CAPÍTULO III. MARCO TEÓRICO

En el capítulo uno se habló sobre la "crisis del software", enfocándonos a su estudio por los problemas que se presentan en los proyectos de desarrollo de sistemas. En este capítulo se tratarán las causas documentadas de tales problemas, para después introducir la ingeniería de software como la solución que la industria ha planteado a los mismos, además de comentar algunos de los diferentes paradigmas para el desarrollo de sistemas. También se presentará la administración de proyectos de software, así como algunos factores para el aseguramiento de la calidad del mismo. Finalmente, y dado que en el siguiente capítulo se propondrán estrategias para la negociación de proyectos de software, al final del marco teórico se introducen algunos conceptos sobre *negociación*.

3.1 Causas de los problemas en los proyectos de desarrollo de software

Al analizar las estadísticas y casos presentados en el capítulo uno, la primera pregunta que surge es: ¿por qué? A nadie le gusta perder tiempo y dinero, y en los proyectos de desarrollo de software que presentan cierta problemática, tanto el usuario como el desarrollador pierden ambos recursos. Por otra parte, es común que este tipo de proyectos presenten una multiplicidad de causas que contribuyen a las dificultades surgidas durante su desarrollo. A continuación hablaremos sobre algunas de las principales causas de los problemas en el desarrollo de software, documentadas en los estudios a que hicimos referencia en la exposición del problema: el de la KPMG y el de *The Standish Group*.

Causas determinadas en el estudio de la KPMG

El estudio de la KPMG [Glass, 1998] al que hicimos referencia en el capítulo uno, concluye seis causas de los problemas en los proyectos de desarrollo de software, mismas que a continuación exponemos:

1. No se especificaron de manera suficiente los objetivos del proyecto 51%
2. Planeación y estimación deficientes 48%
3. Tecnología nueva para la organización 45%

4. Administración de proyecto inadecuada o inexistente 42%

5. Personal experimentado insuficiente en el grupo de desarrollo 42%

6. Mala actuación de proveedores de hardware y software 42%

Glass añade otra categoría [Glass, 1998]: Problemas en la eficiencia del rendimiento y desempeño.

Ampliaremos ahora los comentarios sobre cada una de estas categorías.

No se especificaron de manera suficiente los objetivos del proyecto

No cabe duda de que los requerimientos del proyecto son la mayor causa de problemas en el desarrollo de software, mismos que ocurren cuando:

1. Hay demasiados requerimientos. Los proyectos grandes presentan más fallas que los proyectos pequeños.

2. Los requerimientos son inestables. Los usuarios no pueden decidir qué problemas quieren resolver en realidad.

3. Los requerimientos son ambiguos. No es posible determinar lo que los requerimientos significan en realidad.

4. Los requerimientos son incompletos. No hay suficiente información para construir el sistema.

Planeación y estimación deficientes

Lamentablemente, la estimación de tiempos y costos es, a menudo, demasiado optimista para responder a la realidad de las tareas a desarrollar. Sin embargo, en proyectos problemáticos las deficiencias en las actividades de planeación y estimación no son las únicas causas de las dificultades, ya que se mezclan con problemas en los requerimientos, con el tamaño de los proyectos, con el manejo de la nueva tecnología, y con deficiencias en la administración del proyecto.

En esta categoría encontramos también el que casi nunca se cumplen los programas de trabajo establecidos al inicio del proyecto. Los desarrolladores de software casi nunca entregamos en la fecha comprometida.

Tecnología nueva para la organización

Puede parecer irónico, pero las innovaciones tecnológicas que muchas veces nos venden los proveedores como la solución a los problemas en la implementación de sistemas, pueden a su vez ser las causas de tales problemas. Entre los tipos de problemas ocasionados por el uso de nuevas tecnologías encontramos:

1. Tecnologías que funcionan bien para proyectos pequeños, pero no para sistemas mayores

2. Tecnologías que no son la solución al problema del cliente; a veces, inclusive, puede suceder que se quiera utilizar la tecnología *per se*, esto ocurre frecuentemente con lo que yo llamo "tecnoperseguidores", es decir, el tipo de personas que quieren tener el último modelo de teléfono celular "por que sí", aunque su teléfono actual funcione de manera adecuada y satisfaga sus necesidades

3. Tecnologías que no son suficientemente maduras, o que jamás lo serán

En esta categoría, una pregunta común es si los problemas se deben a la tecnología en sí, o al personal que la maneja. Definitivamente, la peor combinación sería personal con poca experiencia que utiliza tecnología no madura.

Administración de proyecto inadecuada o inexistente

Las demás categorías de causas de problemas tienen posibilidad de ser minimizadas con una administración de proyecto adecuada. Lamentablemente, no existe ninguna "receta" para convertir a una persona en un buen administrador de proyectos; la administración de proyectos de software es, hasta cierto punto, un arte, que bien puede ser apuntalado por técnicas, métodos y herramientas, pero depende en gran medida de las habilidades y aptitudes del administrador del proyecto.

Entre los factores de mayor riesgo en este sentido, se encuentran:

1. Planeación
2. Contrato
3. Seguimiento al desarrollo
4. Conocimientos técnicos del administrador del proyecto
5. Elaboración de métricas
6. Impulso a estándares de producto y de proceso
7. Manejo de problemas culturales, políticos y organizacionales
8. Seguimiento a tiempos, costos, beneficios, y fechas de entrega
9. Administración del riesgo
10. Continuidad

Personal experimentado insuficiente en el grupo de desarrollo

Es indudable que la calidad del personal involucrado en un proyecto de desarrollo de software incidirá de manera determinante en el desahogo del mismo. Este factor aplica para todos los niveles del desarrollador: directivo, de gestión, líderes de proyecto, analistas, programadores, evaluadores, documentadores, personal de soporte y atención a usuarios, en fin, no hay rol que escape a la apremiante necesidad de contar con personal con la capacidad y habilidades necesarias para afrontar los retos de cada proyecto.

Mala actuación de proveedores de hardware y software

En muchos de los proyectos de software que presentan problemas, es frecuente encontrar que no sólo el cliente y el desarrollador los afrontaron, sino que también el proveedor de hardware o de software "se hundió en el mismo barco". Este factor es particularmente aplicable en el entorno actual de sistemas distribuidos, donde, a diferencia del ambiente de cómputo de la era de los mainframes, se tienen diversos proveedores para los elementos que componen las tecnologías de la información: equipos de cómputo, sistemas operativos, telecomunicaciones, sistemas manejadores de bases de datos, compiladores, periféricos, etc.

Problemas en la eficiencia del rendimiento y desempeño

A pesar de que ahora contamos con equipos de cómputo y telecomunicaciones mucho más poderosas, es común encontrar sistemas que no cumplen con las necesidades de operación de los usuarios en términos de eficiencia del rendimiento y del desempeño del software. La optimización y el afinamiento ("tuning") de hardware, software y comunicaciones son ahora tanto o más valiosos que en épocas pasadas.

Causas determinadas en el estudio de *The Standish Group*

El estudio de *The Standish Group* concluyó 19 criterios de éxito, ordenados por la importancia otorgada a través de un cierto número de puntos, como sigue:

	Criterio de éxito	**Puntos**
1	Participación del usuario	19
2	Apoyo de los niveles directivos	16
3	Establecimiento claro de requerimientos	15
4	Planeación adecuada	11
5	Expectativas realistas	10
6	Dividir el proyecto en fases (milestones)	9
7	Personal competente	8
8	Asignación de recursos	6
9	Visión y objetivos claros	3
10	Personal dedicado y comprometido	3

Tabla 2. Criterios de éxito de acuerdo con el estudio de *The Standish Group*

Los factores que se consideraron como las principales causas de retrasos en los proyectos fueron, en orden de importancia:

1. Falta de información del usuario
2. Requerimientos y especificaciones incompletas
3. Requerimientos y especificaciones cambiantes
4. Falta de apoyo del nivel directivo
5. Tecnología incapaz de soportar el sistema
6. Falta de recursos
7. Expectativas irreales
8. Objetivos no claros
9. Estimación de tiempos no realista
10. Tecnología nueva

Los factores que se consideraron como las principales causas de cancelación de proyectos fueron, en orden de importancia:

1. Requerimientos incompletos
2. Falta de participación del usuario
3. Falta de recursos
4. Expectativas irreales
5. Falta de apoyo del nivel directivo
6. Requerimientos y especificaciones cambiantes
7. Falta de planeación
8. No se necesitaba ya la aplicación
9. Falta de administración de tecnologías de la información
10. Falta de cultura informática

3.2 Ingeniería de software

Existen autores que afirman que el software puede ser considerado como un producto de ingeniería tanto como un avión, automóvil, televisión o cualquier otro objeto que requiera un alto nivel de habilidad para transformar la materia prima en un producto utilizable [Thayer, 1997]. Fue F. Bauer quien acuñó el término *ingeniería de software* en 1967, durante una reunión previa a la conferencia del Comité de Ciencia de la OTAN para temas de desarrollo de sistemas de software a gran escala, en Garmisch, Alemania Oriental. Esta conferencia se llevó a cabo en el otoño de 1968, reuniendo alrededor de cincuenta programadores de alto nivel, científicos de la computación y líderes de la industria, para tratar el tema de la "crisis del software", donde la ingeniería de software fue visualizada como la solución a este problema. La ingeniería de software se aplicó como tecnología, por primera vez, a mediados de la década de los setenta, y fue a finales de la misma que fue aceptada como ocupación en los Estados Unidos.

Existen diversas definiciones de *ingeniería de software* [Thayer, 1997]:

1. La aplicación práctica de la ciencia de la computación, la administración, y otras ciencias, al análisis, diseño, construcción y mantenimiento de software y de la documentación necesaria para usar, operar, y mantener el sistema de software entregado.

2. Una ciencia de la ingeniería que aplica el concepto de análisis, diseño, codificación, pruebas, documentación, y administración, a la terminación exitosa de grandes programas de computadora elaborados para resolver necesidades específicas.

3. El establecimiento y uso de los sólidos principios de la ingeniería con la finalidad de obtener software económico que sea confiable y que trabaje de manera eficiente en máquinas reales (Bauer) [Naur, 1969].

4. La disciplina tecnológica preocupada de la producción sistemática y del mantenimiento de los productos de software que son desarrollados y modificados en tiempo y dentro de un presupuesto definido.

5. La aplicación de un método sistemático, estructurado y cuantificable, al desarrollo, operación y mantenimiento del software. (IEEE)

Algunas características que hacen que la ingeniería de software presente diferencias respecto de otras ingenierías son:

1. El software es un producto enteramente conceptual.

2. El software no tiene propiedades físicas como peso, color o voltaje, y, en consecuencia, no está sujeto a leyes físicas o eléctricas.
3. La naturaleza conceptual del software crea una distancia intelectual con el problema que pretende resolver.

4. Para probar el software es necesario disponer de un sistema físico.

5. El mantenimiento del software no es sólo una sustitución de componentes.

La ingeniería de software se compone de tres elementos fundamentales:

- Técnicas a emplear en la construcción del software: Entre ellas se encuentran técnicas a emplear durante la planificación de proyectos, análisis de requerimientos, diseño de software, diseño de estructuras de datos, validación de sistemas software, mantenimiento, etc.

- Herramientas que dan soporte al desarrollo de software.

- Procesos y metodologías, nexo de unión entre las técnicas y las herramientas que definen la secuencia en que se aplican las técnicas, las entregas que se requieren, los controles necesarios para asegurar la calidad, la coordinación de cambios, etc.

Roger S. Pressman, en su libro "*Software Engineering: A Practitioner's Approach*", visualiza a la ingeniería de software como una tecnología de capas, como sigue:

Figura 5. Tecnología de capas de la ingeniería de software

Los métodos indican cómo construir técnicamente el software. Las herramientas suministran un soporte automático o semiautomático para los métodos, y los procedimientos sirven como el medio de cohesión de los métodos y herramientas. Adicionalmente, como Pressman señala [Pressman, 1993], el cimiento de toda la ingeniería de software es el enfoque en la calidad, así como todas las demás ingenierías deben descansar en un compromiso organizacional con la calidad, como se observa sobre todo en las tendencias administrativas y de ingeniería de nuestros días, entre las que se encuentra el enfoque de calidad total. La cultura de la calidad lleva, en última instancia, al desarrollo de estrategias más maduras para la ingeniería de software.

Por otra parte, la capa de los procesos es la base y el "pegamento" que mantiene unidas a las capas, favoreciendo el desarrollo de software de manera efectiva, racional y oportuna, al definir áreas clave que forman la base de la administración del proyecto y establecen el contexto en el que:

- se aplican métodos y técnicas,
- se producen entregables (modelos, documentos, reportes de datos, formas, etc.)
- se establecen métricas y puntos de revisión,
- se asegura la calidad, y
- se administra el cambio de manera apropiada.

Los métodos dicen el "cómo" se va a construir el software, agrupando una amplia variedad de tareas a realizar que incluyen: análisis de requerimientos, diseño, construcción de programas, pruebas y mantenimiento. Los métodos de la ingeniería de software descansan en una serie

de principios básicos que gobiernan cada área de la tecnología e incluyen actividades de modelado, así como técnicas descriptivas.

Las herramientas proporcionan soporte automatizado o semiautomatizado para los procesos y los métodos. Cuando se integran las herramientas para que la información creada por una herramienta pueda ser utilizada por otra, se establece lo que se conoce como CASE (computer-aided software engineering - ingeniería de software asistida por computadora). CASE combina software, hardware, y una base de datos (un repositorio que contiene información importante sobre el análisis, el diseño, la construcción de programas y las pruebas), para crear un ambiente de ingeniería de software análogo al CAD/CAE (computer-aided design/engineering - diseño/ingeniería asistida por computadora) para la creación de hardware.

A treinta años de aquella conferencia que dio nombre a la ingeniería de software, los problemas en el desarrollo de sistemas continúan. La "crisis" no ha sido superada, pero la colaboración entre el sector académico y la industria, así como el apoyo de los gobiernos, permite generar nuevas técnicas y métodos de trabajo, como sucede con la incorporación de métricas, con lo que la ingeniería de software dejará, con el tiempo, de ser un proceso todavía artesanal para convertirse en un proceso más industrial. Pero la dinámica es lenta, como señala un estudio del Software Engineering Institute de la Universidad de Carnegie Mellon, que dice que típicamente se requieren dieciocho años para que una innovación en ingeniería de software, producto de la investigación, sea adoptada por la industria. Como ejemplo de esto vamos a hablar del caso de la programación estructurada.

La actitud hacia la programación ha cambiado. En los años cincuenta se veía la programación como una actividad simple cuya función era codificar un cálculo; una década después, ya representaba una habilidad difícil pero hasta cierto punto individual; mientras que ahora se le concibe como una disciplina de equipo, compleja, para la resolución de problemas. La programación estructurada fue un importante parteaguas, tanto como concepto como conjunto de metodologías, pues llamó la atención sobre la importancia y la necesidad de abordar la programación de manera disciplinada.

Cuando fue introducida a la industria del software a principio de la década de los setenta, la programación estructurada fue definida como un conjunto de convenciones de codificación cuya función era estandarizar la forma del código fuente. Posteriormente se le describió como el proceso de diseñar, codificar y probar software de manera ordenada, con un enfoque *top-down* y de refinamiento por pasos. Más tarde se ha constituido en un concepto que refiere a metodologías que imponen una disciplina en cada etapa del ciclo de vida del software y en la organización del equipo de desarrollo.

La evolución de la definición de programación estructurada ilustra los comienzos de una mayor preocupación por la administración de proyectos de desarrollo y mantenimiento de software, más allá de los problemas técnicos que se presentan en ellos, así como por todo el ciclo de vida y las fases que lo componen. El seguimiento de metodologías estructuradas favorece el

cumplimiento de los programas de trabajo, el mejoramiento de la corrección y calidad del software producido y el incremento de las posibilidades de éxito de los proyectos.

3.3 Algunos paradigmas de la ingeniería de software

La ingeniería de software integra abarcando, en una estrategia de desarrollo, los procesos, métodos y herramientas descritos con anterioridad, así como los controles y los productos entregables, constituyendo un *modelo de procesos* o un *paradigma*, mismo que cubre tanto el desarrollo del software como la documentación del mismo. Existen varios modelos, o paradigmas, aplicables a diferentes tipos de proyectos o de aplicaciones. Entre ellos se encuentran las siguientes cinco clases o modelos [Pressman, 1997]:

- Modelo lineal (secuencial)
- Prototipaje
- Modelo incremental
- Espiral
- Métodos formales

Antes de explicar brevemente estos paradigmas, comentaremos el concepto de "ciclo de vida". La formalización del proceso de desarrollo de software se define con un marco de referencia denominado "ciclo de desarrollo del software" o "ciclo de vida del desarrollo del software", que es "el periodo de tiempo que comienza con la decisión de desarrollar un producto de software y finaliza cuando se ha entregado éste". Este ciclo, por lo general, incluye una fase de requisitos, fase de diseño, fase de implantación, fase de prueba, y a veces, fase de instalación y aceptación, aunque esto varía dependiendo del paradigma empleado. El ciclo de desarrollo del software se utiliza para estructurar las actividades que se llevan a cabo en el desarrollo de un producto de software.

A continuación hablaremos brevemente de los cinco paradigmas de la ingeniería de software:

Modelo lineal o secuencial

El *modelo lineal o secuencial*, llamado también "ciclo de vida clásico" o "modelo de la cascada", divide el ciclo de vida del producto de programación en una serie de actividades sucesivas; cada fase requiere información de entrada, procesos y resultados, bien definidos. Se denomina "de cascada" porque los productos pasan de un nivel a otro con suavidad. Este modelo requiere de una estrategia sistemática y secuencial que comienza a nivel del sistema y continúa a través del análisis, diseño, codificación, prueba y mantenimiento. Es el paradigma más antiguo y también el más usado en la ingeniería de software. Sin embargo, con el paso de los años, se han producido críticas, incluso por seguidores activos, que cuestionan su aplicabilidad a todas las situaciones.

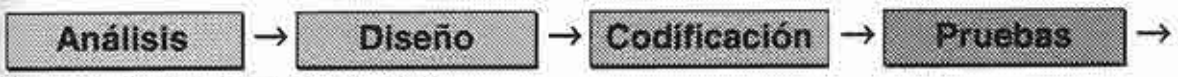

Figura 6. Modelo lineal o secuencial

- Análisis: Es indispensable comprender perfectamente los requisitos del software, para que éste no presente problemas.

- Diseño: El diseño del software es realmente un proceso de varios pasos que se enfoca sobre cuatro atributos distintos del programa: la estructura de los datos, la arquitectura del software, el detalle de los procedimientos y la definición de la interfaz. El proceso de diseño traduce los requisitos en una representación del software que pueda ser establecida de forma que obtenga la calidad requerida antes de que comience la codificación. Al igual que los requisitos, el diseño se documenta y forma parte de la configuración del software.

- Codificación: El diseño debe traducirse en una forma legible para la máquina. Si el diseño se realiza de una manera detallada, la codificación puede realizarse de manera más eficiente.

- Prueba: Una vez que se ha generado el código, comienza la prueba del programa. La prueba se centra en la lógica interna del software, asegurando que todas las opciones se han probado, y en las funciones externas, realizando pruebas que aseguren que la entrada definida produce los resultados que realmente se requieren.

Existe una fase no definida claramente pero siempre presente que es la de mantenimiento, ya que es indudable que el software una vez entregado al cliente sufrirá cambios (posible excepción es el software inmerso). Los cambios ocurrirán debido a que se hayan encontrado errores, o a que el software deba adaptarse a posibles cambios.

No obstante la velocidad de cambio y del crecimiento en las tecnologías de la informática de los últimos años, sorprende el hecho de que la forma en que el software es desarrollado ha cambiado poco desde los años setenta. El ciclo de vida clásico que, como ya señalamos, representa el desarrollo de software como un proceso de atravesar una serie de fases definidas, define productos finales (entregables) que cada fase pasa al siguiente nivel, y así, como en una cascada, el agua fluye de un nivel a otro sin regresar, así pasan dichos productos de fase a fase sin posibilidad de retorno, aunque es verdad que, en la práctica, puede haber iteraciones en ciertas fases.

Dado que la filosofía del paradigma implica completar con un alto grado de exactitud cada paso antes de empezar el siguiente, los principales problemas que se han detectado en esta aproximación son debidos a que se comienza estableciendo todos los requisitos del sistema. Sin embargo, en demasiadas ocasiones no es posible disponer de unas especificaciones

correctas desde el primer momento, porque puede ser difícil para el usuario establecer al inicio todos los requisitos. En otras hay cambio de parecer de los usuarios sobre las necesidades reales cuando ya se ha comenzado el proyecto, siendo probable que los verdaderos requisitos no se reflejen en el producto final. Otro de los problemas de esta aproximación es que los resultados no se ven hasta muy avanzado el proyecto, por lo tanto la realización de modificaciones, si ha habido un error, es muy costosa.

A pesar de lo extendido de su uso, el modelo clásico presenta algunas desventajas, resumidas como sigue:

- Los proyectos "reales" rara vez siguen el flujo secuencial propuesto por el modelo, de manera que la iteración entre las fases se da de manera indirecta. Por esto los cambios afectan tanto su aplicación.

- Es difícil para el cliente, o usuario, establecer de manera explícita sus requerimientos. El modelo requiere dichos requerimientos como entrada al análisis, por lo que la incertidumbre natural que se da sobre todo al inicio de los proyectos es difícil de manejar al aplicarlo.

- El cliente debe tener paciencia, ya que no verá versiones de programas ejecutando aquello que requiere sino hasta fases avanzadas del proceso. Cualquier detalle malentendido al expresar o recabar sus requerimientos no será visto sino hasta que los programas que integran el sistema sean revisados y probados.

Prototipaje

Un prototipo es una representación o modelo del producto de software a desarrollar. Por lo regular, un prototipo tiene un funcionamiento limitado en cuanto a capacidades, confiabilidad o eficiencia.

Hay varias razones para desarrollar un prototipo; una de ellas es ilustrar los formatos de datos de entrada, mensajes, informes y diálogos al cliente, constituyéndose en un mecanismo adecuado para explicar opciones de procesamiento y tener un mejor entendimiento de las necesidades de él, por lo que el *prototipaje* es una buena estrategia en situaciones en las que existe demasiada incertidumbre sobre los requerimientos del usuario, o sobre la eficiencia de algún algoritmo, o sobre el tipo de interfaz para la interacción hombre-máquina.

En este paradigma se comienza cuando usuario y desarrollador, conjuntamente, establecen los objetivos del proyecto, identifican los requerimientos "conocidos", y enlistan áreas que requerirán ser definidas más adelante. Entonces se hace un "diseño rápido" que se enfoca en las entradas y las salidas que verá el usuario (típicamente las pantallas y los reportes). El "diseño rápido" lleva a la construcción de un prototipo, mismo que es evaluado por el usuario o cliente y que es utilizado para refinar los requerimientos del software que será desarrollado.

El proceso se repite sucesivamente hasta que el prototipo se ajusta a las necesidades del usuario, mismas que así serán más claras para el desarrollador. Muchas veces se construye un prototipo al que se le incorporan algunas funciones reales (o procesos) que el sistema debe cubrir, reutilizando código de otros sistemas o aplicando herramientas como generadores de reportes.

El prototipaje beneficia tanto al usuario como al desarrollador, permitiendo al primero ver cómo será el sistema, mientras que el segundo puede presentar resultados más inmediatos. Sin embargo, tiene algunas desventajas:

Relacionadas con las expectativas del cliente:

- En los prototipos "huecos", aquellos donde no se incorporan funciones reales y que entonces son una serie de pantallas y reportes no muy distintas a una presentación en computadora, el cliente puede llegar a pensar que sólo será cuestión de incluir algunos detalles para hacer que su prototipo trabaje en ambiente real, presionando así al desarrollador a entregar resultados antes de lo factible.

1. En prototipos "funcionales", el cliente ve trabajando lo que parece ser una primera versión del software, ignorando que, por las prisas en hacer que funcione, el desarrollador no ha considerado los aspectos de calidad o de mantenimiento del software a largo plazo. Cuando se le informa de que el producto debe ser reconstruido, el cliente piensa que sólo solicitando que se apliquen "unas cuantas mejoras" será suficiente para hacer del prototipo un producto final que funcione. Y el desarrollador del producto cede demasiado a menudo.

Relacionadas con la forma en que el desarrollador produjo el prototipo:

- El desarrollador, frecuentemente, impone ciertos compromisos de implementación con el fin de obtener un prototipo que funcione rápidamente. Puede que utilice un sistema operativo inapropiado, o un lenguaje de programación equívoco o simplemente porque ya está disponible y es conocido, puede que implemente ineficientemente un algoritmo, sencillamente para demostrar su capacidad.

- El desarrollador puede comprometer la incorporación de algunas funciones que no puedan ser implementadas en la plataforma escogida para el desarrollo del software real.

- El desarrollador puede implementar el prototipo en una plataforma conocida o dominada por sus programadores, sin haberse detenido a evaluar la conveniencia técnica de dicha plataforma para las características del proyecto, dado que conocía éstas de inicio.

A pesar de las desventajas presentadas, el prototipaje puede ser un paradigma efectivo si se definen las reglas del juego desde el principio, esto es, si tanto el cliente como el desarrollador

acuerdan que el prototipo será construido únicamente para efectos de definir requerimientos, por lo que será después descartado (al menos en parte), para dar paso a un desarrollo de software enfocado a la calidad y al posterior mantenimiento del sistema.

Con el advenimiento y uso extendido de los llamados lenguajes de programación de cuarta generación (4GLs), el prototipaje ha venido incrementando su popularidad como técnica de desarrollo para los sistemas *front-office*, ya que una característica de la mayoría de los 4GLs es la facilidad con la cual se puede producir el front-end, es decir, la interfaz de usuario, permitiendo una construcción y modificación rápida de los prototipos electrónicos del software que se está desarrollando.

Modelo incremental

Cuando se utiliza el *modelo incremental*, el primer incremento es un *producto central*. Esto es, se cubren los requerimientos básicos, pero las características suplementarias (conocidas y desconocidas) no son incorporadas al producto a entregar. El usuario utiliza o prueba al detalle el producto central, y como resultado de este uso o evaluación, se elabora un plan para el siguiente incremento. Dicho plan aborda la modificación del producto central para satisfacer mejor las necesidades del cliente, así como para incorporar algunas características y funcionalidad adicionales. Este proceso se repite de manera sucesiva hasta que se llega al producto completo.

Como se puede observar, tanto el prototipaje como el modelo incremental y el espiral que veremos más adelante, son paradigmas iterativos por naturaleza. Sin embargo, el modelo incremental se enfoca a la entrega de un producto funcional en cada incremento, o iteración. Cada incremento constituye una versión "recortada" del producto final, pero permite al usuario utilizar un producto y, al desarrollador, evaluarlo.

El modelo incremental es útil cuando no se cuenta con el personal suficiente para llevar a cabo la implementación de todo el producto en el tiempo establecido para el proyecto, ya que los primeros incrementos se pueden ir implementando con poca gente. El grupo de desarrollo puede ir creciendo, si se requiere, en el siguiente incremento. Además, se pueden planear los incrementos de manera que los riesgos técnicos sean controlados, por ejemplo, si se planea utilizar una plataforma de hardware que aún no sale al mercado, al elaborar los primeros incrementos evitando el uso de ese nuevo hardware (de ser esto posible), y entregando así algo al usuario final sin demoras fuera del alcance del desarrollador.

Modelo en espiral

El *modelo en espiral* combina la naturaleza iterativa del prototipaje con los aspectos sistemáticos y de control del modelo lineal. Al usar este paradigma, el software es desarrollado en una serie de versiones incrementales. Durante las primeras iteraciones, la

versión incremental puede ser un prototipo, mientras que en las últimas iteraciones se producen versiones más completas. Cada pasada por la espiral se mueve a través de seis tareas (cada tarea se divide a la vez en subtareas que se adaptan a las características del proyecto que se está abordando):

- Comunicación con el usuario, con la finalidad de establecer un canal efectivo entre el cliente y el desarrollador.

- Planeación: Determinación de objetivos, alternativas y restricciones, es decir, cuáles serán los alcances del software a desarrollar, recursos, tiempos, límites de presupuesto (costos razonables), las diferentes alternativas en personal, tecnología, dirección del proyecto, y demás información relacionada.

- Análisis de riesgo: Análisis de alternativas e identificación y resolución de riesgos, a fin de que los riesgos en el proceso del desarrollo del software no quebranten las restricciones establecidas desde el principio. Involucra realizar análisis de costos de modelos, de proyectos extraordinarios, etc., midiendo así el impacto tanto de los riesgos técnicos como de administración del proyecto.

- Ingeniería: Construcción de una o más representaciones de la aplicación. Desarrollo del proceso de "siguiente nivel". Creación de prototipos (desde prototipos iniciales hasta prototipos finales del software) y aplicación de ingeniería para el desarrollo del producto.

- Liberación, esto es, probar, instalar y proporcionar soporte al usuario (por ejemplo, documentación y capacitación).

- Evaluación con y del cliente: Obtener retroalimentación basada en la evaluación de las representaciones del software creadas durante la etapa de ingeniería (hasta ese nivel), e implementadas en la de liberación, para determinar los puntos críticos en el funcionamiento del mismo.

En opinión de Gilb [Gilb, 1988], el modelo en espiral es una estrategia más realista para el desarrollo de sistemas de gran escala, ya que al ir evolucionando el software que se va construyendo, tanto el desarrollador como el cliente pueden entender y reaccionar antes los riesgos presentados por cada nivel, o iteración en la espiral. Se puede usar el prototipaje como un mecanismo para la reducción de riesgos, en cualquier etapa de la evolución del producto final. Se toman las ventajas sistemáticas de las fases del ciclo de vida clásico, incorporándolas en un marco de iteraciones que refleja el mundo real. Si se aplica correctamente, la espiral puede reducir riesgos antes de que se conviertan en problemas, ya que requiere de la consideración de éstos en todas las etapas del proyecto.

Pero también presenta desventajas. Puede ser difícil convencer a los clientes, especialmente al momento de definir los términos del contrato de desarrollo, que esta estrategia basada en la evolución del producto puede ser controlable. Requiere de mucha experiencia en la

evaluación y medición de riesgos. El paradigma fue presentado por Barry Boehm en un artículo llamado *"A Spiral Model for Software Development and Enhancement"* [Boehm, 1988], y no ha sido tan ampliamente utilizado como el modelo lineal y el prototipaje, por lo que se requerirá más tiempo antes de que se compruebe su eficacia.

El modelo planteado por Boehm [Boehm, 1988] vino a definir un proceso para el desarrollo de software completamente diferente al de cascada, tratando de resolver la rigidez del mismo y agregando la medición de los riesgos que implica el continuar con el desarrollo del producto. Es un método que implementa los aciertos del prototipo y del ciclo de vida clásico, añadiendo un análisis de riesgo y considerando una dimensión radial [Pressman, 1997].

Al aplicar este tipo de modelo se debe tener cuidado en la aplicación de cada nivel del proceso evolutivo, es decir entender y reaccionar a los riesgos en cada nivel, así como la reducción de los riesgos generados en el mismo proceso de desarrollo.

Métodos formales

El paradigma de los *métodos formales* abarca un conjunto de actividades que llevan a la especificación matemática formal del software, al establecer una notación matemática rigurosa para especificar, desarrollar y validar un sistema.

Cuando se utilizan los métodos formales, el análisis matemático permite encontrar y corregir problemas tales como la ambigüedad y la falta de consistencia, en lugar de descubrirlas a través de un proceso de prueba y error. Es un modelo promisorio para alcanzar la meta de software libre de defectos, aunque presenta algunas desventajas:

- El desarrollo de métodos formales requiere una importante inversión de tiempo y dinero.
- Dado que pocos desarrolladores tienen la preparación y experiencia suficientes para aplicar los métodos formales, se requiere mucha capacitación.
- Es difícil usar los modelos como mecanismo de comunicación con usuarios no técnicos.

Por lo anterior, este paradigma es más aplicable en proyectos donde cada defecto pueda tener consecuencias fatales, como puede ser el software de control en aparatos de misión crítica (como un avión o un equipo biomédico).

3.4 Administración de proyectos de ingeniería de software

La administración de proyectos surge como una forma de prevenir los problemas que se presentan en el desarrollo de productos, al ser un sistema de procedimientos, prácticas, tecnologías, y *know-how* que apoya la planeación, organización, integración, dirección y control necesarios para administrar de manera exitosa un proyecto de ingeniería. Si el