3 El Proceso para el Desarrollo de Software

Un *proceso* está definido como una serie de acciones u operaciones que conducen a un fin. En general, una empresa u organización requiere de uno o más procesos para lograr sus objetivos, los cuales por lo general involucran la utilización de sistemas de software. En el caso de una empresa que se dedica al desarrollo de software, se requieren procesos que abarquen desde la creación de un sistema de software hasta su mantenimiento. Todo esto es conocido como el *ciclo de vida* del software. Como hemos visto en el Capítulo 1, el desarrollo de sistemas de software es algo muy complejo. ¡De lo contrario todos haríamos siempre software perfecto! Un aspecto básico para manejar la complejidad inherente en los sistemas de software es contar con un modelo de proceso a seguir, como se discutirá en el resto del capítulo.

3.1 Modelo del Proceso

El *modelo de proceso* define un orden para llevar a cabo los distintos aspectos del proceso. El modelo se puede definir como un grupo de estrategias, actividades, métodos y tareas, que se organizan para lograr un conjunto de metas y objetivos. El modelo de proceso abarca aspectos como la *planeación*, *autoridad*, *predicción*, *evaluación* y *rastreabilidad* (“traceability”).

?? La planeación involucra definir cómo se llevarán a cabo las diversas etapas del proceso sin limitarse a aspectos de desarrollo si no también por ejemplo, los organizacionales.
?? La autoridad define cómo se puede influir para llegar a donde se quiere.
?? La predicción describe a donde se va a llegar.
?? La evaluación describe donde se encuentra el proceso actualmente.
?? La rastreabilidad describe cómo se logró un resultado particular.

En particular, el proceso de desarrollo es considerado como un conjunto de personas, estructuras organizacionales, reglas, políticas, actividades, componentes de software, metodologías y herramientas usadas o creadas específicamente para conceptualizar, desarrollar, ofrecer un servicio, innovar o extender un producto de software, es decir la forma en que la organización realiza sus distintos proyectos de generación de software.

Los modelos de proceso varían mucho entre sí y dependen de las diversas opiniones o máximas generales en las cuales se basan [Goldberg & Rubin 1995], donde obviamente cada persona puede tener una opinión distinta al respecto. Por ejemplo algunas creencias en el desarrollo de software son:

?? Es mejor comprender el problema antes de desarrollar una solución.
?? El proceso para resolver un problema debe dar un resultado predecible, sin importar del individuo que hace el trabajo.
?? Debe ser posible planear y calcular el proceso con gran precisión.
?? Evaluar y administrar el riesgo es importante para el éxito del proceso.
?? Etapas bien definidas con entregas intermedias aumentan la confianza que se tiene en el resultado final.

En general, todas las creencias luego actúan como base para definir las *estrategias*, *actividades*, *métodos*, y *tareas* del modelo de proceso. Estos conceptos se describen a continuación.

?? Una estrategia es un plan para llevar a cabo un objetivo, en nuestro caso el desarrollo de software. Existen diversas estrategias para lograr mejor calidad en el software final. Una estrategia básica se relaciona con el tipo de arquitectura que se desea crear, por ejemplo, utilizando elementos sencillos como bloques y componentes o como elementos prefabricados de más alto nivel. Esta arquitectura puede incluso integrar diversos niveles de sofisticación en los elementos. Las estrategias básicas escogidas afectan directamente el tipo de programación y los lenguajes que se utilizarán. En cierta manera, para este libro ya hemos definido nuestra estrategia básica de desarrollo de software, la cual es el uso de tecnología orientada a objetos, en particular usando el lenguaje Java. Sin embargo, aún dentro esta estrategia de orientación a objetos puede refinarse aún mas. (Obviamente, se puede utilizar una estrategia distinta, incluso que no sea orientada a objetos.) La estrategia no sólo afecta la arquitectura del sistema sino tambien cómo se llevarán a cabo las actividades del proceso. Mientras no se tengan conflictos, es posible combinar múltiples estrategias, donde las distintas actividades del proceso de software pueden hacerse bajo estrategias diferentes, definiendo implícitamente la estrategia global del modelo de proceso. Dos estrategias importantes son el uso de prototipos y reutilización. Hablaremos de esto más adelante.
?? Una actividad es una unidad o paso organizacional para llevar a cabo cierto aspecto de un proceso. En nuestro caso las actividades definen los distintos pasos necesarios para lograr las metas y objetivos definidos en el modelo de proceso, o sea en el desarrollo de software. Las actividades dependen de la arquitectura de software y deben ser simples de aprender y usar; deben simplificar la comprensión del sistema, deben ser suficientemente

poderosas para expresar la información requerida para modelar el sistema, deben ser lo suficientemente descriptivas para poder discutir el sistema sin ambigüedades y deben proveer un modelo evolucionable del sistema. Las actividades básicas necesarias para el proceso de desarrollo de software son las siguientes: (i) *requisitos* para capturar los aspectos funcionales correspondientes, cómo un usuario interactuaría con el sistema; (ii) *análisis* para dar al sistema una estructura robusta y extensible bajo un ambiente de implementación ideal; (iii) *diseño* para adoptar y refinar las estructuras al ambiente de implementación particular; (iv) *implementación* para codificar el sistema; (v) *pruebas* para validar y verificar el sistema; (vi) *integración* para pegar componentes del sistema; (vii) *documentación* para describir los distintos aspectos el sistema y (viii) *mantenimiento* para extender la funcionalidad del sistema.

?? Un método es un procedimiento definiendo las tareas que deben llevarse a cabo para satisfacer la actividad. Existen métodos, por ejemplo, para asegurar la calidad del software, seguir el progreso del proyecto y probar el software. Durante el análisis, el método debe ayudar en la identificación de los objetos necesarios para la arquitectura del sistema. Análisis estructurado y análisis orientado a objetos son ejemplos de diferentes métodos para hacer análisis, cada uno con sus propias tareas. Una *metodología* se refiere al estudio de los métodos, existiendo un gran número de metodologías para el desarrollo de software. En general, distintas metodologías llevan a cabo las actividades del desarrollo de software de diferente manera. En este libro buscamos aplicar las metodologías más evolucionadas utilizando tecnología orientada a objetos. En el apéndice del libro contrastaremos algunas de estas metodologías.

?? Una tarea es un grupo relacionado de acciones contribuyendo a una acción mayor. Cada método define un conjunto de tareas a llevarse a cabo para lograr los objetivos deseados. La tarea puede incluir condiciones de entrada y de salida que serán satisfechas antes y después de su realización.

Existen procesos de acuerdo al tipo de proyecto como se verá en la Sección 3.1.1 y aunque no hay límite a los diversos modelos de proceso que puedan existir, describiremos los más “clásicos”: el Modelo de Cascada en la sección 3.1.2 y el Modelo de Espiral en la sección 3.1.3. Cada uno de estos modelos de proceso está definido con un propósito particular y posee distintas estrategias para especificar las diferentes actividades, métodos y tareas. El tema de la madurez del modelo será tratada en la sección 3.1.4.

3.1.1 Procesos Adaptados a Tipos de Proyectos

Una creencia común pero equivocada en la industria del software es que hay un sólo modelo de proceso que sirve para todo tipo de proyecto basados en tecnología orientada a objetos. En general, el modelo de proceso depende del tipo particular de proyecto que se esté llevando a cabo. Algunos de estos tipos de proyectos son:

?? **Primer proyecto de su tipo**, donde se va a crear la mayoría del software desde cero, aunque obviamente se pueden aprovechar componentes genéricos para su desarrollo. Por ser la primera vez que se crea este tipo de software, se requiere más tiempo para analizar el dominio del problema que para otros proyectos. Incluso aunque el dominio del problema sea familiar pudiera ser ésta la primera versión de un sistema de software de este tipo. En un primer proyecto en su tipo, la incertidumbre crea riesgos adicionales.

?? **Segundo proyecto en su tipo**, donde se busca agregar nueva funcionalidad a un proyecto conocido. El desarrollador típicamente tiende a excederse agregando demasiada funcionalidad en comparación al proyecto anterior (“featuritis”). El sistema se vuelve muchos más grande que el original significando retrasos en el sistema, como ocurre con muchos de los paquetes comerciales de la actualidad.

?? **Variación de un proyecto**, donde se extiende un sistema ya existente. Esto puede involucrar introducir componentes de software reutilizables como un marco (“framework”), crear nuevos componentes o simplemente extender la aplicación existente mediante nueva funcionalidad. Dependiendo de la estrategia particular, el modelo de proceso debe variar. Por lo general, el riesgo en este tipo de proyectos es mucho menor que en los primeros proyectos de su tipo. Lo que se debe hacer ya está definido por la naturaleza del software existente, sin embargo se debe comprender las nuevas extensiones en el software en especial si éstos involucran componentes reutilizables.

?? **Proyecto de reescritura de legado** (“legacy”), donde se busca transformar o hacer una “reingeniería” de un sistema ya existente desarrollado bajo tecnologías anteriores, a un sistema desarrollado bajo nuevas tecnologías, tales como las orientadas a objetos. Este ha sido el enfoque más importante para tratar el problema del año 2000. En lugar de remendar sistemas se aprovechó para rescribirlos. Como la organización ya ha escrito el sistema por lo menos una vez antes, el proyecto de reescritura de legado tiene varias características en común con otros modelos, como variación de un proyecto donde por ejemplo, se incluirá actividades para examinar el sistema existente, para extraer requisitos y para comprender la arquitectura anterior. También tiene aspectos comunes con un primer proyecto en su tipo, ya que, se debe crear una nueva arquitectura sin poder contar con

software reutilizable del proyecto anterior. Además, existen todos los riesgos involucrados con un primer proyecto usando una nueva tecnología.

?? **Proyecto de creación de software reutilizable**, donde se busca crear uno o más componentes de software reutilizables. Este tipo de proyecto es muy similar a otros proyectos de desarrollo de software, siendo necesario comprender requisitos y desarrollar el diseño completo del componente. Sin embargo, es diferente de otro tipo de sistemas, en que los requisitos tienen que considerar las necesidades de múltiples proyectos, asegurando que el diseño es suficientemente general para ser útil en otras situaciones desconocidas. Por lo general, esto requiere de esfuerzos mucho mayores que para software no reutilizable, razón por la cual la mayoría del software existente no es reutilizable.

?? **Proyecto de mejora de sistema o mantenimiento**, donde se busca modificar los componentes básicos de un sistema para apoyar nueva funcionalidad. Tales proyectos a menudo son relativamente pequeños en alcance y no incluyen rescribir componentes o la aplicación completa. Se debe tener una buena comprensión de los componentes a ser mejorados y cómo estos cambios afectan el resto del sistema.

3.1.2 Modelo Cascada

El modelo de cascada clásico data de la década de los 60s y 70s (Royce 1970, Boehm 1981). El modelo de cascada se define como una secuencia de actividades a ser seguidas en orden, donde la estrategia principal es definir y seguir el progreso del desarrollo de software hacia puntos de revisión bien definidos (“milestones” o “checkpoints”). En su época de esplendor, este modelo tuvo gran aceptación en la comunidad de contratistas gubernamentales estadounidenses, ya que éstos recibían sus pagos del gobierno en base a entregas basadas en horarios (“schedule”) predefinidos. El desarrollo de software implicaba una secuencia de actividades a realizarse y cuyo seguimiento era verificar que cada actividad haya sido completada. La ejecución del modelo era muy lineal, por lo cual el modelo fue sencillo y atractivo; donde se especificaba las actividades para luego hacerlas de principio a fin. Se consideraba que una vez terminada una actividad se continuaba con la siguiente. La Figura 3.1 muestra un diagrama conceptual del modelo describiendo el orden a seguir de las actividades del desarrollo de software. No se muestra una etapa explícita de “documentación” dado que ésta se llevaba a cabo durante el transcurso de todo el desarrollo.

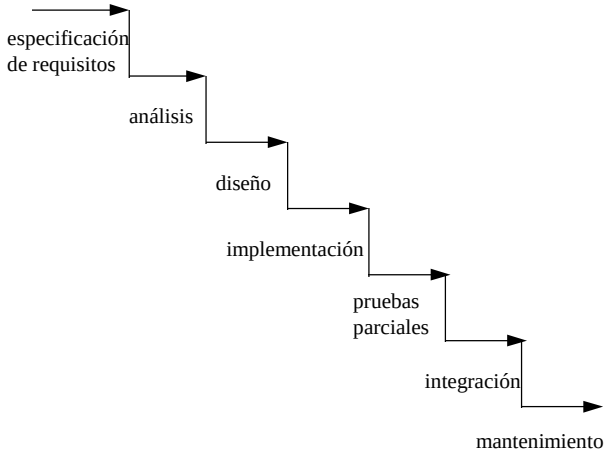

Figura 3.1 Diagrama del modelo de cascada.

Las siguientes máximas sirven de base para el Modelo de Cascada (Goldberg y Rubin 1995):

?? Las metas se logran de mejor manera teniendo como fin puntos de revisión bien definidas y documentadas, dividiendo el desarrollo en etapas secuenciales bien definidas.

?? Documentos técnicos son comprensibles para usuarios y administradores no-técnicos, y estos participantes no-técnicos pueden comunicarse de forma efectiva durante las diversas actividades. Cada detalle sobre los requisitos puede conocerse de antemano antes de desarrollarse el software, y estos detalles son estables a través del desarrollo.

?? Pruebas y evaluaciones pueden llevarse a cabo eficientemente al final del desarrollo.

El modelo de cascada fue inicialmente bien recibido ya que identificaba etapas razonables y lógicas para las diversas actividades. Lamentablemente, el modelo no explicaba entre otras cosas cómo modificar un resultado. No existía una guía del por qué y cuándo se debía revisar un resultado previo para sus posibles cambios, en especial

considerando que es extremadamente difícil definir todos los requisitos de un sistema al inicio y que estos se mantengan estables y sin cambios a lo largo del desarrollo. Esta rigidez trajo dudas sobre la utilidad del modelo. La ironía en la mayoría de los proyectos de desarrollo que usan este modelo es que los administradores no están de acuerdo con las máximas básicas, aunque eligen modelos de proceso basados en ellas. A menudo, el requisito de producir entregas intermedias (mayormente documentos) para ser seguidos por financiamiento obliga a seguir este enfoque secuencial, separando drásticamente las actividades, aún cuando los administradores crean que otro enfoque sería mejor. Por lo tanto, el modelo de cascada dejó de ser utilizado de acuerdo a su definición original, llevando a los usuarios a utilizar variantes del modelo básico.
Ed Yourdon, en su libro *Decline and Fall of the American Programmer* (Yourdon 1992), discute los problemas con el Modelo de Cascada:

?? Los documentos a entregar rigen el proceso de software.
?? Toma demasiado tiempo ver resultados.
?? Depende de requisitos estables correctos.
?? Hace difícil rastrear (ver la dependencia) de los requisitos iniciales y el código final.
?? Retrasa la detección de errores hasta el final.
?? No promueve el reuso de software.
?? No promueve el uso de prototipos.
?? No se practica de manera formal.

3.1.3 Modelo Espiral

El modelo de espiral es una modificación al modelo de cascada desarrollado durante la década de los 80s (Boehm 1988). El modelo de espiral se basa en una estrategia para reducir riesgo, al contrario del modelo de cascada que es dirigido por documentos. Como parte del manejo de riesgo el modelo incorpora una estrategia de uso de prototipos, algo muy aceptado en la actualidad. El modelo enfatiza ciclos de trabajo, cada uno de los cuales estudia el riesgo antes de proceder al siguiente ciclo. Cada ciclo comienza con la identificación de los objetivos para una parte del producto, formas alternativas de lograr los objetivos, restricciones asociadas con cada alternativa, y finalmente procediendo a una evaluación de las alternativas. Cuando se identifica incertidumbre, se utilizan diversas técnicas para reducir el riesgo en escoger entre las diferentes alternativas. Cada ciclo del modelo de espiral termina con una revisión que discute los logros actuales y los planes para el siguiente ciclo, con el propósito de lograr la incorporación de todos los miembros del grupo para su continuación. La revisión puede determinar si desarrollos posteriores no van a satisfacer las metas definidas y los objetivos del proyecto. En tal caso, se terminaría el espiral. Para utilizar este modelo se debe ser particularmente bueno en identificar y manejar riesgos. La Figura 3.2 muestra un diagrama conceptual del modelo de cascada describiendo los distintos ciclos del espiral. Nuevamente, no se muestra una etapa explícita de “documentación” dado que ésta se llevaba a cabo durante el transcurso de todo el desarrollo.

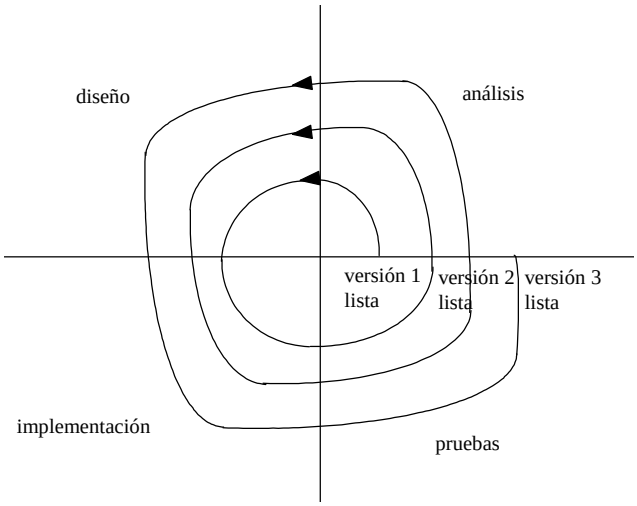

Figura 3.2 Diagrama del modelo en espiral.

El modelo de espiral contempla que el desarrollo de sistemas es un proceso de cambios progresivos. Un sistema normalmente se desarrolla mediante cambios en la especificación de la versión anterior del sistema que son incorporados a nuevas versiones, donde un cambio se conoce como un *delta en la especificación de requisitos* o *versión*. En la Figura 3.3 se ilustra este concepto.

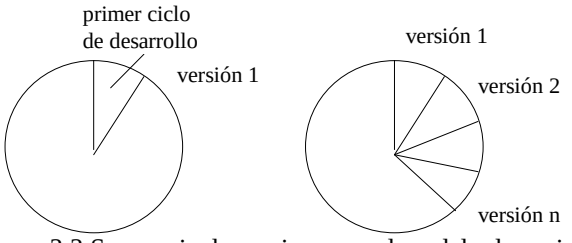

Figura 3.3 Secuencia de versiones en el modelo de espiral.

Aunque modificaciones a sistemas existentes constituyen la mayor parte del costo total durante el ciclo de vida del software, la mayoría de los métodos de desarrollo de software se concentran en nuevos desarrollos, tratando revisiones como algo menor. El modelo de proceso debiera enfocarse en los cambios del sistema.
Las máximas del modelo de espiral (Goldberg y Rubin 1995) son:

?? Una actividad comienza con un entendimiento de los objetivos y riesgos involucrados.
?? Basado en la evaluación de soluciones alternas, se usa las herramientas que mejor reduzcan los riesgos.
?? Todo el personal relacionado debe involucrarse en una revisión que determina cada actividad, planeando y comprometiéndose a las siguientes actividades.
?? El desarrollo puede proceder en incrementos en cada etapa, permitiendo prototipos sucesivos del producto.

Con algunas variantes, este es el modelo de proceso más importante en la actualidad.

3.1.4 Modelo Win-Win

El modelo "Win-Win" [Boehm 1998] se basa en el modelo de espiral y da énfasis en la identificación de las condiciones de *ganancia* para todas las partes implicadas. Se crea un plan para alcanzar las condiciones ganadoras, determinando los riesgos involucrados.
El principal objetivo del modelo es establecer las reglas para la definición del proceso de desarrollo del proyecto tomando en cuenta a todos los implicados. Son cuatro los ciclos del modelo consistiendo de cuatro actividades principales cada uno:

?? Definición de los objetivos del proceso y elaboración del sistema y subsistemas del producto.
?? Evaluación de las alternativas con respecto a los objetivos del proyecto. Identificación y resolución de las fuentes principales de riesgo en el proceso de desarrollo de los productos.
?? Elaboración de la definición de los productos y procesos.
?? Planeación del siguiente ciclo. Calendarización del ciclo de vida del plan, incluyendo la partición del sistema en subsistemas para llevar el proceso en ciclos paralelos.

Una vez revisadas las actividades principales, los ciclos manejados en el modelo marcan líneas muy específicas a seguir:

Ciclo 0. Aplicación básica. Se determina la viabilidad de la plataforma para el desarrollo de la aplicación.
Ciclo 1. Aplicación de los objetivos del ciclo de vida. Se desarrolla los objetivos del ciclo de vida, incluyendo prototipos, planes, especificaciones de aplicaciones básicas y verificación de la existencia de una arquitectura viable para cada capa de la aplicación.
Ciclo 2. Aplicación de arquitectura del ciclo de vida. Se genera la especificación del proyecto, detallando la arquitectura del ciclo de vida.
Ciclo 3. Capacidad de operación inicial. Se define el alcance inicial para cada proyecto.

Las máximas o creencias del modelo son las siguientes:

1. Crear software basado en componentes para lograr mayor calidad en sistemas de mayor tamaño.
2. Escribir software reutilizable para eficientar el proceso de desarrollo.
3. Medir la calidad del sistema como aspecto clave del desarrollo del producto.
4. Lograr mayor calidad en el proceso de ensamblaje a partir de componentes menores.
5. Usar tecnología basadas objetos como aspecto básico para lograr la calidad.
6. Poder lograr sistemas más rápidamente, sencillos, confiables y de calidad a través de procesos bien definidos.

7. Utilizar el modelo de espiral como base del proceso.
8. Flexibilizar el proceso de desarrollo del software para lograr los objetivos generales de eficiencia.
9. Involucrar al cliente mediante el manejo de prototipos.
10. Analizar los riesgos en el proceso del desarrollo del software para asegurar la calidad final del sistema.

No hay límite en el alcance o tipo de proyectos donde pueda ser aplicado el modelo "Win-Win". No se necesita mucho tiempo de gestión, de forma que se puede utilizar en proyectos pequeños, tanto como proyectos más grandes.

3.2 Calidad de Software y Madurez del Proceso

La *calidad de software* significa diferentes cosas para distintos grupos. Para la IEEE la calidad de software es el grado en que un sistema, componente o proceso cumple con los requerimientos especificados y con las necesidades o expectativas del cliente o usuario [American National Standard, 1984]. En la definición de la norma ISO 9000, la calidad de software es el grado (pobre, bueno o excelente) en que un conjunto de características inherentes del software cumplen con los requisitos.

La calidad del software está directamente ligada con el proceso de desarrollo de software. En general, se supone que un proceso bien conocido y ampliamente utilizado, sustentado en medición y predicción de eventos, debe permitir controlar en buena medida la producción de software [De Marco, 1982] y consecuentemente la calidad de estos productos. Sin embargo, la producción de software sigue siendo compleja y difícil de obtener, aunque se están haciendo esfuerzos importantes para facilitar la producción y aumentar su calidad. Uno de los esfuerzos que han logrado mejores frutos es el desarrollo de modelos de *madurez del proceso* de producción de software que permite no sólo la estandarización de la producción a manera de cualquier otro producto sino el permitir una mejora continua.

Una de las principales razones es que los modelos plantean un cambio de cultura de la organización y una fuerte inversión en recursos, como son financieros, tecnológicos y principalmente humanos.

La industria del software sólo lleva medio siglo, razón por la cual una gran parte de los líderes de proyectos, análisis, diseñadores y desarrolladores de productos siguen trabajando de manera artesanal. Es lamentable ver que no sólo empresas pequeñas, sino también medianas y grandes siguen viendo al software cómo algo difícil de predecir y controlar.

Los factores implicados en la obtención de un producto de calidad:

?? el cliente/usuario, participante primordial en el proceso de desarrollo del producto y responsable en definir los requisitos del producto final (sistema).
?? el desarrollador, responsable del proceso de producción y del aseguramiento de la calidad del producto.
?? el proceso, (definido anteriormente).
?? el producto, correspondiente al sistema a ser desarrollado.

Todos estos aspectos tienen una estrecha y continua interrelación que determinan no sólo aspectos de la ingeniería del producto, sino también la organización, soporte y administración. En general, la organización debe primero establecer los estándares, el proceso de desarrollo y el proceso de evaluación a través de métricas bien establecidas, para luego poder lograr mejoras en los productos. La evaluación de los procesos evita especificaciones incompletas o anómalas, la aplicación incorrecta de metodologías, etc. [Jones, 1993]. Para ello se utilizan distintos modelos de madurez de procesos que tienen como objetivo apoyar distintas estrategias de desarrollo y evaluación para así lograr una mejora continua en los productos. Cabe resaltar que no se debe aplicar alguno de estos modelos de madurez bajo el supuesto de mejorar en su calidad sin antes establecer y definir los procesos correspondientes. En particular, la calidad de un sistema de software está gobernada por la calidad del proceso utilizado para desarrollarlo y mantenerlo [Humphrey, 1995].

Por lo general, la calidad en los productos se da a través del control de los procesos de producción o procesos de desarrollo. Tomando en cuenta la definición y medición de los procesos de desarrollo como paso previo hacía una mejora, revisaremos a continuación los enfoques de los modelos más conocidos y sus implicaciones.

3.2.1 CMM (Capability Maturity Model)

El modelo "clásico" en el tratamiento de la *capacidad* de los procesos de desarrollo y su madurez es CMM (*Capability Maturity Model*) del SEI (*Software Engineering Institute*). CMM tiene como objetivo evaluar los procesos en sus distintos niveles de madurez, identificar los niveles a través de los cuales una organización debe formarse para establecer una cultura de excelencia en la ingeniería de software. El *modelo de madurez de procesos* fue generado a través de la experiencia colectiva de los proyectos más exitosos de software, generando así un conjunto de prácticas importantes que deben ser implantadas por cualquier entidad que desarrolla o mantiene software.

En particular, CMM es un marco de trabajo especificando guías para organizaciones de software que quieren incrementar su capacidad de procesos, considerando los siguientes puntos:

?? Identificar fortalezas y debilidades en la organización.
?? Identificar los riesgos de seleccionar entre diferentes contratos y monitorear los mismos.
?? Entender las actividades necesarias para planear e implementar los procesos de software.
?? Ayudar a definir e implementar procesos de software en la organización a través de una guía.

Los procesos son evaluados a través de distintos niveles de madurez, que van desde prácticas desordenas o *ad-hoc*, hasta lograr una mejora continua de procesos y por ende de producto, como se muestra en la Figura 3.4.

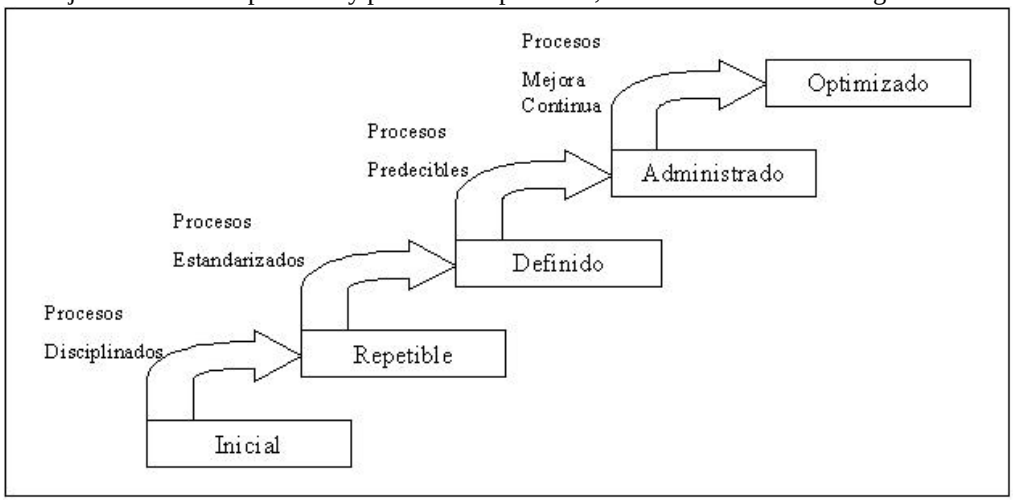

Figura 3.4. Los cinco niveles de madurez del proceso de software.

Se describe con mayor detalle los niveles de madurez en la Tabla 3.1.

	Nivel	Características	Transición al siguiente nivel
1	**Inicial**	Ad Hoc, poca formalización, junto con herramientas informalmente aplicadas al proceso.	Iniciar una administración rigurosa del proyecto, y asegurar la calidad.
2	**Repetible**	Se logró un proceso estable con un nivel repetible de control estadístico.	Establecer un grupo de proceso y una arquitectura de proceso de desarrollo de software. Introducir métodos y tecnologías de ingeniería de software.
3	**Definido**	Se logró una base para un progreso mayor y continuo.	Establecer un conjunto básico de administraciones de proceso para identificar la calidad y costo de los parámetros y una base de datos del proceso. Juntar y mantener datos del proceso. Calcular la calidad relativa de cada producto e informar a la administración.
4	**Administrado**	Mejoras sustanciables en calidad junto con medidas comprensivas del proceso.	Apoyar la recopilación automática de datos del proceso. Usar datos para analizar y modificar el proceso.
5	**Optmizado**	Mejorías en base a mayor calidad y cantidad.	Continuar mejorando y optimizando el proceso.

Tabla 3.1. Los 5 niveles del modelo de CMM.

3.2.2 ISO-9000

El ISO-9000 (International Standard Organization)

En contraste a CMM, la certificación ISO-9000 para la calidad del software fue adaptada de estándares generales y no incluye múltiples niveles, por lo cual o se está certificado o no se está. La certificación es equivalente al nivel 3 de la escala de SEI. Se calcula que hasta el año 1994, aproximadamente el 90% de las organizaciones de USA estaban por *debajo* del nivel 3.

La madurez de los procesos es determinada por las distintas *KPA* (*Key Process Area*), es decir por las áreas básicas que componen a las organizaciones y su evolución. Al definir los procesos se tiene luego la capacidad de repetirlos,

estandarizarlos en toda la organización, predecirlos para luego administrarlos (planificarlos, organizarlos, dirigirlos y controlarlos) y por último optimizarlos continuamente (hacerlos eficientes y eficaces).
A pesar de lo anterior, la implementación de los modelos no ha sido cosa fácil, por ejemplo CMM a pesar de tener casi diez años de liberado sólo cuenta con 60 empresas en todo el mundo han logrado estar en el nivel 5 optimizado a Octubre del 2001 [Software Engineering Community, 2001].

3.2.3 PSP/TSP

El PSP (Personal Software Process) es una tecnología que tiene como justificación la premisa de que la calidad de software depende del trabajo de cada uno de los ingenieros de software y de aquí que el proceso diseñado debe ayudar a controlar, manejar y mejorar el trabajo de los ingenieros [Humphrey, 1998].
El objetivo de PSP es lograr una mejor planeación del trabajo, conocer con precisión el desempeño, medir la calidad de productos y mejorar las técnicas para su desarrollo. La instrumentación de esta tecnología consiste en lo que se denomina "evolución del PSP". Se siguen ciertos pasos comenzando con las líneas base *PSP0* y *PSP0.1*, el proceso personal de planeación *PSP1* y *PSP1.1*, el manejo personal de calidad *PSP2* y *PSP2.1* y por último el proceso personal cíclico *PSP3*, como se muestra en la Figura 3.5.

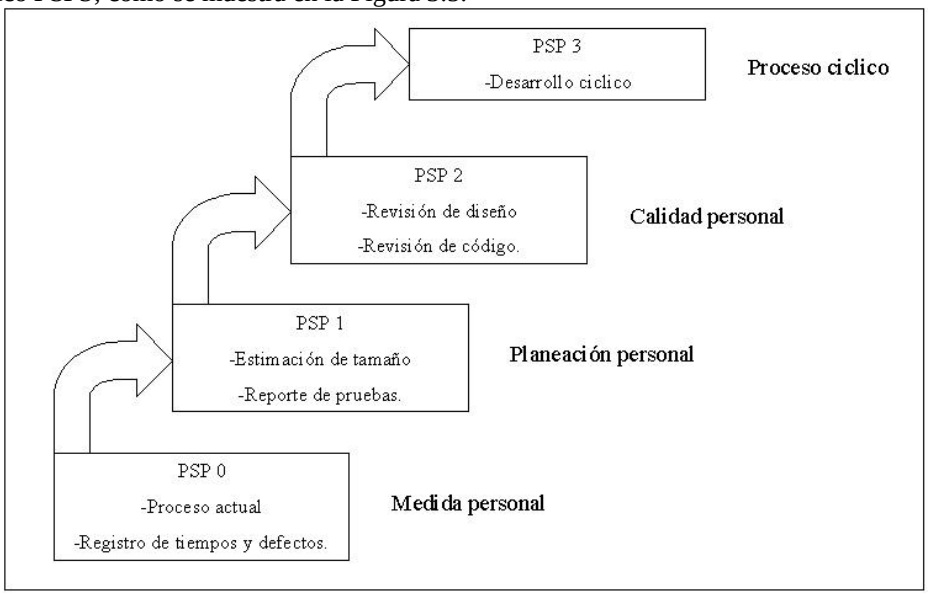

Figura 3.5. Los niveles de PSP.

?? *PSP0* (i) define el proceso de trabajo personal identificando y ordenando las principales (ii) introduce la recolección de datos para medir la productividad y calidad a través del registro de tiempo y defectos (iii) establece las bases para las mejoras en planificación de trabajo por tiempos y evaluación de resultados y (iv) documenta el proceso usando formas específicas. *PSP0.1* (i) registra el tamaño del producto a través de puntos funcionales y estandarización de la codificación y (ii) registra los problemas y propuestas de mejora.
?? *PSP1* (i) mejora la planeación introduciendo la estimación del tamaño del producto y (ii) introduce los reportes de pruebas. *PSP1.1* (i) introduce las estimaciones de recursos e (ii) introduce la calendarización.
?? *PSP2* (i) introduce las actividades de detección temprana de defectos a través de revisiones de diseño, código y uso de listas de verificación. *PSP2.1* (i) introduce formas para el diseño detallado facilitando así la revisión del diseño.
?? *PSP3* (i) introduce el proceso cíclico para desarrollar programas de mayor tamaño, (ii) introduce el registro de seguimiento de asuntos y (iii) lleva el resumen de planeación y registro de tiempo, tamaño y defectos por ciclo.

El PSP se considera la solución para pasar rápidamente entre niveles de CMM al lograr un mejor entendimiento de nuestras capacidades y habilidades y un mejor control sobre nuestro trabajo. Sin embargo, PSP tiene el problema de que es implementada a nivel individual. Al momento de la integración colectiva existen conflictos en el nivel organizativo, por lo cual se definió TSP (*Team Software Process*).
El TSP se concentra en los aspectos del desarrollo de software realizados por equipos de trabajo, definiendo aspectos como la asignación y control de tareas para los diversos miembros del equipo.

3.2.4 SPICE

SPICE (*Software Process Improvement and Capability dEtermination*) [Dorling, 1995] es un modelo de madurez de procesos internacional.

SPICE fomenta productos de calidad, promueve la optimización de procesos y facilita la evaluación del producto a través de los procesos de desarrollo. SPICE tiene diversos alcances, se aplica tanto a nivel directivo como a nivel de usuarios para asegurar que el proceso se encuentra alineado con las necesidades del negocio, apoya en que los proveedores de software tengan que someterse a una sola evaluación para aspirar a nuevos negocios y busca que las organizaciones de software dispongan de una herramienta universalmente reconocida para dar soporte a su programa de mejoramiento continuo.

SPICE tiene tres características principales: el *marco de valor* que contempla una dimensión funcional del procesos, la *evidencia* para la evaluación y la *recurrencia* dada por la selección de instancias de proyectos o productos.

SPICE está conformado por 9 documentos que permiten instrumentar paso a paso el modelo con su correspondiente evaluación, como se muestra en la Figura 3.6:

?? Información del modelo - (parte 1) conceptos y guía introductoria, (parte 4) guía para conducción de aseguramiento, (parte 6) calificación y entrenamiento de asesores, (parte 7) guía para mejora del proceso, (parte 8) guía para determinar capacidad del proceso de un proveedor y (parte 9) vocabulario general.

?? Normatividad del modelo - (parte 2) modelo de referencia de procesos y capacidad, (parte 3) realización de evaluación, (parte 5) construcción, selección y uso de aseguramiento de instrumentos y herramientas.

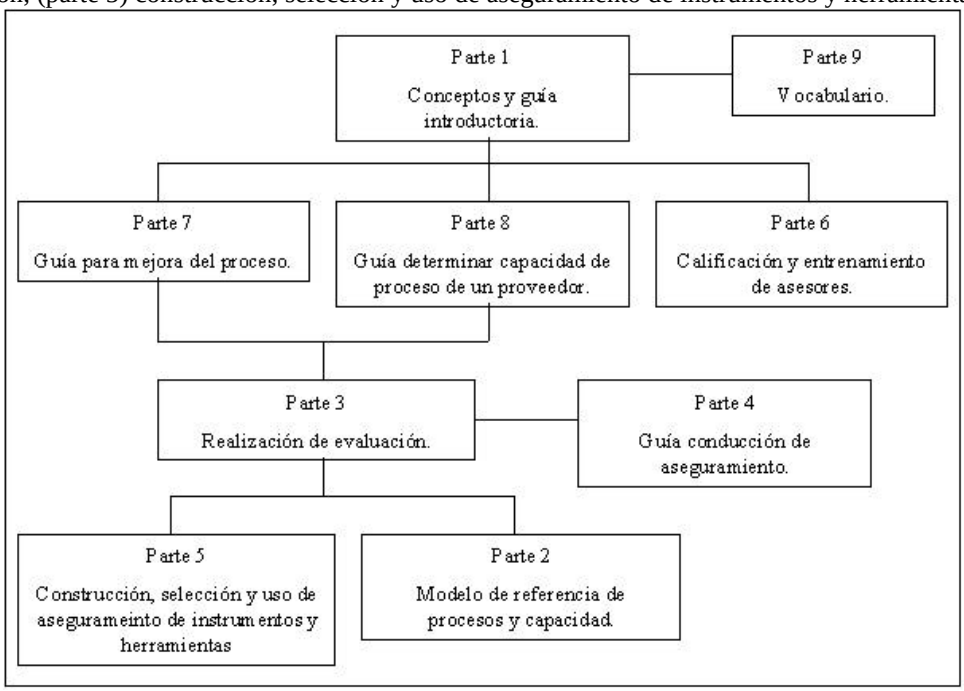

Figura 3.6. Componentes del modelo SPICE.

El modelo establece un común denominador para una evaluación uniforme de los procesos de software, aunque la evaluación no pretende ser una nueva instancia de certificación, sino que a través de los resultados se pretende demostrar lo adecuado del mismo.

Al igual que CMM (*CMMi - Capability Maturity Model Integrated*), SPICE integra una serie de niveles por la que sus procesos deberán pasar para obtener cómo resultado final la madurez. Los niveles son: Nivel 0 *Incompleto,* Nivel 1 *Fabricado informalmente*, Nivel 2 *Planeado*, Nivel 3 *Bien definido*, Nivel 4 *Controlado cuantitativamente*, y Nivel 5 *Mejora continua.*

Adicionalmente hay una definición de procesos generales que abarcan a toda la organización y a través de los cuáles se identifica el cómo lograrlos: Cliente – Proveedor *CUS*, Ingeniería *ENG*, Administración *MAN*, Apoyo o soporte *SUP* y Organización *ORG*. Los procesos generales son soportados por prácticas específicas que deberán cumplirse para lograr un paso de niveles, además de la estrecha relación entre los mismos.

SPICE hace hincapié en la calidad y actualización, así como en la vigencia del producto. Ya que la tecnología es cambiante, las fases que marca el modelo SPICE son sin duda uno de los pilares en que se tendrá que trabajar con la

mayor dedicación para obtener calidad en el producto y que el servicio del mismo sea excelente, además de generar la confianza necesaria hacia la dirección y hacia el usuario de donde se obtiene la información.

3.2.5 PEMM

PEMM (*Performance Engineering Maturity Model*) [Scholz, 1999] presenta un modelo para evaluar los niveles de integración, aplicación, ejecución y diseño, llamado ingeniería de la ejecución del modelo de madurez. Al igual que SPICE se apoya en el modelo de madurez de capacidades CMM. El objetivo de PEMM es poder evaluar la Ejecución de la Ingeniería (EI) así como la integración del proceso. El modelo sirve tanto para evaluar una organización como los propios desarrollos de procesos tecnológicos específicos. Sirve también para definir el criterio al escoger un proveedor de software para los productos críticos o semi-críticos de la compañía.
Al igual que el CMM, PENN cuenta con 5 niveles, los cuales determinan la mejora del comportamiento de ejecución y el decremento del riesgo de ejecución a través de estos niveles, como se muestra en la Figura 3.7.

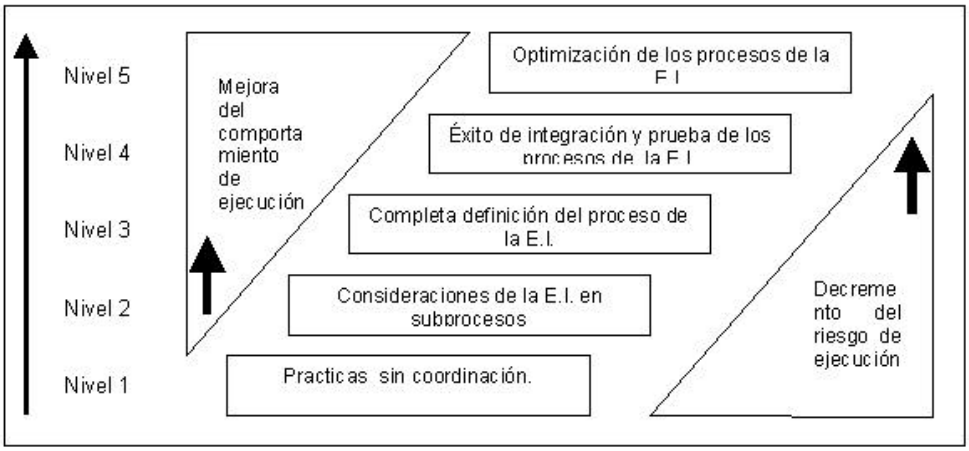

Figura 3.7. Modelo general PEMM.

La evaluación de una compañía se hace a través de la medición de aspectos generales, la organización, la definición de procesos de ingeniería, el proyecto de la dirección y la tecnología, a través de 34 preguntas, utilizando el método Meta-Pregunta-Métrico (GQM, Goal Question Metric). GQM se usa para encuestas expertas identificando y midiendo los objetivos a través de preguntas con respuestas cuantificables (en la actualidad sólo ' Si' o ' No').

3.2.6 TickIt

Tick It [Tick It, 1992], desarrollado por el Departamento de Comercio e Industria del Reino Unido, surge por la poca adopción de las normas internacionales de calidad ISO 9000 para el área de desarrollo de software. TickIt es primordialmente una guía que presenta las estrategias para lograr la certificación en la producción de software a través de la interpretación de los estándares ISO.
Los objetivos principales de TickIt son, además de desarrollar un sistema de certificación aceptable en el mercado, estimular a los desarrolladores de software a implementar sistemas de calidad, dando la dirección y guías necesarias para tal efecto. El objetivo de certificación es demostrar que las prácticas necesarias para asegurar la calidad durante el desarrollo de software existen y son verificables [Blackman, 1995]. En general el modelo permite certificar cualquier tipo de proyecto a través de una estructura más flexible.
La guía de auditoria provee la liga necesaria para que la conformación (o no) del sistema auditado respecto al modelo TickIt, pueda ser expresada en función de los criterios de la ISO 9001, logrando así la aplicación de esta última al desarrollo de software. Finalmente, los requerimientos de experiencia y conocimientos que se piden a los auditores hacen posible la aplicación del modelo, suponiendo que la experiencia y conocimientos de los auditores no se vean obsoletos frente a prácticas y técnicas nuevas dentro del medio de desarrollo de software.
Esta guía se compone de (i) un capítulo de conceptos de calidad, (ii) la norma ISO 9000-3, (iii) una serie de guías para proveedores y compradores, (iv) una guía para la auditoria del sistema de calidad, (v) el proceso de certificación y (vi) guías complementarias.

3.3 Estrategias

Como mencionamos anteriormente, existen múltiples estrategias que afectan el modelo del proceso de software. En esta sección analizaremos algunas de las más importantes como son los tipos de *tecnología*, *arquitectura*, *desarrollo*, *prototipo* y *reutilización*.

3.3.1 Tecnología

Uno de los factores más importantes es el tipo de tecnología que se utilizará. Este aspecto tiene grandes repercusiones en todo el proceso y debe escogerse con sumo cuidado. En general, la selección del tipo tecnología tiene que ver con diversos aspectos del sistema, algunos de los cuales ya se han mencionado en el Capítulo 2, como la robustez, extensibilidad, etc. Tomemos el caso de extensibilidad, donde el desarrollador debe lograr una arquitectura muy estable que minimice los efectos de cambios en el sistema. Existen por ejemplos algunas estadísticas informales (Coad y Yourdon 1991) que muestran la tendencia a cambios en varios elementos de un sistema, como se muestra en la Tabla 3.2.

Elemento	Probabilidad de cambio
Interfaces	Alto
Funcionalidad	Alto
Datos	Medio
Funciones	Medio
Objetos	Bajo
Información	Bajo

Tabla 3.2 Probabilidad de cambios futuros en el software de acuerdo al tipo de elemento de diseño.

Esta tabla resalta dos aspectos importantes en el desarrollo de software: (i) la arquitectura del sistema y el lenguaje de programación deben apoyar elementos lo más estable posible, en otras palabras elementos con menor probabilidad de cambios; y (ii) la arquitectura del sistema debe distinguir al máximo entre los distintos tipos de elementos de manera que aquellos de mayor probabilidad de cambio no “arrastren” a los más estables. En el capítulo 2 se dieron algunas razones para la utilización de tecnología orientada a objetos en lugar de la estructurada. La tabla anterior apoya lo dicho anteriormente. ¡Si el lector aún no está convencido de las razones dadas, considere que los autores anteriores han sido de los primeros en apoyar, a inicio de los 90s, el enfoque orientados a objetos para la construcción de sistemas, y el propio Yourdon fue en la década de los 80s el pionero de los enfoques estructurados! Este tema lo discutiremos con mayor detalle en la sección de metodologías. Nótese en la tabla anterior que los últimos dos elementos, interfaces y funcionalidad, son elementos de más alto nivel que objetos, información, datos y funciones, y son manejados a través de las metodologías, por lo cual éstas deben integrar también los conceptos orientados a objetos si esperamos sacarle provecho a los lenguajes de programación.

3.3.2 Arquitectura

La *arquitectura* de software se define como la estructura general del sistema incluyendo aspectos de como cambiarlo, verificarlo y mantenerlo. La arquitectura general se especializa a través de las distintas actividades del modelo de proceso hasta llegar a una arquitectura particular implementada por el código final. En el caso de desarrollo de sistemas orientados a objetos, la arquitectura general estará basada en clases y objetos.
Las arquitecturas deben especializarse de acuerdo a los diferentes tipos de sistemas. Algunos tipos de sistemas comunes son:

?? **Transformación en lote** (“batch”), son sistemas de transformación secuencial sobre un conjunto de entradas, resultando en un conjunto de salidas que se procesa sin interacción con el mundo externo. Un ejemplo de un sistema de este tipo es un compilador.

?? **Transformación contínua**, son sistemas en los cuales las salidas dependen activamente de las entradas que cambian frecuentemente y deben ser actualizadas de manera correspondiente. Ejemplos de estos sistemas son los sistemas de control de señales.

?? **Interfaz interactiva**, son sistemas regidos por interacciones externas, típicamente por un usuario. Las interfaces interactivas son por lo general parte de una aplicación de mayor alcance. Ejemplos de estos sistemas son los clásicos sistemas de ventanas como Windows u Office. Estos sistemas son controlados por manejadores de eventos, donde el manejador responde a cada evento externo, generalmente un “click” del ratón o la presión de una tecla en el teclado.

?? **Simulación dinámica**, son sistemas que simulan objetos del mundo real y que evolucionan con el tiempo. El mayor problema en la simulación es proveer un rendimiento adecuado. Idealmente un número arbitrario de procedimientos paralelos ejecuta la simulación. Ejemplos de estos sistemas son simuladores de sistemas financieros, redes neuronales [Weitzenfeld et al, 2000], sistemas eléctricos (como SPICE), etc.
?? **Sistemas de tiempo real,** son sistemas regidos por restricciones estrictas en el tiempo. Por lo general se requieren garantías en el tiempo de respuesta, siendo este tipo de sistemas de los más complejos en desarrollar. Ejemplos de estos sistemas son los controladores de procesos industriales y dispositivos de comunicación.
?? **Administración de transacción**, son sistemas que se ocupan de consultar, guardar y actualizar bases de datos y que incluyen, por lo general, acceso concurrente y distribuido para múltiples usuarios. En particular las transacciones deben ser atómicas y no deben tener interferencia con otras transacciones. Ejemplos de estos sistemas son los de reservaciones de vuelos y los de control de inventario.

3.3.3 Desarrollo

A nivel de actividades, existen dos estrategias básicas que vale la pena describir con más detalle: el desarrollo de actividades de forma iterativa y de manera incremental. Lamentablemente los términos “iterativo” e “incremental” a menudo se usan indistintamente, sin embargo, las dos son estrategias de desarrollo distintas e independientes, aunque pueden utilizarse de forma separada o en conjunto.

?? **Iterativo:** Se conoce como desarrollo iterativo el acto de revisar un resultado previo para ser luego modificado. Desarrollo iterativo apoya rehacer porciones del sistema. La idea detrás del desarrollo iterativo es revisar parte del sistema de forma controlada para remover errores o hacer mejoras basadas en la retroalimentación del usuario. Después de cada iteración, se evalúa el resultado y se planea, en la iteración siguiente, mejorar el sistema y la calidad del diseño. Aunque lo ideal es no tener que hacer dos veces lo mismo, a veces esto se justifica cuando un problema es bastante nuevo o difícil. Por ejemplo, en el caso de un marco reutilizable a menudo se requiere que este se utilice y revise varias veces antes de tener suficiente confianza en su calidad y potencial de reuso. Por otro lado, esto puede servir de excusa para hacer un trabajo incompleto, y no hacerlo bien la primera vez.
?? **Incremental:** Se conoce como desarrollo incremental el acto de incrementar o añadir desarrollo. Desarrollo incremental apoya dividir los sistemas para desarrollarse en diferentes momentos para obtener progreso en pequeños pasos. El desarrollo incremental se puede ver como una forma explícita para evitar la teoría del “Big Bang” para desarrollo de software, donde una gran explosión de desarrollo de repente se transforma de una sola vez de forma milagrosa en el sistema completo. Al contrario, el desarrollo de sistemas se considera un proceso lento, el cual puede durar varios años, dependiendo del tamaño del sistema. El enfoque incremental requiere que un problema se divida en varios subproblemas para que cada cual se desarrolle a su vez. Según se completa cada sección, se verifica e integra con las demás secciones ya completadas del sistema. En cada paso, el sistema parcialmente completado se puede evaluar en relación al desarrollo de secciones futuras. El conocimiento sobre el sistema crece progresivamente según el trabajo progresa. En la mayoría de los casos es mejor desarrollar el sistema paso a paso, comenzando con algunas de sus funciones básicas, permitiendo posteriormente añadir nueva funcionalidad. El sistema se agranda incrementalmente hasta llegar al nivel deseado, ofreciendo retroalimentación más frecuente durante el proceso de desarrollo. Esta es la idea detrás del modelo de espiral, donde cada ciclo significa un incremento en el sistema. El término “software factory” describe la división en procesos y subprocesos de las actividades del desarrollo. Para que la secuencia de las etapas de desarrollo sean exitosas, es esencial definir etapas que no requieran cambiar los resultados anteriores al introducir nuevas etapas. Por lo tanto, una comprensión inicial de los requisitos que sirven de base para el sistema completo es importante. Cada etapa se desarrolla como un ciclo y es verificada antes de completar la etapa. Es la regla más que la excepción que los requisitos de los sistemas no son totalmente conocidos al iniciarse el proyecto.

En general, el desarrollo de software es un proceso incremental y muchas iteraciones ocurrirán antes de poder completar el sistema. Estas iteraciones ocurrirán y no tiene sentido impedirlas, sino encontrar la forma de manejarlas.

3.3.4 Prototipo

Un *prototipo* es una versión preliminar, intencionalmente incompleta o reducida de un sistema. El término *prototipaje rapido* (RAD – Rapid Application Development) se refiere al proceso de construir y evaluar uno o más prototipos rápidamente. Los prototipos son estrategias aplicadas a la mayoría de actividades del proceso de software, las cuales pueden estar relacionadas con aspectos técnicos, funcionales, eficiencia o interfaces de usuario. Los prototipos rápidos permiten el desarrollo sencillo con resultados inmediatos. Ya que un prototipo se concentra en las

propiedades que requieren mayor investigación, aspectos adicionales pueden dejarse de lado, siendo mostrados únicamente de forma esquemática. El propósito de los prototipos es buscar de manera preliminar información necesaria para ayudar en la toma de decisiones.

Los prototipos complementan el desarrollo de sistemas incrementales y pueden ayudar a reducir el riesgo en la especificación de requisitos o en el diseño de la arquitectura del sistema. Una ventaja de los prototipos es que sirven como medio de comunicación entre el desarrollador y el cliente, ayudando a visualizar rápidamente la dinámica del sistema. Por lo general un prototipo no es lo suficientemente robusto para ser el producto final. Sin embargo, la existencia de la demostración apoya la creencia de que el producto completo puede desarrollarse de manera satisfactoria. Si el prototipo es diseñado con cuidado puede inclusive utilizarse en el sistema final.

A pesar de las múltiples formas de interacción con el usuario final, éstos raramente son capaces de expresar claramente lo que quieren y a menudo prefieren algo diferente de lo que reciben. Por desgracia, este es el aspecto más problemático del desarrollo de software ya que a menudo ocurre temprano durante la especificación del proyecto y se detecta muy tarde durante la entrega del proyecto. Esta situación ocurre porque los encargados de la toma de decisiones a menudo les faltan información clave.

En general, se puede pensar en un prototipo de software como un medio para la especificación de requisitos o un enlace de comunicación entre el usuario final y el diseñador. También se le puede ver como un modelo incompleto demostrando las capacidades del sistema final, que provee al usuario con una representación física de las partes más importantes del sistema antes de su implementación completa.

Los prototipos son una manera rápida para comprender los modelos de requisitos, análisis, diseño, implementación y pruebas de un sistema. Sin embargo, existe la idea errónea de que gracias a la tecnología orientada a objetos los prototipos se pueden evolucionar iterativamente hasta llegar al sistema final. Se cree que simplemente se comienza a codificar y se deja que el producto se desarrolle. Esto realmente varía y depende del tipo de prototipo que se esté desarrollando:

?? **Prototipos de requisitos**: Un prototipo de requisitos permite que los usuarios interactúen con una “interfaz” del sistema que muestre toda o casi toda la funcionalidad requerida del producto final sin que realmente se implemente. El objetivo es ayudar a clarificar los requisitos y solicitar nuevas ideas. Como las interfaces complejas son difíciles de documentar, el prototipo puede incluirse como parte de la documentación de requisitos.

?? **Prototipos de análisis**: Un prototipo de análisis permite generar una arquitectura general del sistema de manera rápida que considere las características principales del sistema, como qué componentes podrían ser utilizados. La idea es mostrar un bosquejo rápido de esta arquitectura que se apegue a los requisitos especificados originalmente.

?? **Prototipos de diseño**: Un prototipo de diseño se desarrolla para explorar y comprender la arquitectura particular del sistema. El prototipo puede servir como base para la evaluación de rendimiento y espacio, y para pruebas de redundancia o inconsistencias en el diseño. Evaluación del rendimiento de un prototipo de diseño puede identificar cuellos de botella en el sistema, dónde se requiere revisar con más detalle antes de poder desarrollar el producto final.

?? **Prototipos verticales**: Un prototipo vertical se usa para comprender una sección o parte de un problema y su solución completa. Esto se hace cuando los conceptos básicos no están bien comprendidos y cuando el desarrollar todos los aspectos de una funcionalidad muy limitada ayuda a aclararlos.

?? **Prototipos de factibilidad**: Un prototipo de factibilidad se usa para demostrar si un aspecto del proyecto es posible. Por ejemplo, si una arquitectura particular es aplicable; si la manera de conectarse a una base de datos ofrece un rendimiento adecuado; si es posible que un grupo de desarrollo aprenda a programar en cierto lenguaje; si la tecnología es apropiada para la organización; o si los costos esperados por un proyecto cubren sus gastos.

En el pasado, los modelos de desarrollo tales como el Modelo de Espiral incorporaban prototipos únicamente como una forma de reducir el riesgo en obtener una interfaz de usuario correspondiente a las necesidades del cliente. Estos prototipos eran secuencias de pantallas mostrando como el usuario vería el producto, dando la ilusión de un sistema funcional el cual podía comentarse con los clientes. Esto es apropiado para lograr una mejor retroalimentación con los clientes.

Sin embargo, la implementación de un prototipo no es necesariamente un producto de calidad, diseñado para ser mantenido a largo plazo. Por el contrario, el prototipo es a menudo pegado rápidamente, para ser probado y luego reescrito. Como tal, la implementación debe tirarse.

Sin embargo, no se puede considerar sólo los extremos; guardar o tirar los prototipos. Por lo general, partes de un prototipo se guardan cuando se crea el siguiente prototipo o versión del producto. En especial el prototipo de diseño

tiene una mayor probabilidad de evolucionar en el software final que otros tipos de prototipos. Es probable, por ejemplo, que el ambiente donde se estudia el diseño sea el mismo a utilizarse en el desarrollo del producto. También un prototipo de análisis pueda evolucionar en el producto final, como en el caso de una biblioteca de componentes de software con interfaces consistentes entre sí, como es el caso de los "JavaBeans". Este también es un ejemplo donde los prototipos pueden hacer uso de los componentes reusables.
Asumiendo que los componentes reusables sean de calidad, es probable que un producto completo pueda evolucionar del prototipo inicial modificando su implementación en respuesta a la retroalimentación del usuario y otros aspectos de diseño. Sin embargo, siempre existirá un conflicto entre un desarrollo rápido y un producto de calidad. Existe también una tendencia que para tratar de ganarle el mercado a los competidores, algunos administradores tratan de enviar el prototipo como si fuera ya el producto final.
La siguiente lista muestra algunas de las razones para crear prototipos, categorizadas según diferentes metas (Goldeberg y Rubin 1995).

?? **Asegurarse que el sistema hace lo deseado**: Probar conceptos, obtener definición de producto, determinar funcionalidad completa, diseñar interfaces de usuario, diseñar subsistemas, asegurar la autorización de los cliente para minimizar cambios y confusión después de la entrega y proveer un contexto de discusión cuando los interesados tienen diferentes perfiles o hablan diferentes lenguajes.
?? **Asegurarse que el sistema hace correctamente lo deseado**: Determinar el modelo de comportamiento del sistema y probar algoritmos alternos.
?? **Mejorar la implementación del sistema actual o futuro**: Ayudar a identificar clases para reuso, diseñar marcos para aplicaciones y ayudar a estimar tiempo de construcción.
?? **Mejorar procesos y recursos**: Crear prueba de concepto para obtener un contrato, probar nuevas herramientas (lenguajes, ambientes de desarrollo), determinar si una tecnología trabaja en un ambiente particular, aprender una tecnología; un lenguaje, un conjunto de herramientas, un conjunto de técnicas, ganar una ventaja de negocio antes de que el sistema final se entregue ofreciendo a los clientes y la prensa una demostración temprana del producto y entrenar temprano el grupo de soporte técnico y ventas.

Existen diversos aspectos de los cuales depende el éxito del prototipo (Goldberg y Rubin 1995):

?? Comprender el propósito del prototipo y usarlo de manera adecuada.
?? Comprender la tecnología a utilizarse y su relación con el proceso de prototipos.
?? Juntar un grupo técnico apropiado para hacer el prototipo: líder de proyecto, documentador, elaborador de prototipos de requisitos y análisis, y otro de diseño.
?? Evaluar al grupo y las entregas finales.
?? Involucrar temprano en el proceso a los usuarios finales.
?? Estar dispuestos a repetir el proceso de prototipos para comprender mejor la arquitectura básica.
?? Imponer criterios de evaluación apropiados al comienzo de cada etapa de prototipos, y basarse firmemente en estos criterios para su terminación.
?? Construir prototipos basados en una biblioteca de código reusable, controlada por el bibliotecario asignado.

Los prototipos fallan cuando (Goldberg y Rubin 1995):

?? No se comprende qué es un prototipo y cómo debe usarse.
?? No se comprende el proceso lo suficientemente bien como para organizar al grupo correctamente.
?? No se sabe cuando dejar de evolucionar el prototipo y comenzar de cero, o sea, se extiende demasiado el proceso.
?? No se sabe cuando continuar para tratar de lograr los criterios de evaluación deseados, o sea, se termina prematuramente el proceso.
?? No se utiliza ambientes o herramientas de apoyo para el desarrollo orientado a objetos, solamente un lenguaje orientado a objetos.
?? Se cree que un prototipo razonable es un producto aceptable.
?? Los prototipos nunca terminan.

3.3.5 Reutilización

En general la reutilización del código se puede hacer dentro de un mismo proyecto o entre proyectos. Reutilización de código dentro de un mismo proyecto involucra simplemente descubrir secuencias de código redundantes en el diseño y usar las facilidades del lenguaje de programación, como procedimientos o herencia. Este tipo de reuso de código es siempre bueno, produciendo programas más pequeños y resultando en correcciones más rápidas. Reuso entre proyectos requiere planeación y representa una inversión. No es muy probable que una clase aislada sirva para

múltiples proyectos pero sí estructuras bien analizadas como tipos de datos abstractos, paquetes gráficos, y bibliotecas de análisis numérico. El consumo de componentes reutilizables es una muy buena estrategia para minimizar el esfuerzo en completar una tarea. Sin embargo, la estrategia tiene que ser complementada por un esfuerzo en producir los componentes reusables. El modelo de proceso de productor/consumidor de software, es un modelo de proceso que mezcla proyectos que producen componentes reusables con aquellos que consumen componentes reusables.

La decisión para reutilizar componentes se basa en la evaluación de costos entre crear una nueva solución o adaptar una existente. Asimismo, una organización debe considerar los costos adicionales en la producción del componente reutilizable planificando la reducción de costos posterior al utilizar los componentes en otros proyectos. Una organización puede administrar costos y hacer del reuso una parte efectiva de su modelo de proceso.

?? **Consumiendo Componentes Reusables.** Cuando se requiere una solución a un problema la primera pregunta que se hace el consumidor es si ya hay una solución disponible. Se debe aprovechar toda solución completa o parcial existente ya que las ventajas incluyen soluciones consistentes entre aplicaciones, además que las mejoras a la solución se propagan a todas las aplicaciones, mejorando la calidad del componente al ser probado por múltiples aplicaciones y sirve para ver si alguna de estas partes ya ha sido resuelta anteriormente. Obviamente para lograr un buen reuso la solución debe adaptarse a los objetivos de la tarea donde el reuso fue considerado. Por lo general, los desarrolladores solamente reusan soluciones existentes donde se confía con el nivel de calidad requerido. El reuso puede ocurrir durante las diversas actividades. Por ejemplo, a nivel de requisitos pueda que ya se haya resuelto el mismo problema anteriormente, y de manera similar durante las demás actividades, incluso a nivel de documentación como en la utilización de ejemplos similares. Las oportunidades ocurren en todo el ciclo de vida de un proyecto.

?? **Produciendo Componentes Reusables.** Crear componentes reusables es una meta importante que cambia la visión del software, de un problema a una ventaja. Los productores de reuso crean resultados que no se evalúan como funcionalidad final independiente y completa, sino como recursos que contribuirán a otros proyectos, por lo tanto, se debe tener una perspectiva de múltiples proyectos. Además, los productores de reuso llevan a cabo tareas especiales para incrementar el potencial de reuso, como el análisis variante, que ayuda a identificar y darle prioridad a posibles variantes de los componentes (Barnes y Bollinger 1991). Otras tareas del productor de reuso incluyen crear documentación orientada a reuso y aplicar evaluaciones especiales de reuso. La documentación debe expresar suposiciones y limitaciones del componente, y describir cómo extender o refinar el componente. La evaluación de reuso se beneficia del análisis variante para identificar los casos de pruebas. Para un marco, se pide una descripción de la categoría de la aplicación que se pueda derivar del marco, y se prueba para ver si la aplicación de este tipo se puede realmente derivar. Para componentes, se pregunta si las descripciones son suficientemente generales para poder participar en diversas aplicaciones de interés para la organización. En la práctica, productores y consumidores de reuso trabajan de forma concurrente. La producción de marcos y componentes reusables, junto con la producción de sistemas, son procesos ligados. Los productores necesitan ver lo que han hecho los consumidores para decidir que es útil. Y los consumidores necesitan probar software de los productores para proveer retroalimentación necesaria para decidir si los resultados son realmente generales y suficientemente valiosos.

A continuación se muestra un resumen (Golberg and Rubin 1995) sobre las razones para utilizar reuso. La mayoría de las preocupaciones con reuso se relacionan con aspectos de tiempo y calidad.

El reuso es valioso porque:

?? Lo común es más fácil de apoyar.
?? Se promueve consistencia externa al incorporar estándares.
?? El reuso incrementa la habilidad para discutir problemas entre diversos grupos.
?? El reuso reduce costos.
?? El reuso hace más fácil comenzar el desarrollo, incluso cuando el componente reusable no se ajusta perfectamente a las nuevas necesidades.
?? El reuso acelera el tiempo de entrega.
?? El reuso de resultados ya probados mejora la calidad del producto.

El reuso no es valioso porque:

?? Los componentes generalizados pueden que no se ajusten a los requisitos de rendimiento.
?? El tiempo de aprendizaje de nuevos componentes puede que no se ajuste a los tiempos del proyecto.
?? Los estándares pueden ser limitantes.

Debe hacerse reuso cuando:

?? Se necesita conservar recursos.

?? Se incrementa la base de funcionalidad.
?? Haciéndolo provee una guía para el diseño.
?? Los componentes reusables mejoran la calidad del producto.
?? Conociendo partes del sistema permite predecir tiempo de desarrollo.
?? Reusar es más rápido.
?? Las suposiciones actuales corresponden a las suposiciones del componente.
?? Los componentes han sido probados operando correctamente en otras situaciones.

No debe hacerse reuso cuando:

?? Los objetivos de diseño no se satisfacen al hacerlo.
?? Se tiene que forzar un ajuste haciendo demasiados cambios.
?? Los componentes reusables proveen aspectos que hacen que el resultado sea inapropiado para el mercado al que va dirigido.
?? Los componentes reusables proveen exceso de funcionalidad.
?? La interfaz del componentes reusable no está bien entendida por los miembros del proyecto.
?? Los componentes reusables son propiedad de un proyecto específico.
?? Los componentes de reuso incluyen demasiada funcionalidad no requerida, aumentando el tamaño del sistema y el esfuerzo de desarrollo.
?? El reuso no disminuye la cantidad de código a ser probada.

3.4 Actividades

La Tabla 3.3 muestra las actividades más importantes para el ciclo de vida del desarrollo de software. Estas actividades corresponden a las mostradas en la Figura 3.1 en el Modelo de Cascada y corresponden de manera similar a las actividades del Modelo de Espiral. La diferencia entre ellas radica en el proceso y orden para llevarlas a cabo junto con las estrategias y métodos utilizados para cada actividad.

Actividad	Descripción
Requisitos	Se especifica las necesidades del sistema a desarrollarse. La especificación de requisitos puede servir como base para la negociación entre los desarrolladores y clientes del sistema y también para planear y controlar el proceso de desarrollo.
Análisis	Se busca comprender los requisitos del sistema logrando la estructuración de una solución, correspondiente a la arquitectura general. Se contesta la pregunta del “qué” del sistema.
Diseño	Se transforma la arquitectura general de análisis, a una arquitectura particular y detallada del sistema que satisfaga todos los requisitos del sistema, donde las condiciones idealizadas durante el análisis se reemplazan por requisitos del ambiente de implantación particular. Se contesta la pregunta del “cómo” del sistema.
Implementación	Se expresa la arquitectura particular del sistema, en una forma aceptable para la computadora, o sea el código.
Pruebas	Se verifica y valida el sistema a nivel de componentes y la integración de ellos. Este es uno de los aspectos más críticos del desarrollo y debe ser aplicado desde el inicio, durante todas las actividades. De tal manera se busca descubrir cualquier defecto en los requisitos, análisis, diseño, implementación e integración. Las pruebas se hacen a varios niveles, desde funciones sencillas hasta el sistema completo.
Integración	Se combinan todos los componentes creados de manera independiente para formar el sistema completo.
Documentación	Se describen los aspectos sobresalientes de los requisitos, análisis, diseño, implementación, integración y pruebas. Esto servirá para usuarios externos e internos, aquellos encargados en mantener el sistema y extenderlo.
Mantenimiento	Se corrigen errores no encontrados durante el desarrollo y pruebas originales del sistema. Se extiende el sistema según existan nuevas necesidades.

Tabla 3.3 Actividades del desarrollo de software.

La transición entre las distintas actividades debe ser natural, debiendo existir una continuidad o rastreabilidad (“traceability”) de una actividad a la siguiente o la anterior. A continuación describimos con mayor detalle cada una de estas actividades.

3.4.1 Requisitos

La actividad o modelo de requisitos tiene como meta definir y delimitar la funcionalidad del sistema de software. El modelo de requisitos puede servir como base de negociación y contrato entre el desarrollador del sistema y el cliente, y por lo tanto debe reflejar los deseos del cliente. Es esencial que los clientes que no tengan un conocimiento de la computación puedan comprender el modelo de requisitos para facilitar la interacción con ellos.
El modelo de requisitos gobierna el desarrollo de todos los demás modelos, siendo central durante el desarrollo del sistema completo. El modelo de requisitos se estructura mediante el modelo de análisis, se realiza mediante el modelo de diseño, se implementa mediante el modelo de implementación y se prueba mediante el modelo de pruebas. Además, todos los demás modelos deben verificarse contra el modelo de requisitos. El modelo de requisitos también sirve como base para el desarrollo de las instrucciones operacionales y los manuales, los cuales son descritos desde el punto de vista del usuario.
El desafío de la especificación de requisitos comienza cuando el experto debe comunicar los conceptos, lo cual no es generalmente posible de hacer adecuadamente por medio de una simple expresión. Como resultado, se provee explicaciones múltiples, verbales o escritas. El desarrollador pide y captura estas explicaciones y las integra en una representación coherente. La especificación de requisitos es particularmente difícil cuando la información es incompleta, los expertos no pueden articular lo que saben, o no están seguros o incluso son incoherentes sobre su conocimiento o creencias. Una meta importante es minimizar las diferencias entre los espacios de concepto y el modelo de requisitos. Si la distancia entre el modelo y la comprensión del experto es grande, será bastante difícil, sino imposible para el experto verificar la precisión. Consecuentemente, una de las necesidades principales de cualquier modelo de requisitos es que sea comprensible para cualquier persona.

3.4.2 Análisis

Después del desarrollo del modelo de requisitos y de haber sido éste aprobado por parte de los usuarios del sistema o clientes, se puede iniciar realmente a desarrollar el sistema. Esto comienza con el desarrollo del modelo de análisis que toma como punto de partida la especificación de requisitos y tiene como meta construir una arquitectura capaz de resolver el problema bajo condiciones ideales. Esto significa que se busca desarrollar una estructura lógica del sistema, la cual debe ser estable, robusta, mantenible y extensible. El análisis se enfoca en qué debe hacer el sistema, en lugar de cómo se supone que lo hará. El alcance del modelo de análisis está directamente relacionado con la naturaleza de los conceptos del modelo. En el caso de la tecnología orientada a objetos, se desea: encontrar los objetos, organizar los objetos, describir cómo los objetos interactúan, definir las operaciones de los objetos y definir los objetos internamente.

3.4.3 Diseño

El propósito del modelo de diseño es extender la arquitectura general de análisis. Este refinamiento se debe a dos razones principales:

?? El modelo de análisis no es suficientemente formal por lo cual para poder llegar al código final se debe refinar las estructuras de la arquitectura general. Se debe especificar las operaciones que deben utilizarse, la comunicación entre componentes, los eventos, etc. Este aspecto es conocido como el *diseño de estructuras* o de manera general como el *diseño de objetos* en el caso de arquitecturas orientadas a objetos.

?? Durante el análisis se asume un mundo ideal para el sistema. En la realidad este mundo ideal debe adaptarse al ambiente donde se implementará el sistema. Entre otros aspectos, se debe considerar los requisitos de rendimiento, aspectos de tiempo real, concurrencia, propiedades del lenguaje de programación, el sistema de manejo de base de datos, etc. Este aspecto es conocido como el *diseño de sistema*.

La razón para no incluir estos aspectos durante el modelo de análisis se debe a que los aspectos anteriores deben influenciar la arquitectura del sistema lo menos posible. En general, la propia aplicación controla la arquitectura y no las circunstancias existentes durante su implementación. Desde otra perspectiva, el modelo de análisis debe ser visto como un modelo conceptual y lógico del sistema, mientras que el modelo de diseño debe acercarse más al código final. Esto significa que se cambia el punto del vista del modelo de diseño a una abstracción del código fuente a ser escrito. Es esencial guardar y congelar el modelo de análisis para un mantenimiento futuro incluso después de terminar el diseño.
En general, se debe comenzar el diseño temprano, preferiblemente al mismo tiempo que se comienza con el modelo de análisis. El primer paso según se comienza a trabajar es identificar el ambiente de implementación y esto se puede hacer en paralelo con el análisis para que esté listo cuando el diseño actual comience. Si se ha hecho un modelo de análisis muy detallado, el grado de refinamiento necesario durante el diseño puede ser muy pequeño. La

decisión de la transición de análisis a diseño depende de cada proyecto, siendo importante decidir esto temprano, lo cual debe basarse en el resultado de la identificación del ambiente de implementación.

?? **Diseño de objetos.** El diseño de objetos consiste de decisiones tácticas, tales como la selección de algoritmos y estructura de datos para satisfacer los objetivos de rendimiento y espacio. El modelo de análisis y el diseño de objetos tienen bastante en común, incluyendo los mismos conceptos, técnicas y notaciones. Como consecuencia, las mismas herramientas de desarrollo pueden utilizarse para llevar a cabo ambas actividades. A menudo, estas similitudes hacen difícil saber que actividad se está llevando a cabo. Uno de los beneficios más importantes de la tecnología orientada a objetos es la representación de la solución como una consecuencia directa de la representación del problema, por lo cual la distinción entre análisis y diseño de objetos no es realmente crítica, a diferencia de otros enfoques más tradicionales. El diseño de objetos especializa la arquitectura general mediante *subsistemas* que agrupan funcionalidad o estructuras comunes, que constan de interfaces bien definidas con otros subsistemas, usualmente identificados por los *servicios* que proporcionan. Los subsistemas pueden definirse en c*apas* correspondientes a un conjunto de subsistemas horizontales construido en término de subsistemas o capas inferiores que pueden ser *cerradas* o *abiertas*. A diferencia de las capas que dividen un sistema de manera horizontal, las *particiones* dividen verticalmente al sistema. Estas divisiones son débilmente conectadas, cada una ofreciendo otro tipo de servicios. Un sistema puede ser descompuesto usando capas y particiones a la vez, donde las capas pueden ser divididas en particiones y las particiones en capas. El diseño de objetos debe definir el *manejo de control*, como son los *sistemas impulsados por procedimientos* y *sistemas impulsados por eventos*. En los sistemas impulsados por procedimientos, el control es mediante procedimientos, mientras que en los sistemas impulsados por eventos, el control reside en un despachador o monitor provisto por el lenguaje, subsistema, o sistema operativo, siendo más apropiado para sistemas interactivos en especial aquellos controlados por el ratón. Otros aspectos que afectan el diseño de objetos son los *componentes* o *bibliotecas* y *herramientas*, donde los componentes permiten construir un sistema mediante estructuras prefabricadas de más alto nivel que lo ofrecido por el lenguaje de programación, mientras que las herramientas son esenciales en la administración de los sistemas. Estas herramientas incluyen ambientes de desarrollo para la escritura y configuración del código, como lo son compiladores, depuradores, preprocesadores y demás.

?? **Diseño de Sistema.** El diseño de sistema define las decisiones estratégicas sobre como se organiza la funcionalidad del sistema en torno al ambiente de implementación. El ambiente de implementación se divide en el ambiente de hardware y el ambiente de software, los cuales están muy ligados entre sí. El ambiente de hardware afecta al ambiente de software, en especial si el software depende de la plataforma. Es importante restringir el efecto del ambiente de implementación sobre la arquitectura de análisis. Para adaptar el modelo de diseño al ambiente de implementación, se debe identificar las restricciones técnicas bajo las cuales el sistema debe ser construido. Esta identificación debe ser hecha temprano, idealmente durante el modelo de requisitos. Aunque no todas las decisiones son hechas durante el diseño de sistema, las prioridades para hacerlas sí deben ser establecidas anteriormente. Existen razones importantes para introducir el ambiente de implementación temprano. No se quiere que el ambiente de implementación afecte la estructura básica del sistema, ya que las circunstancias actuales probablemente serán cambiadas de una manera u otra durante el ciclo de vida del sistema. No se quiere que el problema se complique aún más por la complejidad introducida a través del ambiente de implementación. De esta manera es posible enfocar lo esencial cuando se desarrollan los aspectos importantes del sistema, o sea, su estructura básica. Entre otras cosas, se identifica la concurrencia en el sistema y se deciden las prioridades durante el diseño, incluyendo rendimiento, memoria, protocolos de comunicación, flexibilidad y extensibilidad. Los subsistemas se asignan a los procesadores y tareas según la arquitectura propuesta, se escoge el manejo de almacenamientos de datos, se escogen los mecanismos para coordinar el acceso a recursos globales, se escoge la implementación del control del software, se escoge el enfoque para el manejo de condiciones de borde, incluyendo manejo de errores. Las decisiones más importantes en relación al ambiente de implementación tiene que ver con el *rendimiento y memoria*, *concurrencia*, *manejo de procesos*, *manejo de almacenamiento de datos* y *manejo de recursos globales*. Los requisitos de rendimiento y uso de memoria tienen un gran efecto sobre la arquitectura del sistema. Por ejemplo, los primeros juegos de video corrían en un procesador con memoria limitada, donde conservar memoria era la prioridad máxima, seguido por ejecución rápida. Actualmente, la prioridad es de mayor rapidez de ejecución sin importar el espacio de memoria necesario. Es bastante común que un buen diseño se arruine por malos rendimientos. Una meta importante del diseño de sistema es identificar cuales componentes deben estar activos concurrentemente y cuales tienen actividades secuenciales. Cada subsistema concurrente debe considerar el manejo de procesos para lograr un mejor rendimiento del sistema, donde se deben seguir los siguientes pasos: (i) estimar los requisitos de recursos; (ii) balancear entre hardware y software; (iii) asignar las tareas a los procesadores; y (iv) determinar la

conectividad física. El manejo de almacenamiento de datos debe considerar aspectos interno como la memoria y externos como los discos. Diferentes tipos de almacenamiento proporcionan diferentes costos, capacidad, y tiempo de acceso. Los archivos son simples pero sus operaciones son de bajo nivel, mientras que las bases de datos organizan el acceso a datos de forma más eficiente, aunque sus interfaces son complejas y muchas de ellas no se integran bien con los lenguajes de programación. El diseñador del sistema debe manejar los recursos globales incluyendo el acceso a unidades físicas como procesadores, controladores, o espacio de disco, o nombres lógicos, como archivos o clases. Más allá de esto, el diseño de sistema debe apoyar aspectos como una terminación inesperada del sistema que puede ocurrir por errores del usuario, agotamiento de recursos, o por fallas externas como de hardware. Un buen diseño considera posibles fallas, incluyendo errores del programa, y debería imprimir o guardar la máxima información sobre el error cuando este ocurra. Estos aspectos son apoyados de manera variada por los distintos lenguajes de programación

3.4.4 Implementación

El modelo de implementación toma el resultado del modelo de diseño para generar el código fuente anotado. Esta traducción debe ser relativamente sencilla y directa, ya que todas las decisiones han sido hechas en las etapas previas. La especialización al lenguaje de programación o base de datos describe cómo traducir los términos usados en el diseño a los términos y propiedades del lenguaje de implementación. Aunque el diseño de objetos es bastante independiente del lenguaje actual, todos los lenguajes tendrán sus especialidades durante la implementación final incluyendo las bases de datos.

Existen ciertamente sistemas que automáticamente traducen descripciones SDL (Specification and Description Language, CCITT [1988?) a código fuente, pero esto requiere que los grafos SDL se extiendan con formalismos similares a los lenguajes de programación. Sin embargo, en su gran mayoría los programadores hacen de manera "manual" la transición final a código fuente. En el modelo de implementación, el concepto de rastreabilidad es también muy importante, dado que al leer el código fuente se debe poder rastrear directamente del modelo de diseño y análisis.

?? **Lenguajes de Programación.** Un aspecto importante durante el diseño de objetos es la selección del lenguaje de programación. El lenguaje de programación no tiene que ser necesariamente orientado a objetos. El uso de un lenguaje de programación orientado a objetos hace más fácil la implementación de un diseño orientado a objetos. La elección del lenguaje influye en el diseño, pero el diseño no debe depender de los detalles del lenguaje. Si se cambia de lenguaje de programación no debe requerirse el re-diseño del sistema. Los lenguajes de programación y sistemas operativos difieren mucho en su organización, aunque la mayoría de los lenguajes tienen la habilidad de expresar los aspectos de la especificación de software que son las estructuras de datos (objetos), flujo dinámico de control secuencial (ciclos, condiciones) o declarativo (reglas, tablas) además de contar con transformaciones funcionales. En el caso de un lenguaje orientado a objetos la estructura básica son las propias clases. Aunque no todos las características mínimas de la orientación a objetos (ver capítulo 2) sean ofrecidas, un lenguaje de programación orientado a objetos implementará de mejor manera todos los conceptos anteriores. En especial, es deseable tener una buena y fácil correspondencia entre un objeto en el modelo de objetos con estructuras en el lenguaje de programación. Dependiendo del lenguaje de programación esto puede hacerse más sencillo o complicado. Los lenguajes más utilizados varían desde los "semi" orientados a objetos como Ada y Modula-2 hasta los estructurados tradicionales, como C, Pascal, Fortran y COBOL. En general, el aspecto más difícil de implementar con estos lenguajes es la *herencia*. Estos temas serán tratados con mayor detalle en el Capítulo de Implementación.

?? **Bases de Datos.** Las bases de datos son parte integral de los sistemas de software, en especial de los sistemas de información. En general, se pueden utilizar bases de datos orientadas a objetos u otros tipos de bases de datos, como las relacionales. Si los aspectos dinámicos y funcionales del sistema son menores en relación a las estructuras del sistema, una base de datos relacional puede que sea suficiente ya que éstas se dedican a almacenar principalmente los aspectos estructurales del sistema. Por otro lado, las bases de datos orientadas a objetos van más allá de las estructuras, guardando aspectos funcionales del sistema además de integrarse de mejor forma con una arquitectura basada en tecnología orientada a objetos. En el Capítulo de Implementación revisaremos estos aspectos.

3.4.5 Integración

El modelo de integración es un aspecto muy importante del proceso de desarrollo. Una característica primordial en todo sistema es mantener la modularidad en los subsistemas, esto significa que inicialmente los subsistemas se desarrollan de manera independiente, llegando el momento para su integración final. Este enfoque maneja de mejor

forma la complejidad del sistema y significa que también deberán hacerse pruebas, primero en los componentes por separado y luego en su totalidad como se describe en la siguiente sección.

3.4.6 Pruebas

El modelo de pruebas es quizás el responsable de revisar la calidad del sistema siendo desarrollado. Los aspectos fundamentales de este modelo son básicamente la *prueba de especificación* y la *prueba de resultado*. Probar un sistema es relativamente independiente del método utilizado para desarrollarlo. Las pruebas comienzan con los niveles más bajos, como son los módulos de objetos, hasta llegar a la *prueba de integración* donde se van integrando partes cada vez más grandes. Una herramienta para pruebas de integración involucra usar el modelo de requisitos para integrar requisitos de manera incremental. En particular, las actividades de pruebas normalmente se dividen en *verificación* y *validación*.

?? **Verificación:** Verificación prueba si los resultados están conformes a la especificación, en otras palabras si se está construyendo el sistema correctamente. La verificación debe comenzar lo antes posible, desde el nivel más bajo mediante la verificación de subsistemas, progresando hacia la verificación de la integración, donde las unidades se verifican juntas para ver si interactúan de forma correcta. Finalmente se verifica el sistema completo. La etapa de verificación en la orientación a objetos es menos dramática que en los sistemas que separan funciones de datos, ya que los objetos son unidades más grandes y durante su diseño las unidades ya se están verificando. Por otro lado, herencia al igual que polimorfismo pueden dificultar la verificación, ya que las operaciones varían según los ancestros o descendientes y los datos de verificación deben ser elegidos cuidadosamente. Incluso la especificación de verificación puede considerarse como una extensión al modelo de requisitos y ser integrada en la arquitectura del sistema. La especificación de verificación debe aplicarse a todos los modelos descritos.

?? **Validación:** Validación prueba si los resultados corresponden a lo que el cliente realmente quería. Este enfoque en la satisfacción del cliente se concentra en obtener la especificación y el resultado correcto. Se hace la pregunta de si se está construyendo el sistema “correcto”, en contraste a la verificación donde se pregunta si se está haciendo el sistema “correctamente”. La validación se captura por medio de análisis extensivo del modelo de requisitos incluyendo interacción constante con los clientes mediante, uso de prototipos, etc. Se debe validar los resultados del análisis. Según el sistema crece y se formaliza, se estudia qué tan bien el modelo de análisis y el modelo de requisitos describen al sistema. Durante el diseño, se puede ya ir viendo si los resultados del análisis son apropiados para el diseño. Si se encuentran puntos que no están claros en el modelo de análisis o en el modelo de requisitos, se les debe clarificar, quizás regresando a la actividad de análisis nuevamente.

3.4.7 Documentación

De manera similar a las pruebas la documentación debe ocurrir a lo largo del desarrollo del sistema y no como una etapa final del mismo. Existen diferentes tipos de documentos que deben ser generados como apoyo al sistema. Cada uno de estos documentos tiene diferentes objetivos y está dirigidos a distintos tipos de personas, desde los usuarios no técnicos hasta los desarrolladores más técnicos. Los siguientes son algunos de los documentos o manuales más importantes:

?? Manual del Usuario, que le permite a un usuario comprender como utilizar el sistema.
?? Manual del Programador, que le permite a un desarrollador entender los aspectos de diseño considerados durante su implementación.
?? Manual del Operador, que le permite al encargado de operar el sistema comprender que pasos debe llevar a cabo para que el sistema funcione bajo cierta configuración de ambiente particular.
?? Manual del Administrador, que le permite al encargado de administrar el sistema comprender aspectos más generales como son los modelos de requisitos y análisis.

Realmente no hay límite al número y detalle que se puede lograr mediante la documentación, de manera similar a que no hay límite a que tanto se puede extender y optimizar un sistema. La idea básica es mantener un nivel de documentación que sea útil aunque es necesario adaptarlo al proceso de la organización.

3.4.8 Mantenimiento

Una visión equivocada del mantenimiento de un sistema es que esto involucra únicamente la corrección de errores. El mantenimiento realmente va más allá de corregir problemas y debe basarse principalmente en considerar las extensiones al sistema según nuevas necesidades. En otras palabras, se basa en generar nuevos desarrollos pero tomando como punto de partida el sistema ya existente. En cierta manera es regresar al resto de las actividades pero

sin partir de cero. Como se mencionó en la sección 3.1, el proceso de mantenimiento tendrá que adaptarse a las nuevas circunstancias del desarrollo.

3.5 Métodos y Metodologías

Los métodos definen las reglas para las distintas transformaciones dentro de las actividades. Las metodologías definen el conjunto de métodos. La selección de las metodologías a utilizarse es otra de las decisiones críticas en el proceso de software. Hasta hace poco, los métodos para análisis y diseño eran muy similares y las herramientas de software como apoyo a los métodos no eran más que herramientas de dibujo asistidas por computadora. En los últimos años, los desarrolladores de software se han hecho más sofisticados en su comprensión de los beneficios de métodos completos y comprensibles, dando mayor énfasis a las metodologías y notaciones correspondientes.
Dado que se ha escogido utilizar tecnología orientada a objetos, los únicos métodos de análisis y diseño de los cuales se debe escoger son aquellos que se basan en conceptos orientados a objetos. Los métodos deben proveer técnicas para crear modelos estáticos y dinámicos del sistema. El modelo estático incluye descripciones de los objetos que existen en el sistema, su relación mutua, y las operaciones que pueden ejecutarse en el sistema. El modelo dinámico incluye la secuencia aceptada de operaciones sobre el sistema, al igual que la secuencia de mensajes entre objetos necesaria para ejecutar las operaciones del sistema.
En general, los métodos difieren en los diversos aspectos que apoyan, tales como el tipo de información recopilada, sus requisitos de consistencia, el dominio de aplicabilidad, modelo de proceso, modelos generados y notaciones.

?? **Tipo de Información Recopilada:** Los métodos de análisis y diseño que se consideran buenos deben proveer un conjunto de técnicas para recopilar información. Estas técnicas se usan para crear una descripción completa del dominio del problema y los medios para crear una solución que satisfaga los objetivos de calidad del sistema. Si la meta del proyecto es crear componentes reusables, entonces una consideración en la selección del método de análisis y diseño es si el método tiene técnicas para desarrollar resultados reusables.

?? **Requisitos de Consistencia:** Consistencia es un atributo de un modelo donde todos los componentes son precisos y relacionados apropiadamente, lo que sirve de base para la integridad del modelo. Los mejores métodos evitan los errores de consistencia e integridad, mientras que métodos aceptables tienen por lo menos técnicas para detectar si existen violaciones. Los métodos deben tener herramientas que apoyen la verificación de los modelos. Este requisito significa que las herramientas sencillas que apoyan sólo diagramas en base a cierta notación no son de interés ya que carecen de manejo de consistencia. Los métodos deben indicar claramente donde falta información y cuando ésta no es crítica para continuar con la siguiente actividad. Además, las herramientas asociadas con el método deben apoyar la liga de modelos que han sido derivados independientemente. Los métodos deben permitir apoyar particiones y trabajo independiente que, para sistemas grandes con múltiples analistas y diseñadores, será uno de los aspectos más importantes.

?? **Dominio de Aplicabilidad:** Algunos métodos sólo se aplican a sistemas basados en comportamiento secuencial, mientras que otros métodos manejan concurrencia, e incluso otros se aplican especialmente para sistemas de tiempo real (Selic, Gullekson, y Ward 1994). Se debe escoger métodos que apoyen las características del dominio escogido.

?? **Modelo de Proceso:** Los métodos seleccionados también deben ajustarse al modelo de proceso preferido, apoyando las entradas esperadas de las distintas actividades, especialmente documentación. Los métodos no deben contradecir el orden deseado de actividades del modelo de proceso, deben proveer guías para revisiones correspondientes y deben apoyar modelos evolucionables. Consideraciones de mantenimiento también pueden influir en la selección de los métodos. Se recomienda seleccionar métodos que administran suficiente información para explicar por qué existe una estructura particular en los distintos modelos. La explicación debe "rastrear" las suposiciones, metas y objetivos que llevaron hacia ese resultado. Los estructuras finales de implementación deben ser consistentes con aquellas especificadas en los modelos anteriores. A veces es útil poder hacer ingeniería en reversa en un sistema, o sea, derivar el modelo de diseño del código final. En tal caso, se debe evaluar la extensibilidad del método para apoyarla.

?? **Modelos Generados:** Una forma para calificar un método es determinar si los modelos que se desean producir pueden derivarse de la información que se obtiene y es representada por el método. Por ejemplo, si en cierto desarrollo se requiere un modelo de seguridad o un modelo de rendimiento antes de poder implementar una solución, entonces los métodos que se deben considerar son aquellos que manejan directamente estos requisitos o aquellos que pueden agregarse para derivar la información deseada. Se evalúa el apoyo provisto por el método y sus herramientas, y cuánto esfuerzo se necesita para extraer los resultados requeridos.

Una *notación* es usada para comunicar el resultado de aplicar un método, donde los elementos de la notación consisten de elementos gráficos, textuales, o alguna combinación de ambos. A menudo se confunde el concepto de

actividad, método, y notación, los cuales están relacionados pero son diferentes. La Figura 3.8 ilustra la relación entre ellos.

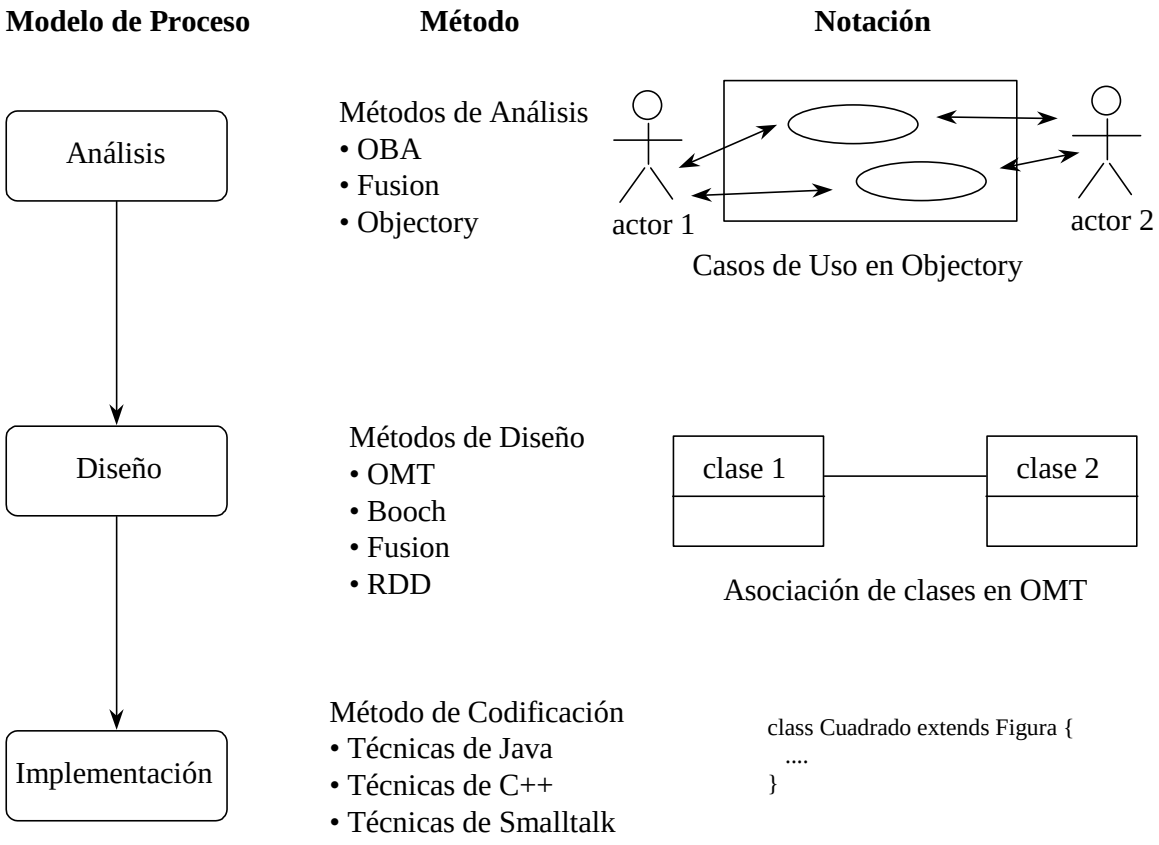

Figura 3.8 Contraste de las actividades de desarrollo de software versus métodos y notación utilizados.

El lado izquierdo ilustra posibles actividades de un modelo, el del medio muestra ejemplos de métodos para llevar a cabo estas actividades, y el lado derecho muestra ejemplos de notaciones para capturar el resultado de los métodos. La mayoría de los métodos pueden compararse examinando cómo se comunican los modelos estáticos y dinámicos, o sea, que tipo de diagrama de representación textual o gráfica se recomienda. Una notación no es simplemente buena o mala, si no más o menos efectiva en comunicar los resultados entre los miembros del equipo. La meta es escoger la notación que sea más efectiva para los equipos. Una buena notación debe tener suficiente poder de expresividad para modelar conceptos a nivel del detalle deseado. Algunas notaciones tienen un vocabulario más extenso que otras por lo cual pueden representar mayores niveles de detalle. También se desea una notación que permita representar modelos a varios niveles de abstracción. Una buena notación es también una que se puede aprender rápidamente, donde sus símbolos tienen sentido y son intuitivos. Las notaciones más pobres expresan grandes cambios semánticos con pequeños cambios en los símbolos. La ubicación, orientación o escala de un símbolo tiene un profundo significado que a menudo sólo el diseñador de la notación puede apreciar. Las notaciones deben comunicar información de manera que minimicen la sorpresa del lector. Como no siempre se tiene apoyo de herramientas para dibujar una notación, se quiere una notación que fácilmente se pueda dibujar a mano.
La mayoría de las organizaciones aún utilizan métodos que no están basados en conceptos orientados a objetos. Utilizan métodos tradicionales, como los *estructurados*, basados en modelos de datos, descomposición funcional, o programación de flujo de datos. Cuando tales organizaciones desean usar lenguajes orientados a objetos y componentes orientados a objetos reusables, el enfoque debe modificarse. Existe una gran disparidad entre los métodos tradicionales y los orientados a objetos, y entre los modelos de análisis y diseño creados con los métodos tradicionales y con los orientados a objetos, incluyendo el ciclo de vida del software. Los métodos tradicionales, especialmente la descomposición funcional, asumen que todos los requisitos primarios pueden representarse como funciones que están relacionados en una estructura jerárquica fija y pueden implementarse de la misma forma. Este enfoque no toma en cuenta las estrategias de desarrollo incremental e iterativo para construir sistemas con

tecnología orientada a objetos, ya que no reconoce que la gente resuelve problema de una forma no estructurada. Más aún, las funciones que están dentro de las jerarquías tienden a apoyar sólo una "superfunción", dificultando el reuso. Los métodos tradicionales se basan en la idea de que los datos y funciones deben separarse para que nuevas funciones puedan añadirse utilizando los datos comunes. La mayoría de las técnicas de los métodos tradicionales tratan con el modelado de datos. La idea que los datos son independientes de las funciones contradice la base conceptual de la tecnología orientada a objetos. En general, pueden existir proyectos que usen métodos tradicionales con tecnología orientada a objetos, pero se trabajaría el doble para obtener los comportamientos del sistema y los datos asociados.

3.5.1 Metodologías Estructuradas

La metodología de análisis y diseño estructurado, conocido por sus siglas en inglés como SA/SD (Structured Analysis and Structured Design) ha tenido mucho seguimientos durante las últimas décadas (Yourdon y Constantine 1978, DeMarco 1979, Page-Jones 1980, Ward y Mellor 1985, Yourdon 1989, Martin y Jackson). Existen múltiples variaciones al concepto básico estructurado como son SADT (Structured Analysis and Design Technique) (Ross, 1985) y RDD (Requirement Driven Design) basado en SREM (Alford, 1985). La metodología estructurada se basa primordialmente en la división entre funciones y datos, como se mencionó en la sección 2.1. Extendiendo este concepto básico, la metodología estructurada identifica durante el análisis las funciones del sistema, mientras que durante el diseño identifica los datos. Otros enfoques como la programación funcional rompen con este esquema (Backus 1977, Bird and Wadler 1988).

Durante las fases de requisitos y análisis se utilizan las siguientes herramientas para describir el sistema lógico: *diagramas de flujo de datos, especificación de procesos, diccionario de datos, diagramas de transición de estados, y diagramas de entidad-relación*. Durante las fases de diseño e implementación, los detalles son incorporados a los modelos anteriores y los *diagramas de flujo de datos* son convertidos a *cartas estructuradas* ("charts") las cuales especifican el código fuente.

?? **Diagramas de flujo de datos** (DFD) sirven para modelar la transformación de los datos en el sistema, y es uno de los modelos más importantes de SA/SD. Un diagrama de flujo de datos se compone de *procesos, flujo de datos, actores* (entidades externas) y *almacenamiento de datos*. Durante el diseño, los procesos del DFD son agrupados en tareas y asignados para su ejecución en el sistema operativo. Procesos del DFD se convierten en funciones del lenguaje de programación, y una *carta estructurada* es creada mostrando el árbol de llamadas de procedimientos.

?? **Especificación de procesos** sirve para describir los procesos a nivel más detallado. Esta especificación comienza desde el nivel más alto del diagrama de flujo de datos, donde los procesos se dividen de manera recursiva, con ayuda de subdiagramas, hasta que existen procesos suficientemente pequeños que sean fáciles de implementar. Esta especificación puede ser expresada con tablas de decisión, pseudo-código, u otras técnicas.

?? **Diccionario de datos** contiene detalles omitidos en los diagramas de flujo de datos, definiendo el significado de los nombres de los flujos y almacenamiento de datos.

?? **Diagramas de transición de estados** modelan el comportamiento que depende del tiempo. La mayoría de los diagramas de transición describen procesos de control o tiempo de ejecución de funciones y acceso a datos causado por eventos.

?? **Diagramas de Entidad-Relación** (ER) (Chen, 76) muestran la relación entre el almacenamiento de datos que de otra forma sólo serían vistos en la especificación de proceso. Cada elemento del ER corresponde a un dato almacenado. (La notación del diagrama de clases es una extensión del diagrama ER.) Este es el enfoque más común para el modelo de información. ER es una técnica gráfica que es popular ya que su notación es fácil de comprender, y suficientemente poderosa para modelar problemas del mundo real. Los diagramas ER son usualmente traducidos directamente a implementaciones de bases de datos.

3.5.2 Metodologías Orientadas a Objetos

Los sistemas orientados a objetos se desarrollan alrededor de entidades del dominio del problema, lo cual resulta en desarrollos bastante estables. El análisis orientado a objeto, a diferencia del estructurado, que considera comportamiento y datos de forma separada, combina ambos. Las características principales de los orientados a objetos son que ofrecen una forma de pensar más que una forma de programar. Además, reducen la complejidad en el diseño de software, y permiten atacar los errores durante el diseño en lugar de durante la implementación, donde el costo de reparación es bastante mayor. Las actividades son: (i) encontrar los objetos (dominio de la aplicación y problema particular); (ii) organizar los objetos (clases, asociaciones); (iii) describir cómo los objetos interactúan

(escenarios, casos de uso); (iv) definir las operaciones de los objetos (interfaces); y (v) definir los objetos internamente (atributos). Existen diversas formas de encarar estas actividades en conjunto y de manera particular. Las técnicas de análisis guían al analista en la transformación de la información provista por los expertos a las representaciones del modelo de análisis en el espacio del modelo de análisis. La naturaleza exacta de los intercambios varían, y es aquí donde los diferentes métodos de análisis divergen. Aunque la mayoría de los métodos están de acuerdo en los conceptos para modelar un problema, los métodos varían según las técnicas a utilizarse. Por ejemplo, se han propuesto diversas técnicas para identificar objetos en el dominio del problema:

?? **Escenarios**: Capturar comportamientos del sistema en guiones (scripts) o casos de usos para derivar los roles y responsabilidades del sistema, como con Object Behavior Analisis (OBA) (Rubin and Goldberg 1992) y Objectory (Jacobson 1992). Este es uno de los enfoques más utilizados en la actualidad.

?? **Tarjetas CRC**: Lluvia de ideas ("brainstorm") entre un grupo de expertos dando como resultado los objetos del dominio del problema que son anotadas en las tarjetas (Wirfs-Brock 1990).

?? **Ingenieria de información**: Identificar la estructura de la información que se guarda y mantiene por las aplicaciones, y mapearla a objetos. En el caso de Shaler y Mellor (1988), se adapta el modelo entidad-relación, estableciendo los objetos como entidades.

?? **Resaltar Texto**: Una técnica muy utilizada es subrayar nombres y verbos en la especificación de requisitos, como se hace en OMT.

Los diagramas que se utilizan tienen cierta similitud con los utilizados en las metodologías estructuradas aunque obviamente tienen sus diferencias. A diferencia de las metodologías estructuradas, los diagramas en las metodologías orientadas a objetos tienden a variar más.

?? **Diagramas de clases** son los más importantes describiendo los componentes básicos para las arquitecturas del análisis y diseño. A diferencia de los diagramas de flujo, que no son utilizados por la mayoría de las metodologías orientadas a objetos, los diagramas de clases muestran relación de asociación entre objetos y no flujo de datos entre ellos.

?? **Diagramas de casos de uso** son los que utilizaremos en este libro y se basan en Objectory correspondientes a la especificación de requisitos. Son completados por documentos en forma de textos.

?? **Diagramas de transición de estado** son probablemente los únicos que tienen su equivalente en las metodologías estructuradas mostrando los cambios de estado en los objetos.

?? **Diagramas de interacción**, también conocidos como diagramas de eventos, muestran aspectos dinámicos de los objetos en relación a la comunicación con otros objetos en el tiempo.

?? **Diagramas de colaboración** son diagramas que resumen los aspectos de comunicación entre objetos de un sistemas.

?? **Diagramas de subsistemas** son diagramas que muestran grupos de clases utilizadas en los distintos subsistemas.

Existe un gran número de metodologías orientadas a objetos, siendo las más importantes las mostradas en la Tabla 3.4.

Nombre del Método	Descripción
RDD	Responsibility-Driven Design [Wirfs-Brock 90]
OOAD	Object-Oriented Analysis and Design [Coad y Yourdon 91]
OOAD	Object-Oriented Analysis and Design [Booch 1991]
OMT	Object Modeling Technique [Rumbaugh et al. 1991]
OOSE/Objectory	Object Oriented Software Engineering [Jacobson 92]
OOK/MOSES	Object-Oriented Knowledge [Henderson-Sellers et al. 92]
OOSA	Object-Orientd System Analysis [Schlaer y Mellor 92]
OOAD	Object-Oriented Analysis and Design [Martin y Odell 92]
OOSA	Object-Oriented Systems Analysis [Embley et al. 92]
OBA	Object Behavior Analysis [Rubin y Goldberg 92]
OORA	Object-Oriented Requirements Analysis [Firesmith 93]
Synthesis	Synthsis Method [Page-Jones y Weiss 93]
OOSD	Object-Oriented System Development [de Champeaux 93]
OOAD/ROSE	Object-Oriented Analysis & Design [Booch 94]
FUSION	Object-Oriented Development [Coleman et al. 94]
Unified	Rational's Unified Software Development Process [Booch et al. 99]

Tabla 3.4 Métodos de desarrollo de software

3.5.3 Integración de Metodologías

Un aspecto importante que resulta de las discusiones de las secciones anteriores es que es sumamente importante escoger una metodología apropiada al tipo de desarrollo que se esté haciendo. Más aún, a veces es necesario integrar metodologías diferentes aplicadas a las diferentes actividades de desarrollo. Esto ocurre generalmente porque las metodologías tienen fortalezas para ciertos aspectos del desarrollo pero no para otros. El integrar metodologías, sin embargo, requiere tener mucho cuidado asegurando que los resultados de una sirvan como entrada a la otra. Esto no significa que las metodologías se puedan "mezclar" durante el desarrollo de una misma actividad.

3.6 Herramientas

Existen *herramientas* que apoyan los diversos aspectos del proceso de software. Al conjunto de herramientas aplicables al desarrollo de sistemas de software se les conoce como CASE ("Computer-Aided Software Engineering"), herramientas para asistir al desarrollador en las diferentes fases del ciclo de vida del proceso del software: planeación, requisitos, análisis, diseño, implementación, pruebas (verificación y validación), documentación, mantenimiento y administración. Las herramientas varían en el tipo de componentes que incorporan, editores (textuales y gráficos), programadores (codificadores, depuradores, compiladores y ensambladores), verificadores y validadores (analizadores estáticos y dinámicos y diagramas de flujos), medidores (monitores), administradores de la configuración (versiones y librerías) y administradores del proyecto (estimación, planeación y costo). La herramienta particular a ser usada debe apoyar el modelo de proceso escogido, y no se debe considerar métodos independientes de herramientas. Si las herramientas son buenas, éstas deben resultar en mejorías notorias en la producción del sistema. Las herramientas deben manejar aspectos de administración de información, generar los distintos tipos de diagramas e incluso el código final. La sofisticación de las herramientas varía desde aquellas que apoyan a un sólo desarrollador en un sólo proyecto hasta aquellas que apoyan múltiples desarrolladores trabajando juntos en un proyecto con almacenamiento compartidos, e incluso múltiples proyectos.

Existen actualmente productos que manejan múltiples métodos y notaciones. La idea llama la atención si se considera a estas herramientas como manejadores de almacenamiento, donde la información obtenida por uno o más métodos se puede almacenar en una representación común y luego mostrarse con la notación preferida, una de las motivaciones detrás de UML. La idea de una sola herramienta que pudiese ajustarse a distintos tipos de proyectos parece ser buena, pero en la práctica puede que no lo sea, dado que se arriesga crear un gran almacenamiento de datos de información no relacionada, sin enfocarse bien a ningún método. Además, se arriesga crear una situación donde los diferentes productos generados no son consistentes. El éxito de las herramientas actuales de proveer múltiples notaciones se asocia con el hecho de que los distintos métodos son fundamentalmente similares en relación a los conceptos básicos que apoyan.

Se debe seleccionar herramientas de acuerdo a los siguientes criterios:

?? Proveer apoyo explícito para cada paso del método.
?? Administrar toda la información que el método requiere obtener o especificar.
?? Poder manejar grandes cantidades de información y ser escalable.
?? Incluir un mecanismo por el cual se pueda probar que la información recolectada es consistente.
?? Apoyar la organización de los diagramas de manera automática.
?? Manejar múltiples usuario simultáneos en uno o múltiples proyectos.
?? Poder generar una implementación inicial junto con la documentación.
?? Apoyar ingeniería en reversa para asegurar que cambios directos en la implementación sean consistentes con los modelos administrados.

La Tabla 3.5 hace un resumen los aspectos más relevantes para seleccionar entre los diversos métodos.

Criterio de Selección	Descripción
Conceptos apoyados	El método debe apoyar conceptos básicos que se cree son significativos en resolver el problema.
Propiedad notacional	La notación debe ser comprensible inmediatamente, y debe haber un subconjunto mínimo para principiantes. Debe ser dibujable a mano.
Cobertura del modelo de proceso	El método debe aplicarse a las actividades identificadas en el modelo de proceso.
Tipos de aplicaciones	El método debe orientarse hacia el tipo de aplicaciones que la organización construye.
Personalización	Si se espera refinar algún método seleccionado, entonces el método debe identificar aspectos apropiados para presonalización. Alternativamente, si se espera componer varios

	métodos, debe asegurarse que las entradas y salidas sean complementarias.
Enfoque evolucionario versus revolucionario	El método debe ser consistente con la habilidad de la organización para incorporar nueva tecnología.
Aprendizaje	El método propuesto debe ser fácil de aprender.
Rastreabilidad	El método debe mantener un mapa claro y consistente entre componentes.
Escalabilidad	El método (y herramientas relacionadas) deben ser apropiados para el tamaño del problema a resolverse. Un método necesita escalar hacia arriba y abajo según las necesidades del proyecto.
Material colateral	El método debe producir los documentos colaterales requeridos por la organización.
Ambiente de la herramienta	Las herramientas que apoyen al método deben integrarse correctamente, con un acceso abierto a la información recolectada.
Momento en el mercado	Se debe tener confianza en que el método y herramientas se mantendrán en el mercado, para los cuales hay amplio entrenamiento y consultores.

Tabla 3.5 Criterio de selección para métodos de desarrollo de software