucciones codificadas que serán procesadas por una computadora [IEEE83]. Además de programas, el software incluye toda la documentación necesaria para instalar, usar, desarrollar y mantener esos programas. El software bien desarrollado es un software que provee los servicios requeridos por sus usuarios y el cual es mantenible, confiable, eficiente, durable y está provisto con una interfaz de usuario apropiada.

Existen tres características principales en el desarrollo de software: su complejidad natural, la carencia de modelos o componentes bien definidos de la disciplina y, el hecho de que el software se desarrolla, no se produce [Basi92]. Esta combinación hace que el software sea una disciplina muy diferente a otras.

La ingeniería de software se puede definir como la aplicación disciplinada de principios ingenieriles, científicos y matemáticos, métodos y herramientas para el desarrollo de software con calidad y con bajos costos de producción [Hump89]. Además, incluye elementos técnicos y no técnicos.

El término ingeniería de software es usado frecuentemente en la literatura técnica refiriéndose a una colección específica de ideas, conceptos, métodos, y demás, perteneciente a la administración del desarrollo de software y a su utilización durante su ciclo de vida útil. Este término fue introducido en 1968 [NaRa68] y es usado como un término general para referirnos al desarrollo y mantenimiento de productos de software. Ya que el tamaño y la complejidad de los sistemas de software se han incrementado considerablemente en los últimos años, producirlos ha venido a ser más una disciplina ingenieril y menos un arte [Knut73].

2.2 Perspectiva histórica

El campo del desarrollo de software es muy joven comparado con otros campos ingenieriles. Al inicio de la era computacional, en los 50's, una aplicación de software era desarrollada por un programador o por un equipo muy pequeño [SoRo96], donde los usuarios finales eran a menudo los mismos programadores. Debido a que la capacidad de las computadoras se ha incrementado considerablemente, también se incrementó el interés y las necesidades de contar con productos de software más poderosos. El incremento en la complejidad de desarrollar sistemas de información, conlleva a desarrollar programas muy grandes y complejos, los cuales difícilmente pueden ser desarrollados sin las herramientas de desarrollo actuales. Esto contribuye a que un equipo de programadores trabaje sobre un mismo producto de software, equipo que introduce más complejidad al proceso de desarrollo; por ejemplo, la coordinación del trabajo, ya que ahora hay que ver cómo distribuir la carga entre el equipo de desarrollo. También introduce más complejidad a los requerimientos de comunicación y comprensión de las necesidades del o de los usuarios, que surgen debido a que los usuarios finales ya no son quienes escriben sus propios programas. El término "crisis del software" fue creado para describir esta situación.

La crisis del software fue citada por primera vez en 1968 [NaRa68]. Desde este tiempo, la industria de la computación ha progresado a gran velocidad, desde la revolución de las computadoras personales, hasta la reciente explosión del uso de redes e Internet. Aún durante el mismo período en que la industria del hardware ha estado ofreciendo mejoras sustanciales en cuanto a desempeño, el software parece una materia impenetrable e inmune a las innovaciones organizacionales, lo que nos hace pensar en la industria del software como una tarea difícil en la madura edad industrial.

Mientras que el término "ingeniería de software" tuvo sus orígenes en lo concerniente con la calidad y la productividad [NaRa68], la percepción general de la "crisis del software" es que esta ha sido una crisis de productividad; aunque cada vez se está viendo más como una crisis de calidad: un problema sin una solución tecnológica rápida [Broo87]. Lograr la calidad en el software, requiere de especial atención en las necesidades del cliente, pero también en las técnicas con las que el programador puede hacer frente a esas necesidades.

Las fallas cometidas en el pasado al desarrollar software y, sus costos actuales, apuntan a las causas fundamentales de la crisis del software. Hemos aprendido de las experiencias pasadas que el software con calidad es difícil de desarrollar. Además, hemos aprendido que la mayoría de los proyectos de software de cualquier magnitud están llenos de problemas y fallas [InSW93] y que esos proyectos usualmente tardan más en ser terminados y cuestan más de lo planeado [TSGI94]. Cuando se ven las razones de estos fallos, se encuentran importantes hechos que deben reconocerse para resolver la crisis del software.

Hoy, 30 años después, la "crisis del software" sigue presente. Aunque ha habido mejoras reales en nuestra búsqueda de la verdadera ingeniería de software, en las herramientas usadas para desarrollar sistemas de información y en la educación de los programadores; la demanda de software se ha incrementado a una tasa más rápida que la productividad de los programadores [NgYe90, Wadl98].

2.3 Objetivos de la modelación de procesos

La modelación de procesos es una técnica para organizar y documentar la lógica, las políticas y los procedimientos que son implementados en un sistema. La modelación de procesos está dividido en dos partes: el modelo lógico y el modelo físico. El modelo lógico de procesos, a menudo llamado modelo conceptual o modelo de negocios, muestra lo que hace el sistema. Es independiente de la implementación; es decir, es independiente de cualquier solución técnica.

Cuando el modelo de procesos está en evolución, este se transforma en un modelo físico de procesos que describe cómo el sistema hace lo que se requiere que haga. El modelo físico se verá afectado por las limitaciones y múltiples caminos de elección de la implementación técnica.

Puede haber diferentes razones que motiven a las organizaciones para modelar los procesos de software. La siguiente lista de metas y objetivos de la modelación de procesos esta incluida en [CuKO92]:

1. facilitar la comunicación humana. Se trata de representar el pro-

ceso en una forma entendible para la gente; esto permite la creación y comunicación de los compromisos generales acerca de los procesos de software. Formaliza el proceso de tal manera que la gente pueda trabajar unida y de una manera más efectiva, proveyendo suficiente información para permitir al individuo o al equipo desempeñar un determinado proceso y, forma una base para el entrenamiento sobre el mismo;

2. soportar a la mejora de procesos. Se trata de identificar todos los componentes necesarios para desarrollar software con alta productividad. Reusa los procesos de software efectivos y bien definidos, compara procesos alternativos de desarrollo, estima el impacto de los cambios potenciales al proceso de software sin ponerlo primero en las prácticas existentes, asiste en la selección e incorporación de la tecnología en los procesos, facilita el aprendizaje organizacional respecto a los procesos de software activos y, ayuda a la evolución de los procesos;

3. apoyar a la administración de procesos. Desarrolla un proceso de software para un proyecto específico con el fin de acomodar los atributos de un proyecto especial, tales como su producto, o ambiente organizacional. Razona acerca de los atributos de la creación o evolución de software, apoya al desarrollo de planes para el proyecto, monitorea, administra y coordina el proceso y, provee una base para la medición del proceso;

4. proveer una orientación automatizada en el desempeño del proceso. Define un ambiente efectivo de desarrollo de software, provee orientación, sugerencias y materia de referencia para facilitar el desempeño humano de los procesos propuestos. Almacena las representaciones de los procesos reusables;

5. proveer soporte a la ejecución automática de procesos. Automatiza partes del proceso de desarrollo, soporta al trabajo cooperativo sobre individuos y equipos automatizando el proceso, recolecta automáticamente los datos de medición que reflejan la experiencia actual con un proceso, y refuerza las reglas para asegurar la integridad del proceso.

Esta lista se puede ver como la contenedora de un principio para las metas de la modelación, desde la meta más básica (número 1) que conjunta los mínimos requerimientos sobre los formalismos usados, hasta la meta más ambiciosa de soportar automáticamente un proceso de software por computadora (número 5).

Actualmente, la mayoría de las organizaciones parecen estar interesadas principalmente en los primeros dos objetivos, por ejemplo: hacer posible la comunicación entre su gente y sustentar el mejoramiento de sus procesos. Sin embargo, una cosa interesante es que la mayoría de las investigaciones en el área se han enfocado en los dos últimos objetivos. Esto a conducido a una amplia diferencia entre las prácticas más modernas en la industria del software y las investigaciones más modernas en las universidades y otros centros de investigación respecto a la modelación de procesos [SoRo96].

2.4 El mejoramiento de los procesos

Como se mencionó en la sección anterior, muchas organizaciones están trabajando actualmente sobre la mejora de sus procesos. Este estímulo es, al menos hasta cierto grado, debido al CMM (*Capability Maturity Model*) [PaCC93], desarrollado por el Software Engineering Institute de la Universidad Carnegie-Mellon. El CMM clasifica los procesos de software en cinco niveles de madurez y determina un camino para ir del nivel más bajo (el menos maduro) al nivel más alto (el más maduro). El CMM es, por una parte, un modelo de referencia [PaCC93]; un modelo idealizado de alto nivel que sugiere qué es lo que debe contener un buen proceso de software y cómo debe trabajar la organización.

El CMM es descrito como:

Nivel 1. Inicial, llamado también *"ad hoc"* o caótico. En este nivel no existen los procesos formalizados, planes de proyecto, estimaciones de costos ni tiempos.

Nivel 2. Repetible. En este nivel, la organización establece control sobre cómo planear y controlar proyectos. La disciplina de procesos necesaria es puesta en practica para repetir los éxitos anteriores en proyectos con aplicaciones similares.

Nivel 3. Definido. Los procesos de software tanto para las actividades técnicas como administrativas, son documentadas, estandarizadas e integradas en un proceso de desarrollo estándar. En todos los proyectos se

usa una versión estandarizada, aprobada y ajustada para desarrollar y mantener sistemas de software.

Nivel 4. Administrado. En este nivel se hacen mediciones detalladas del proceso de desarrollo de software y la calidad del producto es obtenida. Los procesos de software y los productos son controlados cuantitativamente. Para alcanzar este nivel, la organización ha acumulado un historial acerca de qué tan bien trabajan sus procesos.

Nivel 5. Optimización. A este nivel, un nivel al que pocas organizaciones han llegado, la organización utiliza métodos de evaluación para mejorar sus procesos existentes. Se logra un proceso de mejora continua mediante la retroalimentación de los procesos, de las ideas innovadoras y de la tecnología.

El mejoramiento que el CMM propone, inicia evaluando a la organización y a sus procesos actuales para averiguar en qué nivel de madurez está y cuáles son sus puntos débiles. Después de eso la mejora procede de acuerdo al CMM.

Otros métodos de mejoramiento de procesos incluyen el QIP (*Quality Improvement Paradigm*) [BaRo88], el *Experience Factory* [Basi93], el método *ami* (Assess, Analyse, Metricate, Improve) [Kunt93] y el PSP (*Personal Software Process*) [Hump94a]. También se está desarrollando un nuevo estándar internacional, conocido como SPICE (*Software Process Improvement and Capability dEtermination*) [Dorl93].

Una de las grandes esperanzas de quienes proponen los ambientes de ingeniería de software centrado en procesos, es que estos sean útiles en la mejora de los procesos de software. Los modelos de madurez son, debido a su alto nivel de abstracción y generalidad, no compatibles como tal para incluirlos dentro del ambiente de ingeniería de software centrado en el proceso. Por otra parte, estos modelos pueden usarse como base para el desarrollo de modelos específicos de la organización, que puedan ser usados por sus ambientes en una forma más detallada.

2.5 La calidad del software

El desarrollo de software es un proceso multidisciplinario compuesto por administración, documentación, aseguramiento de la calidad y programación. Cada una de estas disciplinas trae técnicas específicas y un concepto propio de las necesidades del usuario. Mediante la creación de este proceso, el software resultante representará un producto completo que cumple con las expectativas de la organización y del cliente. Sin la aplicación de estas disciplinas en el proceso de desarrollo, no se logrará la calidad.

La calidad de un producto de software radica en gran medida en la calidad de la ejecución del proceso que se emplea para desarrollarlo. Es decir, el proceso es importante, pero la forma en la que se ejecuta el mismo, es fundamental. El desarrollo de software es un proceso que puede ser controlado, medido y mejorado progresivamente. Adicionalmente, la calidad del proceso de desarrollo puede ser afectada por la tecnología que se usa para soportarlo.

La calidad del software debe incluirse en el proceso mientras este es desarrollado. Se trata de prevenir errores, no de corregirlos. Así que, el programador debe estar capacitado y entrenado en los principios y herramientas usadas para lograr la calidad a través del proceso de desarrollo. El mejoramiento de la calidad requiere de un compromiso y de la participación de todos los niveles de la organización.

Los sistemas de software no son objetos estáticos. La característica dinámica más importante de un sistema de software es la confiabilidad [Musa84]. La razón de esto es que los costos de las fallas del sistema a menudo exceden los costos de su desarrollo en la primera etapa.

La meta principal de las investigaciones en la ingeniería de software ha sido lograr la productividad de los programadores y mejorar la calidad de los productos que elaboran [Radi88].

El software tiene dos características de calidad: la del proceso de desarrollo y la del producto. Las características de calidad del producto son

definidas por el usuario mientras que las características de calidad del proceso son definidas por los programadores [NuMe95].

Calidad de proceso de desarrollo

1. Mantenibilidad. El código del programa debe poder modificarse fácilmente para cambiar, suprimir o añadir nuevas funciones, mejorar el desempeño o corregir defectos.
2. Reusabilidad. El código del programa debe poderse usar para desarrollar nuevos productos.
3. Legibilidad. El código del programa de be poderse leer y comprender con facilidad

Calidad del producto

1. Usabilidad. El producto debe ser fácil de usar y de aprender.
2. Eficiencia. El producto debe utilizar los recursos del sistema en forma moderada.
3. Confiabilidad. El producto debe ejecutar las tareas sin causar errores aun en condiciones ambientales estresantes.
4. Integridad. El producto debe prevenir accesos impropios y no autorizados a su programa y a sus datos.
5. Adaptabilidad. El producto debe poderse utilizar, sin modificaciones, en ambientes diferentes a aquel en el que fue diseñado.

Logrando la calidad se reducen los costos de desarrollo. Este es el principio de la calidad del desarrollo de software. La mejor manera de lograr la productividad y la calidad es reduciendo el tiempo invertido en la repetición del trabajo, ya sea que esa repetición venga del cambio de requerimientos, del diseño, o de las pruebas.

2.6 Ciclos de vida de desarrollo de software

El ciclo de vida de desarrollo de un sistema de software es una serie de etapas y reglas sobre cómo el proceso de desarrollo de software debe proceder a través de él.

Un ciclo de vida de desarrollo de software no es la definición de los procesos de software que una organización debe seguir, tampoco puede

considerarse una metodología, ya que no provee reglas para el desarrollo. Más bien, se define como un modelo de referencia para un proceso de desarrollo de software [Come91]

Desde principios de los 80's, se ha identificado un gran número de modelos de ciclo de vida. Algunos han sido muy diferentes entre sí, mientras que otros han tenido otros nombres pero básicamente han sido semejantes en algunos aspectos. Los modelos más conocidos son: cascada, evolutivo, RAD (*Rapid Advanced Development*) /prototipos y, adaptable. Una revisión concisa de los primeros tres, que son ampliamente conocidos y usados, nos provee de un marco para discutir el último.

CICLO DE VIDA CASCADA.

El primer modelo ampliamente aceptado fue el llamado modelo cascada (ver Figura 2.1), presentado por Royce [Royc70], y el cual es el derivado de otros modelos de proceso ingenieril. El modelo cascada ve el desarrollo de software como un proceso secuencial, en el que al final de cada etapa se entregan algunos documentos o productos que serán necesarios para iniciar la siguiente fase.

El tradicional ciclo de vida cascada se ha usado durante muchos años y ha sido el modelo aplicado en un gran porcentaje de productos de software existentes.

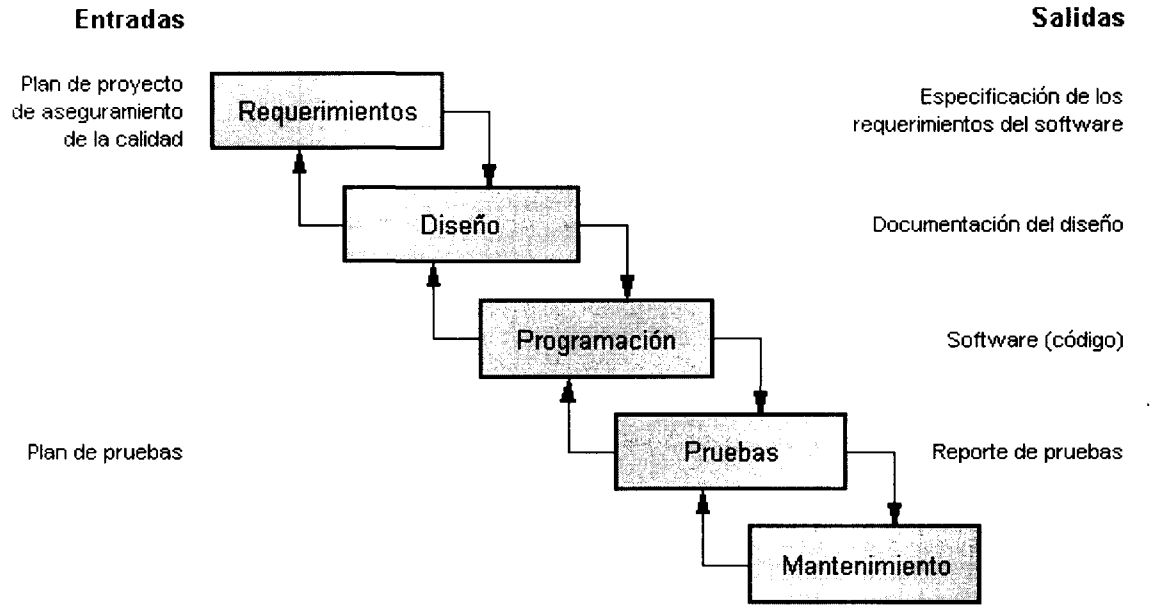


Figura 2.1. El ciclo de vida cascada

El modelo cascada funciona mejor en proyectos pequeños y de bajo riesgo donde las expectativas de cambio externo sean bajas. También en proyectos donde los cambios de negocios sean menos frecuentes y menos drásticos y, en general, cuando las aplicaciones sean simples (contabilidad básica, nómina, etc.).

Algunas de las fortalezas del ciclo de vida cascada son: recalca la terminación de una fase antes de iniciar la siguiente; enfatiza en la planeación anticipada, la participación del usuario y en el diseño; enfatiza en las pruebas como una parte integral del ciclo de vida. Mientras que algunas de sus debilidades son: depende de la captura y paralización de los requerimientos con anterioridad en el ciclo de vida; no es políticamente practicable en algunas organizaciones; enfatiza en el producto más que en el proceso.

CICLO DE VIDA EVOLUTIVO

Hay muchas variaciones del ciclo de vida evolutivo [BaTu75]. Tom Gilb usó el término evolutivo a mediados de los 80's para describir un modelo iterativo con ciclos de desarrollo pequeños y bien planeados. El ciclo

de vida espiral propuesto por Barry Boehm [Boeh87a] incorpora el concepto de administrar el desarrollo de software basado en riesgos.

El modelo espiral es muy flexible, y no tiene etapas fijas. Más bien, el modelo prescribe un ciclo de cuatro etapas que deben ser ejecutados las veces que sea necesario.

Los ciclos de vida evolutivos, como el espiral, son iterativos por naturaleza. Manejan la incertidumbre tomando pequeños pasos y probando los resultados. Estos modelos surgieron para ayudar en la administración de riesgos e incertidumbre y no para incrementar la velocidad del desarrollo. Tienden a involucrar planificación significativa y documentación, al menos en las formas propuestas por Gilb y Boehm. Los modelos evolutivos han tendido a incorporar "mini-cascadas" dentro de cada ciclo de desarrollo. El concepto fundamental es el de optimización en cada etapa del desarrollo.

El ciclo de vida RAD/Prototipos

Mientras que los ciclos de vida evolutivos surgieron para administrar los riesgos, el ciclo de vida RAD/prototipos surgió para acelerar la velocidad del desarrollo. Como los modelos RAD fueron usados en proyectos grandes, ellos adoptaron prácticas fuertemente evolutivas. Otro aspecto importante de los modelos RAD/prototipos fue su enfoque a la participación del cliente/usuario. RAD surgió a mediados de los 80's, como la mejor herramienta disponible para incrementar la interacción entre los programadores y los usuarios, ya que proveían algo más significativo (para el usuario al menos) que modelos de ingeniería de software sobre el papel.

Algunas de las fortalezas del ciclo de vida de prototipos son: Los requerimientos pueden ser conjuntados con anterioridad y más confiablemente; los requerimientos pueden ser comunicados con mayor claridad y completitud entre los programadores y clientes; las opciones de los requerimientos y el diseño pueden ser investigadas rápida y económicamente; la mayoría de los errores de diseño y requerimientos pueden ser descubiertos rápidamente. Mientras que algunas de sus debilidades son: requiere de herramientas RAD y de experiencia para usarlas, lo que se refleja en un costo de desarrollo para la organización.

El ciclo de vida adaptable

Los ciclos de vida evolutivo y RAD/prototipos proporcionan una herencia de los modelos adaptables. Este ciclo de vida, ilustrado en la figura 3, combina aspectos de estos dos tipos de ciclo de vida con dos características explícitas adicionales.

Primero, el modelo adaptable tiene una base conceptual explícita y adaptable. Esto está reflejado en los nombres de las fases iterativas (especulación, colaboración, y aprendizaje). Mientras que los modelos RAD/prototipos tienen una inclinación adaptable en muchas de sus variantes, las diferencias conceptuales sobresalientes a menudo no son bien expresadas.

Segundo, el ciclo de vida adaptable es también un modelo basado en componentes en lugar de un modelo basado en tareas. Esto es crucial para escalar a grandes proyectos mientras se mantenga un ambiente que conduzca hacia la innovación necesaria para abordar los problemas complejos.

Otros

Existen otros modelos de ciclo de vida. Entre los cuales están las transformaciones formales [Balz81] y reuso. Estas propuestas, aunque tienen aplicaciones importantes, parecen no tener más popularidad que el modelo cascada o espiral.

Los mayores méritos de los ciclos de vida del proceso de software son que nos han hecho cuidadosos del proceso de software y nos han provisto de una visión global de las mayores actividades del desarrollo de software y sus interrelaciones. Los modelos de ciclo de vida no incluyen información del proceso al nivel de detalle necesario para ejecutar exitosamente los proyectos de software.

2.7 El ambiente de desarrollo de software

El término "ambiente" se refiere a la colección de herramientas de hardware y software que un programador de sistemas usa para desarrollar sistemas de software. Debido a que las mejoras tecnológicas y las necesidades de los usuarios crecen día a día, la funcionalidad de los ambientes

de desarrollo tienden a cambiar. En los últimos 20 años, el conjunto de herramientas disponibles para los programadores se ha incrementado considerablemente. Podemos reseñar este cambio observando algunas distinciones en la terminología. "Ambiente de programación" y "ambiente de desarrollo de software" son a menudo usados como sinónimos, pero existen algunas diferencias entre ambos.

Los ambientes de ingeniería de software son construidos generalmente por las organizaciones para apoyar sus propios esfuerzos de desarrollo de productos. Esas organizaciones emprenden la construcción de un ambiente automatizado de desarrollo para hacer mejor sus productos, para hacer su trabajo más fácil y, para reducir costos de producción. Su meta es maximizar sus recursos creativos minimizando la cantidad de tiempo, dinero y esfuerzo cognitivo gastado en tareas tediosas, repetitivas o improductivas. Una organización bastante grande con un proceso estable y bien definido, una línea de productos atractivos y suficientes recursos, tendrá la capacidad de construir un ambiente que provea apoyo a cada aspecto de sus procesos de desarrollo. La mayoría de las organizaciones miran hacia las herramientas CASE para conseguir apoyo comercial y tecnológico.

La gente con talento es un elemento muy importante en cualquier organización de software pero, de la misma manera, los mejores profesionales necesitan un ambiente organizado en el cual puedan hacer un trabajo cooperativo.

Un ambiente ideal de desarrollo ayuda al programador en el proceso de definición del problema y la implantación del sistema. Debe ser un ambiente cooperativo y útil en el cual el programador pueda fácilmente:

- identificar problemas y resolverlos,
- concentrarse en la resolución del problema sin preocuparse por escribir código eficiente y,
- aprender de las actividades realizadas.

Un ambiente de desarrollo de software es un sistema que apoya cualquier tipo de actividad los programadores de sistemas lleven a cabo. Una secuencia de actividades que los programadores realizan es llamada "proceso de desarrollo de software". La primera aproximación de un ambiente de desarrollo de software puede ser un conjunto de herramientas usadas

en varias situaciones dentro del proceso de desarrollo de software, pero este tipo de ambiente debe construirse con mucha más preparación y cuidado. Uno de los tópicos más interesantes y urgentes en los recientes estudios de la ingeniería de software, es la investigación de cómo obtener una arquitectura mejor y más efectiva para este tipo de ambiente.

Los ambientes de desarrollo de software están diseñados para soportar todas las etapas del proceso de desarrollo de software desde los estudios iniciales de factibilidad hasta la operación y el mantenimiento.

2.8 Elementos críticos del proceso de desarrollo

Tres componentes clave manipulan todo el progreso en la productividad del desarrollo de software [KhPa94]: la gente involucrada, la organización de los procesos de desarrollo y la tecnología usada.

2.8.1 Proceso

Un proceso es el conjunto de acciones, tareas y procedimientos involucrados en la realización de una actividad. Un buen proceso de software debe ser predecible; las estimaciones de costos y calendarios deben cumplirse y el producto resultante debe ser robusto para ofrecer la funcionalidad requerida.

Dentro de la asignación predecible de recursos y la estimación de tiempo, muchas organizaciones intentan encontrar formas de producir mejor software a costos más bajos [NaSh82]. Recientemente se ha descubierto que la forma para progresar es estudiar y mejorar la manera en cómo se desarrolla el software, ya que aun con la mejor tecnología, esta sólo ayuda una vez que el ambiente organizacional esté preparado.

Definir los procesos del desarrollo de software proporciona una oportunidad para un esfuerzo colaborativo, debido a que genera una base de conocimientos que pertenece a la organización completa. Los procesos definidos pueden proporcionar una base común para que las acciones puedan ser usadas colectivamente, las consecuencias de acciones puedan ser observadas a través del sistema y, los procesos puedan ser mejorados. Las oportunidades de manejar los procesos pueden ser identificados por más gente, porque más gente se da cuenta o conoce el proceso.

2.8.2 Tecnología

La tecnología no puede ser completamente efectiva sin un marco organizacional. Hay evidencia de que destinarse sólo a la nueva tecnología en lugar de mejorar el proceso puede ocasionar errores costosos [DoD87].

Durante el proceso de desarrollo de software, se ejecutan varias actividades y tareas que van desde recopilar los requerimientos del usuario hasta entregar el producto terminado. Las máquinas a menudo pueden ayudar a la gente, algunas veces en forma automática, para evitar errores. Trabajan más rápido y concentran la atención al nivel conceptual.

Los programas que ayudan a desarrollar software se llaman "herramientas de desarrollo de software", son muy especializados y algunos son diseñados para soportar una actividad específica del ciclo de vida, mientras que otros, para una acción específica y limitada a una fase del ciclo de desarrollo.

2.8.3 El factor humano

El estudio sobre el factor humano relacionado con el desarrollo de sistemas de información es una nueva área de investigación, donde algunos de los primeros trabajos fueron desarrollador hace ya algunos años [Wein71, Shne80]. Este tema comenzó a ganar importancia debido a que el factor humano es tan importante en el desarrollo de software como los grandes sistemas de información que con él se desarrollan.

La función del factor humano en el desarrollo de software se reconoce como muy crítica [PeSV94]. Un aspecto importante en la involucración de la gente en el proceso de desarrollo, es averiguar cuánto puede aportar en el mejoramiento del mismo después de un período de tiempo.

El objetivo del programador de sistemas de información es aplicar su conocimiento en la forma más efectiva para producir software con calidad. Para lograr este fin, el programador necesita emplear herramientas y técnicas dentro del margen de un ambiente disciplinado. La calidad de cualquier sistema de software depende en gran medida de la calidad de la gente que lo desarrolla. Cualquier programador que no entienda el dominio de su aplicación no podrá desarrollar sistemas con calidad.

Capítulo 3. Introducción al PPDS

3.1 Introducción

El objetivo de este capítulo es presentar el marco dentro del cual está construido el PPDS.

Este capítulo inicia describiendo a La Quinta Disciplina y la Dinámica de Sistemas como las bases fundamentales del PPDS. Después, se definen las cinco disciplinas del PPDS y sus principios. Finalmente, se muestra y se describe su arquitectura.

3.2 Fundamentos del PPDS

3.2.1 La Quinta disciplina

Senge describe a la Quinta Disciplina refiriéndose a las "organizaciones aprendedoras" [Seng90], organizaciones donde la gente expande continuamente su capacidad para crear los resultados que buscan, donde los nuevos y crecientes patrones de pensamiento son cultivados y, donde la gente está aprendiendo continuamente.

Senge describe las cinco disciplinas que son vitales para una organización aprendedora. Organización aprendedora, según Senge, es un concepto que describe a una organización ideal construida sobre una visión, trabajo en equipo, apertura, flexibilidad, habilidad para actuar bajo condiciones de cambio, etc. Es una organización donde la gente no sólo se cultiva en su área profesional y sus habilidades, sino también donde se compromete a tomar responsabilidades con sus compañeros en favor de su futuro compartido, trabajando en la creación de la máxima sinergia y la máxima habilidad para hacerse cargo de una situación compleja.

Cada una de las cinco disciplinas del aprendizaje pueden ser abordadas en tres niveles:

- prácticas: qué hacer;
- principios: ideas rectoras y conceptos;
- esencia: el estado de ser de quienes tienen un gran dominio de la disciplina.

Las prácticas constituyen el aspecto más tangible de toda disciplina. También constituyen el núcleo primordial de los individuos o grupos cuando comienzan a practicar una disciplina.

El dominio de cualquier disciplina requiere un esfuerzo para comprender sus principios y para seguir sus prácticas. Existe el impulso de pensar que con la comprensión de ciertos principios se ha "aprendido" la disciplina, se trata sólo de la difundida trampa de confundir comprensión intelectual con aprendizaje. El aprendizaje siempre implica nueva comprensión y nueva conducta, "pensar" y "hacer". Por eso es posible distinguir entre principios y prácticas. Ambos elementos son esenciales.

La esencia de las disciplinas consiste en ese estado de ser que llegan a experimentar los individuos o grupos que poseen un alto nivel de dominio de las disciplinas. Aunque son difíciles de expresar con palabras, son vitales para aprender plenamente el significado y propósito de cada disciplina. Cada disciplina modifica a quien la practica en modos básicos. Por eso son denominadas como disciplinas personales, aunque algunas se practiquen en ambientes colaborativos.

Estas son las cinco disciplinas:

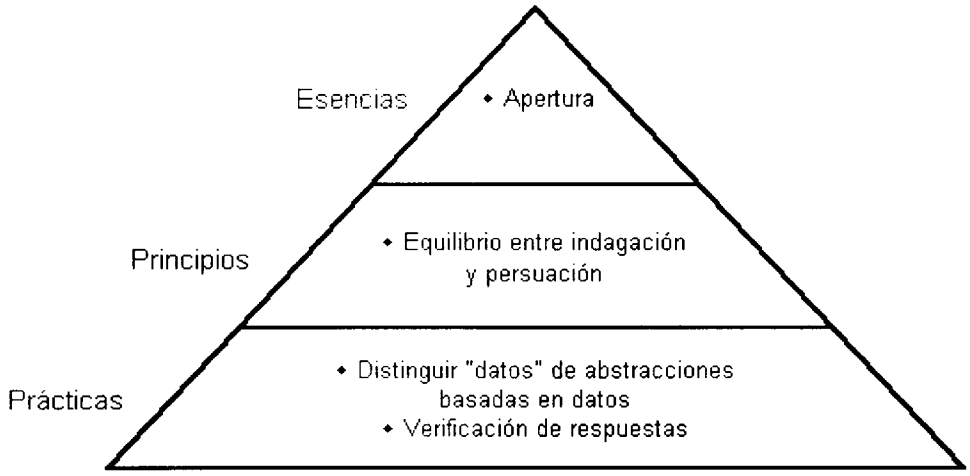


Figura 3.1. Modelos conceptuales

MODELOS CONCEPTUALES. Se refiere a reflexionar, clarificar y mejorar continuamente las imágenes internas y personales del ambiente que nos rodea y, ver cómo estas influyen en nuestras acciones y decisiones. En la Figura 3.1 se puede ver este modelo.

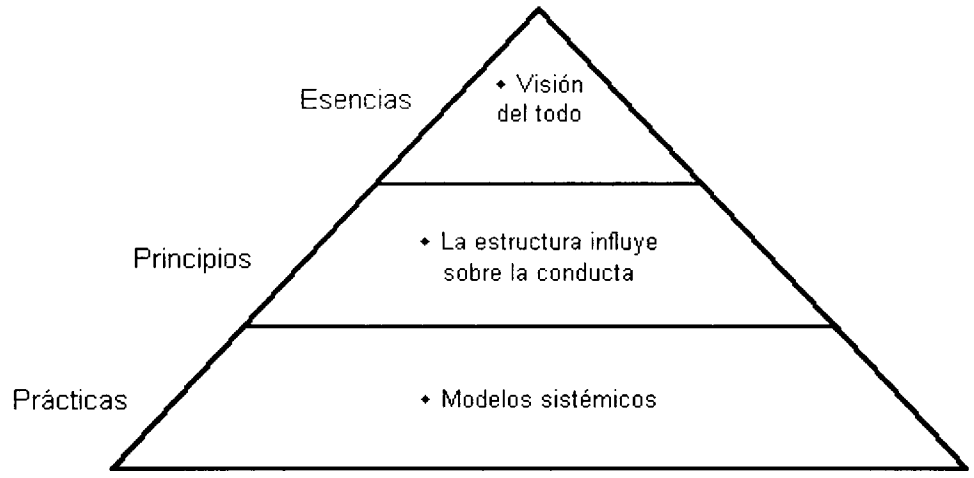


Figura 3.2. Enfoque sistémico

ENFOQUE SISTÉMICO. El enfoque sistémico es una forma de pensar y un lenguaje para describir y entender las fuerzas e interrelaciones que influyen en el comportamiento de los sistemas. Esta disciplina ayuda a ver cómo es posible cambiar los sistemas más efectivamente y a actuar más a tono con los procesos que en este se ejecutan

Figura 3.3. Desarrollo personal

DESARROLLO PERSONAL. Aprender a extender la capacidad personal para lograr los resultados más deseados y crear un ambiente organizacional, el cual anime a todos sus integrantes a desarrollarse a sí mismos hacia las metas y propósitos que eligieron.

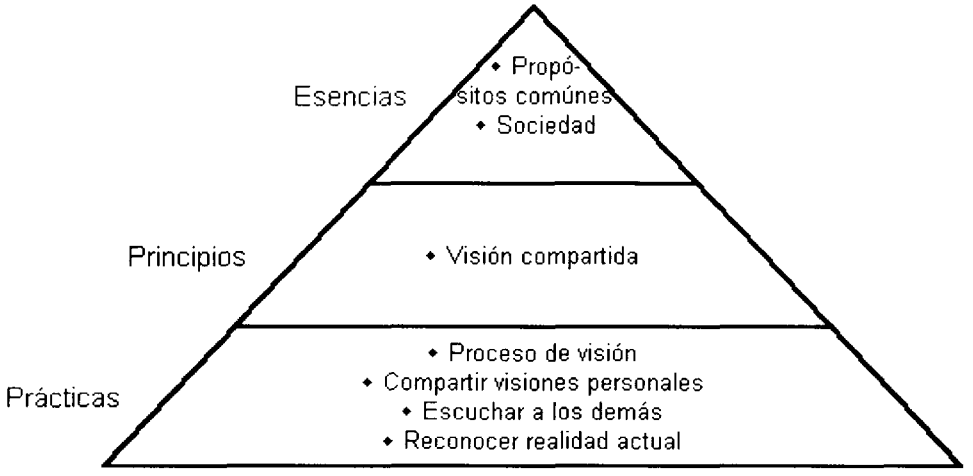


Figura 3.4. Visión compartida

VISION COMPARTIDA. Construir una cultura de compromisos en un equipo, mediante el desarrollo de intereses compartidos del futuro que se busca crear, además de los principios y prácticas con las cuales se espera alcanzarlos. Si los miembros de un equipo realmente comparten las imágenes del futuro y si ellos se entusiasman con lo que están creando juntos, entonces actuarán por motivación propia y contribuirán a la causa voluntariamente.

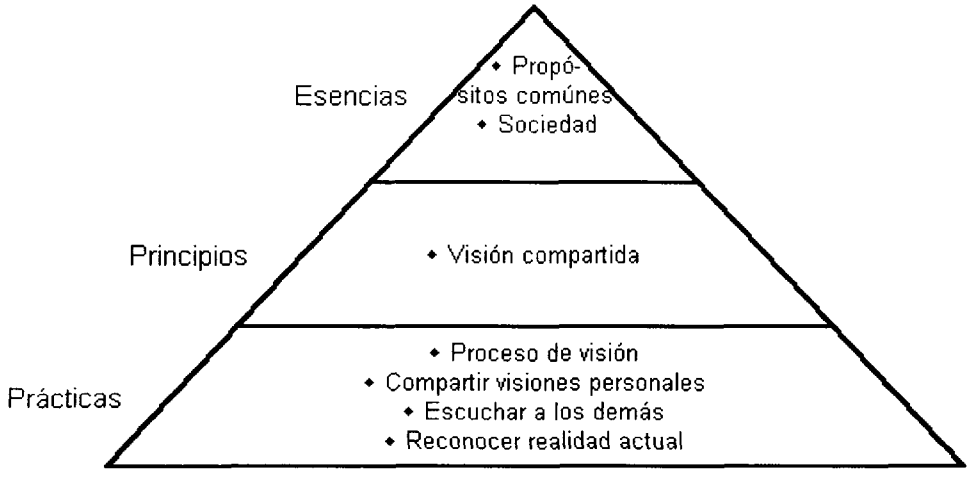


Figura 3.5. Aprendizaje en equipo

APRENDIZAJE EN EQUIPO. Se trata de transformar las habilidades colectivas conversacionales y de pensamiento, de tal forma que los equipos puedan desarrollar una inteligencia y una habilidad más grande que la suma e los talentos individuales de sus integrantes.

3.2.2 La Dinámica de Sistemas

La dinámica de sistemas es la aplicación de los principios y técnicas de los sistemas de control de retroalimentación para modelar, analizar y entender el comportamiento dinámico y complejo de los sistemas. Es decir, el patrón de comportamiento que generan a través del tiempo. Sus orígenes se remontan al trabajo de Jay W. Forrester dentro de la dinámica de la industria [Forr61]. La dinámica de sistemas ve a las organizaciones, las economías, las sociedades y en general, a todos los sistemas humanos, como sistemas que contienen retroalimentación. Los procesos de retroalimentación son, de esta forma, vistos como la clave para estructurar y aclarar relaciones dentro de tales sistemas y para entender su comportamiento dinámico.

Desde que se publicó "Industrial Dynamics" de Forrester [Forr61], el alcance de la aplicación de este método se ha hecho extremadamente amplio, tales aplicaciones incluyen: las fluctuaciones económicas, políticas antidrogas, dinámica y administración de ecosistemas e ingeniería de software.

El proceso de desarrollo de software, como la mayoría de los sistemas organizacionales se caracteriza por un complejo conglomerado de ciclos de retroalimentación interconectados [AbMa91]. El comportamiento de tales sistemas a menudo confunde a la intuición común y al análisis, aunque las implicaciones dinámicas de ciclos aislados pueden ser razonablemente obvias.

El desarrollo de software es un proceso dinámico y complejo ya que hay muchos factores que interactúan a través de su ciclo de vida, impactando así en el costo final, el tiempo de entrega y la calidad [Boeh84]. Se han utilizado nuevas formas de pensamiento y herramientas afines para ayudar en la administración del desarrollo de software en ambientes competitivos y de recursos limitados. La dinámica de sistemas proporciona el

conocimiento necesario para mejorar el entendimiento de las complejidades de los sistemas dinámicos tales como el proceso de desarrollo de software.

3.3 Definición de las disciplinas

3.3.1 Ajustar los modelos conceptuales

En general, los modelos conceptuales son patrones de pensamiento que los humanos construyen para explicarse el porqué de las cosas [GeSt83]. Un modelo conceptual es la forma en la que cada individuo interpreta un evento basándose en su experiencia y de acuerdo al contexto del problema a ser resuelto. Estos modelos se generan de las interpretaciones de eventos pasados; tienden a ser traducidas inicialmente a representaciones visuales, aunque generalmente son multisensoriales, para que puedan ser comunicadas con mayor facilidad. Estos patrones influyen en cómo entender las cosas que pasan en el ambiente y cómo reaccionar para responder a ellas. Las diferencias entre modelos conceptuales explican por qué dos personas pueden observar el mismo acontecimiento y describirlo de maneras distintas: prestan atención a distintos detalles.

El estudio de los modelos conceptuales ha incluido análisis detallados en dominios tales como el movimiento, la navegación marítima, la electricidad y el desarrollo de representaciones computacionales [GeSt83]. También se ha relacionado a formalismos lógicos usados en mecánica [DeBr81], electrónica [KiBo84], aprendizaje [WhFr85] y otras disciplinas formales para poder describir una amplia variedad de conceptos semánticos.

Norman [Norm93] señala que los modelos conceptuales son un tipo de ayuda externa para la cognición humana. Sin esa ayuda externa, se limita la memoria, los pensamientos y el razonamiento. Pero la inteligencia humana es muy flexible y adaptable y es muy buena para inventar procesos y objetos que superan a sus propios límites. ¿Cómo se puede incrementar la capacidad de la memoria, el pensamiento y el razonamiento? creando ayudas externas: cosas que permitan desarrollar la creatividad. Algunas de

estas cosas pueden obtenerse de un comportamiento social cooperativo, otros, surgen del aprovechamiento de la información presente en el ambiente actual. Norman añade que el poder de la cognición proviene de la habilidad de representar, exentos de detalles irrelevantes, las percepciones, las experiencias y los pensamientos, por medios diferentes a aquellos en los cuales han ocurrido.

3.3.2 Pensar con un enfoque sistémico

El enfoque sistémico está basado en la dinámica de sistemas y es altamente conceptual. El enfoque sistémico se puede ver como un lenguaje para expresar y observar un fenómeno complejo, a través de aprender a expresarse, compartir y, manejar interconexiones complejas. Se refiere a una forma de pensamiento y a un lenguaje para describir y entender las fuerzas e interrelaciones que condicionan el comportamiento de los sistemas. Esta disciplina ayuda a hacer cambios en los sistemas de una manera más efectiva. También ayuda a saber cómo actuar de acuerdo con los grandes procesos del mundo natural y a conocer la estructura organizacional, la estructura procedimental del desarrollo de software y los estándares de trabajo.

Esta forma de análisis requiere que el individuo utilice e incremente su desarrollo personal en cuanto a sus capacidades de conceptualización. El enfoque sistémico proporciona un marco de trabajo para efectuar cambios simultáneamente en las ideas, innovaciones en infraestructura y en la aplicación de nuevas tecnologías.

3.3.3 Extender el desarrollo personal

Senge define al desarrollo personal como: aprender a desarrollar la capacidad individual para producir intencionalmente los resultados más deseados. Se refiere también a la creación de un ambiente organizacional que fomente en todos sus integrantes el deseo de desarrollarse hacia sus metas y propósitos personales.

El desarrollo personal consiste en una serie de actividades que refuerzan el continuo mejoramiento de los procesos de trabajo del individuo. Este programa implica desarrollar planes y metas para actividades de trabajo personal, estableciendo y usando procesos personales definidos, midiendo y analizando la efectividad de tales procesos, e implementando mejoras a los mismos.

El individuo define sus procesos de trabajo y usa esos procesos definidos para planear su trabajo; mide sus procesos de trabajo y establece metas mensurables para mejorara su desempeño. También analiza sus procesos de trabajo para identificar oportunidades de mejoras e identifica el conocimiento y habilidades necesarias para lograr esas mejoras. Mejora sus procesos de trabajo y elimina las causas de defecto en su trabajo.

El programador debe practicar su visión personal durante todo del proceso de desarrollo de software. Un programador exitoso necesita entender la definición del proceso y el producto, evaluar éxitos y fracasos, realimentar información para el control del proyecto, aprender de las experiencias [Seng90], documentar y revisar experiencias exitosas para ser competitivos en su área de trabajo. Esto requiere de aprendizaje, retroalimentación y mejoramiento continuo [Seng90].

Nadie puede aumentar el desarrollo personal de otra persona. Sólo es posible crear condiciones que alienten y respalden a las personas que deseen aumentarlo.

El desarrollo personal constituye, en potencia, el motivador más poderoso, pues a diferencia de otros motivos, nunca quedará realmente saciado.

3.3.4 Construir una visión compartida

Senge define la visión compartida como: el hecho de compartir un interés común, es construir un sentido de compromiso en un grupo con intereses y prácticas comunes, para satisfacer esos intereses. Idealmente, la visión compartida es representada de tal manera que se extiende y se sostiene dentro de una organización y todo lo que la rodea. Como Senge expli-

ca, para poder extender una visión, esta debe parecer una invitación a asociarse a la visión compartida; después, se desarrolla un compromiso hacia la visión compartida, y un proceso que observe su acatamiento, manteniendo la visión sobre el camino a lo largo del recorrido y hasta su realización.

Para que la visión compartida sea sostenible, esta debe estar sustentada en visiones personales de cada individuo. Senge explica que el punto de partida para crear una visión compartida es animar a esa visión personal. La creatividad es una herramienta para afinar la visión personal.

3.3.5 Adquirir el aprendizaje en equipo

La disciplina del aprendizaje en equipo inicia con el diálogo, la capacidad de los integrantes del equipo para interrumpir sus suposiciones y entrar a un "pensar juntos" auténtico y acoplarse en las acciones coordinadas.

Aunque un programador puede trabajar como parte de un equipo, el equipo es necesario porque el sistema de software requerido llega a ser tan grande que no puede ser desarrollado por una sola persona en una cantidad de tiempo razonable. Dentro del equipo, el trabajo es dividido y cada individuo trabaja en la parte del sistema que le corresponde, desarrollando una parte del sistema.

El aprendizaje en equipo se refiere a la capacidad de transformar las habilidades de pensamiento colectivo y conversacional, de tal manera que los grupos de personas puedan desarrollar confiablemente una inteligencia y una habilidad más grande que la suma de sus talentos individuales.

En el ambiente de programación, los proyectos y los reportes en equipo son una cuestión crítica. Estos demandan habilidades tales como: encontrar y evaluar información; generar información al tiempo que se conduce una investigación; habilidades para hablar y explicar en público; habilidades técnicas de escritura; habilidad para funcionar eficientemente en un ambiente de trabajo en equipo y, ejercitar las habilidades de liderazgo y administración dentro del equipo.

3.4 Los Principios del PPDS

El PPDS está fundamentado sobre cinco disciplinas; estas son:

- modelos conceptuales,
- enfoque sistémico,
- desarrollo personal,
- visión compartida y,
- aprendizaje en equipo.

Las tres primeras disciplinas tienen particular aplicación en el individuo, mientras que las últimas dos, aplicación grupal.

El aprendizaje individual (desarrollo personal) prepara al individuo para formar parte de un equipo; esto que el programador aprende, debe a su vez hacerlo más receptivo a otros conocimientos, experiencias, preguntas y formas de pensar de sus compañeros. Hace que aprenda a desarrollar su capacidad personal para crear los resultados más deseados, creando un ambiente organizacional, que anima a todos sus integrantes a desarrollarse a sí mismos hacia las metas y propósitos que eligieron. Se trata de reflexionar, aclarar, y mejorar sus conceptos internos acerca del ambiente y ver cómo influyen en sus acciones y decisiones (modelos conceptuales). Un punto de vista con el cual sea capaz de entender el ciclo de vida del desarrollo de software y las interrelaciones del sistema es lo que se requiere para trabajar hacia mejores relaciones tanto con los sistemas como con la gente (enfoque sistémico). El enfoque sistémico es una forma de pensar, un lenguaje para describir y entender las fuerzas e interrelaciones que influyen en el comportamiento del sistema. Esta disciplina ayuda a ver cómo cambiar el sistema de trabajo para realizarlo más eficientemente y, a actuar más a tono con los grandes procesos del ciclo de vida de desarrollo de software.

Sin un propósito dirigido ni valores compartidos (visión compartida), el esfuerzo del equipo de desarrollo tendrá problemas de comunicación provocados por la confusión; es necesario construir un sentido de compromiso en un equipo, desarrollando los objetivos compartidos del futuro que se busca crear y, los principios y prácticas por los cuales esperamos obtenerlos. Para que todos aprendan juntos (aprendizaje en equipo), es ne-

cesario un proceso receptivo de atención hacia los otros, transformar las habilidades convencionales y colectivas del pensamiento, de tal manera que los grupos de personas puedan desarrollar inteligencia y habilidad mayor que la suma de los talentos de los individuos.

3.5 La estructura del PPDS

Para que un programador logre desarrollarse personal y profesionalmente en un ambiente de trabajo, necesita conjuntar cinco disciplinas críticas en su método de trabajo. La Figura 3.6 muestra esas disciplinas, cada una provee un conjunto de prácticas que si se ejecutan satisfactoriamente ayudarán al programador a lograr su crecimiento personal y profesional.

Los Modelos Conceptuales, el Enfoque Sistémico y el Desarrollo Personal, constituyen el entorno individual de trabajo, mientras que el Aprendizaje en Equipo y la Visión Compartida constituye al entorno colectivo.

El crecimiento individual de un programador inicia con un cambio de actitud, es decir, primero que nada necesita estar dispuesto al cambio. El aprendizaje no puede ser duradero a menos que esté respaldado por el interés y la curiosidad personal. En el entorno individual, el programador cambia su percepción del proceso de desarrollo de software mediante la conciliación de sus modelos conceptuales y su visión del todo.

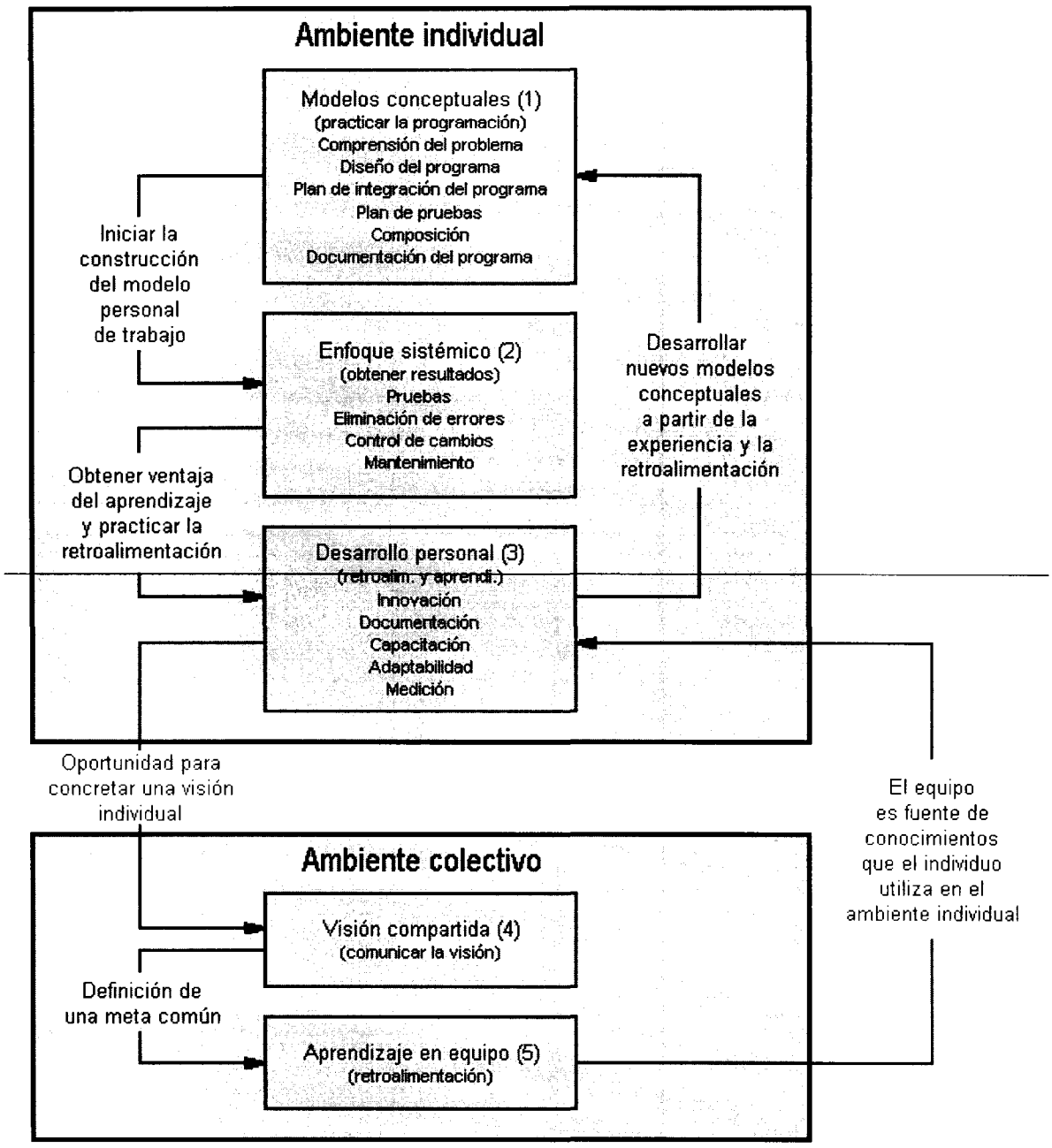


Figura 3.6. Las etapas del PPDS

En la transición de los modelos conceptuales al enfoque sistémico, el programador comenzará a pensar en un modo de construir, definir y organizar sus actividades a medida que vaya descubriendo sus modelos con-

ceptuales dominantes; de tal manera que respalde a su nuevo modelo laboral. Mientras se tengan interpretaciones más realistas de los eventos, se puede también tener una mejor explicación del sistema en su totalidad. El sistema de trabajo puede comenzar a construirse a partir de una interpretación.

En la transición del enfoque sistémico al desarrollo personal, el programador se dará cuenta de que forma parte de un sistema de trabajo, y que las transformaciones más radicales pueden ser el resultado de los cambios en su orientación y autoimagen. Esto requiere trabajo en el desarrollo personal: desarrollar una visión personal y aprender a ver el entorno desde una perspectiva creativa e interdependiente.

En la transición del desarrollo personal a los modelos conceptuales, el programador observará que sus teorías y modelos del entorno también forman parte de la realidad actual. Podrá desarrollar y probar nuevos modelos conceptuales, tales como la idea de que la automotivación es más eficiente que el incentivo o premio por su desempeño, lo cual le permitirá guiar mejor su aprendizaje y desarrollo.

En la transición del ambiente individual al ambiente colectivo, el programador se sentirá con la obligación de entender con qué finalidad ha elaborado su modelo de trabajo. Este es un buen momento para trabajar en visiones compartidas con la finalidad de concretar su visión individual, ya que inevitablemente necesitará que sus compañeros de equipo formen parte de esa visión.

El aprendizaje en equipo puede ser un paso natural después del proceso de visión compartida. Las aspiraciones colectivas brindan a los integrantes del equipo una motivación para el aprendizaje colectivo. Un nuevo nivel de cooperación y comunicación debe ser definido e implementado por el equipo de desarrollo para trabajar juntos hacia una meta claramente definida.

3.6 Conclusiones

El enfoque sistémico es un lenguaje para describir y entender las fuerzas e interrelaciones que moldean el comportamiento de los sistemas. De acuerdo a Senge, el enfoque sistémico y los modelos conceptuales son las herramientas más poderosas para conceptualizar la problemática de las organizacionales. El desarrollo personal y la visión compartida ayudan sobre las diferencias entre el aprendizaje individual y el aprendizaje en equipo, mediante el incrementar la calidad de la conversación, el diálogo y la comunicación del equipo. Si bien, esos son los productos de muchos pensamientos y conversaciones, no son necesariamente intocables. Más bien, las ideas conductoras constantemente asumen nueva apariencias en expresión, evolución, conversación, decisiones y acción.

Capítulo 4. Las cinco disciplinas del PPDS

4.1 Introducción

En este capítulo se trata en detalle cada una de las disciplinas del PPDS. También se discuten las actividades contenidas en cada disciplina y que deben ser desempeñadas por los programadores.

4.2 Las tareas dentro del PPDS

4.2.1 Los modelos conceptuales y el PPDS

Es una tarea difícil generar modelos conceptuales efectivos en los programadores de sistemas debido a la subjetividad en la interpretación del diseño y de los procesos, la forma de percibir el ambiente de desarrollo, las herramientas y las técnicas de desarrollo, etc. Por ejemplo, es una cuestión crítica cuando se trata de reuso de software, porque la gente tiende a pasar por alto o a mal utilizar componentes que son difíciles de entender. Por supuesto, la implementación de los componentes tiene modelos conceptuales asociados a ellos.

La retroalimentación da acceso al descubrimiento de modelos conceptuales. La habilidad de estar dispuesto a participar en un proceso de retroalimentación, permite regular el comportamiento de esos modelos para evaluar las opiniones que ocasionan esas interpretaciones. Esta habilidad reflexiva permite evaluar si una opinión dada aún tiene valor práctico en un ambiente en constante cambio.

4.2.1.1 Comprensión del problema

El proceso de idear y escribir un programa es, básicamente, una situación de solución del problema. El problema debe ser comprendido primero; una estrategia de solución general debe ser aplicada y, finalmente, esta estrategia debe ser traducida en acciones específicas.

Al principio, el programador inicia con una comprensión imprecisa y poco detallado de cuáles funciones deben estar contenidas en el sistema de información, cuáles salidas corresponden a cada posible entrada, cómo una decisión afecta a otra, etc.

Muchas decisiones acerca del comportamiento del sistema dependerán de otras decisiones aún no tomadas; así, las perspectivas estarán cambiando conforme el problema sea entendido de una mejor manera. Sin embargo, de esta manera se obtendrá una comprensión rica y precisa del comportamiento esperado, que es necesario para crear diseños efectivos y desarrollar código correcto.

Para poder iniciar el diseño de una aplicación, es necesario conocer lo siguiente:

- entender el propósito y los objetivos del programa (qué es lo que se quiere que el programa haga),
- definir las entradas al sistema y,
- definir las salidas del sistema.

Con estos tres pasos, se estará en condiciones para iniciar el diseño del programa. Aun sin un método real de diseño a seguir, debe estar claro que el programa tomará los datos de entrada y los transformará para poder crear las salidas requeridas.

4.2.1.2 Diseño y revisión del programa

El programador alcanza una pequeña parte de la fase del diseño del sistema. Su trabajo es definir el sistema al más bajo nivel de detalle hasta que alcance un nivel que pueda programar. A esto se le llama diseño del programa.

Uno de los obstáculos para desarrollar un buen sistema es un diseño que cambia constantemente. Cada cambio significa reescribir el código que ya ha sido escrito, cambiando los planes en pleno proyecto y corrompiendo la consistencia interna del sistema.

El problema es que a menudo nadie sabe que es lo que el programa hace hasta que haya una versión preliminar que pueda correr. Una vez que el diseñador elabore la estructura básica del sistema, cualquier cambio que no sea crítico debe esperar hasta la siguiente versión. Esto es una postura difícil de ejercer, pero que los programadores disciplinados pueden adoptar para proteger la integridad del código.

4.2.1.3 Revisión del diseño del código

Revisar el diseño del código antes de implementarlo. Este proceso es sencillo: es probable que sólo el programador, su supervisor y tal vez otro programador necesiten participar. El propósito de la revisión del diseño del código es asegurarse de que se ha hecho el mejor diseño, asegurarse de que todas las funciones estén implementadas y todas las eventualidades estén consideradas. El principal objetivo de la revisión es el de asegurarse que se ha comprendido el diseño original del sistema mediante la localización de los defectos de la traducción del diseño al diseño del código en una especificación (por ejemplo: omisiones, adiciones no solicitadas y contradicciones). También se trata de considerar funcionalidad alternativa, objetivos de desempeño o representación.

De acuerdo a Fagan [Faga86], existe evidencia de que la experiencia adquirida de la revisión de código provoca que los programadores reduzcan la cantidad de errores en las fases siguientes. Los reportes de la industria [Russ91] sugieren que las revisiones de código pueden ser mucho más eficientes que las pruebas. En un experimento que involucró a programadores profesionales, se observó que en la lectura del código se detectaron más fallas que en las pruebas funcionales o estructurales [BaSe94].

4.2.1.4 Planeación de la integración del sistema

El sentido común nos dice que no se puede programar todo y después ponerlo todo junto ya que se requerirá de un acoplamiento secuencial. Es necesario planear el orden en el cuál se armará el sistema, hay que pla-

near este orden de integración ahora, ya que hay que estar escribiendo los programas en el orden en el cual serán integrados.

4.2.1.5 Planeación de las pruebas de los programas

El programador debe preparar el plan de pruebas para el módulo antes de que este sea implementado. Planear las pruebas después de la implementación puede llegar a ser prejuicioso, ya que se tiende a probar sólo las partes "duras" del código.

Es importante hacer pruebas de funcionalidad con el usuario al terminar la etapa de diseño, generando un listado específico y evolutivo de actividades en las fases de pruebas.

4.2.1.6 Composición

Desde el inicio de la computación, han sido desarrolladas muchas formas de hacer la programación más fácil y más eficiente. La programación ha sido siempre una parte necesaria en el desarrollo de software, y es probable que siga siendo así por lo menos hasta en un futuro cercano.

Todos los programas computacionales y el código fuente usado para crearlos, están sujetos a muchas limitaciones, y aun cuando la programación se ha venido transformando en una disciplina de precisión, está limitada por muchos casos.

A continuación se listan algunos de estos casos. No se trata de discutir cómo programar, sino de sumarizar qué es lo que se constituye como un programa bien definido. Este debe responder a las siguientes preguntas:

- la configuración del hardware sobre el cual el programa va a ejecutarse. En algunos casos, el software está ligado directamente al hardware; esto es una limitación debidos a que hay que ver de qué manera la aplicación depende del hardware,
- la elección del lenguaje de computación con el cual el programa será escrito,

- la elección del sistema operativo sobre el cual el programa será ejecutado,
- otros programas computacionales, tales como el administrador de base de datos, etc., con los cuales el programa debe interactuar,
- el estilo personal e individual del programador,
- cualquier limitación de tiempo, en función de cuándo el programa debe correr y qué tan rápido y,
- cualquier limitación de tamaño en el volumen de los datos o en el tamaño de las piezas individuales de los datos (registros).

Un programa terminado está sujeto a todos estos tipos de limitaciones. El resultado es un producto de todas estas limitaciones siendo aplicadas, en cierto grado, dependiendo de la naturaleza del programa computacional, las circunstancias bajo las cuales ha sido escrito y bajo las cuales debe operar.

Cuando se escribe un programa, las opciones para elegir un lenguaje de programación serán limitadas a aquel que está disponible en la computadora particular y a la configuración de software que se esté usando.

Ya que las habilidades de programación son independientes del lenguaje, es relativamente fácil para los programadores quienes están familiarizados con un lenguaje de programación aprender un lenguaje nuevo del mismo tipo. Todo lo que tienen que aprender es una nueva sintaxis, ya que el concepto en sí ya ha sido comprendido.

El desarrollo de la solución (el programa) involucra la creación de un modelo semántico interno del modelo y un modelo correspondiente de la solución. Una vez que este modelo esté construido, puede ser representado en alguna notación sintáctica apropiada. Un programador experimentado quien entienda un determinado número de lenguajes de programación tendrá aproximadamente el mismo grado de dificultad en representar esa solución, independientemente de cuál lenguaje esté siendo usado actualmente.

Así, el proceso de solución del problema es independiente del lenguaje y las habilidades técnicas con lenguajes particulares y no son garantía de habilidades de programación.

4.2.1.7 Documentación

A la mayoría de los programadores no les gusta documentar. Esta actitud realmente ayuda a generar desastres porque una buena anotación acerca del diseño básico del sistema es valorada cuando llega la hora de la actualización, las modificaciones y la corrección de errores. La fiabilidad se conseguirá si los programadores se aseguran de escribir la documentación y mantenerla actualizada.

4.2.1.8 Beneficios de los Modelos Conceptuales

Los modelos conceptuales ofrecen grandes oportunidades para el cambio. Aunque al principio puede ser considerada como un simple ejercicio intelectual, con poca relevancia para el "mundo real", esta disciplina es la más práctica ya que tiene que ver con la forma de comunicar el pensamiento y escuchar las opiniones de los demás. Afina la capacidad para enfrentarse a los actuales tiempos de cambio.

4.2.1.9 Dificultades

Lamentablemente, este es el lugar más difícil para comenzar la construcción de un cambio conceptual. Se requiere mucha perseverancia para dominar la disciplina de los modelos mentales, tal vez porque muy pocas personas han aprendido a incorporar la investigación y la reflexión a sus pensamientos, emociones y conducta cotidiana. Cuando se comienza a practicar estas aptitudes, se envían a la superficie algunas de las respuestas inconscientes y automáticas. Observamos lo que hemos hecho de nosotros y los demás mediante un pensamiento automático. Aun después de percibir nuestros modelos mentales, no es tan sencillo aprender a actuar de otra manera.

4.2.2 El enfoque sistémico y el PPDS.

El defecto clave en la educación de los programadores es la falla para adoptar una visión global del proceso de desarrollo de software [Shaw90]. Por ejemplo, debido a la modificabilidad del software, una forma sencilla de arruinar un programa es añadirle una serie de características sin considerar el tiempo para integrarlos apropiadamente. Bajo presión, la tendencia natural es ensamblar las nuevas funcionalidades donde se pueda, sin pensar cómo se está afectando al diseño original del programa. Después de haber hecho esto repetidas veces, el programa resultante se convierte en una colección difusa e inmanejable de partes de código.

La tendencia de cada integrante del equipo para ir por su propio camino implica que puede llegar a ser necesario un control estricto de administración para asegurar que las metas individuales no rebasen las metas del equipo.

Este control puede ser más fácilmente logrado si todos los integrantes del equipo se dan la oportunidad de tomar parte activa en cada etapa del proyecto. La iniciativa individual es más propicia cuando a un integrante del equipo se le asigna la realización de un trabajo sin tener conocimiento de la parte que ese trabajo juega en el proyecto completo. Por ejemplo, digamos que el diseño de un programa es presentado a un programador para que lo implemente. Ese programador puede ver cómo el diseño puede ser implementado pero, sin entender cómo ese diseño fue realizado, esa implementación puede tener implicaciones en otras partes del sistema. Si el programador se involucra con el diseño justo desde que este inicia, el programador estará más propenso a identificarse con ese diseño y a esforzarse por mantenerlo, en lugar de modificarlo.

A los equipos que trabajan con la disciplina del PPDS no se les impide la creatividad individual de sus integrantes. Una vez que la arquitectura del sistema se ha establecido y las interfaces entre subsistemas se han definido, el individuo generalmente hace su trabajo aislado sobre un componente dado del sistema.

4.2.2.1 Pruebas

Glen Myers [Myer79] establece tres reglas que pueden servir muy bien como objetivos de las pruebas:

> Las pruebas son el proceso de ejecutar un programa con la intención de encontrar errores.
> Un buen caso de pruebas es aquel que tiene una alta probabilidad de encontrar un error no descubierto hasta ahora.
> Una prueba exitosa es aquella que descubre un error no descubierto hasta este momento.

Estos objetivos podrían implicar un cambio radical en el punto de vista del programador, ya que van en contra de la visión comúnmente tomada de que una prueba exitosa es aquella en la cual no se han encontrado errores. El objetivo es, pues, el de diseñar pruebas que sistemáticamente descubran diferentes clases de errores y hacerlo con un mínimo de tiempo y esfuerzo.

Si las pruebas son dirigidas exitosamente, de acuerdo a los objetivos expuestos arriba, descubrirán errores en el software. Como un beneficio secundario, las pruebas demuestran que las funciones del software estén trabajando de acuerdo a las especificaciones, que los requerimientos de desempeño parezcan haber sido alcanzados. Adicionalmente, los datos obtenidos mientras las pruebas son ejecutadas proveen una buena indicación de la credibilidad del software e indican, de manera general, la calidad del software. Pero hay una cosa que las pruebas no pueden hacer: las pruebas no pueden mostrar la ausencia de errores, sólo pueden mostrar su presencia. Es importante mantener esto en mente mientras las pruebas son ejecutadas.

Durante la fase de programación, el diseño detallado es implementado a través del código, resultando un programa o sistema listo para ser instalado. Existen tres tipos de pruebas que pueden ser ejecutados antes de su implantación: unitaria, de integración y sistémica. Si bien el programador es responsable de las pruebas unitarias, la responsabilidad para las pruebas de integración y las pruebas sistémicas son determinadas por el administrador del proyecto, dependiendo del tamaño del proyecto y su naturaleza.

Excepto para sistemas muy pequeños, no es realista intentar la prueba de los sistemas como si se tratara de una sola unidad. Los sistemas grandes se componen de subsistemas formados por módulos que, a su vez, pueden estar compuestos de procedimientos o funciones. Si se intenta probar el sistema como una sola entidad, es posible que no se identifique más que un pequeño porcentaje de errores. El proceso de prueba, al igual que el de programación, debe avanzar en etapas, siendo cada una de ellas la continuación lógica de la etapa anterior.

En el proceso de pruebas pueden identificarse cuatro etapas:

PRUEBAS DE FUNCIONES (UNITARIAS)

Las pruebas de funciones o de unidades es el nivel básico en donde se prueban las funciones que componen un módulo para garantizar que operan de manera correcta. En un sistema de diseño apropiado, cada función debe tener una sola especificación definida con claridad. No debe contener demasiada dificultad diseñar casos de prueba para asegurar que las funciones cumplen con su especificación. Las funciones no deben depender de otras de su mismo nivel, para posibilitar la prueba de cada función como una entidad aislada, sin la presencia de otras funciones.

PRUEBA DE MÓDULOS

Un módulo está compuesto por varias funciones que pueden interactuar entre sí. Una vez que se han probado las funciones individuales, es necesario probar la interacción entre estas funciones cuando componen un módulo. Debe ser posible probar un módulo como una entidad aislada, sin la presencia de otros módulos en el sistema.

Los módulos codificados son probados individualmente por el programador quien los escribió para asegurarse que cada uno ha sido correctamente implementado y satisface los requerimientos especificados.

Los módulos pueden ser probados en dos fases. La primera fase es llamada prueba de la "caja blanca". En esta prueba el programador sabe qué hay dentro del módulo, y suministra datos de tal manera que cada camino lógico en el programa es ejecutado. Después de esto, la segunda fase, o prueba de "caja negra" puede ser realizada. En las pruebas de caja

negra el programador ignora el interior del módulo, donde los datos son suministrados en el orden y frecuencia aproximada al uso real.

PRUEBAS DE SUBSISTEMAS (INTEGRACIÓN DE MÓDULOS)

Las pruebas de integración consisten en la integración de los módulos desarrollados por separado en el producto final. Adicionalmente, las pruebas de integración son hechas para asegurar la correctitud del sistema.

Las pruebas de integración pueden iniciarse cuando el plan de pruebas de integración esté terminado y los módulos individuales hayan sido implementados. Adicionalmente si hay una revisión de código o pruebas unitarias para cualquiera de los módulos entonces esta etapa debe también estar finalizada.

Esta prueba es el siguiente paso del proceso en el cual los módulos se agrupan para formar subsistemas. Puesto que los módulos interactúan entre sí, la prueba de integración de módulos se debe centrar en la prueba de las interfaces de aquéllos, dando por supuesto que los módulos son correctos.

Las pruebas de integración se enfocan en probar la intercomunicación de los módulos. Si el módulo principal hace llamadas a submódulos, el programador debe integrar y probar todos esos módulos para que trabajen juntos. De igual manera si él no es responsable de escribir los módulos, debe revisar todas las llamadas a y, retornos de esos módulos.

El objetivo de las pruebas de integración es ejecutar la integración de los módulos separados y probar su interacción hasta que el producto final esté ensamblado.

PRUEBAS DEL SISTEMA

Las pruebas del sistema (a veces llamadas pruebas de integración) se llevan a cabo cuando se integran los subsistemas para conformar el sistema completo. En esta etapa, el proceso de prueba tiene que ver con el hallazgo de errores en el diseño y la codificación. También se relaciona con la confirmación de que el sistema total proporciona las funciones especifica-

das en los requisitos y que sus características dinámicas cumplen con las planteadas en la definición de requisitos.

Las pruebas sistémicas examinan la operación del sistema como una entidad, algunas veces en un ambiente operativo simulado. Este tipo de pruebas asegura que los requerimientos de software hayan sido satisfactorios.

4.2.2.2 Eliminación de errores (depuración)

La depuración del sistema es el proceso de eliminar los errores lógicos y sintácticos detectados durante la codificación. Con la meta principal de obtener una pieza de código ejecutable, la eliminación de errores comparte con la verificación de código, ciertas técnicas y estrategias pero difiere en su aplicación y alcance [FIPS83].

Una vez que se ha escrito el código, este debe ser verificado en su funcionalidad con datos reales e irreales.

Algunos programadores tienden a pensar que su trabajo termina cuando el código es compilado sin errores, particularmente porque piensan que la responsabilidad de encontrar errores corresponde al área de Aseguramiento de Calidad.

La eliminación de errores es generalmente considerada a ser la fase más desgastante en el desarrollo de software y ha sido explorada empíricamente en algunos estudios [Youn74]

4.2.2.3 Mantenimiento del software

Históricamente, el término "mantenimiento" se ha aplicado al proceso de modificar un programa cuando éste ya ha sido entregado y está en operación. Esas modificaciones pueden implicar cambios sencillos para corregir errores de codificación, cambios mayores para corregir errores de diseño o reescrituras drásticas para corregir errores de especificación o introducción de nuevos requisitos.

El término "mantenimiento" se usará aquí para denotar la modificación de un programa con el fin de corregir errores y proporcionar nuevas posibilidades. Hay tres categorías de mantenimiento del software:

- Mantenimiento de perfeccionamiento.
- Mantenimiento adaptable.
- Mantenimiento correctivo.

El mantenimiento de perfeccionamiento comprende los cambios solicitados por el usuario o por el programador del sistema; el mantenimiento adaptable se debe a cambios en el ambiente del programa, y el mantenimiento correctivo es la corrección de errores del sistema descubiertos hasta el tiempo de operación. Un estudio hecho por Lientz y Swanson [LiSw80] descubrió que alrededor del 65% del mantenimiento era de perfeccionamiento, el 18% adaptable, y el 17%, correctivo.

Es imposible desarrollar sistemas, independientemente de su tamaño, que no necesiten mantenimiento. Durante la vida del sistema, sus requisitos originales se modificarán para reflejar necesidades cambiantes, el ambiente del sistema cambiará y surgirán errores ocultos, no descubiertos durante la etapa de comprobación del sistema.

Los costos de mantenimiento son muy difíciles de calcular con anticipación. La evidencia de los sistemas existentes muestra que los costos de mantenimiento son, por mucho, los más cuantiosos del desarrollo y uso de un sistema.

4.2.2.4 Beneficios del Enfoque sistémico

Entre los beneficios del Enfoque Sistémico se incluyen:

- mediante la especial atención de los elementos de la calidad del software, los programadores están más consientes de la necesidad de desarrollar software libre de errores;
- instituir una actividad de pruebas sistemáticas provee un marco en el cual nuevas tecnologías de aseguramiento de la calidad pueden ser aplicadas tan pronto como estén disponibles.

4.2.2.5 Dificultades

No es posible practicar el enfoque sistémico en forma individual, no porque la disciplina sea difícil, sino porque en un sistema complejo los buenos resultados necesitan la mayor cantidad posible de perspectivas.

4.2.3 El desarrollo personal y el PPDS

Por medio del mejoramiento de sus capacidades, el programador aumenta su conocimiento acerca del proceso de software; le permite determinar de una mejor manera la calidad y tiempo de entrega del software que debe ser desarrollado.

La investigación requiere de la adquisición de conocimientos sobre el cómo y el por qué un determinado tipo de herramienta podría ser útil. También requiere confirmar que dicha herramienta posea ciertas propiedades o produzca ciertos efectos; esto puede lograrse mediante el cuidadoso diseño de un experimento que mida dichas propiedades y permita la comparación de la herramienta con otras alternativas. El método científico puede usarse para comprender los efectos del uso de una herramienta en particular sobre un determinado medio ambiente. También puede usarse para confirmar hipótesis referentes a las formas de crear o desarrollar software de la mejor forma posible.

Como programador, se toma un incontable número de decisiones durante el desarrollo. Algunas de estas, son hechas de acuerdo a la experiencia del propio programador, así como a su estilo particular (o método) de desarrollo de software. Idealmente, todas las decisiones son hechas de acuerdo a un criterio objetivo pero en realidad, muchas decisiones no son prescritas por su técnica personal de solución de problemas, ni pueden ser justificadas o puramente objetivas. En lugar de eso, son hechas de acuerdo a sus valores.

Así que, desde esta perspectiva el programador requiere de un marco progresivo en el cual pueda enfocar su investigación y sus procesos de desarrollo, lógica y físicamente integrados para producir y tomar ventaja de

los modelos de la disciplina, que se evalúan y ajustan de acuerdo al ambiente de aplicación.

4.2.3.1 Creatividad e innovación

La clave de la innovación reside en combinar el proceso racional [Fetz88] con el proceso creativo [Zaro84] del desarrollo de software; donde la creatividad productiva casi siempre está orientada a la mejora de los productos y los procesos existentes.

El primer paso que la innovación exige es identificar las oportunidades y establecer los objetivos y los criterios. Estas son labores analíticas. Pero a continuación, basándose en esos objetivos y criterios, se ha de pensar en la creatividad: el surgimiento de nuevas ideas. Lo siguiente es proceder a otro análisis para separar las posibles soluciones y determinar qué ideas pueden convertirse en realidad.

En el supuesto caso de que exista la necesidad de innovar y, el ambiente apropiado, cuatro son los pasos básicos a seguir.

Objetivo. ¿Cuál es el resultado final deseado?
Criterios. ¿Cuáles son sus limitaciones?
Ideas. ¿Qué ideas nuevas pueden ser generadas?
Soluciones. ¿Cómo se pueden combinar las ideas para conseguir soluciones viables?

Primer paso. Lo primero que se necesita es aclarar la necesidad de la innovación decidiendo cuál es el propósito.

Antes de generar ideas y soluciones, es necesario conocer qué resultados específicos se desean y cuáles son las restricciones a las que hay que ajustarse. Los criterios hacen que la generación de ideas sea más fácil. También servirán para evaluar la viabilidad de las soluciones y su posibilidad de funcionamiento en la realidad. En esta etapa resulta útil recopilar las opiniones de los compañeros, pero a veces, el programador mismo será la única persona que tenga que aprobar la innovación.

Segundo paso. No todos los criterios tienen igual valor. Unos sugieren más ideas que otros. Para una mejor innovación deben seleccionarse

aquellos criterios que estimulen el mayor número posible de ideas. Estos suelen ser los criterios más abiertos. También se refieren directamente a la formulación del objetivo y evitan limitaciones tales como las restricciones de presupuesto o de tiempo. Los criterios que no pasen esta prueba a menudo sirven como prueba de viabilidad en etapas posteriores del proceso.

Tercer paso. En el tercer paso se generan las ideas, es aquí donde el programador puede usar su creatividad natural. No es fácil; requiere un cambio de percepción. El programador necesita suspender los juicios acerca de *cómo deberían* ser las cosas. En su lugar, debe centrarse en *cómo podrían* ser. Esto implica aceptar unos riesgos mayores en su forma de pensar.

Éste es otro momento en el que convendrá consultar con los compañeros. Las reuniones de grupo son muy útiles porque generan muchas ideas con rapidez.

Cuarto paso. Rara vez una sola idea brillante da como resultado una innovación significativa. La mayoría de las innovaciones provienen de una combinación creativa de diferentes ideas, ya de por sí buenas y factibles, que se ajustan hasta alcanzar una solución adecuada. Es aquí donde se practica la lluvia de ideas, ya sea con los compañeros o los superiores.

Ahora hay que repasar la lista de ideas y elegir las mejores, las más interesantes o las más novedosas, para luego eliminar las ideas "extravagantes" que sirvieron sólo para generar otras ideas.

No es recomendable ser demasiado estricto respecto a las ideas en este momento, ya que todavía se desea recopilar la más amplia variedad de sugerencias. Si se es demasiado selectivo ahora, se limitará la creatividad.

Una de las claves de la innovación es, como se ha visto, no hacer juicios prematuros. Esto resulta esencial para desarrollar el mayor número de ideas creativas. Una vez que se ha desarrollado un número determinado de posibles soluciones hay que empezar a evaluarlas. El hacerlo así da la oportunidad de mejorarlas y perfeccionarlas, e incluso idear otras nuevas.

Las soluciones existentes se pueden combinar para generar una solución mejor. Así, la evaluación conlleva una mayor creatividad. Lo que determina que una posible solución pueda funcionar es que esta debe ser viable; tiene que funcionar de la forma deseada y; no puede implicar riesgos inaceptables.

Para evaluar la viabilidad, se compara la solución con los criterios preliminares. Hay que comparar la solución con cada uno de los límites para ver si cumple con cada uno de ellos. Si no lo hace, hay que elegir entre modificar la solución para que satisfaga los límites o eliminar esa solución por ser impracticable.

A continuación, hay que evaluar las posibilidades de operación de las funciones. Esto se lleva a cabo comparando cada solución con los criterios deseables, comprobando sus puntos fuertes y débiles en cada una de las tareas. Puede que resulte necesario perfeccionar las soluciones para mejorar su funcionamiento.

En el cálculo de riesgos se examina lo que puede fallar en una solución, calculando la probabilidad y la repercusión de cada riesgo posible. Esto puede llevar a introducir mejoras en algunas soluciones. La mayoría de las innovaciones contienen un riesgo, y no es posible preocuparse por todos y cada uno de ellos. Lo que hay que hacer es ordenar los riesgos principales en términos de probabilidad y repercusiones.

Hay que preguntarse cuál de los riesgos es el más probable. Y si las consecuencias fuesen graves en caso de que el riesgo se materializase. Si se encuentra un riesgo probable con una repercusión grave será necesario mejorar la solución de forma que se elimine o se reduzca el riesgo.

Una vez que se haya valorado la viabilidad, el funcionamiento y el riesgo, se pueden realizar los últimos retoques. Esta puede ser una de las etapas más creativas del proceso.

El encontrar la mejor solución implica a menudo la búsqueda de una forma de añadir o suprimir algo de la solución. Y normalmente es algo bastante evidente.

4.2.3.2 Documentación personal

Desgraciadamente, los programadores sienten a menudo que deben ser capaces de recordar la forma en cómo funciona el código que escriben. Si ya se cuenta con teorías como la "programación sin ego" [Wein71], es posible entonces desarrollar otra teoría llamada "programación sin memoria". Esto no quiere decir programación sin pensar, sino programación sin sentir que se debe recordar "de memoria" el contexto e implicaciones de lo que se está programando. Además, no es posible confiar en la memoria si no hay garantía de cuánto tiempo se estará trabajando sobre las mismas tareas.

Aun cuando no haya comunicación con otras personas, la formalidad al escribir las cosas pueden ofrecerle al programador algo de comprensión extra, o traerle a la mente una consecuencia que no se había considerado anteriormente.

En numerosas ocasiones, el programador regresa a un proyecto que había terminado, y se siente mal porque no recuerda muchos detalles importantes acerca de él, es entonces cuando descubre las anotaciones que hizo cuando estaba trabajando en él.

A continuación unas recomendaciones para evitar dejarlo todo a la memoria y dejar ese espacio en la mente para almacenar otros datos.

El diario de ideas. En este diario se anotan las ideas, sin profundizar en el análisis de la misma. Está escrito en formato libre, así que es posible capturar ideas rápidamente antes que desaparezcan. Este diario contiene:

- ideas innovadoras,
- posibles soluciones para resolver problemas vigentes y,
- acciones alternativas a las actualmente practicadas,

El diario de decisiones. Este diario provee las bases para la toma de decisiones, los argumentos que la soportan y los que las rechazan. Este diario es clave para evitar perder el tiempo tratando de reorganizar las ideas que se generaron tiempo atrás. Este diario contiene:

- una descripción pequeña de cada asunto a ser decidido. Cualquier cosa que el programador tenga que seleccionar entre múltiples alternativas y las elecciones que no pueden ser documentadas en el código;
- una lista de alternativas de diseño de código;
- argumentos en pro y en contra del diseño que se haya elegido y;
- una descripción de la decisión final.

El diario de modificaciones realizadas en el código. Este diario contiene, en una manera detallada, las modificaciones que se han hecho sobre el código, de tal manera de que no quede duda del por qué del cambio. Este diario contiene:

- anotaciones que indiquen las razones más fuertes para efectuar el cambio. A menudo son las mismas anotaciones escritas, a manera de comentario, en el código, pero estas deben estar a un nivel de detalle superior;
- usualmente, también una descripción detallada por cada cambio a cada procedimiento, método, función, etc.;
- y si es necesario, una descripción línea por línea, si el cambio se ha realizado en un código complejo. Es recomendable dejar en el código las líneas sustituidas de tal manera que queden anuladas a manera de comentarios.

El uso de los diarios. Una táctica clásica de administración es la de delegar tareas para evitar disturbios en el flujo del trabajo personal. El diario de pensamientos puede ser útil aquí. Delegar para el futuro. Mientras el programador piensa en algo, debe anotarlo en un párrafo conciso. No debe tomarse el tiempo para explorar la factibilidad de la idea, solo debe registrarla en detalle suficiente para refrescar su memoria.

El programador debe hacer de su trabajo, una actividad reflexiva. Tan pronto haga un movimiento al código, debe remitirse al diario para hacer las anotaciones recomendadas.

El programador debe revisar periódicamente los diarios de pensamientos y decisiones, ya que es muy probable que el diario de pensamientos debe tener elementos que no haya resuelto aún. En ambos casos, releer las anotaciones, a menudo provoca la generación de ideas nuevas.

4.2.3.3 Autoaprendizaje y mejora continua

El propósito de la mejora continua es proveer una base para el autodesarrollo del programador. Con un proceso de autoaprendizaje y mejora continua, el programador mejora constantemente sus conocimientos y habilidades en las actividades relacionadas con su trabajo.

Para soportar el autoaprendizaje, puede verse que existen cuatro fuentes básicas de información para un programador:

- documentación impresa y publicaciones,
- sistemas de ayuda en línea,
- consultas con los compañeros y,
- el conocimiento que el programador adquiere en el transcurso de su trayectoria profesional.

Existe una tendencia natural a preferir las fuentes impresas de información, así como al intercambio de información persona a persona. Esta última, puede llegar a ser más provechosa que la primera, ya que ofrece la posibilidad de intercambiar ideas y despejar posibles dudas que puede generar la información escrita.

En lo referente a la mejora continua, puede verse que dos de los puntos más sobresalientes son: corregir errores y aprender de ellos, evitando cometer los errores del pasado.

Evitar errores: los individuos pueden obtener enormes beneficios reduciendo o eliminando errores. Estas son algunas formas de reducir o eliminar los errores cometidos:

Evitar equivocaciones por descuido. Es importante saber distinguir entre equivocaciones aceptables y equivocaciones no aceptables. Las equivocaciones aceptables son equivocaciones que surgen de la creatividad. Son una señal de desarrollo, indican una prueba de la existencia de nuevos conocimientos y de experimentación. Por otra parte, las equivocaciones inaceptables son "equivocaciones por descuido", por la repetición de errores. Este tipo de equivocaciones regularmente resultan costosos y perjudicial.

Revisar el trabajo propio. Revisar la calidad del trabajo antes de entregarlo. Es un buen hábito utilizar listas de control (*checklists*).

Evitar equivocaciones:

- considerar todos los posibles riesgos que pueden aparecer en la realización de una tarea;
- eliminar las causas básicas de los defectos y errores, no sólo los síntomas;
- esforzarse por conseguir simplicidad: soluciones "elegantes", generar y/o utilizar líneas directas de comunicación, crear procedimientos sencillos, etc..

Corregir los errores y aprender de ellos:

- es bueno descubrir errores cometidos, también es bueno admitirlos cuando los compañeros señalan esos errores;
- aceptar la crítica legitima y constructiva. Ignorar la crítica injustificada;
- estar consciente de que los errores existen y que muchos de ellos son a veces inevitables. Los errores también forman parte del desarrollo individual.

4.2.3.4 Adaptabilidad al cambio

Todo proyecto de cambio contiene un grado de incertidumbre en lo relativo a sus probabilidades de éxito. El hecho de aceptar lanzarse a lo desconocido equivale a aceptar encontrarse tanto con resultados buenos como con malos.

El programador debe ver al cambio desde una perspectiva optimista, debe alejar las fuentes de resistencia y concentrarse en buscar obtener los beneficios que este ofrece.

A continuación se discuten las oportunidades que el programador puede encontrar en un proceso de cambio, para adquirir o afinar sus habilidades y, para eliminar prácticas negativas.

Hábitos. El programador adquiere hábitos a lo largo de su carrera profesional; esos hábitos son, con gran frecuencia, más fáciles de mantener que de destruir. Ahora bien, cuando un proceso de cambio obliga al programador a abandonar una conducta "fácil", lo libera de esta y al mismo tiempo le da la oportunidad de adoptar una conducta más provechosa.

Habilidades. El cambio provee al programador de oportunidades para obtener nuevas habilidades y conocimientos que antes que este llegara no hubiera sido posible obtenerlas. El programador puede asegurar en cualquier momento que su desarrollo personal está representada por un gran numero de experiencias, algunas positivas, otras no, que le han permitido crear un ambiente satisfactorio y estimulante.

Necesidades. Cubrir las necesidades personales es un asunto cotidiano. El cambio puede traer consigo una vía para cubrir esas necesidades que posiblemente en la situación y el ambiente actual no puedan lograrse.

Modelos conceptuales. El cambio es en sí un facilitador de la generación de modelo conceptuales. A medida en la que el programador descubre sus modelos conceptuales dominantes, comienza a pensar en un modo de construir su sistema de trabajo de tal manera que respalde al modelo organizacional y a su vez que sea compatible con este.

4.2.3.5 Capacitación

Existen varios componentes para realizar una evaluación efectiva de la capacitación. Uno de los modelos de evaluación más completos y más ampliamente referenciados es el modelo de Donald Kirkpatrick [Kirk79]. Los cuatro niveles de este modelo son los siguientes:

- reacción;
- aprendizaje;
- comportamiento y;
- resultados.

NIVEL 1. EVALUACIÓN DE LA REACCIÓN.

Kirkpatrick utiliza el término "reacción" para referirse a qué tan bien los capacitados califican como agradable un programa particular de capacitación. La evaluación de la reacción de los capacitados consiste en medir sus impresiones respecto al programa de capacitación; pero no contempla la medición de su aprendizaje.

Por otra parte, Antheil y Casper [AnCa86] hacen hincapié en que la recopilación de datos con respecto a la reacción de los capacitados refleja sólo sus opiniones personales y no debe considerarse como una prueba de que se ha adquirido conocimiento nuevo.

Carnevale y Schulz [CaSc90] van más allá. Ellos afirman que las reacciones de los capacitados son fáciles de recopilar, y que proveen información sustantiva y significativa acerca de la capacitación. Aseguran también que porque la información concerniente a la recopilación de datos no revelan el aprendizaje real que han adquirido, esa información no indica con exactitud la relación existente entre la ganancia obtenida y la ganancia esperada del esfuerzo de la capacitación.

Nivel 2. Evaluación del aprendizaje.

De acuerdo a Kirkpatrick, el segundo nivel de análisis en el proceso de evaluación es el de aprendizaje. Kirkpatrick define "aprendizaje" como todo aquel principio, hechos y técnica que los capacitados comprenden y absorben. Y asegura que la evaluación del aprendizaje es mucho más difícil de medir que la reacción. De acuerdo a las pautas marcadas por Kirkpatrick, es esencial obtener mediciones exactas y significativas por medio de procedimientos estadísticos.

Endres y Kleiner [EnKl90] establecen que son necesarias las evaluaciones anteriores y posteriores a la capacitación cuando se desea evaluar la cantidad de aprendizaje que se ha adquirido. Sin un punto de comparación, la medición del aprendizaje al final del programa de capacitación, no revelará con exactitud qué tanto conocimiento se ha adquirido de tal programa. Las pruebas de "papel y lápiz", son las herramientas más frecuentemente usadas para medir el conocimiento, aunque existen otros medios para recolectar este tipo de información.

De acuerdo a Carnevale y Schulz [CaSc90], las herramientas de medición que se usan para evaluar el aprendizaje deben reflejar cada objetivo

particular del programa de capacitación. Advierten que tal medición de aprendizaje debe indicar que el método de instrucción ha sido efectivo, pero no muestra si los capacitados aplicarán esos nuevos conocimientos al trabajo, o no.

NIVEL 3. EVALUACIÓN DE LA TRANSFERENCIA DE APRENDIZAJE.

El tercer nivel en el modelo de evaluación de Kirkpatrick es la transferencia de aprendizaje. Kirkpatrick advierte que la evaluación de los programas de capacitación, en términos de comportamiento en el trabajo, es más difícil que la evaluación de las reacciones y el aprendizaje.

Antheil y Casper [AnCa86] enfatizan la importancia de recopilar y presentar la información de tal manera que sea significativa y relevante para los involucrados. Este nivel de evaluación no solamente evalúa el desempeño de la persona que recibe la capacitación sino que también provee de retroalimentación valiosa y útil para rediseñar los programas de capacitación existentes o, para diseñar programas que cumplan con futuras necesidades. Antheil y Casper recomiendan la recopilación de datos tanto cualitativos como cuantitativos.

Según Nanda [Nand88], para que el conocimiento adquirido en el proceso de capacitación sea valioso tanto para el individuo como para la organización, tal conocimiento debe repercutir en un cambio de actitud. Desgraciadamente, un factor que obstaculiza la transferencia de conocimientos es el clima organizacional, el cual puede ser inconsistente con lo que se enseñó en el programa de capacitación. Esta inconsistencia a menudo convierte a tales programas de capacitación en algo completamente inefectivo.

NIVEL 4. EVALUACIÓN DE RESULTADOS

El cuarto nivel de evaluación de Kirkpatrick es el de los resultados o impacto en la organización. Si bien, la medición de los programas de capacitación en términos de resultados puede ser la mejor manera de medir la efectividad, Kirkpatrick mismo, señala que existen muchos factores que complican y hacen muy difícil, si no imposible, evaluar ciertos tipos de programas en términos de resultados. La separación de variables para medir qué tanto se debe al mejoramiento adquirido al entrenamiento es extremadamente difícil.

Trapnell [Trap84] señala que la evaluación del impacto no es una ciencia debido al gran número de variables diferentes que están en función de la capacitación y que pueden afectar los resultados a largo plazo.

Los cuatro niveles de evaluación de la capacitación de Kirkpatrick son un marco útil al considerar las técnicas de evaluación, la evidencia es la frecuencia con la que se referencia en la literatura actual que trata de la evaluación de los programas de capacitación.

Esta sección ha ofrecido una revisión de los cuatro niveles de evaluación de la capacitación de Kirkpatrick. La experiencia reflejada en la literatura sugiere que debe incorporarse al menos los primeros tres niveles rutinariamente en el diseño de los programas de capacitación. De hecho, muchos autores recalcan la importancia de considerar con anticipación cuál de los niveles de evaluación será enfocado en el proceso del diseño del programa de capacitación. La capacidad de seguir la pista y reportar con regularidad la efectividad de los programas de capacitación, más allá de la reacción de los capacitados, documentando el aprendizaje y los cambios en el desempeño, puede ser crítico para el éxito de un programa de capacitación.

Los resultados obtenidos con el uso de los cuatro niveles de evaluación de Kirkpatrick, son aún difíciles de medir. La dificultad estriba, como el mismo Kirkpatrick señala, en la habilidad de separar la capacitación de la multitud de variables externas que pueden impactar al desempeño de los capacitados en el largo plazo.

Por último, es bueno recordar que el proceso de aprendizaje influye en el comportamiento de las personas. El aprendizaje inicia cuando la persona se interesa en un tópico y finaliza cuando se siente satisfecho con el tópico. En la transición entre estas posiciones, un individuo puede mostrar ansiedad y desesperación, y eso le resta capacidad de aprendizaje. Un resultado de esas sensaciones puede ser renunciar a apoyar el proceso de cambio.

4.2.3.6 Beneficios del Desarrollo Personal

Practicar el desarrollo personal ofrece una opción para las personas que desean cambiar su estructura organizacional. Siempre puede esforzarse, como individuo, para fomentar su desarrollo personal.

4.2.3.7 Dificultades

Un proyecto de desarrollo personal supone abandonar la premisa de que la gente está motivada principalmente por el dinero, los premios y el temor. En contraste, se debe partir de la premisa de que, en el ambiente apropiado, todos cooperan porque desean aprender, trabajar bien y ser reconocidos como personas. Aunque esta actitud puede ser difícil de cambiar; un enfoque consiste en iniciar simultáneamente un proyecto de visión compartida donde se permite a los empleados decir qué esfuerzos de dominio personal contribuirán a la evolución de la totalidad.

4.2.4 La visión compartida y el PPDS

Entender la dinámica de equipos ayuda a los programadores a trabajar en equipo. Los programadores que trabajan en equipo pueden lograr mejores resultados y mejorar las condiciones armoniosas de trabajo si entienden cómo interactúan los integrantes del equipo y cómo el equipo, siendo una entidad aislada, toma su lugar dentro de la organización.

El primer obstáculo en el uso de las prácticas de calidad es la cultura común en la práctica de desarrollo de software, la supremacía de la creatividad individual muchas veces no es compatible con la extensa aplicación de las revisiones, con la estricta adherencia a los estándares o, con la documentación detallada del trabajo.

Las observaciones sugieren que la mayoría de los individuos involucrados en la programación de computadoras son individuos orientados al trabajo [BaDu63], motivados principalmente por su trabajo. Esto implica que los equipos de programación pueden estar compuestos de individuos donde cada uno tiene su propia idea de cómo un mismo proyecto debe abordarse. Esto lo corroboran por los problemas reportados frecuente-

mente: los estándares de interfaces ignorados, sistemas rediseñados cuando ya están codificados, añadiduras de detalles innecesarios al sistema, etc.

4.2.4.1 Comunicación de la Visión Compartida

Una de las primeras razones por la que los proyectos reinician o simplemente fallan, es la carencia de una misión de proyecto la que, en este punto, significa un cuidadoso análisis de los problemas u oportunidades y su posible impacto en la organización. Los integrantes del equipo, clientes y demás personas involucradas con el sistema, necesitan comprender en buena medida los componentes fundamentales del proyecto: metas, objetivos, alcance, restricciones y visión.

4.2.5 El aprendizaje en equipo y el PPDS

La interacción entre los integrantes del equipo representa un papel muy importante en el desempeño global del equipo. Un equipo bien dirigido cuyos integrantes reconocen las habilidades y deficiencias de los demás y se apoyan mutuamente tiene más probabilidades para desempeñarse mejor que un equipo que ha sido formado al azar y cuyos integrantes tienen muy poco en común en cuanto a metas personales y profesionales y, personalidad.

4.2.5.1 Retroalimentación

Un concepto clave en los equipos de alto desempeño se refiere al aprendizaje individual y colectivo. Inicialmente, un equipo de alto desempeño tiene mucho que aprender: aprender a trabajar juntos, cómo tomar decisiones de equipo, cómo desarrollar y reforzar normas de trabajo, así como también las capacidades, talentos y habilidades de cada integrante del equipo.

Debido a que el equipo desarrolla soluciones e implementa planes, este necesita detenerse frecuentemente para revisar su entendimiento colectivo para poder llegar a acuerdos. Los detenimientos frecuentes también

son necesarios para revisar la calidad de los resultados obtenidos por el equipo.

Las mejoras a los procesos necesitan ser compartidos y comprendidos por la totalidad del equipo. ¿Cuáles son las prácticas que funcionan mejor? ¿cuáles son las que no funcionan como antes? ¿por qué sí funcionan y por qué no?. Este tipo de revisión del aprendizaje necesita ser arraigado a través del tiempo para que el equipo pueda funcionar como un equipo de alto desempeño.

La retroalimentación inmediata es crítica para que el equipo pueda desarrollarse consistentemente.

4.3 Conclusiones

El desarrollo de software es una actividad que requiere de habilidades cognoscitivas y, como cualquier habilidad, está sujeta a las limitaciones intelectuales de cada persona. Existe una gran diversidad en las habilidades individuales, estas se reflejan en diferencias en la inteligencia, educación y experiencia. Esto resulta de la manera en cómo la información es almacenada y modelada en la mente; el programador debe examinar sus modelos de procesamiento de conocimiento e intentar identificar los efectos que este puede tener en el proceso de desarrollo de software.

Como los métodos formales no garantizan por sí solos la eliminación de los errores del código, las pruebas cuidadosamente realizadas son la única manera de asegurarse que un programa funcione adecuadamente. Consecuentemente, los desastres pueden retrasar las pruebas sistémicas del producto hasta que la codificación esté casi terminada. A este punto, los programadores no pueden corregir fácilmente las malas decisiones del diseño, ya que los errores son difíciles de detectar.

La incorporación de nuevas tecnologías como las herramientas CASE, los lenguajes de programación, etc., motivan a la gente a aprender y a ajustar o adoptar los procedimientos y procesos en transición para reflejar los resultados del aprendizaje.

El aprendizaje y la evolución son dos aspectos fundamentales en el proceso de desarrollo de software: la palabra desarrollo lo justifica por sí misma. Aprendizaje y evolución pueden ser altamente restringidas, desviados y suprimidos por acciones burocráticas; esto siempre daña a la tarea primaria: el desarrollo no tendrá éxito sin la habilidad del equipo de tener la capacidad de lograr sus metas.

Capítulo 5. El ambiente en el trabajo

5.1 Introducción

En este capítulo se estudian las condiciones ambientales y personales con las cuales se puede lograr la implantación exitosa de PPDS. El capítulo inicia con una discusión acerca de cuáles deben ser las actitudes del programador para asegurar la implantación del PPDS. Después, se habla del ambiente físico de trabajo y de cómo este influye en el programador. Las características del equipo de trabajo son discutidas después de esto. Finalmente, se destaca la importancia de la comunicación, la cultura participativa y la cultura de compromisos.

5.2 Actitudes del programador

Las actitudes del programador juegan un papel muy importante en la interacción con sus compañeros y pueden impactar significativamente en su aprendizaje y su desempeño. Las actitudes positivas del programador logran resultados positivos hacia su persona y hacia su empresa. Si esas actitudes se deterioran, se verá reflejado en el cumplimiento de sus compromisos y, más importante aun, en su desempeño.

A continuación se discuten las actitudes más relevantes que influyen en el aprendizaje y el desempeño del programador.

- Identificación: la habilidad para percibir el punto de vista de otra persona como si esta fuera propia.
- Compañerismo: aligerar la tensión cuando esta no conduce al programa de desarrollo. La tensión controlada es buena, ayuda en la motivación. Sin algo de tensión no hay necesidad de lograr una tarea, no hay motivación para ir más allá.
- Proactividad: el aprendizaje es un proceso dinámico, y el programador debe tener la iniciativa de buscar la oportunidad para cubrir sus necesidades de aprendizaje y solución de problemas.

- Disponibilidad: el grado en que el programador está dispuesto a aceptar el cambio. Un factor importante para vencer a la resistencia natural al cambio es la motivación. Los hábitos que el programador encuentra confortables son normalmente resistentes al cambio. La percepción individual de las necesidades existentes y la innovación, deben ser tales que provean de ímpetu para más acción, así vencerá a esa y a otras resistencias.
- Empatía: la habilidad del programador para identificarse con las actividades de sus compañeros, especialmente con aquellos que son diferentes a los de él.
- Aceptación del cambio: es ver los cambios como normales. El programador debe monitorear y evaluar constantemente su capacidad de cambio, en flexibilidad, nuevas ideas y, qué tan rápido se adapta al cambio.

5.3 El ambiente físico en el trabajo

El ambiente de trabajo tiene efectos cuantificables muy importantes para el rendimiento de quienes trabajan en él.

Los factores ambientales incluyen elementos físicos tales como:

- tamaño y estructura de la habitación destinada,
- luminosidad,
- temperatura,
- nivel de ruido y,
- grado de privacidad

Un ambiente físico de trabajo agradable es útil. Un ambiente pobre de trabajo perturba el trabajo de calidad, es desmoralizador y es también un buen motivo para buscar trabajo en otra parte.

McCue [McCu78] da una descripción de las consideraciones arquitectónicas en la construcción del Laboratorio de Desarrollo de Software de IBM en Santa Teresa. Cada programador tiene una oficina privada con una

terminal, un escritorio adecuado y una gran ventana. Esas oficinas rodean una sala de conferencias para las reuniones de equipo.

El diseño de este edificio se llevó a cabo junto con los programadores que usarían esas instalaciones. Los factores ambientales más importantes identificados en ese estudio fueron:

- intimidad. Cada programador necesita un lugar donde pueda concentrarse y trabajar sin interrupciones;
- conciencia del exterior: la gente prefiere trabajar con luz natural y vista al exterior y;
- personalización: los individuos adoptan distintas prácticas de trabajo y tienen diferentes opiniones sobre el decorado. Es importante tener la posibilidad de reorganizar el lugar para adaptarlo a las prácticas de trabajo y personalizar así ese ambiente.

5.4 Las características del equipo

Algunas de las características más importantes para el correcto desempeño de los equipos de trabajo son:

EL TAMAÑO DEL EQUIPO.

El tamaño de los equipos de desarrollo ha de ser relativamente pequeño: entre 4 y 10 integrantes [KVHS97]. Cuando se emplean equipos pequeños, los problemas de comunicación se minimizan.

Además de disminuir los problemas de comunicación, los equipos pequeños tienen otros beneficios:

- Se puede desarrollar un estándar de calidad del equipo. Puesto que dicho estándar se logra por consenso, es más probable que se cumpla este que los estándares arbitrarios impuestos al equipo por la administración del proyecto. Los integrantes del equipo trabajan juntos y pueden aprender unos de otros [Seng90].

- Se puede practicar la programación sin ego [Wein71]. En este ambiente, los programas son considerados propiedad del equipo, y no propiedad personal.

LA ESTRUCTURA DEL EQUIPO.

La estructura del equipo tiene mucha relación con la del sistema. En forma ideal, la estructura del equipo se forma de manera que su estructura concuerde con la estructura del proyecto.

En la práctica esto suele no suceder. El administrador del proyecto debe utilizar el personal disponible cuando se inicia el proyecto y sólo en ocasiones se puede permitir el lujo de reunir a un equipo nuevo específico para cada proyecto.

Existen dos formas para contrarrestar esta inconcordancia:

- El equipo se puede estructurar de forma que toda comunicación pase a través de un coordinador central.
- El tamaño del equipo puede mantenerse en un número fijo y usar todos los canales de comunicación.

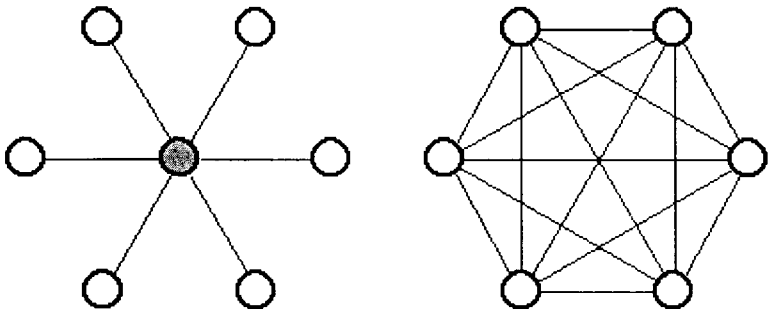


Figura 5.1. Patrones de comunicación grupal

Las investigaciones hechas por Leavitt [Leav51] y Shaw [Shaw71] sugieren que la segunda alternativa es la más efectiva. En sus experimentos, los equipos estructurados de esta manera prefirieron trabajar en grupos estructurados libremente. También concluyeron que el rendimiento en la solución de problemas de los equipos de estructura libre es superior al de los equipos de estructura centralizada.

JERARQUÍA, PERSONALIDAD Y SEXO DE CADA INTEGRANTE DEL EQUIPO.

La eficiencia de la comunicación del equipo es influida por la jerarquía, la personalidad y sexo de cada integrante del equipo. La comunicación entre los integrantes de mayor y menor jerarquía del equipo tiende a ser dominada por los de la jerarquía más alta. Los integrantes de baja jerarquía pueden inhibirse al iniciar la comunicación a causa de su jerarquía. Este efecto de jerarquía en la comunicación puede ser minimizado por el esfuerzo activo de los integrantes de más jerarquía animando a la comunicación a los de menos jerarquía. De cualquier manera, es casi imposible eliminar esto en organizaciones jerárquicas donde el progreso de un integrante novato depende de sus colegas más experimentados.

5.5 Comunicación

¿Cuántas veces puede una persona echar a perder la comunicación entre las diferentes fases del proyecto? Desde un punto de vista optimista, el programador en sí no debe tener ningún problema dialogando con él mismo, el problema inicia cundo se encuentra en un equipo que está conformado por una serie de individuos con pensamiento similar.

La comunicación es el intercambio de información entre las entidades que trabajan hacia una visión compartida. El propósito de la comunicación es el de establecer un ambiente social que ayude a la interacción efectiva y asegure que el individuo (o equipo) tenga las habilidades para compartir información y coordinar sus actividades eficientemente. La comunicación efectiva entre los integrantes de un equipo de desarrollo de software es esencial si se desea que ese equipo trabaje eficientemente.

Para efectuar una comunicación es necesario que se establezcan mecanismos que vayan de arriba a abajo y de abajo a arriba dentro de la or-

ganización, asegurando así que todos los individuos tengan las habilidades necesarias de comunicación para realizar sus tareas, coordinarlas con eficiencia, y resolver los problemas.

Establecer la comunicación efectiva inicia cuando se comunican los valores, políticas y procedimientos de la organización hacia el individuo. La capacidad de comunicación oral y escrita es mejorada mediante la capacitación. Es aquí, cuando se desarrolla la habilidad de la comunicación interpersonal, necesaria para mantener la efectiva relación de trabajo. Es necesario asegurarse que el tiempo sea usado con más efectividad, que los problemas de comunicación sean resueltos a través de los medios apropiados, que las opiniones individuales acerca de las condiciones de trabajo sean consideradas y que los procedimientos formales sean establecidos para sacar a flote las quejas y resolverlas.

Cuando el programador se desempeña en un equipo, puede dedicar parte de su tiempo a la comunicación con sus compañeros. Parte de esa comunicación es esencial ya que el programador puede compartir sus conocimientos.

5.6 Participación

El propósito de la cultura participativa es el de asegurar el flujo de la información dentro del equipo, incorporar el conocimiento de los individuos al proceso de la toma de decisiones, y obtener su ayuda para establecer los compromisos. Establecer una cultura participativa traza los cimientos para construir equipos de alto desempeño.

La cultura participativa inicia estableciendo la comunicación efectiva dentro de los equipos de trabajo y a través de todos los niveles de la organización para buscar la intervención de los individuos, involucrándolos en los compromisos y en la toma de decisiones. Mejorar la comunicación requiere de la identificación de las necesidades de información y desarrollar mecanismos para compartirla. La participación en la toma de decisiones es establecida como un mecanismo estratégico para la organización.

De acuerdo con Schrage [Schr90], la participación es el proceso de la creación compartida: dos o más individuos con habilidades complementarias interactúan para crear una apreciación compartida que nunca había sido obtenida o que no podía haberse generado por sí misma. La participación crea un modelo conceptual compartido acerca de un proceso, un producto, o un evento. Schrage lo enfatiza en una relación participativa, la labor eficiente es el asunto central; la comunicación y el trabajo en equipo existen para apoyar esto.

5.7 Compromisos

Una cultura de compromisos se basa en la confianza en que lo que se ha prometido se cumplirá. Dentro de una cultura así, los proyectos se terminan exitosamente debido, no a un esfuerzo excepcional de la gente sino, a la aplicación competente y consistente de técnicas eficientes. Las fechas se cumplen porque han sido negociadas sobre una base de realismo y honestidad. Todas las partes del proceso de desarrollo (administrador, programadores y cliente) están conscientes de lo que pueden lograr hacer, a qué precio y en cuánto tiempo.

Dentro de una cultura de compromisos, las promesas son cumplidas, en parte, porque las promesas que son hechas son realistas y alcanzables. Para saber que un plan de desarrollo es alcanzable, es necesario saber qué tan grande es un proyecto y qué puntos pueden ser logrados con los recursos disponibles.

Para iniciar una cultura de compromisos hay que ser cuidadoso con lo que se promete. Siendo más cuidadoso sobre los compromisos, se estará menos propenso a caer en la propia trampa.

No deben hacerse promesas que no se puedan cumplir. Cumplir con las fechas y el presupuesto es más fácil cuando no se compromete a lo imposible en la primera oportunidad. Esto significa que es necesario saber lo que uno es capaz o incapaz de hacer.

El programador, como tal, necesita iniciar consigo mismo. Necesita estar preparado para negociar con sus superiores compromisos alcanza-

bles. Esto significa evitar jugar el popular juego psicológico administrativo de la "motivación insustancial," en el cual las expectativas irrealistas son asimiladas por aquellos a quienes se les convence fácilmente y se les dice que todo lo que se obtenga será gracias a ellos.

El programador debe aprender a no aceptar acuerdos que sabe son irracionales. Construyendo una reputación de compromisos sustentados, el programador incrementará su poder de negociación. Demostrando confiabilidad y desempeño predecibles en el transcurso del tiempo, mejoran su credibilidad.

Los programadores que son realistas en cuanto a compromisos y progresos son a menudo objeto de burla por no ser "trabajadores en equipo" o criticados por ser de "pensamiento negativo". Realismo y escepticismo no deben ser confundidos con pesimismo. Los programadores necesitan del escepticismo y del pensamiento crítico para cuestionar acuerdos riesgosos.

5.8 Conclusiones

Algunos programadores trabajan solos, otros en equipo. Un buen administrador de proyectos que sea sensible a estas preferencias coloca al programador en la mejor situación la cuál se adecue a su estilo.

Además de su efectividad individual, la efectividad del equipo, la comunicación y el aprendizaje también son afectados por el medio físico. Los equipos de programación requieren áreas donde todos los integrantes se puedan reunir y discutir su proyecto, de manera formal e informal. Las necesidades de intimidad individual y de comunicación entre el grupo parecen ser objetivos mutuamente excluyentes, pero la solución de este problema, descrito por McCue, es agrupar oficinas individuales alrededor de cuartos centrales más grandes que se puedan usar para encuentros y discusiones del equipo.

La comunicación interna del equipo es compleja, y es muy importante que se posibilite el encuentro de sus integrantes. Estas salas deben ser capaces de acomodar a todo el equipo con intimidad; no es razonable esperar que estos encuentros tengan lugar en el rincón de una oficina.

Además de contar con los conocimientos técnicos de desarrollo de software, el programador necesita tener conocimientos sobre relaciones personales para ser capaz de comunicarse tanto en forma oral como escrita.

Capítulo 6. La implantación del PPDS

6.1 Introducción

En este capítulo se discute el proceso para implantar el PPDS. El capítulo inicia exponiendo las cuatro etapas para implantar el PPDS, las cuales son: análisis, adaptación, asimilación y adopción; en esta sección se ven también las tareas y consideraciones que el programador debe realizar en cada etapa. Después, se examina la estructura del proceso de implantación, Finalmente, se exponen las conclusiones del capítulo.

6.2 Las cuatro etapas de la implantación del PPDS

La disciplina del PPDS cubre la totalidad de la etapa de la programación de sistemas; por ello, la implantación de la misma requiere de una planeación cuidadosa. Las metas personales y los objetivos a lograr mediante la aplicación de la disciplina deben ser claramente expuestos.

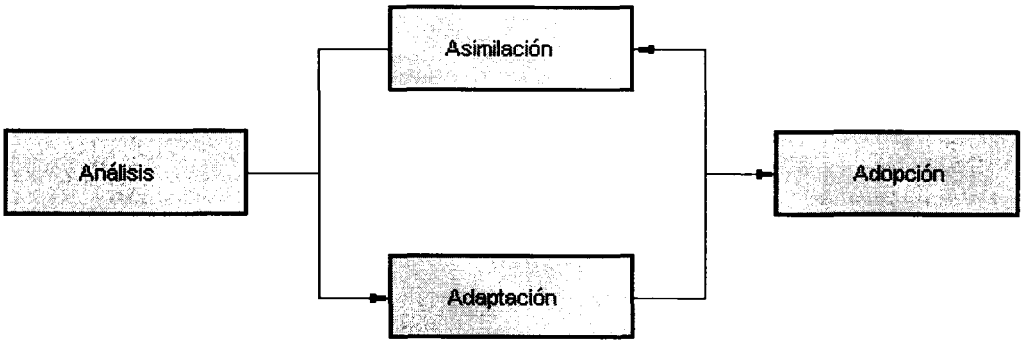


Figura 6.1. Las cuatro etapas de la implantación del PPDS

Una adecuada planeación para la introducción de la disciplina del PPDS es crítica para asegurar la correcta aplicación de la misma. Introducir una disciplina de aprendizaje dentro de un ambiente de desarrollo existente no es cosa fácil y, en el caso del PPDS, podría ser más complicado puesto que implica un cambio significativo en el sistema de trabajo del programador.

Los programadores tienen la responsabilidad de establecer eventos para medir el progreso de sus actividades dentro de las actividades del proyecto.

A continuación se explica el método de implantación de la disciplina del PPDS, el cual está compuesto por cuatro etapas: análisis, asimilación, adaptación y adopción. En la Figura 6.1 se muestra el proceso de implantación.

- Análisis. Durante esta etapa, el programador invierte una apreciable cantidad de tiempo documentándose acerca del PPDS, hablándolo con los compañeros, examinando la disciplina, y discutiendo las posibles maneras de usarlo y evaluar su utilidad.
- Adaptación. En esta etapa, el programador comienza a usar el PPDS con algunos posibles cambios con respecto a la configuración original. Estudia al PPDS como modelo universal.
- Asimilación. En esta etapa el programador comienza a darse cuenta de lo que hace, de la manera en la que el aprendizaje puede cambiar, para poder obtener los beneficios del uso de la disciplina. También el programador reconoce que el soporte institucional es requerido para transformar algunas de sus prácticas tradicionales para una mayor integración tecnológica con enseñanza y aprendizaje.
- Adopción. Después de varias iteraciones del ciclo adaptación–asimilación, el uso del PPDS se hace operacional durante la etapa de adopción. El nivel de adopción representa un nivel de compromiso para el individuo, con el uso continuo de la disciplina del PPDS.

6.2.1 Análisis

En esta sección se discutirán las tres partes principales que forman la fase de análisis en el proceso de implantación del PPDS.

La primera parte del análisis es el estudio del PPDS; después, el programador identifica sus necesidades de aprendizaje, solución de problemas y desarrollo profesional. Una vez que tenga esa base, se dispondrá a plantear sus objetivos y metas profesionales que desea lograr al trabajar con el PPDS.

6.2.1.1 Estudio del PPDS

En lo que se refiere al estudio del material a estudiar, un principio general es que primero deben recogerse todos los aspectos fundamentales del PPDS y remitirse a los detalles en etapas posteriores de la implantación. Hacer una revisión meticulosa de los capítulos de este trabajo, página por página, puede producir un exceso de detalles en perjuicio del conocimiento principal.

Un aspecto psicológico importante en la aproximación que va de lo general a lo particular es que se descubren rápidamente los grandes beneficios del completo conocimiento de los hechos más importantes en determinado campo de conocimiento, esto da ánimo para expandir el sistema de conocimiento en áreas más detalladas del PPDS. De tal manera que mientras se va avanzando, los nuevos hechos y reglas irán encajando sobre la base de conocimientos que ya se ha adquirido.

Una base fuerte facilita la tarea de ajustar las nuevas estructuras de conocimiento, proporciona ayuda adicional para trabajar y, libera de la impresión de que se está aprendiendo cosas sin sentido.

Lo primero que el programador debe estudiar es la introducción al PPDS. Aunque la base fundamental inicia desde el capítulo 2, es en el capítulo 3 donde puede inicial su estudio formal. El material contenido en estos capítulos es esencial para iniciar a practicar el PPDS.

Se recomienda que este estudio se realice en comunicación con los compañeros, a fin de llegar a concretar una percepción reforzada con diferentes opiniones o puntos de vista.

Una vez que el programador tenga bien clara esta información, estará listo para entrar a las etapas de adaptación y asimilación.

6.2.1.2 Identificar necesidades

Primero que nada, el programador necesita identificar con claridad las áreas de conocimiento que considere más útiles para el presente o un futuro cercano. Mientras más rápido se haga, más tiempo se tendrá para aplicar los conocimientos. La clave está en determinar las necesidades de aprendizaje en el contexto de las demandas de habilidades y ocupacionales. Esta es una tarea individual.

6.2.1.3 Establecer objetivos y metas

Antes de iniciar, el programador debe declarar un conjunto de objetivos claramente definidos, estos objetivos identifican dónde y cómo la disciplina del PPDS se implantará y cuáles metas y objetivos personales serán perseguidos. Debe hacerse un compromiso apropiado y realista para lograr esos objetivos. Sin ese compromiso, el uso exitoso de la disciplina del PPDS y la realización de sus beneficios en calidad y productividad podrían estar en duda.

Las metas específicas deben ser definidas por cada evento mediante la identificación de una cuantificación de un aspecto resultante de la introducción del PPDS en un proyecto en particular. El sentido debe ser el de evaluar cada evento para ver si los aspectos resultantes de la introducción del PPDS han ocurrido y, por lo tanto, si esos aspectos deben ser aceptados para continuar de acuerdo a lo planeado.

Establecer objetivos y metas es una tarea predominantemente individual aunque, si el programador lo considera necesario, puede solicitar la intervención de sus compañeros en caso de que dude si alguno de sus objetivos o metas sea alcanzable.

6.2.2 Adaptación

En esta sección se discutirán las tres partes principales que forman la fase de adopción en el proceso de implantación del PPDS.

La primera parte de la adopción es el estudio de la complejidad de las modificaciones o adaptaciones que puedan realizarse en el PPDS, Después, se discute la experimentabilidad de esas modificaciones para finalmente hablar del grado de compatibilidad que presentan respecto a las necesidades del programador.

6.2.2.1 Complejidad

La complejidad es el grado en el cual el uso y la aplicación de una innovación son vistas como una tarea difícil. Los investigadores sugieren que una innovación compleja reduce la probabilidad de adopción ya que esta requiere más habilidades y esfuerzo para poder adoptarla [CoZm90]. La complejidad ha sido ampliamente reconocida como un inhibidor de la adopción [Roge83). Es por eso, que los cambios o ajustes que se hagan sobre la disciplina del PPDS deben ser lo más simplificados y claros posibles.

6.2.2.2 Experimentabilidad

La experimentabilidad es el grado en el cual el uso de una innovación puede ser realizada en un escenario limitado antes de la adopción. Roger [Roge83] argumenta que los posibles adoptadores tienden a sentirse más cómodos con las innovaciones que pueden ser experimentadas antes de la adopción, de esta manera se incrementa la probabilidad de realizar su adopción.

La adaptación del PPDS orienta la personalización y configuración de las capacidades genéricas (procesos, métodos, herramientas, etc.) para producir instancias especificas de aquellas capacidades que pueden ser aplicadas dentro de un contexto específico.

La implantación de la disciplina del PPDS debe tomar lugar en una manera gradual. Un proyecto piloto puede proveer una oportunidad de poner a tono las prácticas del PPDS en una cultura local y, las nuevas prácticas pueden ser introducidas como resultados piloto.

6.2.2.3 Compatibilidad

Para el caso particular del PPDS, la compatibilidad es el grado en el cual el uso de una innovación es considerado como consistente con los valores existentes del individuo, su experiencia y sus necesidades. En este contexto, puede ser evaluado en términos de compatibilidad con las prácticas y cultura existentes, como son los métodos, los procesos, las herramientas, etc. Grover [Grov93] encontró que la compatibilidad fue un pronosticador de la innovación en la tecnología informática. También ha sido encontrada una asociación empírica positiva respecto al comportamiento entre compatibilidad y adopción [HoLe90].

6.2.3 Asimilación

A este grado de avance, el programador será capaz de determinar los cursos de acción que le proporcionen los resultados esperados desde que estableció sus metas y objetivos (ver sección 6.1.1.3). En esta sección se discutirán los temas de la determinación de cursos de acción y la coordinación de las actividades.

6.2.3.1 Desarrollar estrategias y cursos de acción

Las estrategias son definidas como metas a largo plazo y los métodos para lograr esas metas. El primer paso de la planeación para un programador es el de comprender lo que el proyecto debe lograr, cómo debe ser logrado, cuáles recursos son necesarios y con cuáles se cuenta.

El programador debe reconocer que las mejoras en los procesos de software es una inversión estratégica y esencial en la organización de su carrera profesional. También debe tener en mente los siguientes puntos:

- Debe enfocarse en los resultados que desea obtener sin dejar de lograr los de la compañía. Debe usar sus conocimientos y habilidades como un medio para lograr ese fin.
- Debe esperar a ver resultados conforme pasa el tiempo, no debe esperar resultados instantáneos. Es necesario tener presente que existe una curva de aprendizaje que marca la pauta de su desarrollo y de su aprendizaje.
- Debe adaptar cuidadosamente los modelos de mejora existentes y las mejores prácticas de la disciplina a su situación particular. Debe elegir modelos establecidos y usarlos como guía, no en forma obligatoria.

El programador debe ver a la mejora de sus procesos como un proyecto personal. Debe desarrollar un plan estratégico de mejora de procesos para su desarrollo personal, debe también planear acciones estratégicas para cada esfuerzo de mejora al que se ha enfocado. Asignar recursos, considerar riesgos, elaborar planes calendarizados, dar seguimiento al progreso comparándolo contra lo planeado y, medir resultados.

6.2.3.2 Coordinación de actividades

Por coordinación se entiende el alinear las entidades del proyecto para trabajar juntos hacia una meta común con un mínimo de fricción. La tarea del programador es resolver diferencias en propuestas, esfuerzos, o calendario y manejar esas diferencias para el beneficio del proyecto.

Coordinación es también la necesidad de informar a la organización de las acciones a tomar o que serán tomadas y que la afectan directamente.

6.2.4 Adopción

La última etapa de la implantación del PPDS es la adopción. En esta sección se discutirán las dos partes que forman la etapa de adopción.

En primero de ellos es la observabilidad de los resultados que genera una innovación puesta en práctica; después, la evaluación de los resultados obtenidos.

6.2.4.1 Observabilidad

La observabilidad es el grado en el cual una innovación genera resultados que son observables y pueden ser comunicados a otros. La observabilidad de una innovación en forma de resultados, tiene un impacto fuerte en la decisión de adopción [ZaDN73]. Rogers y Shoemaker [RoSh71] sugieren que la facilidad y la efectividad con los que los resultados de usar una innovación pueden ser comunicados a otros tienen una influencia significativa en la decisión de adopción. Algunos estudios han reportado falta de relación significativa entre observabilidad y el comportamiento de la adopción [HoLe90].

La adopción del PPDS da orientación al programador en la definición de sus objetivos actuales y futuros y las necesidades tecnológicas correspondientes, identificando las tecnologías que cumplen con esa necesidad, insertando esas tecnologías dentro de su trabajo en forma efectiva y desarrollando esos conocimientos para cumplir con las necesidades y objetivos futuros.

6.2.4.2 Evaluación de los resultados

El programador es responsable de medir los resultados de la aplicación del PPDS durante y al final de la misma. La evaluación le permite juzgar el progreso obtenido por el uso del PPDS. El programador puede decidir entonces si es necesario realizar cambios sobre la teoría del PPDS. Los eventos deben ser puestos en el calendario del proyecto de tal manera que exista un rango de movilización suficiente que esté siempre disponible para confrontar las decisiones contraproducentes sin arriesgar la terminación exitosa del sistema. La evaluación a cada evento debe ser enfocada sobre si los propósitos acordados están siendo logrados y si está conforme con las técnicas seleccionadas.

Las diferentes actividades encaminadas a garantizar un seguimiento constante y sistemático del cambio, corresponden a lo que podría ser denominado como "administración del cambio". Esta administración del

cambio podrá llevarse a cabo mediante actividades de control y evaluación. Si durante la implantación, el cambio no es aceptado, entonces debe regresarse a la configuración anterior al cambio, resultará más importante aun garantizar un esmerado seguimiento.

Las actividades de control se traducen en verificaciones encaminadas a averiguar si, efectivamente, se actuó de acuerdo al plan de acción establecido para proceder, en caso necesario, a ajustar los movimientos a fin de facilitar el cambio. Así, se podrán corregir errores de trayectoria durante la implantación.

Las actividades de evaluación se traducirán en acciones que permitan averiguar hasta qué punto los objetivos van camino de su realización, si se trata de una evaluación continua, o, tratándose de una evaluación final, si dichos objetivos han sido o no alcanzados. También en este caso, la evaluación permitirá proceder, si es necesario, a ajustes durante el transcurso de la implantación.

La evaluación efectiva no es un evento que ocurre al final del proyecto, sino que es un proceso continuo que ayuda en la toma de decisiones para entender mejor el proyecto; cómo está impactando al programador, y cómo está siendo afectado por factores internos y externos.

La evaluación no debe ser conducida simplemente para probar que un proyecto está trabajando, sino para mejorar la manera en la que trabaja. Por consiguiente, el programador no debe ver la evaluación sólo como una responsabilidad impuesta en el proyecto, sino más bien como una herramienta de administración y aprendizaje que vendrán para su beneficio.

El número de eventos para medir el progreso y su ubicación dentro del calendario variará de un proyecto a otro pero, como una regla general, debe aparecer frecuentemente en la parte inicial del calendario del proyecto.

6.3 Los ciclos de implantación del PPDS

En la transición del análisis a la adaptación, el programador inicia enfocándose a ajustar sus modelos conceptuales de sus actividades propias al contexto de su ambiente de trabajo y actividades particulares. El ciclo adaptación/asimilación será completado las veces que sean necesarias hasta que el programador considere que ha descubierto sus modelos conceptuales dominantes.

Una vez que el programador termine con la etapa de los Modelos Conceptuales, deberá continuar con el Enfoque Sistémico. En el ciclo adaptación/asimilación el programador trabajara tanto con el Enfoque Sistémico como con los Modelos conceptuales, este ciclo será completado tantas veces como el programador considere necesarias hasta que el Enfoque Sistémico, junto con los Modelos Conceptuales, sean eficientemente aplicados.

En los ciclos subsecuentes, se irán añadiendo las disciplina restantes de acuerdo al orden especificado en la sección 3.5. Se requiere que el programador respete el orden de las mismas para lograr una implantación más efectiva. El criterio que adoptará el programador para decidir si avanza con la siguiente disciplina será el mismo que con las disciplinas anteriores.

Se requiere de entrenamiento para implantar la disciplina del PPDS. Los programadores necesitan un completo entendimiento de los imperativos del PPDS y, un equipo sólido de profesionales necesita el apoyo suficiente por parte de la administración para practicar los principios del PPDS. De esta manera se podrá configurar el proceso consultando con otros integrantes del equipo.

En los entornos en los que usualmente los sistemas son desarrollados debido a la creación personal de los programadores individuales, debe ser incluido un periodo de transición, durante el cual, los individuos se convierten en un equipo funcional. Las actividades en el periodo de transición incluyen el desarrollo de estándares para el equipo en cuanto a estilo y estructura. Estos estándares pueden facilitar las revisiones y consensos y el proceso de comunicación. Con la experiencia, los individuos aprenderán

a apreciar la ayuda que le otorgan sus compañeros en aras de mejorar su trabajo y en aras de promover su propio crecimiento profesional

6.4 Conclusiones

Adaptar el PPDS al ambiente local de desarrollo es la parte que consume más tiempo y esfuerzo en la implantación del PPDS, pero la definición del proceso es necesaria a pesar de la propia disciplina. Una vez que exista un PPDS localmente definido, el esfuerzo más efectivo debe comenzar tan pronto el proceso sea usado, clarificado y refinado en la práctica.

Los recursos existentes que son utilizados por una organización pueden a menudo ser una base, un elemento o un complemento a la disciplina del PPDS. El PPDS no rechaza el uso de otras técnicas de ingeniería de software siempre y cuando estas no sean incompatibles con los principios del mismo. Los estándares de detección de errores por ejemplo, pueden servir como base para configurar la fase de verificación y validación dentro del PPDS.

Capítulo 7. Conclusiones

Desde el punto de vista de la investigación, el programador necesita establecer una base científica e ingenieril para realizar su trabajo. Esto es, necesita practicar la investigación constante que le permita entender los diferentes procesos, productos y otras experiencias y construir modelos descriptivos, entender los problemas asociados con el desarrollo de software, desarrollar soluciones enfocadas en el problema, experimentar con estos y analizar y evaluar sus efectos, refinar y ajustar esas soluciones para lograr el mejoramiento continuo y la efectividad y entender ampliamente sus efectos, y construir modelos de las experiencias obtenidas del desarrollo de software.

Es difícil desarrollar sistemas de información complejos que funcionen bien. En la búsqueda de la solución a este problema, o lo que el Dr. Frederick Brooks llamó "the silver bullet" (la bala de plata), muchas organizaciones centraron su atención hacia los modelos, técnicas y herramientas de desarrollo. En una época, esas soluciones fueron la programación estructurada, los lenguajes de alto nivel; hoy, hay constructores de aplicaciones, "componentware", técnicas de programación orientadas a objetos y ambientes integrados de programación. Sin embargo, una verdad incómoda ha sido ignorada: un buen programa de software puede ser escrito usando lenguajes de bajo nivel y, ciertamente, un software sin calidad puede ser desarrollado usando técnicas y lenguajes de cuarta generación.

Una de las razones por las cuales la calidad a menudo es desplazada a segundo término, es que esta no es gratis. Todavía existe una gran necesidad de mejorar herramientas, técnicas y métodos y, tal vez lo más importante: mejorar la educación y el entrenamiento para los programadores. No hay truco o técnica que elimine la complejidad de las aplicaciones modernas, pero sí hay métodos para sobreponerse a ellas; esta tesis contiene a uno de ellos.

Cuando una compañía u organización tiene poca o nula competencia, se presta muy poca atención a la calidad de las personas cuyos esfuerzos son esenciales para lograr la calidad del producto. El esfuerzo y el trabajo

individual determinan la percepción de la calidad del servicio por parte de los clientes, la cual se convierte casi en sinónimo de calidad personal.

El mejor lugar para comenzar a desarrollar la calidad en una compañía u organización es en la forma de actuar y en la actitud de su gente con respecto a la calidad [Boeh84]. La calidad personal y profesional del programador repercuten en una serie de mejoras de calidad: un proceso de desarrollo con calidad.

Con elevados niveles de calidad se crean productos y servicios de calidad superior. Por ahora, y quizá por mucho tiempo más, los programadores, seguirán siendo humanos, gente que responde a las necesidades de otro grupo de gente, creando programas computacionales diseñados para satisfacer esas necesidades. Esos sistemas de información son la salida tangible de un proceso razonado: la conversión de un proceso de pensamiento a una acción, con la cual se obtiene un producto. La meta de un programador es, o debe ser, la construcción de un producto que cumpla con las necesidades reales del grupo de personas para quienes se ha desarrollado el software.

Observar, aprender y saber evaluar son tres procesos obligatorios de todo programador moderno. Estas tres acciones resumen su proceso cognitivo. Pero antes que nada, los programadores deben tomar conciencia de la fuerza con la cual se unen estas tres acciones. También deben entender la importancia del paso que va desde el conocimiento hasta la conciencia, la cual puede llegar al final del proceso cognitivo.

El programador debe aprender a percibir la responsabilidad intrínseca de la acción de aprender. Tal responsabilidad involucra no sólo a los programadores mismos sino también a las otras personas.

En conclusión, el PPDS es una disciplina que prepara al programador para trabajar en equipo, condición que resulta indispensable en los tiempos modernos. Es una disciplina de trabajo orientada a la solución del problema, cuya implantación gradual permite al programador aplicarla sobre la marcha obteniendo así, los resultados en forma rápida.

El PPDS provoca la mínima resistencia al cambio en el programador, ya que este, es adaptado por el programador para su propio ambiente de trabajo, el programador depositará la mayor credibilidad en esta disciplina.

También, esta disciplina motiva al programador para organizar su información, fomenta la comunicación, el autoaprendizaje y, sobre todo, es económica.

Apéndice A

Uno de los criterios iniciales para diseñar este cuestionario es el de proveer un mecanismo que los desarrolladores de software puedan utilizar para evaluar su desempeño como profesionales. Al mismo tiempo busca evaluar el aprendizaje y las ventajas obtenidos del uso del PPDS así como detectar sus debilidades como desarrolladores.

La necesidad de crear este cuestionario se origina a partir de que, según la experiencia, los rangos inconsistentes en la productividad lograda entre un proyecto y otro, pueden ser el indicativo de que no se está siguiendo un proceso estandarizado de trabajo y aprendizaje. En el desarrollo de software, como en otras áreas, la productividad está definida como el diferencial (la razón) de esfuerzo aplicado sobre resultados obtenidos. En el caso del desarrollo de software, la productividad está definida como la cantidad de esfuerzo requerido para obtener un determinado grado de funcionalidad [Boeh87].

En la mayoría de las organizaciones, el software es desarrollado mediante muchos y muy diferentes procesos (y en muchos casos, ningún proceso como tal). Si el software está siendo desarrollado mediante procesos inconsistentes, entonces los rangos de productividad de los desarrolladores también serán inconsistentes. Por lo tanto, mientras el desarrollador aplique en su trabajo un proceso estándar de desarrollo y se discipline a medirlo, los rangos de su productividad se estabilizarán y lo convertirán en un desarrollador más eficiente.

Para lograr medir el desempeño del desarrollador, este análisis considera evaluar las diferentes actividades que el desarrollador realiza a lo largo de las diferentes etapas en el desarrollo del software.

FORMA DE CONTESTAR EL CUESTIONARIO

El presente cuestionario mide las 5 disciplinas del PPDS y requiere la mayor veracidad y honestidad en las respuestas del encuestado, de manera que el resultado obtenido sea un apoyo confiable para el mejoramiento del profesional.

A la derecha de cada pregunta hay marcos para las cuatro posibles respuestas: NUNCA, A VECES, CASI SIEMPRE y SIEMPRE.

Marca **NUNCA** cuando:

- La práctica no está bien establecida o está siendo ejecutada inconsistentemente.
- La práctica no se ejecuta nunca o en un número de veces poco significativo.

Marca **A VECES** cuando:

- La práctica es ejecutada en algunas ocasiones.
- La práctica puede ser ejecutada algunas veces, o con cierta frecuencia, pero es omitida bajo circunstancias difíciles.

Marca **CASI SIEMPRE** cuando:

- La práctica es ejecutada en la mayoría de las ocasiones.
- La práctica puede ser ejecutada con una gran frecuencia y es omitida sólo en contadas ocasiones.

Marca **SIEMPRE** cuando:

- La práctica está bien establecida y se ejecuta consistentemente.
- La práctica debe ser ejecutada casi siempre para poder ser considerada como bien establecida y ejecutada consistentemente como un procedimiento estándar de operación.

CUESTIONARIO

El propósito de los Modelos Conceptuales.

Los Modelos Conceptuales son representaciones sicológicas de la realidad que traemos en nuestras mentes. Mediante la interacción con nuestro medio ambiente, con otros, y con los elementos de la tecnología, formamos modelos conceptuales internos de nosotros mismos y de los sistemas con los cuales interactuamos. Estos modelos conceptuales proveen de

una capacidad predictiva y explicativa para entender las interacciones o el comportamiento de los sistemas y cómo trabajan. Los modelos conceptuales proveen mecanismos simplificados para nosotros para tratar los problemas y resolverlos.

	MODELOS CONCEPTUALES	Nunca	A veces	Casi siempre	Siempre
1	¿Revisa cuidadosamente la especificación de los requerimientos antes de implementarlos?	❑	❑	❑	❑
2	¿Elabora un diseño del programa de tal manera que cumpla con las especificaciones del cliente?	❑	❑	❑	❑
3	¿Realiza una revisión eficiente del diseño del código?	❑	❑	❑	❑
4	¿Planea sistemáticamente la integración del sistema?	❑	❑	❑	❑
5	¿Realiza pruebas de funcionamiento basándose en un plan predefinido?	❑	❑	❑	❑
6	¿Conoce y comprende claramente las políticas y normas rectoras de la empresa y se identifica con ellas?	❑	❑	❑	❑
7	¿Conoce claramente sus funciones y responsabilidades y se identifica con ellas?	❑	❑	❑	❑
8?	¿Sus objetivos son claros y sabe que sus actividades lo llevarán eficazmente al logro de los mismos?	❑	❑	❑	❑

El propósito del Enfoque Sistémico

El Enfoque Sistémico es el lenguaje para observar y expresar un fenómeno complejo, a través de aprender a expresar, compartir y, manejar interconexiones complejas. Se refiere a una forma de pensamiento y a un lenguaje para describir y entender las fuerzas e interrelaciones que condicionan el comportamiento de los sistemas. El Enfoque Sistémico ayuda a saber cómo actuar de acuerdo con los grandes procesos del mundo natural y a conocer la estructura organizacional, la estructura procedimental del desarrollo de software y los estándares de trabajo.

	ENFOQUE SISTÉMICO	Nunca	A veces	Casi siempre	Siempre
1	¿Sabe encontrar y eliminar errores en su código?	❑	❑	❑	❑
2	¿Sabe identificar riesgos que pueden presentarse en el momento de realizar sus tareas?.....	❑	❑	❑	❑
3	¿Evalúa el impacto de sus decisiones en su cliente y en la empresa?	❑	❑	❑	❑
4	¿Sabe obtener el mayor provecho de las condiciones de su ambiente laboral?	❑	❑	❑	❑
5	¿Sabe lo necesario respecto a lo que pasa en las demás áreas del equipo y está en constante, oportuna y eficaz coordinación con ellos?.....	❑	❑	❑	❑
6	¿Se relaciona con las demás áreas de acuerdo con una estrategia racional y uniforme, buscando una imagen de servicio y ayuda a los demás?	❑	❑	❑	❑

El propósito del Desarrollo Personal

El Desarrollo Personal es aprender a desarrollar la capacidad individual para producir intencionalmente los resultados más deseados. Es la creación de un ambiente organizacional que fomente en todos sus integrantes el deseo de desarrollarse hacia sus metas y propósitos profesionales.

El desarrollo personal consiste en una serie de actividades que refuerzan el continuo mejoramiento de los procesos de trabajo del individuo. Este programa implica desarrollar planes y metas para actividades de trabajo personal, estableciendo y usando procesos personales definidos, midiendo y analizando la efectividad de tales procesos, e implementando mejoras a los mismos.

	DESARROLLO PERSONAL	Nunca	A veces	Casi siempre	Siempre
1	¿Realiza investigaciones especificas en su área técnica?	❑	❑	❑	❑
2	¿Solicita retroalimentación?	❑	❑	❑	❑
3	¿Identifica áreas de oportunidad en los proyectos en los que participa?	❑	❑	❑	❑
4	¿Organiza sus actividades y se concentra en altas prioridades?	❑	❑	❑	❑
5	¿Establece compromisos y los cumple bajo los acuerdos negociados?	❑	❑	❑	❑
6	¿Tiene iniciativa y disposición para aprender y mejorar su desempeño?	❑	❑	❑	❑
7	¿Cumple sus metas de desarrollo personal?	❑	❑	❑	❑
8	¿Sabe obtener la información que necesita?	❑	❑	❑	❑

9	¿Lleva un control en la estimación de tiempos para realizar sus tareas?	❑	❑	❑	❑
10	¿Compara los tiempos ocupados para cada tarea con los tiempos estimados en cada una de ellas y aprende de la diferencia?	❑	❑	❑	❑
11	¿Utiliza consistentemente un estilo de programación?	❑	❑	❑	❑
12	¿Mejora su proceso personal de desarrollo de software basándose en los defectos que detecta y corrige?	❑	❑	❑	❑
13	¿Documenta sus actividades, problemas y soluciones y experiencias?	❑	❑	❑	❑
14	¿Al presentarse problemas recaba toda la información necesaria, y con métodos racionales busca siempre las mejores soluciones?	❑	❑	❑	❑
15	¿Constantemente está haciendo aportaciones para innovar y mejorar el método de trabajo?	❑	❑	❑	❑
16	¿Ayuda espontánea y oportunamente y colabora eficazmente con el resto del equipo?	❑	❑	❑	❑

El propósito de La Visión Compartida

La Visión Compartida representa el hecho de compartir un interés común. Es construir un sentido de compromiso en un grupo con intereses y prácticas comunes para satisfacer esos intereses.

La Visión Compartida debe estar sustentada en visiones personales de cada individuo. El punto de partida para crear una visión compartida es animar a esa visión personal. La creatividad es una herramienta para afinar la visión personal.

	VISIÓN COMPARTIDA	Nunca	A veces	Casi siempre	Siempre
1	¿Se comunica en forma clara y se asegura que sus mensajes hayan sido comprendidos?	❑	❑	❑	❑
2	¿Considera los puntos de vista y las posturas de los demás?	❑	❑	❑	❑
3	¿Escucha con atención y comprende la información que recibe?	❑	❑	❑	❑
4	¿Se involucra en la toma de decisiones y sabe por qué y para qué se han tomado tales decisiones?	❑	❑	❑	❑
5	¿Conoce y comprende los objetivos del equipo? ..	❑	❑	❑	❑

El propósito del Aprendizaje en Equipo

El aprendizaje en equipo se refiere a la capacidad de transformar las habilidades de pensamiento colectivo y conversacional, de tal manera que los grupos de personas puedan desarrollar confiablemente una inteligencia y una habilidad más grande que la suma de sus talentos individuales.

	APRENDIZAJE EN EQUIPO	Nunca	A veces	Casi siempre	Siempre
1	¿Trabaja en equipo y se integra efectivamente para el logro de los objetivos?	❑	❑	❑	❑
2	¿Muestra empatía con los compañeros?	❑	❑	❑	❑

3	¿Participa en la solución de problemas y en las propuestas?	❑	❑	❑	❑

MODO DE EVALUAR

En cada sección calificada sume 0 puntos por cada respuesta **NUNCA**; 1 punto por cada respuesta **A VECES**; 2 puntos por cada respuesta **CASI SIEMPRE**; y 3 puntos por cada respuesta **SIEMPRE**. Divida el resultado de esta suma entre la cantidad máxima de puntos que se pueden obtener en la sección y multiplíquelo por 100.

CÓMO INTERPRETAR LOS RESULTADOS DE LA ENCUESTA

MODELOS CONCEPTUALES		
Baja calificación	Escala de puntuación	Alta calificación
La persona es estática. No valora los comentarios de sus compañeros ni se preocupa por comprender mejor el porqué de las cosas.	0 ⟵⟶ 100	Esta persona está lista para el cambio. Está dispuesta a participar en un continuo proceso de retroalimentación porque puede evaluar las opiniones y críticas que recibe.

ENFOQUE SISTÉMICO		
Baja calificación	Escala de puntuación	Alta calificación
La persona no puede predecir las consecuencias de sus acciones y no entiende las fuerzas en interrelaciones que afectan en el comportamiento de los sistemas.	0 ⟵⟶ 100	La persona es capaz de evaluar las consecuencias de las acciones que ha de realizar o está realizando. Puede describir y entender las fuerzas e interrelaciones que influyen en el comportamiento de los sistemas.

DESARROLLO PERSONAL		
Baja calificación	Escala de puntuación	Alta calificación
La persona no ha logrado extender su desarrollo personal.	0 ⟷ 100	La persona ha aprendido a extender su capacidad personal para lograr los resultados que desea obtener.

VISIÓN COMPARTIDA		
Baja calificación	Escala de puntuación	Alta calificación
La persona no ha logrado incorporarse a una visión compartida en su equipo de trabajo, posiblemente porque no ha logrado descubrir sus modelos conceptuales dominantes.	0 ⟷ 100	La persona es partícipe de una cultura de compromisos con su equipo de trabajo, mediante el desarrollo de intereses compartidos del futuro que desean crear.

APRENDIZAJE EN EQUIPO		
Baja calificación	Escala de puntuación	Alta calificación
La persona no ha podido obtener conocimiento y aprendizaje mediante la interacción con sus coequiperos	0 ⟷ 100	La persona ha podido desarrollar una inteligencia y una habilidad más grande que la suma de los talentos individuales.

CÓMO MEJORAR

- EL DESARROLLO PERSONAL.

  - Concentrarse en los medios y no en el resultado, es decir, tener la capacidad para cencentrarse en metas personales relevantes, no sólo en metas secundarias. Es necesario tener intereses genuinos para así, comprometerse con naturalidad.
  - Comprometerse a mantener una visión personal y una visión clara de la realidad que nos rodea. La disciplina del desarrollo personal sugiere que podemos cultivar un modo de pensar que nos aproxime gradualmente a generar resultados tangibles.
  - Adoptar una cultura de compromisos donde las promesas se cumplan; en parte, porque son hechas en una base de realismo y honestidad. Para iniciar una cultura de compromisos hay que ser cuidadosos con lo que se promete, es necesario saber lo que uno es capaz o incapaz de cumplir.

- Los Modelos Conceptuales

  - Adoptar modelos conceptuales que capaciten para adaptarse a ámbitos o circunstancias cambiantes.
  - Los miembros de las organizaciones aprendedoras deben aprender acerca de su rol y cómo desempeñarse de una mejor manera; alineándose así con la organización; el futuro; retando las normas organizacionales; y desarrollando memoria organizacional.
  - La visión para la empresa debe ser desarrollada, compartida y entregada por sus miembros, no crecida aisladamente y "vendida" a las masas.

- La Visión Compartida

  - Decisión y compromiso. Decidirse a formar parte de la visión por elección propia además de sentirse plenamente responsable de alcanzar la visión.

- El Aprendizaje en Equipo

  - Practicar la lluvia de ideas. Esto permite la estimulación de ideas y ayuda a generar soluciones alternativas a los problemas.
  - Círculos de conocimiento. La técnica de los círculos de conocimiento es altamente motivante y es una técnica ideal para refor-

zar las habilidades en algún tópico o área de conocimiento. Provee un marco para revisar en el cual todos aprenden más o solidifican lo que ya sabían.

- Líder técnico. El integrante del equipo que tiene más conocimientos sobre el tópico reúne al equipo y le presenta conocimientos nuevos o difíciles.

- El Enfoque Sistémico

  - Estudiar las partes del sistema.
  - Ver el sistema como un todo.
  - Desarrollar conciencia del rol o función del sistema que se encuentra contenido dentro de un sistema más grande.

CONCLUSIONES

Este método semicuantitativo de evaluación provee al programador de una métrica muy práctica en el ciclo de desarrollo de software. Es una herramienta sencilla y fácil de usar para conocer el progreso y desempeño de los desarrolladores. Requiere un poco de adaptación al ambiente real de desarrollo. Este mecanismo refleja el progreso del desarrollador entre proyectos y muestra inmediatamente el impacto y el esfuerzo requerido para mantener intacto el plan original de proyecto. Este sistema puede ser usado para lograr rangos de productividad estables y puede ser usado como una herramienta de planeación para los nuevos proyectos y proveer así un plan más exacto y alcanzable.

Referencias

[AbMa91] Abdel-Hamid, Tarek and Madnick, Stuart E., "Software Project Dynamics: An Integrated Approach", Englewood Cliffs, New Jersey: Prentice Hall. 1991.

[AdCl91] Adler, P. and Clark, K., "Behind the Learning Curve: A Sketch of the Learning Process," Management Science, vol. 37, no 3, May 1991, pp. 267-281.

[AnCa86] Antheil, J.H., and Casper, I.G., "Comprehensive evaluation model: A tool for the evaluation of non-traditional educational programs." Innovative Higher Education, 11 (1), 55-64. 1986.

[ArBS92] Argote, L., Beckman, S. and Epple, D. "The Persistence and Transfer of Learning in Industrial Settings," Management Science, vol. 36, no. 2, pp. 140-154, Feb. 1992.

[Arro62] Arrow, K. "The Economic Implications of Learning by Doing," Review of Economic Studies, vol. 29, pp. 166-170, April 1962.

[BaDu63] Bass, B.M., and Duntenman, G., "Behavior in groups as a function of self, interaction and task orientation", *J. Abnorm. Soc. Psychol.*, 1963.

[BaRo88] Victor R. Basili and H. Dieter Rombach, "The TAME Project: Towards improvement--oriented software environments," IEEE Transactions on Software Engineering, vol. SE-14, pp. 758--773, June 1988.

[Basi92] Basili, V.R., "The Experimental Paradigm in Software Engineering", Proceedings of Dagstuhl-Workshop; Springer-Verlag, Sep. 1992.

[Basi93] Basili, V.R. "The (E)xperience (F)actory and its Relationship to Other Improvement Paradigms." In Proceedings of the 4th European Software Engineering Conference. Garmisch-Partenkirchen. Germany, September 13-17, 1993.

[BaTu75] Basili, V.R. and Turner, A.J. "Iterative enhancement: a practical technique for software development", IEEE Transactions on Software Engineering, vol. 1, No. 4, 1975, pp. 390-396.

[Boeh84] Boehm, Barry, "Software Engineering Economics," IEEE Transactions in Software Engineering, vol. SE-10, Number 1, Jan 1984, pp. 4-21.

[Boeh87] Boehm, Barry, "Improving Software Productivity," IEEE Computer, September 1987, pp. 43-57.

[Boeh87a] Boehm, Barry, "A Spiral Model of Software Development and Enhancement," IEEE Computer, vol. 21 pp. 61-72, may 1988.

[Boeh87b] Boehm, Barry, "Industrial Software Metrics Top 10 List," IEEE Software, September 1987, pp. 84-85.

[Broo75] Brooks, Frederick P., "The Mythical Man-Month," Addison-Wesley, Readings, Mass., New York, 1975

[Broo87] Brooks, Frederick P., "No Silver Bullet: Essence and Accidents of Software Engineering," IEEE Computer, vol. 20, no. 4, april 1987, pp. 10-19.

[CaSc90] Carnevale, A.P., and Schulz, E.R., "Return on investment: Accounting for training." Training & Development Journal, pp. 51-531. July 1990.

[Come91] Comer, E.R., "Alternative Life Cycle Models", *Aerospace Software Engineering: A Collection of Concepts*, American Institute of Aeronautics, 1992.

[CoZm90] Cooper, R.B. and Zmud, R.W.. "Information technology implementation research: a technological diffusion approach". Management Science, 36, pp. 123-139. 1990.

[CuKo92] Curtis, Bill, Kellner, Marc, and Over, Jim. Process Modeling. Communications of the ACM, September 1992, pp. 75-90.

[DaBC88] Davis, A.M., E.H. Bersoff, and E.R. Comer, "A Strategy for

Comparing Alternative Software Development Life Cycle Models," IEEE Transactions on Software Engineering, vol. SE-14, no. 10, October 1988, pp. 1453-1461.

[DeBr81] DeKleer, J., and Brown, J.S., "Mental models of physical mechanisms and their acquisition," In J.R. Anderson (ed.), Cognitive Skills and their Acquisition. Hillsdale, NJ: Erlbaum. 1981.

[Diaz97] Diaz-Herrera, J., "Object Oriented Software Development," lecture information, September 1997.

[DiSt98] Dishaw, Mark T., and Diane M. Strong "Assessing software maintenance tool utilization using task-technology fit and fitness-for-use models", Journal of Software Maintenance: Research and Practice. vol. 10, no. 3, May/June 1998, pp. 151-179.

[EnKl90] Endres, G.J., and Kleiner, B.H., "How to measure management training and development effectiveness". Journal of European Industrial Training. 14(9), 3-7. 1990

[Faga86] Fagan, M. "Advances in Software Inspections," IEEE Transactions on Software Engineering, vol. SE-12, no 7, Jul. 1986, pp. 744-751.

[Fetz88] Fetzer, J.H., "Program Verification: The Very Idea", Common ACM, vol. 31, no. 9, 1988, pp. 1048-1063.

[FIPS83] FIPS Publication 101, Guideline for Lifecycle Validation, Verification, and Testing of Computer Software, Federal Information Processing Standards Publication 101, Institute for Computer Science and Technology, National Bureau of Standards, Gaithersburg, MD, 1983.

[Forr61] Forrester, Jay. "Industrial Dynamics". Cambridge Massachusetts: MIT Press, 1961.

[GeSt83] Gentner, D. and Stevens, A. "Mental Models". Hillsdale, NJ: Erlbaum, 1983.

[Grov93] Grover, V. and Goslar, M. D., "The Initiation, Adoption and Implementation of Telecommunications Technologies in U.S. Organizations," Journal of Management Information Systems, vol. 10, No.1, pp. 141-163. 1993.

[Herb94] Herbsleb, J. et al. "Benefits of CMM-Based Software Process Improvement: Initial Results", (CMU/SEI-94-TR-13, ADA283848). Pittsburgh, PA; Software Engineering Institute, Carnegie-Mellon University, 1994.

[HoLe90] Holak, S.L. and Lehmann, D.R., "Purchase Intentions and the Dimensions of Innovation: An Exploratory Model", Journal of Product Innovation Management, Vol. 7, No. 1, pp. 59-73. January 1990.

[Hump94a] Humphrey, W.S., "The Personal Software Process - Overview, practice, and results." S.P.I. Forum, 1994

[Hump94b] Humphrey, W.S., "The personal process in software engineering." Article of SEI, CMU, 1994.

[IEEE83] Adapted from *IEEE Standard Glossary of Software Engineering Terminology*, ANSI/IEEE Std 729-1983, IEEE, Inc., NY 1983.

[IEEE90] Standard Glossary of Software Engineering Terminology, Std 610.12-1991, IEEE, 1990

[IEEE94] Survey of Existing and In-Progress Software Engineering Standards, Business Planning Group, IEEE Software Engineering Standards Committee, Version 1.1, august 1994.

[InSW93] Ince, Darrel; Sharp, Helen; Woodman, Mark, "Introduction to Software project Management and Quality Assurance," McGraw-Hill. England, 1993.

[Isaa93] Isaacs, W. N. "Taking flight: Dialogue, Collective Thinking, and Organizational Learning." Organizational Dynamics, Winter, 24-39. 1993.

[Ives94] Ives, Blake, "Probing the Productity Paradox", Management Information System Quarterly, Volume 18, Number 2, June, 1994.

[Jone86] Jones, C. "Programming Productivity," New York: McGraw-Hill Book Company, 1986.

[KhPa94] Khodabandeh, Arash; Palazzi, Paolo; "Software Development: People, Process, Technology." CERN/ECP Report 95/5.

[KiBo84] Kieras, D. & Bovair, S., "The role of mental models in learning to operate a device." Cognitive Science, 8, 255-273. 1984

[Kirk79] Kirkpatrick, D.L., "Techniques for evaluating training programs." Training and Development Journal, 33(6), 78-92. 1979.

[Knut73] Knuth, D., "The Art of Computer Programming." Addison-Wesley, 1973.

[KVHS97] A. Kamel, S. Voruganti, H. J. Hoover and P. Sorenson, "Software Process Improvement Model for a Small Organization: An Experience Report." Workshop annual '97 in metrics of Software in Oregon, Coeur d'Alene, Idaho, May 1997.

[Leav51] Leavitt, H.J., "Some effects of certain communication patterns on group performance," J. Abnorm. Soc. Psychol., 1951.

[Lehm94] Lehman, M., "Introduction to FEAST", in Proceedings of the FEAST Workshop, London, June 1994.

[Lehm95] Lehman, M., "FEAST - Feedback, Evolution And Software Technology," Software Process Newsletter, Committee on Software Process Technical Council on Software Engineering, IEEE Computer Society, No. 3, Spring 1995.

[Ling93] Ling, David; Stant Salot and Robert Daigle, "Industry's Opposition to the United States Proposed Establishment of a Software-Sector Accreditation Program."

[Marc94] Marciniak, John J. Editor. "Encyclopedia Of Software Engineering", Wiley & Sons, 1994.

[McCu78] McCue, G.M., "IBM's Santa Teresa Laboratory - Architectural design for program development," IBM Systems J., 1978.

[Musa84] Musa, J.D., "Software Reliability," in Handbook of Software Engineering, C. Vick and C.V. Ramamoorthy (eds.), Van Nostrand Reinhold Co., NY, 1984, pp. 392-412.

[Myer79] Myers, G., "The Art of Software Testing", Wiley, New York, N.Y., 1979.

[Nand88] Nanda, R., "Organizational performance and supervisor, skills." Management Solutions, 33(6), 22-28. 1988.

[NaRa68] Naur, P. and Randell, B. "Software Engineering," Proc. NATO Conf. Scientific Affairs Division, NATO, Brussels, 1968.

[NaSh82] Naumann, Justus D., and Shailendra Palvia "A Selection Model for Systems Development Tools", MIS Quarterly, vol. 6, no. 1; pp. 39-48.

[NCBM89] Norman, R., Corbitt, G., Butler, M., and McElroy, D. "CASE Technology Transfer: A Case Study of Unsuccessful Change," Journal of Systems Management, (40:5), May 1989, pp. 33-37.

[NgYe90] Ng, Peter A., Yeh, Raymond T. "Modern Software Engineerong: foundations and current perspectives", Van Nostrand Reinhold, 1990.

[Norm93] Norman, D. "Things that make us smart: Defending human attributes in the age of the machine." Reading, MA: Addison-Wesley. 1993

[Orli93] Orlikowski, Wanda, "CASE Tools as Organizational Change: Investigating Incremental and Radical Changes in Systems Development", Management Information Systems Quarterly, vol. 17, no. 3, September 1993.

[PaCC93] Paulk, M.C. et al. "Capability Maturity Model for Software, Version 1.1". Technical Report CMU/SEI-93-TR-24. Pittsburgh, PA, USA: Software Engineering Institute, 1993.

[PeSV94] Perry, D.E.; Staudenmayer, N.A.; Votta, L.G.; "People, Organizations and Process Improvement", IEEE Software, July 1994, pp. 36-45.

[Radi88] Radice, R.A., and R.W. Phillips, "Software Engineering: An Industrial Approach", vol. I, Prentice-Hall, N.J., 1988.

[Rama84] Ramamoorthy, C.V.; Prakash, Atul; Tsai, Wei-Tek; Usuda, Yutaka, "Software Engineering: Problems and Perspectives." IEEE Computer, pp. 191-229. October 1984,.

[Roge83] Rogers, Everett M. "Diffusion of Innovations". The Free Press. Third Edition. New York, N.Y., 1983.

[RoSh71] Rogers, Everett M. and F. Floyd Shoemaker, "Communication of Innovations: A Crosscultural Approach", The Free Press. New York, N.Y., 1971.

[Rout92] Rout, T., "Quality, Culture and Education in Software Engineering". Australian Computer Journal. Vol. 24, No. 3, 1992.

[Royc70] Royce, W., "Managing the development of large software systems: Concepts and Techniques," proc. WESTCON, California, USA, August 1970.

[Russ91] Russell, G. "Experience with Inspection in Ultra-Scale Developments," IEEE Software, Jan 91, pp. 25-31.

[Seng90] Senge, Peter M., "The Fifth Discipline: The Art and Practice of the Learning Organization." New York: Doubleday, 1990.

[Shaw71] Shaw, M.E., "Group Dynamics. The Psychology of Small Group Behavior." McGraw-Hill, New York, 1941

[Shaw90] Shaw, Mary. "Prospects for an Engineering discipline of software", IEEE Software, November 1990, pp. 15-24.

[SoRo96] Sommerville, I. and Rodden, T. "Human, Social and Organizational Influences on the Software Process". In: Fuggetta, Alfonso & Wolf, Alexander (eds.). Software Process. Chichester, United Kingdom: John Wiley & Sons, 1996, pp. 89-109. (Trends in Software).

[Szaj94] Szajna, Bernadette, "Software Evaluation and Choice: Predictive Validation of the Technology Acceptance Instrument", MIS Quarterly, vol. 18, no. 3, pp. 319-324, 1994.

[Trap84] Trapnell, G., "Putting the evaluation puzzle together". Training and Development Journal, 38(5), 90-93. 1984.

[TSGI94] The Standish Group International, Inc., "The Chaos Report," Dennis, Mass., 1995.

[Wadl98] Wadlin, M., "Productivity Trends Downward." Premia Corporation. http://www.premia.com/whitepaper/prodtrends.html. April, 1998.

[Wein71] Weinberg, G.M., "The Psychology of computer Programming," New York, Van Nostrand Reinhold, 1971.

[WhFr85] White, B. & Frederiksen, J. (1985). "Qualitative models and intelligent learning environments." In R. Lawler & M. Yazdani (Eds.), Artificial Intelligence and Education. Norwood, NJ: Ablex.

[Yell79] Yelle, L. "The Learning Curve: Historical Review and Comprehensive Survey," Decision Sciences, vol. 10, pp. 302-328, Feb. 1979.

[Yell90] Yellen, R. "Systems Analysts Performance using CASE versus Manual Methods," in Proceedings of the Twenty-Third Annual Hawaii International Conference on System Sciences, Hawaii, January 1990.

[Youn74] Youngs, E.A., "Human errors in programming", Int. J. Man-Mach, Stud. 6, 1974, pp. 361-376.

[ZaDN73] Zaltman, G.; Duncan, R. and Nolbeck, J., "Innovations and Organizations", John Wiley & Sons, New York, 1973.

[Zaro84] Zaron, E., "Zaron and the Art of Motorcycle Maintenance", Creative Computing, 1984, pp. 124-129.