Capítulo 12
Ingeniería de software: el proceso para el desarrollo de software

Por Alfredo Weitzenfeld Ridel y Silvia Guardati Buemo

El desarrollo de software es una de las actividades más importantes de la computación, ya que está presente tanto en el desarrollo de aplicaciones ajenas a la computación —por ejemplo, un programa que controla la asignación de salas de embarque en un aeropuerto— como en el desarrollo de programas básicos en el área —por ejemplo, un sistema operativo—. Otro aspecto relevante que debe tenerse en cuenta es que el desarrollo de software no es una tarea solamente técnica, en la cual lo único que importa es la tecnología y los desarrolladores. La producción de software generalmente también involucra a terceros (es decir, en la mayoría de las situaciones se desarrolla un programa para satisfacer una necesidad específica de un usuario que no es el mismo programador). Por lo tanto, el éxito de un programa está sujeto a que éste haga lo que se espera que haga, que haya sido desarrollado con los recursos estimados y que sea confiable. Considerando lo mencionado, el diseño, desarrollo y mantenimiento de software debe realizarse con la misma seriedad y responsabilidad con la que se llevan a cabo cualquiera de las actividades propias de las ingenierías tradicionales.

En este capítulo se presenta una introducción al proceso de desarrollo de software, así como los conceptos básicos acerca de metodologías que existen para el trabajo en equipo, dado que el software que se desarrolla hoy en día ya no son programas de cien líneas de código, hechos por un solo programador, como lo eran durante los primeros años de la computación. De esta manera, el lector podrá conocer los fundamentos de la ingeniería de software aplicables en el desarrollo de software tanto para las áreas de la computación estudiadas en este libro, como para cualquier área del saber.

12.1 Modelo del proceso

Un **proceso** define quién hace qué, cuándo y cómo lo hace, para alcanzar cierto objetivo. En general, el éxito de las empresas u organizaciones depende en gran medida de la definición y seguimiento adecuado de sus procesos. En el caso de una empresa que se dedica al desarrollo de software, se requieren procesos especializados que abarquen desde la creación hasta la administración de un sistema de software. Como se ha visto en capítulos anteriores, los sistemas de software pueden llegar a ser extremadamente complejos. Para administrar la complejidad de tales sistemas es necesario contar con modelos de procesos y tecnologías de software apropiadas. En este capítulo se describe en qué consiste el proceso de software.

Un **modelo de proceso de software** define cómo resolver la problemática del desarrollo de sistemas de software. Para desarrollar software se requiere resolver ciertas fases de un proceso que se conocen en su conjunto como el **ciclo de vida**[1] del desarro-

[1] Scacchi, W., 2001, "Process models in software engineering", en J. Marciniak (Ed), *Encyclopedia of software engineering* (second edition), Wiley.

llo de software. Un modelo de proceso debe considerar una variedad de aspectos, como el conjunto de personas, estructuras organizacionales, reglas, políticas, actividades, componentes de software, metodologías y herramientas utilizadas.

A continuación se describen aspectos esenciales que definen un proceso: arquitecturas, actividades, métodos y metodologías, estrategias y herramientas para la administración de software.

12.1.1 *Arquitecturas*

Una **arquitectura de software** define la estructura general de un sistema. Las arquitecturas varían de acuerdo con el tipo de sistema a desarrollarse. Pueden ser arquitecturas basadas en elementos sencillos o en componentes prefabricados de mayor tamaño. Además de depender del tipo de sistema a desarrollar, la selección de una arquitectura afecta aspectos como la **extensibilidad** del sistema (qué tan fácil es extenderlo en el futuro para incorporar más funcionalidad o mayor capacidad). Por lo tanto, la arquitectura debe ser escogida de manera que minimice los efectos de los cambios que pueda haber en el futuro en el sistema. Para hacer esta elección existen ciertas heurísticas que muestran la tendencia a cambiar en varios elementos de un sistema, como se muestra en la tabla 12.1.[2] Las **interfaces** representan los elementos gráficos, la **funcionalidad** son las reglas del negocio (requisitos del usuario), los **datos** y **funciones** son los elementos internos que se usan para describir a los objetos (correspondientes a las estructuras de datos básicas de la programación orientada a objetos), mientras que la **información** representa el dominio del problema en una aplicación.

Esta tabla resalta dos aspectos del desarrollo de software. Primero, la arquitectura del sistema debe distinguir entre elementos con mayor y menor probabilidad de cambios. Segundo, el desarrollo de software debe considerar un modelo de proceso en el que aquellos elementos de mayor probabilidad de cambio no "arrastren" a los más estables. Este tema se discute con mayor detalle en la sección de metodologías.

12.1.2 *Actividades*

Una **actividad** es una unidad (un paso) básica de un proceso. En el proceso de software

Tabla 12.1 Probabilidad de cambios futuros en el software de acuerdo con el tipo de elemento de diseño.

Elemento	Probabilidad de cambio
Interfaces	Alta
Funcionalidad	Alta
Datos	Mediana
Funciones	Mediana
Objetos	Baja
Información	Baja

[2] Coad, P., y Yourdon, E., 1991, *Object-oriented analysis*, Yourdon Press.

las actividades definen los pasos necesarios para lograr las metas y objetivos (por ejemplo, especificar los requisitos del sistema). Las actividades deben ser fáciles de definir y seguir, deben simplificar la comprensión del sistema, y ofrecer flexibilidad, precisión y extensibilidad. Las actividades básicas del proceso de desarrollo de software, conocidas como el ciclo de vida del software, son las siguientes:

(i) *especificación de requisitos* para capturar los aspectos funcionales del sistema, describiendo cómo interactuaría un usuario con la aplicación.
(ii) *análisis* para dar al sistema una estructura o *arquitectura* robusta y extensible independiente del ambiente de implementación final.
(iii) *diseño* para adoptar y refinar la arquitectura del sistema y adaptarla al ambiente de implementación específico.
(iv) *implementación* para programar el sistema.
(v) *pruebas* para validar y verificar el sistema.
(vi) *integración* para combinar los diferentes componentes del sistema.
(vii) *documentación* para describir los diversos aspectos del sistema.
(viii) *mantenimiento* para extender la funcionalidad del sistema.

La tabla 12.2 muestra las actividades más importantes del ciclo de vida del proceso de software.

La transición entre las distintas actividades del ciclo de vida del software debe ser natural: debe existir continuidad o **rastreabilidad** (en inglés, **traceability**, la capacidad de entender por qué razón surgió y a qué resultados llevó cada decisión realizada durante el desarrollo de software) de una actividad a la siguiente o la anterior. A continuación describimos con mayor detalle cada una de las actividades listadas arriba.

12.1.2.1 Requisitos

La actividad de especificación de requisitos tiene como meta definir y delimitar la funcionalidad del sistema de software. La especificación de requisitos sirve como base de negociación entre el desarrollador del sistema y el cliente, y debe reflejar los deseos de éste. El resultado es el **modelo de requisitos**. Es esencial que los clientes que no tengan conocimientos en computación comprendan el modelo de requisitos para facilitar la interacción de los desarrolladores del software con ellos.

El modelo de requisitos gobierna el desarrollo de los demás modelos —los demás modelos se deben basar en éste—. El modelo de requisitos también sirve como base para elaborar las instrucciones de operación y los manuales, los cuales deben ser redactados desde el punto de vista del usuario.

El desafío en la especificación de requisitos comienza cuando el experto debe comunicar los conceptos importantes para su aplicación. En general, no es posible hacer esto adecuadamente de manera sencilla. Como resultado, se provee de múltiples explicaciones, orales o escritas, que el desarrollador trata de integrar en una forma 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 en su información o se contradicen.

12.1.2.2 Análisis

Después de desarrollar el modelo de requisitos y de que los usuarios del sistema o clientes lo aprueben, se puede continuar con la elaboración del **modelo de análisis**, teniendo como meta construir una arquitectura capaz de resolver el problema bajo condiciones ideales. Esto significa desarrollar una estructura lógica del sistema que debe ser estable y extensible. El análisis se enfoca en qué debe hacer el sistema, en lugar de cómo se

Tabla 12.2 Actividades del desarrollo de software.

Actividad	Descripción
Requisitos	Se especifican las necesidades del sistema a desarrollar. La especificación de requisitos sirve 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 con el propósito de estructurar la arquitectura del sistema. Responde a la pregunta del "qué" del sistema.
Diseño	Se transforma la arquitectura obtenida durante el análisis en una arquitectura especializada, donde se considera el ambiente de implantación particular del sistema. Obedece al "cómo" del sistema.
Implementación	Se expresa la arquitectura del sistema en una forma aceptable para la computadora (es decir, el código programado).
Integración	Se combinan los componentes creados de manera independiente para formar el sistema completo.
Pruebas	Se verifica y valida el sistema a nivel de componentes individuales y su integración. Éste es uno de los aspectos más críticos del desarrollo y debe desarrollarse de manera concurrente con el resto de las actividades. 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.
Documentación	Se describen los aspectos sobresalientes de los requisitos, análisis, diseño, implementación, integración y pruebas. Esto servirá para usuarios externos e internos, aquellos encargados de mantener el sistema y extenderlo.
Mantenimiento	Se corrigen errores que no se hayan encontrado durante el desarrollo y las pruebas originales del sistema. Se extiende el sistema si surgen nuevas necesidades.

supone que lo hará. El alcance del modelo de análisis está directamente relacionado con la naturaleza del problema. En el caso de un análisis orientado a objetos, se desea identificar los objetos y describir cómo interactúan entre sí.

12.1.2.3 Diseño

El propósito del **modelo de diseño** es extender la arquitectura de análisis. La razón para no hacer esta extensión durante el modelo de análisis es que la propia aplicación controla la arquitectura del sistema y no las circunstancias existentes durante su implementación. En otras palabras, el modelo de análisis debe ser visto como un modelo conceptual y lógico del sistema, mientras que el modelo de diseño debe definir todo lo que es necesario hacer para alcanzar el código final. Dado que los ambientes de implementación tienden a cambiar, es necesario guardar y congelar el modelo de análisis para cualquier manteni-

miento que se requiera en el futuro, incluso después de terminar el diseño. El modelo de diseño se concentra en dos aspectos principales, el diseño de objetos y el diseño de sistemas, como se describe a continuación.

El modelo de análisis no es lo suficientemente formal, por lo cual para llegar al código final se deben refinar las estructuras de la arquitectura de análisis. Es necesario especificar el detalle de cada clase (en otras palabras, sus operaciones y atributos). Este aspecto se conoce como el **diseño de estructuras** o, de manera general, como el **diseño de objetos**, en el caso de arquitecturas orientadas a objetos. El diseño de objetos incluye la selección de algoritmos y estructuras de datos para satisfacer los objetivos de rendimiento y espacio del proyecto de software. El modelo de análisis y diseño de objetos tienen bastante en común, incluyendo conceptos, técnicas y notaciones similares. Como consecuencia, las mismas herramientas de desarrollo se utilizan para llevar a cabo ambas actividades. A menudo estas similitudes hacen difícil saber qué actividad se está llevando a cabo. Uno de los beneficios de la tecnología orientada a objetos es representar 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 de desarrollo de software.

Durante el análisis se considera un mundo ideal para el sistema. En la realidad este mundo ideal debe adaptarse al ambiente en el que se va a implementar el sistema. Entre otros aspectos, se deben considerar los requisitos de rendimiento, uso de memoria, protocolos de comunicación, tiempo real, concurrencia, propiedades del lenguaje de programación, el sistema de manejo de base de datos, etc. Este aspecto se conoce como el **diseño de sistema**, que define las decisiones estratégicas sobre cómo organizar la funcionalidad del sistema en torno al ambiente de implementación, incluyendo tanto hardware como software. Para adaptar el modelo de análisis al ambiente de implementación, es necesario identificar las restricciones técnicas bajo las cuales se tiene que construir el sistema. Además, el diseño del sistema debe apoyar aspectos adicionales como una terminación inesperada durante la ejecución del sistema por agotamiento de recursos o por fallas externas del hardware.

12.1.2.4 Implementación

El modelo de implementación toma el resultado del modelo de diseño para generar el código final del sistema. Esta traducción debe ser relativamente sencilla y directa, ya que todas las decisiones importantes han sido tomadas 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 específico de implementación. Muchas herramientas, como se ve más adelante, apoyan el proceso de generación automática de código a través de un proceso de **ingeniería hacia adelante** (en inglés, **forward engineering**), como en el caso de herramientas CASE basadas en UML. Sin embargo, la generación automática de código es parcial, y se requiere que el desarrollador la complete de manera manual. En el modelo de implementación, el concepto de rastreabilidad también es importante, dado que al leer el código fuente se debe poderlo relacionar con los modelos de diseño y análisis.

Aunque existen muchos tipos de lenguajes de programación, el uso de un lenguaje orientado a objetos facilita 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, de tal manera que si se cambia de lenguaje de programación no debe ser necesario el rediseño del sistema. Por razones de rastreabilidad, es deseable siempre tener una buena y fácil correspondencia entre los objetos del modelo de diseño y los objetos del lenguaje de programación.

En la actualidad, las bases de datos son parte integral de los sistemas de software. Aunque existen bases de datos orientadas a

objetos, aún siguen siendo más populares las bases de datos relacionales. El modelo de implementación debe contemplar el diseño de la base de datos que se vaya a utilizar en la generación del sistema final.

12.1.2.5 Integración

El modelo de integración es un aspecto importante del desarrollo de software. En todo diseño es deseable mantener una buena modularidad en el sistema, de manera que el desarrollo actual junto con las futuras extensiones puedan hacerse con base en componentes independientes y no en la totalidad del sistema. Cuando esto ocurre, es necesario integrar los diversos componentes para obtener como resultado el sistema final.

12.1.2.6 Pruebas

El modelo de pruebas es el responsable de revisar la calidad del sistema. Este modelo consta de la validación del sistema (también conocida como prueba de especificación) y la verificación (también conocida como prueba de resultado). De manera adicional, el modelo de pruebas combina pruebas de unidad y pruebas de integración.

En la validación se prueba si la funcionalidad del sistema corresponde a la especificación del cliente. Se cuestiona si se está construyendo el sistema "correcto", en contraste con la verificación, donde se cuestiona si se está haciendo el sistema "correctamente". Durante el diseño e implementación se revisa el sistema de acuerdo con las especificaciones de los modelos de análisis y de requisitos.

En la verificación se prueba si se está construyendo el sistema "correctamente". La verificación debe comenzar lo antes posible, desde el nivel más bajo, con la revisión de los componentes individuales, prosiguiendo con la integración de éstos, hasta verificar el sistema completo. La especificación de verificación del sistema debe ser una extensión del modelo de requisitos e integrarse en la arquitectura del sistema.

12.1.2.7 Documentación

La documentación se debe hacer durante la elaboración del sistema y no como una etapa final del mismo. Existen diferentes tipos de documentos que se deben generar como apoyo al sistema. Cada uno tiene diferentes objetivos y está dirigido a distintos tipos de personas, desde los usuarios no técnicos hasta los desarrolladores más técnicos. Los siguientes son algunos documentos o manuales más importantes.

El **manual del usuario** permite a un usuario comprender cómo utilizar el sistema. El **manual del programador** contiene información para que un desarrollador entienda los aspectos más relevantes de diseño. El **manual del operador** permite al operador del sistema comprender qué pasos debe seguir para que el sistema funcione bajo cierta configuración y con base en un ambiente de implementación particular. El **manual del administrador** permite que el encargado de administrar el sistema comprenda sus aspectos más generales, como son los modelos de requisitos y análisis.

En realidad no hay límite en cuanto a la cantidad de manuales y el detalle de la documentación, así como tampoco hay límite con respecto a cuánto se puede extender y optimizar un sistema. El objetivo es mantener un nivel de documentación que sea útil y extensible.

12.1.2.8 Mantenimiento

El mantenimiento de un sistema es la continuación del ciclo de vida luego de haber completado una primera versión del sistema. Aunque parte del objetivo involucra resolver problemas, durante el mantenimiento se deben considerar las extensiones del sistema de acuerdo con nuevas necesidades. El mantenimiento significa seguir un nuevo ciclo de actividades de desarrollo, pero partiendo de un sistema ya existente.

12.1.3 *Métodos y metodologías*

Los **métodos** definen las reglas para las transformaciones internas de las actividades, mientras que las **metodologías** definen el conjunto de métodos. Un método es un procedimiento que define tareas o acciones a realizar, donde cada tarea incluye condiciones de entrada y de salida que se deben satisfacer antes de realizarse y después de completarse. Las diferentes metodologías varían en el alcance del apoyo que proporcionan al desarrollo de software.

Los métodos deben apoyar conceptos básicos que se consideren significativos para resolver el problema. Se deben poder utilizar los métodos en diferentes dominios de aplicación, y aplicar a sistemas basados en diferentes arquitecturas, incluyendo secuencial, concurrente, distribuida e incluso en tiempo real.

Los métodos deben ajustarse al ciclo de vida del proceso, apoyando las distintas actividades, incluyendo la documentación. Deben explicar las suposiciones, metas y objetivos que llevaron hacia un resultado particular. Los métodos no deben contradecir el orden establecido para las actividades del modelo de proceso, sino proveer guías para llevarlas a cabo. El mantenimiento de un sistema también debe estar apoyado por los métodos.

Los métodos deben proveer técnicas para recopilar información de acuerdo con el proceso de desarrollo. Por ejemplo, si el proceso se basa en tecnologías orientadas a objetos, los métodos deben apoyar la identificación de objetos en el sistema. Por otro lado, si el proyecto tiene como objetivo crear componentes reutilizables, los métodos deben incluir técnicas para la obtención y verificación de estos componentes.

Los métodos deben apoyar su propia extensibilidad, identificando qué aspectos del método pueden ser modificados por el desarrollador para adaptarlo a sus necesidades particulares (por ejemplo, el formato de la documentación).

Los métodos deben permitir la generación de modelos a partir de la información recopilada. Por ejemplo, si en cierto desarrollo se requiere un modelo de seguridad o uno de rendimiento, entonces los métodos deben considerar estos requisitos o poder extenderse para obtener la información deseada. Se debe evaluar esta capacidad, además del esfuerzo necesario para obtener tales resultados.

Los métodos deben apoyar la integridad de los modelos generados, verificando y evitando errores de coherencia, además de incluir técnicas para detectar problemas. Esto significa que las herramientas que sólo apoyan la diagramación son muy limitadas como apoyo a métodos, ya que carecen de manejo de coherencia. Los métodos deben permitir el desarrollo independiente, algo esencial para sistemas de gran tamaño con múltiples analistas y diseñadores.

Los métodos deben ofrecer entradas y salidas bien definidas que permitan la integración de diversos métodos, incluso pertenecientes a distintas metodologías. A veces es deseable aplicar diferentes metodologías a distintas actividades de desarrollo. Esto ocurre cuando ciertas metodologías son más apropiadas para ciertos aspectos del desarrollo, como análisis o diseño.

Los métodos y herramientas correspondientes deben ser apropiados para el tamaño del problema a resolver. Un método necesita escalar hacia arriba o hacia abajo según las necesidades del proyecto.

Es muy importante contar con una notación estandarizada para representar los modelos desarrollados. Estos modelos deben incluir elementos gráficos, de texto o alguna combinación de ambos. Una notación no es simplemente buena o mala, si no más o menos eficaz en comunicar los resultados. Una buena notación debe tener la suficiente expresividad para modelar conceptos al nivel del detalle deseado. Algunas notaciones tienen un vocabulario más extenso que otras, por lo cual permiten mostrar más detalles. También se hace necesaria una notación que permita representar modelos con varios ni-

veles de abstracción. Una buena notación también facilita su comprensión y aprendizaje, teniendo un subconjunto mínimo como apoyo a los principiantes. Las notaciones más pobres expresan grandes cambios semánticos con pequeños cambios en los símbolos. Las notaciones deben comunicar información de manera que minimicen el factor sorpresa. Como no siempre se tiene apoyo de herramientas para dibujar una notación, se buscan notaciones que sean fáciles de dibujar.

Se debe tener confianza en los métodos y herramientas correspondientes, en que éstas se mantendrán en el mercado (seguirán estando disponibles) y en que haya la posibilidad de capacitación y apoyo técnico.

Existe una gran variedad de métodos y metodologías en apoyo al proceso de software. A continuación se describen las metodologías estructuradas o tradicionales y las orientadas a objetos.

12.1.3.1 METODOLOGÍAS ESTRUCTURADAS

Las **metodologías tradicionales** o **metodologías estructuradas** se enfocan principalmente en la descomposición funcional de un sistema. El objetivo es lograr una definición completa del sistema en término de funciones, estableciendo los datos de entrada y salida correspondientes a cada función que debe cumplir el sistema. A estas metodologías se les conoce como de **análisis y diseño estructurado** (en inglés, **SA/SD—structured analysis and structured design**). Durante las actividades de desarrollo se utilizan diferentes herramientas de modelado.

Los **diagramas de flujo de datos** (DFD) sirven para modelar la transformación de datos a través de las funciones del sistema. Un diagrama de flujo de datos se compone de procesos, flujo de datos, actores (entidades externas) y almacenamiento de datos. Durante el análisis, los procesos del DFD se descomponen hasta convertirse, durante el diseño, en funciones de programación, creando una carta (chart) estructurada del sistema.

Los **diagramas de transición de estados** sirven para modelar el comportamiento en el tiempo. Los diagramas de transición describen el efecto de eventos externos en los procesos y funciones de un sistema.

Los **diagramas de entidad-relación** (o entidad-vínculo) sirven para modelar el almacenamiento de datos (la forma en la que éstos están organizados).

12.1.3.2 METODOLOGÍAS ORIENTADAS A OBJETOS

Las **metodologías orientadas a objetos** se enfocan principalmente en el modelado de un sistema en término de los objetos (y clases de objetos) que va a manipular. A diferencia de las metodologías estructuradas, inicialmente se identifican los objetos del sistema, para luego especificar su comportamiento (funciones). Durante las actividades de desarrollo se utilizan diferentes herramientas de modelado.

Los **diagramas de clases** sirven para describir los componentes esenciales de la arquitectura de un sistema. A diferencia de los diagramas de flujo de datos, los diagramas de clases muestran relaciones de asociación entre clases, y no flujo de datos entre ellas.

Los **diagramas de casos de uso** sirven para especificar un sistema en términos de su funcionalidad. A diferencia de las metodologías estructuradas, los diagramas de casos de uso no son descompuestos en funciones de programación.

Los **diagramas de transición de estado** sirven para describir los cambios de estado en los objetos, siendo equivalentes a sus similares en las metodologías estructuradas.

Los **diagramas de secuencia** sirven para describir los aspectos dinámicos del sistema, mostrando el flujo de eventos entre objetos a través del tiempo.

Los **diagramas de colaboración** sirven para describir la comunicación entre objetos en un sistema.

Los **diagramas de subsistemas** sirven para describir agrupaciones de clases en un sistema.

A menudo se confunden algunos de los conceptos descritos hasta el momento, como los de actividad, método y notación. La figura 12.1 ilustra la relación entre ellos. El lado izquierdo de la figura muestra diferentes actividades, la parte media muestra ejemplos de métodos para llevar a cabo las actividades correspondientes, mientras que el lado derecho contiene ejemplos de notaciones para capturar el resultado de estos métodos.

12.1.4 *Estrategias*

Una **estrategia** se define como un plan para lograr un objetivo. Las estrategias afectan aspectos como la arquitectura del sistema, el orden en que se llevan a cabo las actividades del proceso y las metodologías a utilizarse. Dada la variedad de posibilidades, es necesario tomar ciertas decisiones iniciales dependiendo del tipo de proyecto que se va a desarrollar. Estas decisiones son parte de una estrategia de desarrollo, lo cual incluye la selección de una tecnología y lenguaje de programación particular (por ejemplo, la tecnología orientada a objetos y el lenguaje Java, respectivamente). Otras estrategias aceptadas en la actualidad son los prototipos y la reutilización, que se describirán a continuación.

12.1.4.1 PROTOTIPOS

Un **prototipo** es una versión preliminar, intencionalmente incompleta o reducida de un

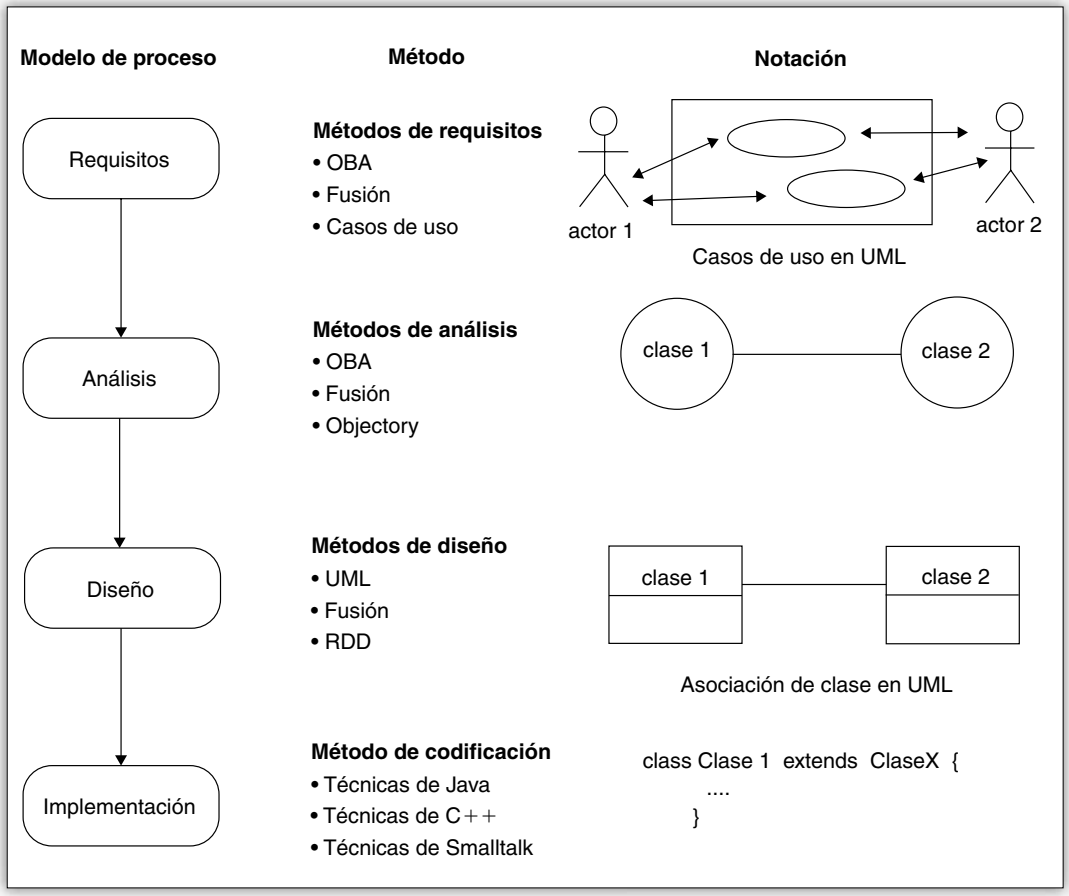

Figura 12.1 Contraste de las actividades, métodos y notaciones de desarrollo de software.

sistema. El uso de prototipos es una estrategia que puede aplicarse en casi todas las actividades del proceso de software. El propósito de los prototipos es obtener de manera rápida la información necesaria como ayuda en la toma de decisiones. Los siguientes son algunos tipos de prototipo.

Los **prototipos de requisitos** permiten que los usuarios perciban la funcionalidad del producto final a través del diseño de interfaces o pantallas del sistema. El objetivo es ayudar a aclarar los requisitos y solicitar nuevas ideas.

Los **prototipos de análisis** permiten generar rápidamente una arquitectura general que considere las características principales del sistema de acuerdo con la especificación de requisitos.

Los **prototipos de diseño** permiten explorar y comprender la arquitectura particular de un sistema para poder evaluar aspectos como cuellos de botella (rendimiento y uso de memoria) o incoherencias en el diseño.

Los **prototipos verticales** permiten comprender parte de un problema y desarrollar su solución completa. Esto se hace generalmente cuando los conceptos básicos no están bien comprendidos (por ejemplo, el seguimiento de cierta metodología).

Los **prototipos de factibilidad** permiten demostrar si es posible lograr ciertos objetivos del proyecto (por ejemplo, aplicar una arquitectura particular, conectarse a una base de datos bajo ciertas restricciones de rendimiento, aprender a programar en cierto lenguaje en un tiempo determinado o predecir los costos de desarrollo de un proyecto).

Un prototipo no es necesariamente un producto de calidad, que deba mantenerse a largo plazo. Por el contrario, los prototipos a menudo son creados y probados rápidamente, para luego ser descartados. Sin embargo, es común que, por presiones de tiempo, se trate de enviar un prototipo al mercado (venderlo) como si éste fuera el producto final. En general, siempre existirá un conflicto entre un desarrollo rápido y un producto de calidad. Las siguientes dos listas muestran algunas consideraciones para el éxito o fracaso de los prototipos[3].

Los prototipos tienen éxito cuando se comprende su propósito y se usan de manera adecuada, se comprende la tecnología que va a utilizarse y su relación con el proceso de prototipos, se integra un grupo técnico apropiado para hacer el prototipo (líder de proyecto, documentador, elaborador de prototipos de requisitos y análisis, y elaborador de prototipos de diseño), se evalúa al grupo y las entregas finales, se involucra temprano en el proceso de software a los usuarios finales, se está dispuesto a repetir el proceso de prototipos para comprender mejor la arquitectura básica, se establecen criterios de evaluación apropiados al comienzo de cada etapa de prototipos y se basa uno firmemente en estos criterios para su terminación y/o se construyen prototipos basados en una biblioteca de código reutilizable, controlada por el bibliotecario asignado.

Los prototipos fallan cuando no se comprende qué es un prototipo y cómo debe usarse, no se comprende el proceso lo suficientemente bien como para organizar al grupo de manera adecuada, no se sabe hasta cuándo dejar de evolucionar el prototipo y comenzar de cero (así extendiendo demasiado el proceso), no se sabe hasta cuándo continuar para tratar de lograr los criterios de evaluación deseados (así terminando prematuramente el proceso), no se utilizan ambientes o herramientas de apoyo adecuados para el desarrollo particular, se cree que un prototipo razonable es un producto aceptable y/o los prototipos nunca terminan.

12.1.4.2 Reutilización

La **reutilización** es la explotación de componentes desarrollados anteriormente dentro de un mismo proyecto o entre proyectos, para

[3] Goldberg, A., y Rubin, K, 1995, *Succeeding with objects: decision framework for project management (primera edición)*, Addison-Wesley.

no comenzar a desarrollar algo desde cero cuando ya se tiene un componente que proporciona cierta funcionalidad necesaria. Dentro de un mismo proyecto, la reutilización se aprovecha mediante estructuras comunes de bajo nivel, como procedimientos, clases o herencia. Este tipo de reutilización produce generalmente programas más compactos. Entre proyectos, la reutilización se aprovecha mediante estructuras comunes de alto nivel, como paquetes gráficos y bibliotecas de análisis numérico. La reutilización entre proyectos requiere planeación y representa una inversión tanto para producir componentes reutilizables como para consumirlos. La decisión para reutilizar componentes se basa en una evaluación de los costos relativos entre crear una nueva solución o adaptar una existente.

Consumir componentes reutilizables requiere identificar si ya existe una solución disponible, ya sea parcial o completa. Las ventajas de la reutilización incluyen soluciones coherentes entre aplicaciones y confianza en la calidad del componente, al haber sido probado anteriormente en múltiples aplicaciones. Para tener éxito con la reutilización, la solución debe adaptarse a las necesidades del sistema. Es común que los desarrolladores sólo reutilicen soluciones si confían en su nivel de calidad. Las oportunidades de reutilización pueden ocurrir durante todo el ciclo de vida de un proyecto.

Producir componentes reutilizables significa tener una perspectiva de múltiples proyectos. Una organización debe considerar los costos adicionales de producir un componente reutilizable, planeando la reducción de costos que ocasiona utilizar los componentes en otros proyectos. Así, el esfuerzo para producir componentes reutilizables es bastante mayor al esfuerzo de desarrollo de componentes "normales", generados específicamente para una aplicación dada.

A continuación se presentan razones a favor y en contra (ventajas y desventajas) de la reutilización.[4] La reutilización es valiosa cuando se desea apoyar lo común, promover la coherencia mediante estándares, incrementar la habilidad para analizar problemas entre diversos grupos, reducir costos, facilitar el inicio de un desarrollo, acelerar el tiempo de entrega o mejorar la calidad del producto. La reutilización no es valiosa cuando se tienen que ajustar los componentes generalizados a los requisitos de rendimiento y el tiempo de aprendizaje para consumir y producir componentes reutilizables a los tiempos del proyecto, se tiene que ajustar algo existente haciendo demasiados cambios, la componente reutilizada provee un exceso de funcionalidad, no se entienden las interfaces de programación de los componentes, o cuando estos últimos son propiedad de un proyecto específico, incluyen demasiada funcionalidad no requerida o no disminuye la cantidad total de código a ser probada.

12.1.5 *Herramientas*

Las **herramientas** son aplicaciones que apoyan diversos aspectos en la administración del proceso de software. El conjunto de estas herramientas se conoce como **ingeniería de software asistida por computadora** (en inglés, *CASE—Computer-Aided Software Engineering*), cuyo objetivo es asistir al desarrollador durante las diferentes actividades del ciclo de vida del proceso de software. Las herramientas varían en su apoyo a los procesos, integrando componentes como editores de texto, generadores de modelos gráficos (diagramas), generadores de código, compiladores, depuradores, verificadores, validadores, medidores (monitores), administradores de configuración y administradores del proyecto. Las herramientas CASE son indispensables en la administración del proceso de software. La selección de estas herramientas debe considerar el apoyo a las metodolo-

[4] Goldberg, A. y Rubin, K., 1995, *Succeeding with objects: decision framework for project management (first edition)*, Addison-Wesley.

gías utilizadas. Se debe tomar en cuenta si proveen apoyo explícito para cada paso del método, administran toda la información que el método requiere obtener o especificar, permiten manejar grandes cantidades de información y hacer proyectos escalables, incluyen un mecanismo que permita probar que la información recolectada es coherente, apoyan la organización de los diagramas de manera automática, permiten usuarios simultáneos en uno o más proyectos, pueden generar una implementación inicial junto con la documentación y apoyan la ingeniería en reversa para asegurar que los cambios directos en la implementación sean coherentes con los modelos administrados.

12.2 Modelos clásicos

Los modelos de proceso dependen de las opiniones o creencias de las personas involucradas en un proyecto.[5] Por ejemplo, algunas de estas opiniones o creencias implican que es necesario comprender el problema antes de desarrollar una solución, el proceso para resolver un problema debe dar un resultado predecible (sin importar qué individuo hace el trabajo), es indispensable planear y calcular el proceso con gran precisión, para que un proceso tenga éxito es importante evaluar y administrar el riesgo y la entrega de etapas intermedias bien definidas aumenta la confianza que se tiene en el resultado final.

A continuación se describen los modelos de procesos "clásicos", analizando las creencias en las cuales se basan.

12.2.1 *Modelo de cascada*

El **modelo de cascada** original se desarrolló entre las décadas de los años 60 y 70[6] y se define como una secuencia de actividades, donde la estrategia principal es seguir el progreso del desarrollo de software hacia puntos de revisión bien definidos (en inglés, **milestones** o **checkpoints**) mediante entregas programadas con fechas precisas (en inglés, **schedule**). La figura 12.2 muestra un diagrama del modelo de cascada que describe el orden de las actividades del desarrollo de software. No se muestra una etapa explícita de documentación dado que ésta se lleva a cabo en el transcurso de todo el desarrollo. El modelo original planteaba que cada actividad debía completarse antes de poder continuar con la siguiente actividad. Sin embargo, en una revisión posterior se extendió el modelo, permitiendo regresar a actividades anteriores.[7]

Algunas de las creencias del modelo de cascada son que las metas se logran mejor cuando se tienen puntos de revisión bien preestablecidos y documentados (dividiendo el desarrollo del software en actividades secuenciales bien definidas), los documentos técnicos son comprensibles para usuarios y administradores no técnicos, cada detalle de los requisitos se conoce de antemano antes de desarrollar el software (dichos detalles son estables durante el desarrollo) y las pruebas y evaluaciones se realizan de manera eficiente al final del desarrollo.

El modelo de cascada fue inicialmente bien recibido, dado que las actividades de las etapas eran razonables y lógicas. Lamentablemente, el modelo no explicaba cómo modificar un resultado, en especial considerando lo difícil que es definir todos los requisitos de un sistema desde el inicio y que éstos se mantengan estables y sin cambios durante el desarrollo. Además, toma demasiado tiempo en obtener resultados, retrasando la detección de errores hasta el final. El modelo también hace difícil rastrear (en otras palabras, ver la dependencia entre los requisitos iniciales y el código final). Esta rigidez trajo dudas sobre la utilidad del modelo, resultando en que éste dejase de utilizarse de acuerdo con su defini-

[5] *Idem*.

[6] Royce, W., 1970, Managing the development of large software systems: concepts and techniques, *Proceedings WESTCON*, IEEE Computer Society Press, San Francisco, CA.

[7] Boehm, B., 1981, *Software engineering economics*, Englewood Cliffs, NJ: Prentice Hall.

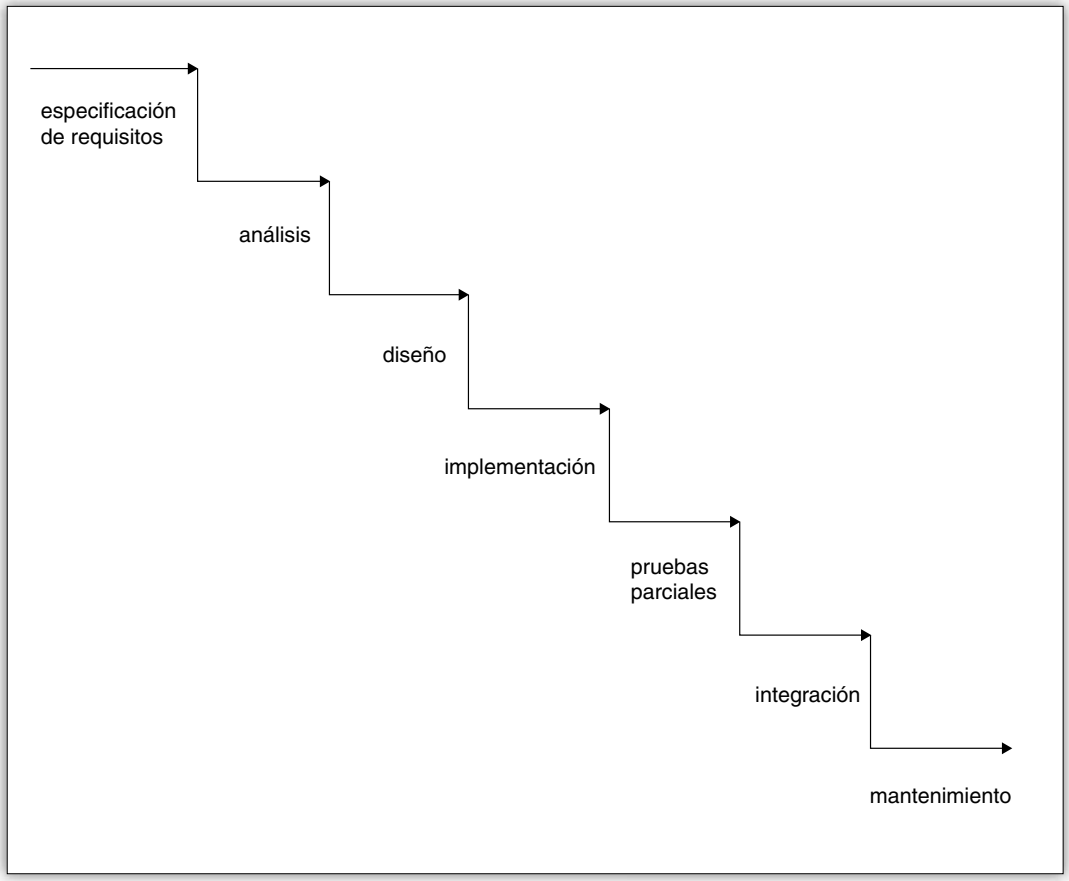

Figura 12.2 Secuencia de actividades para el modelo de cascada.

ción original, llevando a los desarrolladores a utilizar variantes del modelo básico, incluyendo el uso de prototipos y la reutilización de software.[8]

12.2.2 *Modelo incremental*

El **modelo incremental** consiste de un desarrollo inicial de la arquitectura completa del sistema, seguida de incrementos y versiones parciales de éste.[9] Cada incremento tiene su propio ciclo de vida, típicamente siguiendo el modelo de cascada. Los incrementos pueden construirse de manera serial o paralela dependiendo de la naturaleza de la dependencia entre versiones y recursos. Cada incremento agrega funcionalidad adicional o mejorada sobre el sistema. Conforme se completa cada etapa, se verifica e integra la última versión con las demás versiones ya completadas del sistema. Durante cada incremento, el sistema se evalúa con respecto al desarrollo de versiones futuras. Las actividades se dividen en procesos y subprocesos, dando lugar al término **fábrica de software** (en inglés, **software factory**). Para que la secuencia de desarrollo sea exitosa, es esencial definir etapas que no

[8] Yourdon, E., 1992, *Decline and fall of the american programmer*, Prentice Hall.
[9] Boehm, B., 1981, *Software engineering economics*, Englewood Cliffs, NJ: Prentice Hall.

requieran cambiar los resultados anteriores al agregar nuevas. Por lo tanto, es importante comprender al inicio los requisitos completos del sistema, algo que normalmente es muy difícil de lograr. El desarrollo incremental evita la teoría del "big bang" para el desarrollo de software, donde una gran explosión de desarrollo se transforma repentinamente en el sistema final.

Algunas de las creencias del modelo de cascada son que la administración de proyectos es más fácil de lograr en incrementos más pequeños, es más fácil comprender y probar incrementos de funcionalidad más pequeños, la funcionalidad inicial se desarrolla más temprano (logrando resultados de inversión en menor tiempo) y hay más probabilidad de satisfacer el cambio en los requisitos de usuario mediante incrementos del software en el tiempo que si fueran planeados todos a la vez en un mismo periodo.

12.2.3 *Modelo evolutivo*

El **modelo evolutivo** es una extensión al modelo incremental en la que los incrementos se hacen de manera secuencial, en lugar de en paralelo.[10] Desde el punto de vista del cliente, el sistema evoluciona según vayan siendo entregados los incrementos. Desde el punto de vista del desarrollador, los requerimientos que son claros al principio del proyecto dictan el incremento inicial, mientras que los incrementos para cada uno de los siguientes ciclos de desarrollo se clarifican a través de la experiencia de los incrementos anteriores. Este modelo considera que el desarrollo de sistemas es un proceso de cambios progresivos mediante deltas (cambios) de especificación de requerimientos. En la figura 12.3 se ilustra este concepto.

El modelo evolutivo es también conocido como **desarrollo rápido de aplicaciones** (en inglés, **RAD—rapid application development**), que se basa tradicionalmente en el uso de prototipos (en inglés, **rapid prototyping**). Un prototipo de software se considera como un medio para especificar los requisitos y un enlace de comunicación entre el usuario final y el diseñador, ayudando a reducir el riesgo de carecer de requerimientos iniciales completos y estables.

Algunas de las creencias del modelo de cascada son la entrega temprana de parte del sistema, aunque no estén completos todos los requerimientos, ya que esto permite utilizarlo como herramienta para la generación de requerimientos faltantes, obteniendo be-

[10] Boehm, B., 1987, A spiral model of software development and enhancement, *Software engineering project management*, IEEE.

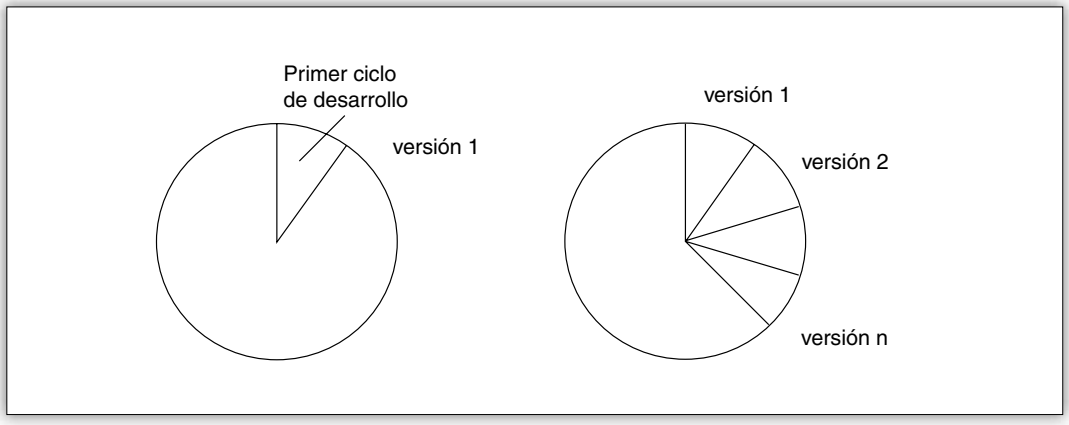

Figura 12.3 Secuencia de versiones en el modelo evolutivo.

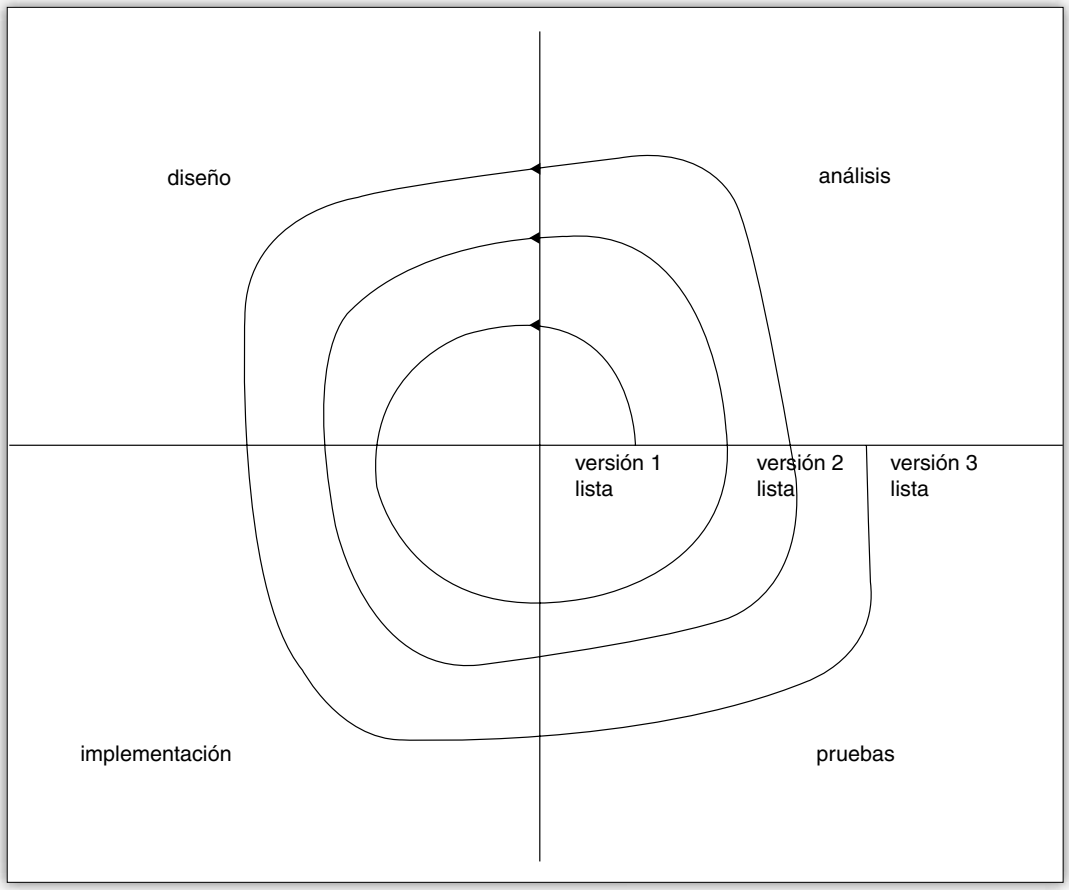

Figura 12.4 Secuencia de actividades para el modelo espiral.

neficios para el sistema mediante entregas iniciales mientras las entregas posteriores se encuentran en desarrollo.

12.2.4 *Modelo espiral*

El **modelo espiral**,[11] desarrollado durante la década de los años 80, es una extensión del modelo de cascada. A diferencia del modelo de cascada, que es dirigido por documentos, el modelo espiral se basa en una estrategia para reducir el riesgo del proyecto en áreas de incertidumbre, como tener requerimientos iniciales incompletos e inestables. 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, soluciones alternas y restricciones asociadas con cada alternativa, y finalmente se procede a su evaluación. Cuando se encuentra que existe cierta incertidumbre, se utilizan diversas técnicas para reducir el riesgo de las distintas alternativas. Cada ciclo del modelo espiral termina con una revisión que discute los logros actuales y los planes para el siguiente ciclo. La figura 12.4 muestra un diagrama conceptual del modelo espiral.

Al igual que el modelo evolutivo, el modelo espiral incorpora una estrategia de uso de prototipos como parte del manejo del riesgo.

[11] *Idem*.

Las creencias del modelo de espiral son que una actividad comienza cuando se entienden los objetivos y riesgos involucrados, se usan las herramientas que mejor reduzcan los riesgos basándose en la evaluación de soluciones alternas, todo el personal relacionado debe involucrarse en una revisión que determine cada actividad (planeando y comprometiéndose con las siguientes actividades) y el desarrollo se incrementa en cada etapa, permitiendo prototipos sucesivos del producto. Con algunas variantes, éste es el modelo de proceso más importante en la actualidad.

12.3 Modelos nuevos

En esta sección se describen algunos de los modelos de procesos "nuevos" que surgieron después de los clásicos.

12.3.1 *Modelo ganar-ganar*

El modelo **ganar-ganar** (en inglés, ***win-win***)[12] extiende el modelo espiral, haciendo énfasis en la identificación de las condiciones de ganancia para todas las partes, creando un plan para alcanzar las condiciones ganadoras y evitar los riesgos correspondientes. Se establecen las reglas para definir el proceso de desarrollo del proyecto, tomando en cuenta todas las partes implicadas. El modelo no necesita mucho tiempo de gestión. Esto permite utilizarlo tanto en proyectos pequeños como grandes.

Se consideran cuatro los ciclos, cada uno compuesto de cuatro actividades. Las cuatro actividades son: elaborar los objetivos, restricciones y alternativas del proceso y producto del sistema y subsistema; evaluar las alternativas con respecto a los objetivos y restricciones (identificando y resolviendo las fuentes principales de riesgo en el proceso y producto); elaborar la definición del producto y proceso; y planear el siguiente ciclo, actualizando el plan del ciclo de vida, incluyendo la partición del sistema en subsistemas para ser considerados en ciclos paralelos, lo cual puede incluir un plan para terminar el proyecto si es muy riesgoso o no es factible, asegurando el compromiso de la administración para continuar según lo planeado.

Una vez revisadas las actividades, los ciclos definen líneas específicas a seguir. En el Ciclo 0 (grupos de aplicación) se determina la viabilidad de un grupo apropiado de aplicaciones. En el Ciclo 1 (objetivos del ciclo de vida de la aplicación) se desarrollan los objetivos del ciclo de vida, incluyendo prototipos, planes y especificaciones de aplicaciones individuales, y se verifica la existencia de al menos una arquitectura viable para cada aplicación. En el Ciclo 2 (arquitectura del ciclo de vida de la aplicación) se establece una arquitectura del ciclo de vida detallado, se verifica su viabilidad, y se determina que no existen riesgos mayores en satisfacer los planes y especificaciones. En el Ciclo 3 (capacidad de operación inicial) se alcanza una capacidad operacional inicial para cada etapa crítica del proyecto en el ciclo de vida del software.

Las creencias del modelo son: crear software basado en componentes para lograr mayor calidad en los sistemas de mayor tamaño, escribir software reutilizable para hacer eficiente el proceso de desarrollo, medir la calidad del sistema como aspecto clave del desarrollo del producto, lograr mayor calidad en el proceso de ensamblaje a partir de componentes menores, usar tecnologías basadas en objetos como aspecto básico para lograr la calidad, producir sistemas rápidamente (sencillos, confiables y de calidad) empleando procesos bien definidos, utilizar el modelo espiral como base del proceso, hacer flexible el proceso de creación del software para lograr los objetivos generales de eficiencia, involucrar al cliente mediante el manejo de prototipos y analizar los riesgos en el proceso del desarrollo del software para asegurar la calidad final del sistema.

[12] Boehm, B., Egyed, A., Port, D., Shah, A., Kwan, J. y Madachy, R., "A stakeholder win-win approach to software engineering education", *Annals of software engineering*, enero de 1999.

12.3.2 *Proceso unificado (UP)*

El **proceso unificado** (en inglés, **UP—unified process**)[13] es una extensión al proceso objectory (del inglés *object factory*),[14] que tiene sus orígenes en la década de los años 80. Estos modelos de proceso se basan principalmente en la especificación de requerimientos de un sistema mediante casos de uso (maneras de utilizar un sistema). El proceso unificado considera como aspecto esencial del desarrollo de software una visión que parte de la arquitectura del sistema, siguiendo un proceso iterativo e incremental. El proceso considera e integra diferentes aspectos, como son los ciclos, fases, flujos de trabajo, mitigación de riesgo, control de calidad, administración de proyecto y control de configuración. De manera adicional, el proceso unificado considera las cuatro P del desarrollo de software: personas, proyecto, producto y proceso.

El proceso unificado tiene las siguientes creencias: para construir un sistema exitoso se debe conocer qué quieren y necesitan los usuarios potenciales; al igual que la arquitectura, en la construcción, permite diseñar edificios desde múltiples puntos de vista, estructura, electricidad, etc., las arquitecturas de los sistemas de software deben permitir visualizar un sistema desde múltiples perspectivas; y el desarrollo de un producto de software comercial puede significar un gran esfuerzo continuando por meses, años o incluso más, por lo que es práctico dividir el trabajo en pedazos, donde cada iteración resulta en un incremento del proyecto.

12.4 Proceso personal y de equipo para el desarrollo de software

Para lograr buenos resultados a nivel de las empresas desarrolladoras de software, primero es necesario que cada uno de sus ingenieros alcancen resultados de excelencia y luego que sus grupos de trabajo también lo hagan. En el caso del desarrollo de software de alta calidad, que responda a las expectativas de los usuarios, es fundamental definir procesos tanto personales como para el trabajo en equipo. En cuanto al proceso personal, aquí se explican dos de los más importantes, el **PSP** (**Personal Software Process**) y el **XP** (**Extreme Programming**). Con respecto al proceso de trabajo en equipos se explica el **TSP** (**Team Software Process**).

12.4.1 *PSP (Personal Software Process)*

El proceso personal (Humphrey, 2004a) consiste en especificar la secuencia de pasos requeridos por un programador para desarrollar o mantener software. Un proceso bien definido proporciona el marco técnico y administrativo para la aplicación de metodologías, herramientas y el esfuerzo personal en el desarrollo de software. Éste determina los papeles y las tareas que deben llevarse a cabo.

PSP propone un proceso. Sin embargo, cada individuo debe adaptarlo a su propia situación. Las principales ideas en las que se basa PSP son:

1. Los desarrolladores entienden mejor lo que hacen cuando definen, miden y controlan su trabajo.
2. Tener un proceso e indicadores bien definidos favorece la evaluación y el aprendizaje a partir de las propias experiencias y de las ajenas.
3. Con el conocimiento y la experiencia se pueden seleccionar aquellos métodos y prácticas que mejor se adapten a las habilidades particulares de cada ingeniero.

[13] Jacobson, I., Booch, G., y Rumbaugh, J., 1999, *The unified software development process*, Addison Wesley.

[14] Jacobson, C., Christensen, M., Jonsson, P., y Overgaard, G., 1992, *Object-oriented software engineering: a use-case driven approach*, Addison-Wesley.

4. Con el uso de un conjunto de prácticas personales de alta calidad, cada individuo se torna en un miembro eficaz de su equipo de trabajo.

12.4.1.1 Los procesos de PSP

En la figura 12.5 (Humphrey, 2004c) se presenta un esquema del proceso personal de desarrollo de software. Es un proceso evolutivo: se parte del estado actual del desarrollador, se van agregando elementos, dando lugar a nuevos procesos, y así se continúa hasta llegar a la última etapa en la cual se repite el proceso personal cíclicamente. A continuación se describe con más detalle cada uno de los subprocesos.

PSP0 es el proceso de desarrollo en sí, incluyendo mediciones básicas que permiten

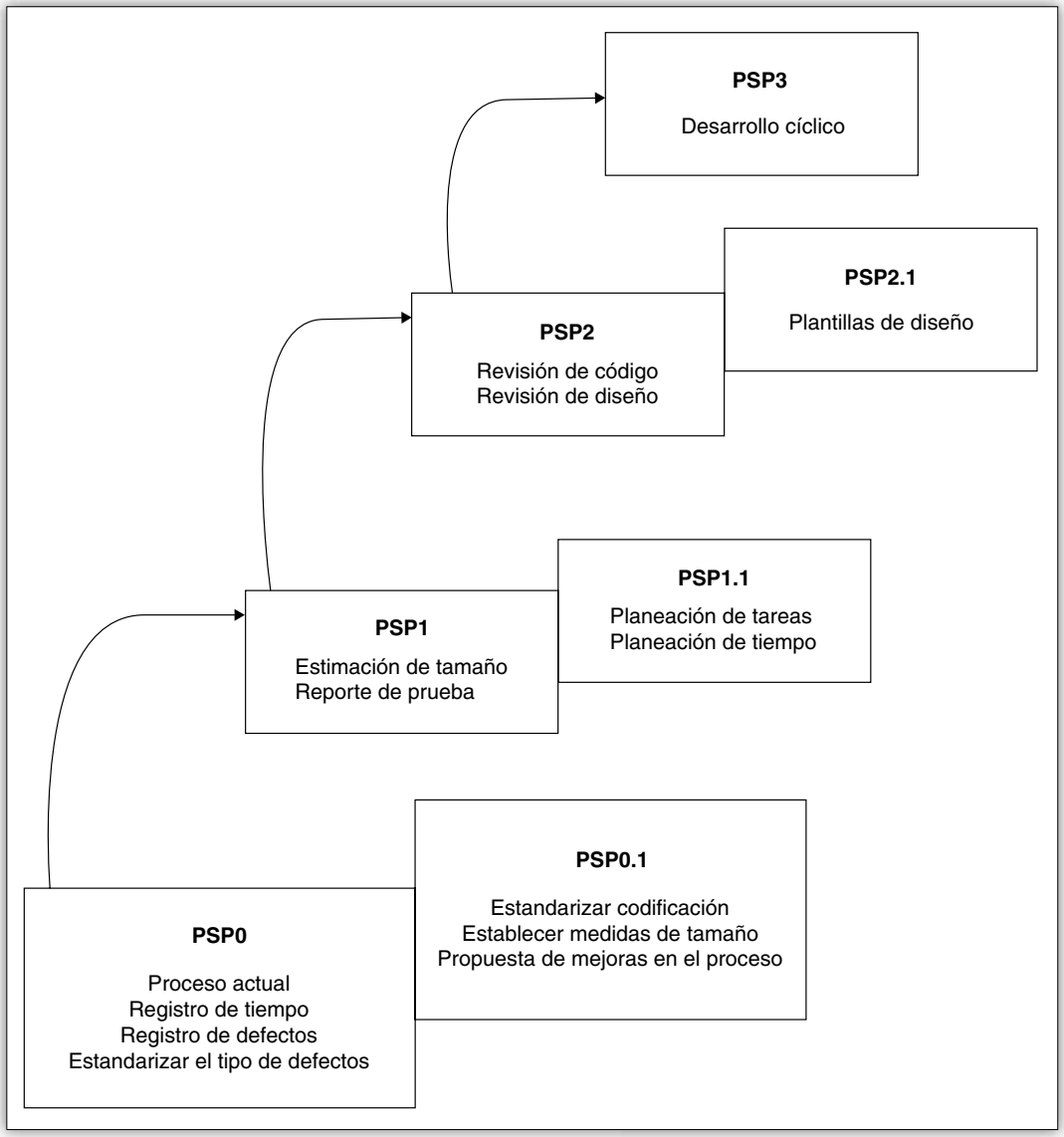

Figura 12.5 La evolución de PSP.

identificar aspectos a mejorar y llevar un seguimiento del progreso alcanzado. Se registra el tiempo y los defectos. Para esto último se requiere definir un estándar de tipos de defectos o se puede tomar el propuesto por PSP.

PSP0.1 se obtiene agregando al PSP0 la estandarización del código, medidas de tamaño y una propuesta para el mejoramiento del proceso. Es importante registrar los problemas encontrados en el proceso y sugerencias para mejorarlo.

En PSP1 se incorpora la actividad de planeación. Se incluye la estimación de recursos y tiempo. Además, se empiezan a realizar reportes de las pruebas efectuadas a los programas desarrollados.

Para alcanzar el nivel de PSP1.1 se agrega la planeación de tareas y el calendario de entregas. La planeación permite al programador entender la relación entre el tamaño de los programas que desarrolla y el tiempo que requiere para ello, establecer metas que puede alcanzar y poder saber en cada momento cuál es el estado de avance de su trabajo.

En PSP2 el énfasis está en la calidad de los productos que se elaboran. Para poder controlar los defectos introducidos en el código, primero se necesita saber cuántos se introducen y de qué tipo son. Con este conocimiento se elaboran listas de los defectos más frecuentes. Teniendo estas listas, se establecen revisiones e inspecciones de diseño y de código, de tal manera que los defectos se detecten en una etapa inicial. Cuanto antes se detecte un defecto, más barato resulta su corrección.

En PSP2.1 se incluyen plantillas de diseño que ayudan a garantizar que el diseño esté completo y sea coherente con las necesidades planteadas. El PSP no dice cómo realizar el diseño; para ello se puede usar cualquiera de las metodologías existentes.

En la etapa PSP3 se plantea un proceso cíclico, útil para el desarrollo de programas o módulos muy grandes. El problema debe subdividirse de tal manera que cada uno de los subproblemas quede en un nivel PSP2.

PSP se apoya fundamentalmente en la planeación del proyecto y en la calidad con la que se lleva a cabo. El mejoramiento continuo del proceso se logra gracias al uso adecuado de los datos obtenidos en experiencias previas, por lo cual se requiere el registro constante de datos durante todas las etapas de PSP. Es importante mencionar que los datos recolectados por los programadores durante las distintas etapas del desarrollo deben ser respetados como propiedad de los programadores. No deben ser usados por los administradores para establecer premios o castigos.

12.4.1.2 Planeación

En el plan del proyecto se define el trabajo y la manera en la que se va a realizar. Asimismo, se especifican las principales tareas, así como una estimación del tiempo y de los recursos requeridos para realizarlas. Para elaborar un plan se recomienda tener en cuenta los siguientes puntos: 1. definir para desarrollar un producto específico; 2. revisar con el cliente; 3. dividir en varias etapas bien definidas y que se puedan evaluar, y 4. debe servir para medir el progreso que se vaya alcanzando en el proyecto.

Considerando lo anterior, un buen plan proporciona información valiosa tanto para el desarrollador como para el cliente. Al desarrollador, el plan le indica todo lo correspondiente al tamaño del trabajo, las etapas en las que lo va a desarrollar, el estado del trabajo en grado de avance y la precisión o no con la que se hizo la planeación de las tareas y de los recursos. Al cliente, el plan le indica todo lo correspondiente al producto que recibirá, el tiempo y el costo involucrados, la evaluación de los progresos y la manera en la que va a ser desarrollado el proyecto.

Para poder planear un proyecto se requiere poder estimar el tamaño del sistema. El método elegido para hacer la estimación debe ser automático, preciso y útil para la planeación. En PSP se usa el método **PROBE** (**PROxy-Based Estimating**). Además, se debe establecer un estándar para el conteo de líneas de código, de funciones o de los

elementos que se vayan a contar para estimar el tamaño.

Otro elemento importante que interviene en la planeación es la estimación del tiempo requerido para desarrollar el producto. En PSP la selección del método para estimar el tiempo depende de la cantidad de datos históricos disponibles.

12.4.1.3 Administración de la calidad

La estrategia seguida por PSP para asegurar la calidad del producto desarrollado se lleva a cabo a través del manejo de la cantidad de defectos insertados en el software. Si se controlan los defectos en cada unidad de software, se puede garantizar la calidad de todo el sistema una vez que las partes sean integradas. La calidad del producto está estrechamente ligada con la calidad del proceso seguido para su producción. Para determinar el nivel de calidad del proceso usado, se deben establecer métricas y hacer seguimientos del proceso.

La calidad también se mide por el servicio prestado al usuario. Este servicio incluye si el sistema es fácil de usar, de instalar y de mantener, si es seguro tanto al protegerse contra los mismos usuarios como de externos, si su desempeño es aceptable y si está acompañado de toda la documentación necesaria. PSP se centra en los defectos, ya que si un sistema de baja calidad entra a la etapa de pruebas es casi seguro que se descuidarán los otros aspectos relacionados con la calidad por detectar y corregir todos los defectos. En el área de la ingeniería de software, la mayoría de los defectos de un programa proviene de errores cometidos por los individuos que lo desarrollaron. Generalmente un proceso de baja calidad produce un producto de baja calidad. De ahí la insistencia de PSP de que cada desarrollador defina un proceso propio de alta calidad.

Un PSP de calidad es aquel que se ajusta a las características del desarrollador y le permite producir software de excelente nivel de manera sistemática. Además, debe ser fácil de entender y adaptar. Cada desarrollador debe definir su propio proceso de calidad. Para ello debe primero definir un proceso y posteriormente medirlo, darle seguimiento y mejorarlo de manera continua. Se recomienda la siguiente estrategia para asegurar la calidad del proceso:

1. Medir la calidad del proceso a través de la calidad del producto, de la productividad alcanzada, del total de defectos corregidos, entre otros.
2. Incorporar los mejores métodos de calidad (por ejemplo, revisiones e inspecciones), encontrar las causas de los errores que se cometen, usar los mejores métodos para el diseño, entre otros.
3. Evaluar periódicamente la estrategia y establecer nuevos objetivos, obedeciendo al carácter dinámico de PSP en cuanto a la mejora del proceso.

12.4.1.4 El uso de PSP

La adopción de PSP es eficaz cuando existe un compromiso por parte de toda la organización, o al menos de un grupo de la misma. Inicialmente implica cambiar los hábitos de trabajo, recopilar datos, realizar planes, escribir reportes y realizar otras actividades que aquellos que no están familiarizados con el proceso podrían considerar como una pérdida de tiempo. Además, la organización que está desarrollando el proyecto debe dar apoyo para formar la base de datos y llevar a cabo el análisis de los datos recolectados. Por lo tanto, es importante que los desarrolladores y administradores entiendan los beneficios que ofrece PSP y colaboren para que el personal se capacite y defina su propio proceso.

En (Humphrey 2004c) se señalan los costos y beneficios relacionados con el uso de PSP. Con respecto a los costos, en primer lugar se indica el tiempo necesario para aprender a usarlo. Se requiere tiempo para aprender PSP y para registrar los datos, tarea que debe hacerse continuamente para poder contar con

información válida cuando haya que hacer planes y estimaciones, y para ir mejorando de manera permanente el proceso propio de desarrollo. Definir y usar un proceso personal puede ocasionar un costo emocional. Se requiere de mucha disciplina para usar un proceso, autoevaluarse y detectar aquellas áreas en las que se puede mejorar. Una vez detectadas las oportunidades de mejoras, se deben establecer objetivos y las medidas que lleven a alcanzar esos objetivos. Por último, se pone en riesgo el ego. Junto al costo emocional de estar buscando una mejora continua está el efecto que esto puede tener sobre la imagen que cada uno tiene acerca de sí mismo. Con el uso de PSP queda explícito el desempeño de cada persona. Sin embargo, es importante señalar que para el desarrollo de software son necesarias diferentes habilidades, conocimientos y aptitudes. Por lo tanto, cada uno debe identificar sus fortalezas y tratar de mejorar las áreas menos fuertes por medio de la experiencia.

Los beneficios son muchos, tanto a nivel personal como a nivel de la organización. En cuanto a los beneficios personales, el desarrollador puede conocer y entender sus propias fortalezas y debilidades, y aprovechar las primeras y buscar superar las últimas. Además, define su propio proceso sobre el que tiene control, y está en condiciones de mejorarlo continuamente y en consecuencia mejorar su desempeño. Cuenta con datos para hacer planes más certeros. Con estos planes puede hacer seguimientos sobre su trabajo, lo cual le permite administrarlo mejor. A nivel de la organización, si PSP es adoptado por todos los programadores, éstos pueden definir un proceso de desarrollo en equipo de mejor calidad. Cada miembro del equipo puede hacer su trabajo sin depender tanto del desempeño de sus compañeros. La organización puede hacer planes más certeros y tener más elementos para negociar con sus clientes: puede planear el tamaño del producto que va a entregar, la calidad del mismo, el tiempo que requiere para su desarrollo y la cantidad de personas que asigna para trabajar en el proyecto.

En la actualidad ya no es posible seguir desarrollando software de manera artesanal, como se solía hacer al inicio de la computación. La industria del software requiere que cada uno de los involucrados haga un trabajo profesional. Los desarrolladores, además de usar PSP, deben utilizar métodos eficaces, tener presente sus fortalezas y debilidades, practicar mucho, aprender de sus experiencias y estar dispuestos a seguir mejorando.

12.4.2 *TSP (Team Software Process)*

El TSP (Humphrey, 2004b) es un proceso de desarrollo de software en equipo. El TSP ofrece un marco de trabajo para los equipos encargados de desarrollar grandes sistemas. Dedica atención al proceso, al producto y al trabajo en grupo. Además, basado en experiencias, guía cómo planear y administrar proyectos de software de gran tamaño que requieren ser realizados por grupos de programadores. Un equipo no se forma simplemente con la unión de varios programadores. Se necesita definir objetivos comunes, un plan de acción aceptado por todos, tener liderazgo adecuado, conocer y respetar las debilidades y fortalezas de cada miembro, ayudarse entre compañeros y ser capaz de pedir ayuda cuando se requiera. Debido a esto, resulta de fundamental importancia definir un proceso para el trabajo en equipo.

Para que el grupo de desarrollo pueda ir mejorando su proceso, TSP, al igual que PSP, requiere del registro continuo de los datos del proyecto. Los datos recopilados, a nivel personal y de equipo, sirven para hacer estimaciones más precisas de tiempo y tamaño en el futuro, así como para conocer el desempeño de los involucrados e implementar medidas para lograr mejoras. TSP provee un conjunto de formas que sirven para ese registro.

TSP está diseñado para grupos de por lo menos veinte ingenieros de software dedicados a proyectos de gran tamaño que requieren de varios años de desarrollo. Se define TSPi (introductory Team Software Process)

como una versión reducida de TSP, que mantiene sus mismos principios, y a partir de la cual se puede escalar fácilmente a TSP. En este libro se describe TSPi. Tanto TSP como TSPi presuponen que todos los miembros del equipo conocen PSP.

12.4.2.1 Los principios de TSPi

TSPi está basado en cuatro principios fundamentales:

1. El aprendizaje es más eficaz si se sigue un proceso definido y se recibe retroalimentación. TSPi provee un marco de trabajo que cumple con estas características. Además, cuenta con mediciones establecidas y está diseñado para realizarse de manera cíclica. El uso de ciclos pequeños permite que el equipo reciba información sobre su desempeño periódicamente. Es decir, el trabajo del grupo se evalúa al terminar cada ciclo y los resultados se analizan y utilizan para mejorar el desempeño del mismo.
2. Para que el trabajo en equipo sea productivo, se requiere definir objetivos, un ambiente de trabajo apropiado y liderazgo adecuado.
3. Es importante recibir la guía apropiada para encontrar soluciones eficaces a los problemas de desarrollo que se originen. TSPi ayuda a definir papeles, métodos y prácticas adecuadas para cada grupo de trabajo.
4. La instrucción es más eficaz cuando se construye sobre la base de conocimientos previamente adquiridos. TSPi se basa en el conocimiento y las experiencias que existen sobre equipos de desarrolladores de software y material disponible sobre este tema.

12.4.2.2 La estructura y el flujo de TSPi

TSPi está formado por ocho procesos: lanzamiento, estrategia, plan, requerimientos, diseño, implementación, prueba y postmortem. Además, usa múltiples ciclos para construir un producto final. En cada uno de los ciclos se aplican los ocho procesos mencionados. Se inicia con una junta de lanzamiento, en la que se presentan los objetivos del producto. Posteriormente, se aplican los otros siete procesos. En el siguiente ciclo se aplican los mismos procesos, pero trabajando sobre lo que haya sido desarrollado en el ciclo anterior, logrando que el producto vaya aumentando sus funcionalidades. La cantidad de ciclos depende del tamaño del proyecto y del tiempo que se disponga. En la figura 12.6 se presenta un esquema de la estructura y del flujo de TSPi.

Para usar una estrategia de desarrollo cíclica se requiere definir el producto básico que se ha de generar con el primer ciclo. Se recomienda que sea una versión pequeña del producto, pero que incluya alguna funcionalidad que dé valor al negocio. Para decidir el contenido y tamaño de cada ciclo se debe tener en cuenta que debe producir una versión que pueda probarse y que represente un subconjunto de la versión final del producto, que debe ser lo suficientemente pequeño para que pueda desarrollarse y probarse en el tiempo disponible y que la combinación de los productos obtenidos en cada ciclo debe dar el producto final deseado.

12.4.2.3 Los procesos de TSPi

Los ocho procesos de TSPi aplicados de manera cíclica permiten a los equipos de ingenieros de software alcanzar productos de alta calidad en el tiempo estimado. TSPi provee una guía en la que se presenta el objetivo, el criterio de entrada, las etapas (con las actividades que involucra cada una de ellas) y el criterio de salida de cada proceso. El criterio de entrada es todo aquello que necesita el proceso para llevarse a cabo y el criterio de salida es el producto o los productos generados por el proceso. Asimismo, la guía proporciona unas formas para el registro de todos los datos relevantes del proyecto. Estos da-

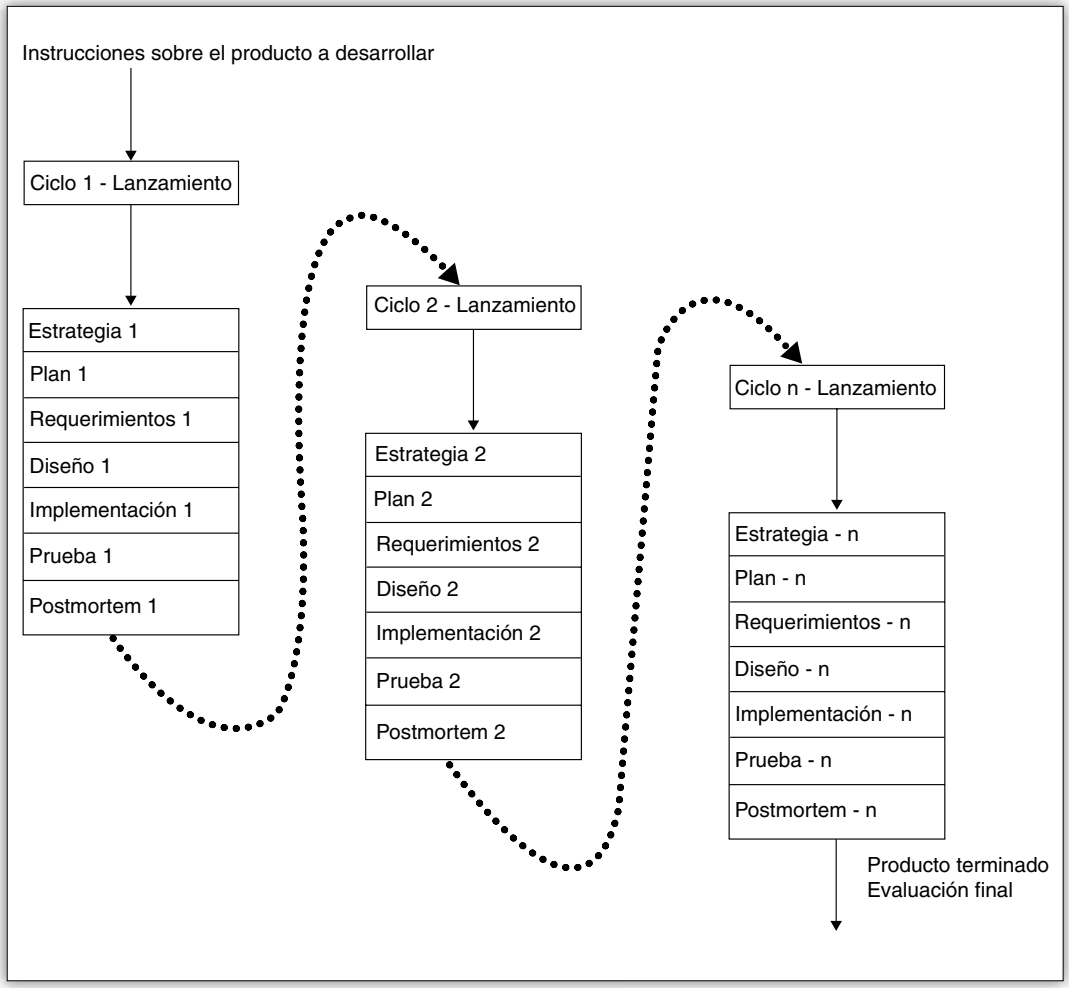

Figura 12.6 Estructura y flujo de TSPi.

tos sirven para evaluar el desempeño del equipo, de cada uno de sus integrantes y definir posibles mejoras al proceso de desarrollo.

12.4.2.3.1 *Lanzamiento*

El establecimiento del equipo de trabajo se lleva a cabo en la etapa de lanzamiento. Se determinan las relaciones de trabajo, los roles de los miembros, quién va a ocupar cada uno de ellos y se llega a un acuerdo sobre los objetivos.

Los roles básicos propuestos por TSPi son: líder de equipo, administrador de desarrollo, administrador de planeación, administrador de la calidad del proceso y administrador de soporte. Los miembros del equipo a quienes se les asignan estos roles realizan también actividades propias de los ingenieros de software.

Establecer objetivos es de fundamental importancia. Se requiere que los mismos estén bien definidos y sean cuantificables. Si no se pueden medir, entonces resulta difícil determinar con precisión la calidad del producto generado y del proceso seguido por el equipo. Se deben fijar objetivos ambiciosos pero realistas, de tal manera que motiven a los integrantes del grupo. Para fijar objeti-

vos con estas características, es importante contar con experiencias previas. Cuando no se cuente con esta experiencia, TSPi sugiere adoptar, como objetivos básicos, elaborar un producto de calidad, llevar a cabo un proyecto productivo y bien administrado, y terminar el proyecto a tiempo. Cada objetivo debe acompañarse de métricas para que pueda evaluarse en qué grado fue alcanzado. En la tabla 12.3 se presentan los objetivos propuestos por TSPi con sus correspondientes métricas.

Después de cada ciclo se debe evaluar el desempeño y establecer objetivos para mejorar el proceso en el siguiente ciclo. Luego se deben definir los cambios necesarios para que los objetivos planteados puedan ser alcanzados.

Además de los objetivos para el equipo, se deben establecer objetivos para cada uno de los miembros del mismo. Si no se cuenta con experiencia previa, TSPi sugiere adoptar los objetivos y las métricas que se presentan en la tabla 12.4.

También se deben establecer objetivos para cada uno de los roles definidos en el equipo. Es importante tener presente que, en caso de que haya un conflicto entre los objetivos del equipo y los personales o de papeles, la prioridad más alta la tiene el objetivo del equipo.

12.4.2.3.2 *Estrategia*

Una vez completada la etapa de lanzamiento, se debe fijar una estrategia para llevar a cabo el proyecto. En esta etapa se debe idear una estrategia para realizar el trabajo, crear un diseño conceptual del producto, y hacer una estimación preliminar del tamaño del producto y del tiempo de desarrollo. El tiempo estimado debe corresponder al tiempo disponible. De no ser así, se debe revisar y ajustar la estrategia hasta que los tiempos coincidan.

TSPi propone un proceso cíclico en el que cada ciclo toma la versión del sistema generada en el ciclo anterior y produce una versión aumentada. La cantidad total de ciclos

Tabla 12.3 Objetivos propuestos para el equipo y sus indicadores.

Objetivos	Indicador 1	Indicador 2	Indicador 3
Producir un producto de calidad	El porcentaje de defectos encontrados antes de la primera compilación debe ser 80%.	La cantidad de defectos encontrados en la prueba del sistema debe ser 0.	Al terminar el proyecto se debió haber incluido el 100% de las funcionalidades requeridas.
Llevar a cabo un proyecto productivo y bien administrado	El error en la estimación del tamaño del producto debe ser menor a 20%.	El error en la estimación del número de horas de desarrollo debe ser menor a 20%.	El porcentaje de datos del proyecto registrados debe ser 100%.
Terminar el proyecto a tiempo	El total de días de retraso o adelanto para completar el ciclo debe ser menor a 4.		

Tabla 12.4 Objetivos propuestos para los miembros del equipo y sus indicadores.

Objetivos	Indicador 1	Indicador 2	Indicador 3	Indicador 4
Ser un miembro del equipo cooperativo y efectivo	El promedio de la evaluación de cada miembro del equipo, efectuada por los pares acerca de la disposición a ayudar y la aportación del trabajo, debe ser mayor a 3.	El promedio de la evaluación de cada miembro del equipo, efectuada por los pares acerca de la contribución general hecha al equipo, debe ser mayor a 3.		
Realizar un trabajo personal disciplinado	El porcentaje de los datos personales registrados debe ser 100%.	El porcentaje de semanas registradas en el documento semanal debe ser 100%.		
Planear y dar seguimiento al trabajo personal	El porcentaje de los datos del proyecto personal registrados en las formas correspondientes debe ser 100%.	El porcentaje de las tareas del proyecto con plan y datos reales registradas en la forma correspondiente debe ser 100%.		
Producir productos de calidad	El porcentaje de defectos encontrados antes de la primer compilación debe ser mayor a 70%.	La densidad de defectos encontrados durante la compilación debe ser menor a 10/KLOC.	La densidad de defectos encontrados durante la prueba de unidad debe ser menor a 5/KLOC.	La densidad de defectos encontrados después de la prueba de unidad debe ser 0.

depende del tamaño del sistema y del tiempo disponible.

El diseño conceptual se requiere para poder hacer posteriormente la planeación del proyecto. Se debe lograr un diseño de alto nivel, definiendo la estructura general del producto. Para esto, TSPi sugiere contestar las siguientes preguntas:

1. Tomando en cuenta experiencias previas, ¿cómo se podría desarrollar este producto?

2. ¿Cuáles son los principales componentes que deben construirse para lograr este producto?
3. ¿Cuáles son las funciones que estos componentes deben proveer?
4. ¿Qué tan grande son estos componentes?

Hecho el diseño conceptual, se puede estimar el tamaño del producto así como el tiempo requerido para su desarrollo.

12.4.2.3.3 *PLAN*

El plan indica todas las actividades y el orden en el que deben realizarse. Proporciona el marco y el contexto de trabajo. Una vez definido el plan, se puede trabajar de manera más eficiente, pues se sabe qué hacer y cuándo hacerlo.

Las actividades planeadas entre los miembros del equipo deben ser distribuidas de manera equilibrada. Es decir, el trabajo asignado a cada miembro del equipo se debe poder realizar en el mismo periodo de tiempo. De esta manera no se ocasionan esperas inútiles para algunos miembros, mientras otros están saturados de trabajo.

Por otra parte, los planes permiten hacer seguimientos del trabajo y avances de cada miembro del equipo. Durante el desarrollo de un producto se realizan varias tareas que se planean para ser ejecutadas en un cierto tiempo. El seguimiento permite saber en cada momento cuáles tareas fueron completadas y en qué tiempo, y de esta manera saber si se está cumpliendo con el calendario establecido. Para que los retrasos no sean excesivamente costosos, es conveniente tener planes detallados. TSPi sugiere subdividir cada actividad en unidades que no consuman más de diez horas de trabajo. De esta forma, en caso de que una tarea se retrase no implicaría más de diez horas de atraso en todo el proyecto antes de que se detecte el problema.

Otro aspecto importante a tener en cuenta es que durante el desarrollo del proyecto pueden surgir algunas tareas no contempladas en el plan inicial. Por lo tanto, es conveniente incluir algunas horas en el plan para realizar esas tareas, especialmente en el primer ciclo.

El resultado de este proceso es el plan completo del equipo y de cada uno de los ingenieros que lo forman, incluyendo tareas, tiempos y calidad.

12.4.2.3.4 *REQUERIMIENTOS*

En esta etapa, las especificaciones de los requerimientos del sistema son generadas por el grupo de trabajo. Éstas deben ser claras y precisas. Además, el equipo debe definir indicadores que permitan evaluar el producto terminado para asegurar que éste cumpla con todas las especificaciones planteadas por el cliente.

Para obtener una buena especificación de los requerimientos, normalmente se necesita dedicar tiempo junto al usuario final del sistema, o un representante del mismo. Es importante que no queden dudas acerca de lo que se espera del sistema. Por lo tanto, a través de preguntas y respuestas los miembros del equipo junto al usuario deben ir definiendo y precisando todos aquellos aspectos que hayan quedado ambiguos. Algunas veces, los usuarios no tienen claro qué necesitan (o no lo saben expresar explícitamente), por lo que durante esta etapa se les debe ayudar a definir y precisar sus expectativas acerca del sistema. Una vez que se llegue a un acuerdo con el cliente sobre los requerimientos, cualquier cambio ocasionará un incremento en el costo y/o tiempo (y esto le debe quedar claro al cliente).

Los requerimientos son obtenidos durante una etapa de extracción en la que se pregunta al usuario final (y/o al cliente) acerca de sus necesidades. Algunas actividades propias de la obtención de los requerimientos son: evaluar la viabilidad del sistema, entender las características de la organización, evaluar las características del negocio, identificar los usuarios del sistema, registrar las fuentes de requerimientos, definir el ambiente de operación del sistema, registrar aquellos reque-

rimientos que fueron bien entendidos, realizar prototipos de aquellos requerimientos sobre los que se tenga alguna duda, definir posibles escenarios de uso del sistema e identificar restricciones del dominio.

En la tabla 12.5 se presentan los requerimientos, clasificados por tipos, en los que se centra TSPi.

Durante la elaboración del documento de requerimientos, el equipo discute acerca de la necesidad del cliente, la posible solución y cómo se va a implementar. De esta forma se genera el documento y se va logrando un acuerdo entre los miembros del grupo sobre el producto que van a desarrollar.

12.4.2.3.5 *Diseño*

En este proceso se elabora un diseño de alto nivel del sistema (es decir, de la estructura general del mismo). El diseño es un proceso creativo que permite identificar los principales componentes, así como la manera en que éstos interactúan. Esto ayuda a decidir la manera en la que se va a desarrollar cada una de las partes y cómo van a ser integradas para formar el sistema. Los estándares más importantes para realizar el diseño de alto nivel son los siguientes:

1. *Convenciones para la identificación*. Se debe establecer el criterio que se va a aplicar para nombrar al sistema, sus componentes, módulos, archivos, variables y demás elementos.
2. *Formatos para la relación*. Se debe especificar cómo va a ser la comunicación entre las partes del software, parámetros de entrada/salida, códigos de error u otras condiciones que deban preverse.
3. *Mensajes*. Se debe establecer un estándar para escribir los mensajes de tal manera que la información proporcionada por el sistema resulte comprensible.
4. *Estándar de defectos*. TSPi sugiere utilizar el estándar de defectos propuesto por PSP que se pueden consultar en (Humphrey 2004a).
5. *Conteo de líneas de código*. Si bien aún no se utilizará, TSPi sugiere que en esta etapa se defina el estándar para llevar a cabo el conteo de líneas de código.

Tabla 12.5 Principales requerimientos manejados por TSPi.

Tipo de requerimientos	Relacionado con
Funcionales	Entradas, salidas, cálculos y casos de uso.
De vinculación del sistema con el exterior	Usuarios, hardware, software, comunicación con otras computadoras o dispositivos.
De software	Manejadores de bases de datos, lenguajes, instalación, etcétera.
De diseño	Estándares, compatibilidad, integración con otros sistemas, etcétera.
Otros	Seguridad, mantenimiento, portabilidad, etcétera.

6. *Estándar para la representación del diseño*. Se debe establecer un estándar para que el diseño generado pueda ser entendido sin ambigüedades por todos los miembros del equipo.

Otros aspectos que deben tenerse en cuenta durante este proceso son la facilidad de uso del sistema y la prueba del mismo. Además, se sugiere que se revise y se inspeccione el diseño generado.

12.4.2.3.6 *Implementación*

Este proceso produce la implementación del producto. Antes de la codificación se debe hacer un diseño más detallado, a partir del diseño de alto nivel generado en la fase anterior. Es conveniente ir subdividiendo cada parte del sistema hasta que los elementos a codificar lleguen a tener 150 o menos líneas. Una vez que se alcance este nivel de detalle, se procede a la codificación de los elementos, la cual se puede hacer de manera paralela por diferentes miembros del equipo.

Aquí se definen algunos estándares de implementación que aumentan y complementan los que se presentaron en la discusión sobre el proceso de diseño:

1. *Revisión de estándares.* Se revisan los estándares definidos en el proceso anterior (diseño) y, si siguen resultando útiles, éstos se emplean. Se recomienda ir mejorándolos a medida que se va adquiriendo experiencia con ellos y que todos los miembros del equipo los utilicen.
2. *Estándar de codificación.* Éste debe ser usado por todos los miembros, ya que esto facilitará la revisión e inspección del código, así como la medición del sistema en líneas de código, funciones, objetos o el criterio que se decida seguir.
3. *Estándar de tamaño.* Permite contar líneas de código para medir el sistema. Sin embargo, durante el proyecto se generan otros productos que deben medirse y para los cuales se requiere definir nuevos estándares. Algunos ejemplos de estos productos son los documentos con los requerimientos y el diseño, pantallas y reportes, bases de datos, y reportes de pruebas.
4. *Estándar de defectos.* Se sugiere adoptar el estándar propuesto en PSP (Humphrey, 2004a). Éste es útil para poder clasificar los defectos y posteriormente analizarlos con el objetivo de mejorar el proceso.
5. *Prevención de defectos.* Es importante conocer las causas que pueden ocasionar los defectos. Un ejemplo de una posible causa de defectos es la falta de conocimiento del lenguaje o del ambiente de desarrollo utilizado. Una vez detectada la causa se pueden tomar las medidas correspondientes para eliminarla. En este caso sería aprender más sobre el lenguaje o el ambiente.

Luego de la codificación se debe revisar y posteriormente inspeccionar el código. Cada unidad debe ser revisada por el programador que la desarrolló. El tiempo estimado para llevar a cabo la revisión es al menos el 50% (preferentemente el 75%) del tiempo usado para la codificación. Luego es conveniente que otros programadores inspeccionen el código generado.

12.4.2.3.7 *Prueba*

Este proceso se aplica para probar el producto obtenido. Es importante que la mayoría de los defectos (preferentemente todos) se hayan detectado y corregido antes de esta etapa, ya que si no, el costo de hacerlo se incrementa notablemente. En TSPi se sugiere seguir las siguientes prácticas de prueba:

1. Construir el sistema integrando unidades ya probadas.
2. Probar el sistema integrado para evaluar si fue construido de manera co-

rrecta y verificar que están todas las partes presentes y que funcionan adecuadamente juntas.
3. Probar el sistema para evaluar que cubre todos los requerimientos.

En esta fase, mientras algunos ingenieros prueban el sistema, otros deben ir elaborando la documentación que se va a entregar al usuario. Dentro de esta documentación se deben incluir todas las explicaciones necesarias para entender la instalación, entrenamiento, uso y mantenimiento del sistema.

12.4.2.3.8 *POSTMORTEM*

En este proceso se revisa todo el trabajo hecho por los ingenieros y todos los datos recolectados durante los procesos previos. Posteriormente se hace un análisis para identificar los puntos en los cuales se puede mejorar el proceso, lo cual provee un medio para aprender y mejorar. Se debe comparar lo planeado con lo hecho. Además, se deben detectar oportunidades para mejorar y decidir cambios en las prácticas para el siguiente ciclo o proyecto.

Para determinar en qué áreas se puede mejorar el proceso se deben realizar revisiones. Algunas de las más importantes son:

1. *Revisión de calidad*. Se debe comparar la calidad planeada a nivel personal y a nivel de equipo con la calidad producida. Deben plantearse las siguientes tres preguntas. ¿Qué se puede aprender de esta experiencia? ¿Qué objetivos se pueden plantear para el siguiente desarrollo? ¿Dónde se podría modificar al proceso para mejorarlo?
2. *Evaluación de roles*. Se deben evaluar los diferentes roles del equipo. Deben plantearse las siguientes tres preguntas. ¿Qué sirvió? ¿Dónde hubo problemas? ¿Qué se puede mejorar?

El éxito de TSPi se basa en que la mayoría de los sistemas que se desarrollan actualmente requieren del trabajo de equipos, no de programadores individuales. Además, los equipos en los cuales cada miembro cuenta con buenas prácticas de trabajo, y trabajan cooperativa y eficazmente juntos, producen resultados de alta calidad. Cuando un equipo encuentra condiciones adecuadas, sus miembros realizan un trabajo superior, son más productivos y disfrutan su trabajo.

12.4.3 *XP (Extreme Programming)*

La programación extrema (Beck, 2004) ofrece un marco flexible, de bajo riesgo y eficiente para el desarrollo de software, basándose en algunos principios muy sencillos. XP se caracteriza además porque tiene en cuenta al ser humano como elemento clave en el desarrollo.

El desarrollo de software es una actividad dinámica que exige mucha flexibilidad y disposición por parte de los clientes y de los desarrolladores. Algunos de los problemas que surgen en los proyectos de software son los siguientes: cambios en el negocio durante el desarrollo del sistema, cancelación de parte o todo el proyecto, introducción de defectos en el diseño y/o en el código, implementación de requerimientos que no fueron solicitados, cambios en el equipo de trabajo, caducidad del sistema, etc. XP toma en cuenta todas estas posibles desventajas del desarrollo de software y genera prácticas para vencerlas o disminuir sus consecuencias.

12.4.3.1 LAS VARIABLES DE XP

XP es un modelo de desarrollo de software en el que hay cuatro variables que juegan un papel fundamental: costo, tiempo, calidad y alcance. La interacción entre las variables y el control de las mismas son decisivos para el éxito del proyecto. El control de estas variables es llevado a cabo por los programadores, administradores y clientes. Aquí se analizan las cuatro variables:

1. *Costo*: hace referencia a cuánto cuesta (en recursos humanos, tecnológicos y de oportunidad) desarrollar y/o mantener un producto de software. XP propone empezar invirtiendo poco e ir gastando más de manera productiva, es decir entregando resultados útiles al cliente.
2. *Tiempo*: hace referencia al tiempo que transcurre hasta que un sistema se libera. XP propone establecer alcances a corto plazo, de tal manera que los programas, con sólo algunas de sus funciones, entren en producción, y con la retroalimentación que se recibe se siga agregando funcionalidades y mejorando lo existente.
3. *Calidad*: hace referencia a que el software desarrollado se ajuste a los requerimientos del cliente, así como a que el mismo esté libre de defectos. El nivel de calidad es medido por los programadores, dando origen a la **calidad interna**. También es medido por los clientes, lo cual se conoce como **calidad externa**. XP propone realizar pruebas de manera continua sobre el producto que se está desarrollando. Por otra parte, XP se apoya en el hecho de que los seres humanos quieren hacer un buen trabajo. Los desarrolladores trabajan más a gusto y desarrollan con mayor calidad si sienten que su trabajo es bueno.
4. *Alcance*: hace referencia a los objetivos que se plantean para el producto a desarrollar. Es una variable difícil de controlar porque, en la práctica, cuando se inicia un proyecto ni el cliente ni los programadores tienen claro adónde se quiere llegar con el desarrollo de un software nuevo. Muchas veces a medida que el sistema entra en producción, el cliente se da cuenta de qué es lo que realmente necesita. XP propone establecer alcances pequeños, empezando siempre por los requerimientos de mayor prioridad para el cliente.

La variable alcance puede condicionarse a las otras tres. Se puede reducir el alcance, siempre que no se descuiden las funcionalidades más importantes, para mantener el costo, la calidad y el tiempo.

12.4.3.2 Los valores de XP

Los valores de XP son la comunicación, la simplicidad, la retroalimentación y la valentía. Éstos deben ser aceptados y respetados por todos los miembros del equipo y de la organización, ya que de ellos depende el éxito de este modelo.

La *comunicación* es básica en el desarrollo de software (¿existirá alguna actividad en la cual no lo sea?). Muchos de los problemas que se presentan en los proyectos de software tienen su origen en la falta de comunicación. Por ejemplo, se pueden ocasionar graves problemas si un programador no avisa a sus compañeros acerca de un cambio que introdujo en el código o reporta de manera inadecuada el grado de avance del producto, o un cliente no avisa a tiempo a los programadores que un requerimiento cambió o no da toda la información necesaria a los programadores. XP ayuda a mantener una buena comunicación por medio de prácticas que, por su naturaleza, exigen que los involucrados se comuniquen. Estas prácticas son las pruebas de unidades, la programación de pares y la estimación de tareas. Además, y complementando lo anterior, XP requiere de la actividad de una persona cuyo trabajo es detectar cuándo la gente no se está comunicando bien y lograr que vuelvan a hacerlo.

La *simplicidad* permite que los productos se diseñen e implementen de la manera más sencilla posible. ¿Por qué usar una estructura de tipo árbol para representar unos datos si el problema se puede resolver igualmente bien usando un arreglo? Si bien es una tendencia natural en el ser humano ver más allá de lo inmediato, la simplicidad es un valor que debe mantenerse en XP. Esta idea se fundamenta en el hecho de que tratar de desarrollar un producto simple y tener un costo adicional, si posteriormente se debe aumentar, es preferible a desarrollar un producto complejo y

caro que posiblemente nunca se use y cuyo desarrollo puede ser muy tardado. Existe una estrecha relación entre la simplicidad y la comunicación. Cuanto más sencillo sea un programa, más fácil será la comunicación entre los involucrados en él. Cuanto más se comuniquen los miembros de un equipo, más fácil será detectar con precisión las actividades que deben realizarse.

La *retroalimentación* acerca del estado actual del proyecto es fundamental para poder tomar decisiones apropiadas. Se lleva a cabo por medio de pruebas. Se definen pruebas de unidades para garantizar que se va avanzando sobre resultados sólidos. De esta manera los programadores prueban la lógica de sus programas. Por otra parte, tanto los clientes como los programadores responsables de las pruebas deben definir pruebas funcionales que permitan determinar el estado actual de todo el sistema. En cuanto se implemente un mínimo de funcionalidades que tengan sentido, el sistema debe ponerse en producción. Ya operando el sistema, el equipo de desarrollo recibirá retroalimentación valiosa sobre el desempeño del mismo y sobre otras funcionalidades que se podrán incorporar. La retroalimentación se relaciona con los valores ya mencionados. Cuanta mayor retroalimentación se tenga, más fácil será la comunicación. Seguramente será más eficaz comunicar el hecho de que cierto código es incorrecto por medio de una prueba que así lo demuestre, que por medio de una discusión entre pares o entre administrador y programador. Por otra parte, cuanto más simple sea un sistema más fácil será probarlo.

La *valentía* permite tomar las decisiones oportunas en el momento exacto, aunque parezcan descabelladas. Por ejemplo, si luego de trabajar todo un día en un trozo de código, éste no funciona correctamente y su desarrollo se está saliendo del plan, entonces con valor se debe desechar y volver a empezar a la mañana siguiente. Si de repente a algún miembro del equipo se le ocurre cómo simplificar el sistema complejo ya desarrollado, la valentía de todos los miembros del equipo permite enfrentar el reto de rehacer todo el sistema de una manera más simple. La valentía es importante siempre que estén presentes los otros tres valores. La comunicación favorece la presencia de valentía entre los miembros del equipo ya que facilita que los mismos intenten, de común acuerdo, tomar riesgo al cambiar un código o un diseño. La simplicidad también la promueve, ya que es más fácil tomar una decisión sobre un sistema simple que sobre un sistema complejo, especialmente si la decisión es tirar el sistema al bote de la basura. La retroalimentación también contribuye a su presencia, ya que es más seguro basar una decisión en la información proporcionada por las pruebas que en la intuición o idea que pueda tener algún miembro del equipo. Por su parte, la valentía produce simplicidad. En cuanto se detecta que un sistema puede simplificarse, se hacen los cambios necesarios para lograrlo.

Existe un elemento adicional señalado por (Beck, 2004), sin el cual ninguno de los otros valores se sostendría: el respeto. Es fundamental que todos los miembros del equipo se respeten mutuamente y respeten el trabajo de los demás. Se requiere compromiso y disfrutar ser parte de un equipo que hace las cosas bien.

12.4.3.3 Principios básicos de XP

Los principios básicos que se generan a partir de los valores, y que ofrecen medios más concretos para aplicarlos son: retroalimentación rápida, simplicidad, cambios incrementales, cambios sin alterar lo ya desarrollado y trabajo de calidad.

La *retroalimentación rápida* consiste en obtener retroalimentación, interpretarla y aplicar lo aprendido al sistema, lo más pronto que se pueda. Todo lo que se puede aprender de la experiencia se debe tratar de llevar al sistema cuanto antes.

La *simplicidad* consiste en hacer un buen trabajo para resolver el problema actual, confiando en que si en el futuro se necesita, se podrá aumentar la complejidad del sistema. XP promueve soluciones simples, que se

ajusten a los requerimientos presentes, en lugar de dedicar esfuerzo buscando soluciones generales que se adelanten a necesidades futuras.

Los *cambios incrementales* consisten en hacer pequeños cambios que aunados lleven a la solución de un problema. XP propone cambios pequeños en la planeación, el diseño, la codificación e incluso en los equipos de trabajo.

Los *cambios sin alterar lo ya desarrollado* consisten en que cada cambio que se realiza no altere lo ya planeado, diseñado o desarrollado. Es decir, que cada cambio sea para incrementar las funcionalidades ya resueltas, no para reemplazarlas por otras.

El *trabajo de calidad* consiste en desarrollar el trabajo de la mejor manera posible. Se basa en que todos los miembros del equipo desean sentir que su trabajo es excelente. De otra manera la motivación baja, el trabajo no se disfruta ni se realiza de manera correcta y finalmente todo el proyecto fracasa.

12.4.3.4 Las prácticas de XP

Considerando los valores y los principios básicos presentados, aquí se plantean las prácticas requeridas para usar XP en el desarrollo de software de alta calidad. Las prácticas son: planeación, alcance pequeño, metáfora, diseño sencillo, pruebas, reestructuración, programación de pares, propiedad colectiva, integración continua, 40 horas de trabajo por semana, cliente en el lugar de desarrollo y estándares de codificación.

1. *Planeación*: permite determinar de manera rápida el alcance de la siguiente entrega. Para ello se deben considerar tanto las prioridades del negocio como las estimaciones técnicas. El éxito de la planeación depende en gran medida del equilibrio entre lo que se desea hacer y lo que es posible hacer. Los responsables del negocio tienen que decidir sobre el alcance del sistema a desarrollar, establecer las prioridades de las diversas funcionalidades, determinar el mínimo de funcionalidades requerido para que el negocio se beneficie al poner el sistema en producción e indicar las fechas en las cuales se debe contar con el software. Por su parte, el personal técnico debe estimar el tiempo para desarrollar una determinada funcionalidad, informar a los responsables del negocio acerca de las consecuencias que puede haber al tomar ciertas decisiones (por ejemplo, la compra de una plataforma de desarrollo), organizar las tareas y el equipo de trabajo para producir el software con alta calidad, y definir el plan de las actividades de tal manera que se asegure realizar aquellas que son de más alta prioridad, así como empezar con aquellas que tienen implícito un mayor riesgo para el negocio.
2. *Alcance pequeño*: se trata de definir entregas tan pequeñas como sea posible, pero que implementen al menos una de las funcionalidades esperadas. XP propone que se hagan planes para periodos cortos, por ejemplo uno o dos meses.
3. *Metáfora*: es una frase que define de manera sencilla todo el sistema. Se hace referencia al proyecto por medio de la metáfora elegida. Ésta permitirá identificar los principales elementos del sistema y la relación entre ellos.
4. *Diseño sencillo*: se debe diseñar la solución del problema de la manera más sencilla. Es decir, el diseño debe incluir sólo los elementos necesarios para cubrir los requerimientos actuales. XP propone no diseñar para el futuro, ya que lo único cierto es lo que actualmente se está resolviendo. Además, se supone que el desarrollador debe ser capaz de realizar cambios a su diseño si en el futuro nuevos requerimientos lo exigen. El diseño se considera correcto si pasa todas las pruebas, no tiene lógica duplicada, expresa todos los propósitos importantes y tiene la menor cantidad posible de elementos.
5. *Pruebas*: se debe probar cada una de las unidades de un programa, así como el

programa completo. XP lleva este concepto al extremo, por lo que sostiene que cualquier función de un programa que no sea probada no existe. Los programadores definen pruebas que validan la operación del sistema. Por su parte, los clientes definen pruebas funcionales que validan la operación del sistema desde el punto de vista del negocio. Cuanto más probado esté un sistema, más fácil será aceptar cambios.

6. *Reestructuración*: se revisa el programa cada vez que se agregan nuevas funcionalidades, buscando simplificarlo. Si se detecta la posibilidad de hacer algo más simple, se hace. Cuando se requiere duplicar código se está en presencia de un caso que exige la reestructuración del sistema.
7. *Programación de pares*: se programa entre dos personas. Es decir, cada trozo de código es generado por un par de personas compartiendo una computadora. Se establecen dos papeles entre los programadores. Uno de ellos se concentra en buscar la mejor manera de implementar el método o la clase asignada. El otro analiza el problema más globalmente (por ejemplo, se cuestiona si se podría simplificar el sistema y así evitar la programación del método o clase actual o si se podrían definir nuevos casos de prueba). La asignación de pareja es dinámica —cambian durante el transcurso del proyecto.
8. *Propiedad colectiva*: todos los miembros del equipo son dueños del sistema. Por lo tanto, todos tienen la posibilidad de modificar o agregar código en cualquier momento y todos tienen la misma responsabilidad ante el sistema.
9. *Integración continua*: se debe ir integrando y probando el código de manera continua. XP propone que se integre luego de algunas horas de trabajo o como máximo una vez al día. El código integrado debe pasar todas las pruebas definidas. De esta manera, cuando otro par de programadores integra su parte y la prueba, si falla tiene la certeza de que es sólo su parte la que tiene defectos.
10. *Trabajar 40 horas semanales*: la cantidad de horas trabajadas por cada desarrollador es un factor importante para el éxito de un proyecto. A pesar de que la cantidad de horas que cada persona puede trabajar de manera eficaz puede variar, se sabe que para que el rendimiento se mantenga, la cantidad de horas no debe exceder las 40 semanales. Además, se considera de fundamental importancia que los programadores dediquen los fines de semana a realizar actividades distintas al trabajo y que tomen vacaciones.
11. *Cliente en el lugar de desarrollo*: el cliente, en este caso, es una de las personas que usará el sistema cuando sea liberado, y debe formar parte del equipo de trabajo. Por lo tanto, debe ayudar al equipo contestando las preguntas que vayan surgiendo, sin detener el trabajo, y definiendo prioridades entre funcionalidades a implementar. Por otra parte, debido a su participación activa durante el desarrollo, el sistema podrá proporcionar mayor valor al negocio.
12. *Codificación estándar*: se requiere establecer un estilo común de codificación. Teniendo en cuenta las prácticas de programación de pares, que todos son dueños del sistema y por lo tanto todos pueden de manera permanente modificar o agregar código y que el sistema se está reestructurando de manera continua, es de fundamental importancia establecer y respetar el estándar. XP propone que el estándar sea lo más sencillo posible, adoptado por los programadores de manera voluntaria y que el mismo debe enfatizar la no duplicación de código y la comunicación entre los miembros del equipo.

XP se apoya en el hecho de que si estas prácticas son buenas, entonces ¿por qué no usarlas de manera intensiva? Es decir, si integrar código es una práctica buena, ¿por qué no integrar código de manera continua? O, si

la simplificación del sistema es una práctica buena, ¿por qué no simplificar continuamente? Lo mismo se podría decir acerca de las pruebas. Si es bueno probar el código, ¿por qué no probarlo todo el tiempo?

Las prácticas se apoyan mutuamente, y muchas de ellas sólo pueden aplicarse en presencia de otras. Por ejemplo, establecer alcances pequeños es posible sólo si se hace una planeación adecuada, si se hacen diseños sencillos que puedan implementarse fácil y rápidamente, y si el código se prueba e integra de manera continua. Por su parte, el estándar de codificación, que implica un esfuerzo adicional para los miembros del equipo, se requiere y justifica si se tienen otras prácticas como la programación de pares, la propiedad colectiva y la integración continua. En la figura 12.7 se presenta un esquema de la relación entre prácticas de XP.

XP resalta la importancia de buscar un equilibrio entre los intereses de los responsables del negocio, los clientes, y los de los responsables de la tecnología, los desarrolladores. El éxito de un proyecto depende de que los grupos colaboren entre sí y mantengan una buena comunicación.

De acuerdo con lo presentado en las secciones precedentes, se puede decir que XP es un modelo de programación novedoso e interesante. Una de las razones para efectuar esa afirmación es la importancia que le da al ser humano como pieza clave en el desarrollo de software. Esto da origen a algunas de las prácticas y estrategias del modelo, como la cantidad de horas trabajadas, la comunicación entre miembros del equipo y el ambiente de trabajo. Otro pilar de este modelo que merece ser destacado es la idea de que toda persona gusta de hacer las cosas de la mejor manera posible, por naturaleza, no por obligación.

Las diferencias más importantes entre XP y otras metodologías de programación se resumen en la tabla 12.6.

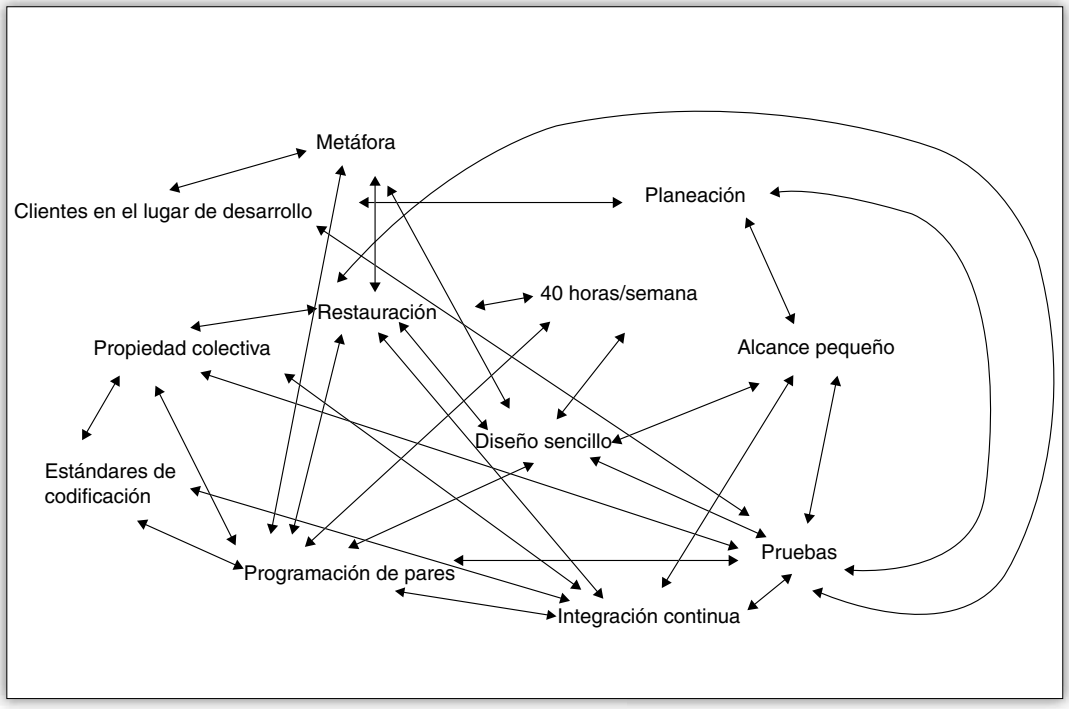

Figura 12.7 Relación entre las prácticas de XP.

Tabla 12.6 Diferencias entre XP y otras metodologías de programación.

	XP	Otras metodologías de programación
Diseño	Evolutivo	Fijo
Codificación	En pares	Individual
Complejidad	Sólo si no se puede evitar	Permitida
Integración	Continua	Al final del desarrollo
Pruebas	Continua	Al terminar cada componente y el sistema
Reuso	No se busca	Se busca
Asignación de tareas	Flexible	Fija
Comunicación	Permanente	Limitada
Clientes	Parte del equipo	Fuera del equipo

12.5 Calidad de software y madurez del proceso

12.5.1 *Calidad de software y modelos de madurez del proceso*

La **calidad de software** tiene diferentes significados para distintos grupos. Para el Instituto de Ingenieros Eléctricos y Electrónicos (en inglés, IEEE—Institute of Electrical and Electronic Engineers)[15] la calidad de software es el grado en que un sistema, componente o proceso cumple con los requerimientos especificados y las necesidades o expectativas del cliente o usuario. En la definición de la norma ISO-9000 de la Organización Internacional para la Estandarización (ISO—International Organization for Standardization)[16] la calidad de software es el grado (pobre, bueno o excelente) de cómo un conjunto de características inherentes del software cumplen con los requisitos del sistema.

La calidad del software está directamente relacionada con el proceso de su desarrollo. Se considera que un proceso bien conocido y ampliamente utilizado, sustentado en medición y predicción de eventos, permite controlar en buena medida la producción de software y, en consecuencia, producir software de calidad[17].

[15] ANSI/IEEE Std. 729-1983, *IEEE standard glossary of software engineering terminology*, IEEE Inc., Nueva York, 1983.

[16] ISO 9000-3:1997, http://www.iso.org

[17] DeMarco, T., 1982, *Controlling software projects: management*, *measurement and estimation*, Prentice Hall.

Los factores que más afectan la obtención de un producto de calidad son: el cliente o usuario (participante primordial en el proceso de desarrollo del producto y responsable en definir los requisitos del producto final), el desarrollador (responsable del proceso de producción y el responsable de asegurar la calidad del producto), el proceso seguido para el desarrollo del producto final y el producto (correspondiente al sistema desarrollado).

Estos factores tienen una estrecha y continua correlación que afecta a la ingeniería del producto, tanto como a la organización, la cual debe establecer estándares para los procesos de desarrollo y evaluación, empleando medidas bien establecidas que permitan mejoras continuas de los productos. La evaluación de los procesos evita especificaciones incompletas o anómalas, la aplicación incorrecta de metodologías, etc.[18] Con el objetivo de evaluar la calidad de los procesos de software se define el concepto de **modelo de madurez** del proceso de producción de software. Estos modelos apoyan no sólo la mejora continua de los procesos de desarrollo de software sino también la estandarización de la producción en toda la organización. Cabe resaltar que no se debe aplicar modelos de madurez, bajo el supuesto de mejorar su calidad, sin antes establecer y definir completamente los procesos de desarrollo. Dada la importancia de los procesos, existen diversas certificaciones basadas en los modelos de madurez de proceso que evalúan que las empresas "digan lo que hacen", "lo hagan como dicen" y "demuestren que lo que dicen así lo hacen"[19]. La razón adicional para estas certificaciones es que los modelos a menudo requieren cambios en la cultura de la organización y una fuerte inversión en recursos financieros, tecnológicos y humanos.

En la tabla 12.7 se muestra una cronología de algunos de los estándares y modelos para la calidad de software más importantes en la actualidad.

En las siguientes secciones se describen algunos de los modelos de madurez más importantes.

12.5.2 *Modelo de madurez de capacidad (CMM)*

El modelo más importante en la actualidad para la evaluación de la madurez de los procesos de desarrollo es el **modelo de madurez de capacidad** (en inglés, **CMM—Capability Maturity Model**) del Instituto de Ingeniería de Software (en inglés, SEI—Software Engineering Institute)[20]. CMM tiene como objetivo evaluar los procesos en sus niveles de madurez e identificar los niveles que una organización debe formar para establecer una cultura de excelencia en la ingeniería de software. Los modelos de CMM se generan gracias a la experiencia colectiva de los proyectos más exitosos de software. Dentro de la familia de modelos CMM, se define el modelo para el área de software, SW-CMM.

En particular, CMM es un marco de trabajo que especifica guías para organizaciones de software que quieren incrementar su capacidad de procesos, considerando los siguientes puntos: identificar fortalezas y debilidades en la organización, ponderar los riesgos de seleccionar entre diferentes contratos y monitorear los mismos, entender las actividades necesarias para planear e implementar los procesos de software, y ayudar a definir e implementar procesos de software en la organización a través de una guía.

El modelo CMM se evalúa según el **área de proceso clave** (en inglés, KPA—Key Process Area), donde cada organización debe incorporar procesos adecuados en cada una de las áreas establecidas. Los procesos se eva-

[18] Jones, C., 1994, *Assessment and control of software risks*, Prentice Hall.
[19] Humphrey, W., 1995, *A discipline for software egngineering*, Addison Wesley.
[20] http://www.sei.cmu.edu/cmm/

Tabla 12.7 Cronología de los estándares y modelos de madurez más importantes para la calidad de software.

Año	Descripción
2000	ISO 9000:2000 SEI CMMI V1.02
1999	PEMM V1.0
1998	ISO 15504 (SPICE) (lanzamiento al público como Reportes Técnicos "tipo 2") TickIT V4.0
1997	ISO 9000-3 (nuevo lanzamiento) SEI para las revisions de SW-CMM apoyando a CMM Integración (CMMI)
1996	IEEE/EIA 12207 (corresponde a ISO 12207) TSP
1995	ISO 12207 (lanzamiento inicial) ISO 15504 (SPICE) (lanzamiento inicial) PSP
1994	ISO 9001 (nuevo lanzamiento)
1993	SEI SW-CMM V1.1
1992	TickIT V2.0
1991	ImproveIT V1.0 (origen de TickIT) ISO 9000-3 (lanzamiento inicial) SEI SW-CMM V1.0 (lanzamiento inicial)
1987	ISO 9001 (lanzamiento inicial)

lúan mediante distintos niveles de madurez, como se muestra en la figura 12.8.

En la tabla 12.8 se describen con mayor detalle los niveles de madurez.

Según información del SEI, hasta abril de 2003 se contaba con cerca de 100 empresas certificadas en el nivel 5 en todo el mundo[21].

Existe una extensión del modelo básico de madurez, **la integración del modelo de madurez de capacidad** (en inglés, **CMMI—Capability Maturity Model Integration**) que tiene como objetivo integrar los diferentes dominios en los que se ha aplicado CMM, más allá de sólo el desarrollo de software. Esto es algo similar al modelo de calidad ISO-9000, que se aplica en múltiples disciplinas, como se explica a continuación.

12.5.3 *Organización Internacional para la Estandarización (ISO)*

Existen diversas normas de la **Organización Internacional para la Estandarización** (en inglés, ISO—International Organization for

[21] http://www.sei.cmu.edu/sema/profile.html, http://www.sei.cmu.edu/sema/pdf/SW-CMM/2003apr.pdf

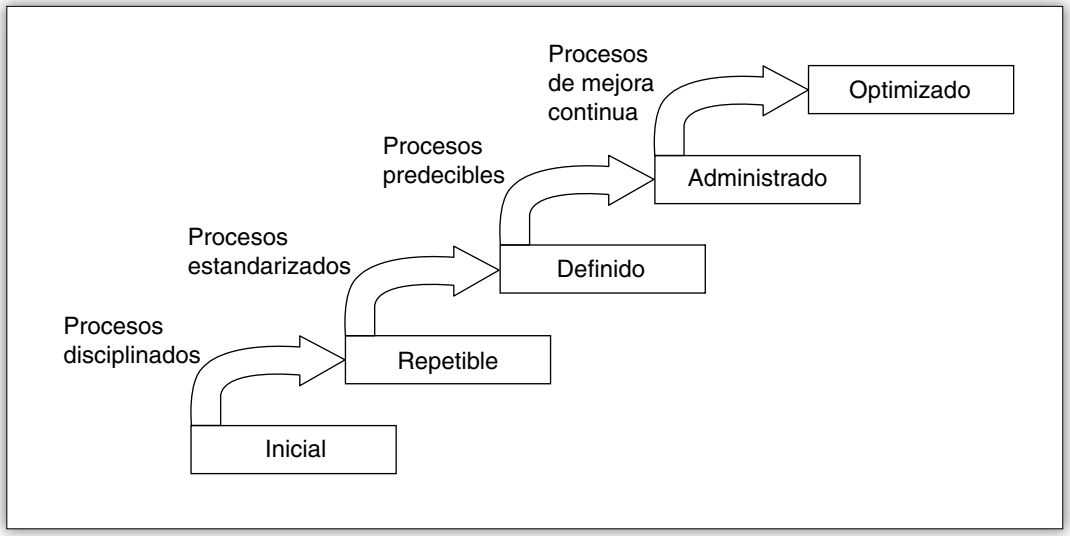

Figura 12.8 Los cinco niveles de madurez del proceso de software.

Tabla 12.8 Los cinco niveles del modelo de CMM.

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

Standarization)[22] que son aplicables al desarrollo de software, como se describe a continuación.

12.5.3.1 ISO-9000

ISO-9000[23] es una familia de normas internacionales relacionadas con la administración de la calidad en productos y servicios que especifica qué requisitos de calidad se deben cumplir, sin especificar cómo se deben cumplir. Dentro de la familia ISO-9000 existen tres estándares, ISO-9001, ISO-9002 e ISO-9003, que se deben considerar dependiendo del alcance de las actividades de la empresa. En el caso del desarrollo de software, ISO-9001 es el estándar más apropiado, ya que considera actividades de diseño y desarrollo. En cierta manera, ISO-9000 es comparable a CMMI por ser una familia de estándares, mientras que ISO-9001 es comparable a CMM. De esta manera, ISO-9001 es un estándar aplicable a la administración de la calidad en el desarrollo de software. A diferencia de CMM, el modelo ISO-9001 no incluye múltiples niveles de calidad a los cuales se puede estar certificado. La certificación es equivalente al nivel 3 de la escala de SEI.

12.5.3.2 ISO-12207

ISO-12207[24] es una familia de normas internacionales relacionadas con la administración de la calidad en los procesos del ciclo de vida del software. Esta norma está dirigida a lograr acuerdos o contratos entre los desarrolladores y clientes en situaciones en las que se requiere el desarrollo, mantenimiento u operación de un sistema de software. Es una norma de alto nivel que deja sin especificar los detalles de cómo llevar a cabo las actividades y tareas de los procesos. Se describen cinco procesos primarios, adquisición, suministro, desarrollo, mantenimiento y operación. Los cinco procesos se dividen en actividades, y las actividades en tareas, agregando requisitos para su ejecución. Se especifcan ocho procesos de apoyo, documentación, administración de configuración, aseguramiento de calidad, verificación, validación, auditoría y resolución de problemas. Además se consideran cuatro procesos organizacionales, administración, infraestructura, mejora y entrenamiento. Esta norma busca que las organizaciones adapten estos procesos según el alcance de los proyectos particulares, eliminando actividades que no se aplican.

12.5.4 *Modelo de madurez de ingeniería de desempeño (PEMM)*

El **modelo de madurez de ingeniería de desempeño** (en inglés, PEMM—Performance Engineering Maturity Model)[25] es un modelo para evaluar los niveles de integración, aplicación, ejecución y diseño, basado en el modelo de madurez de capacidades de CMM. El objetivo de PEMM es evaluar la ejecución de la ingeniería y proceso del modelo de madurez. El modelo sirve tanto para evaluar una organización como para los propios procesos de desarrollo tecnológicos particulares. Sirve también para definir el criterio para escoger un proveedor de software para los productos críticos o semicríticos de la empresa.

Al igual que CMM, PEMM cuenta con cinco niveles que determinan la mejora del comportamiento de ejecución y el decremento del riesgo de ejecución, como se muestra en la figura 12.9.

La evaluación de una compañía se hace midiendo los aspectos relacionados con la organización, la definición de procesos de

[22] http://www.iso.org

[23] http://www.iso.org/iso/en/iso9000-14000/iso9000/iso9000index.html

[24] http://www.12207.com/

[25] Schmietendorf, A., y Scholz, A., 1999, "The Performance Engineering Maturity Model", en *Metrics News,* vol. 4, núm. 2.

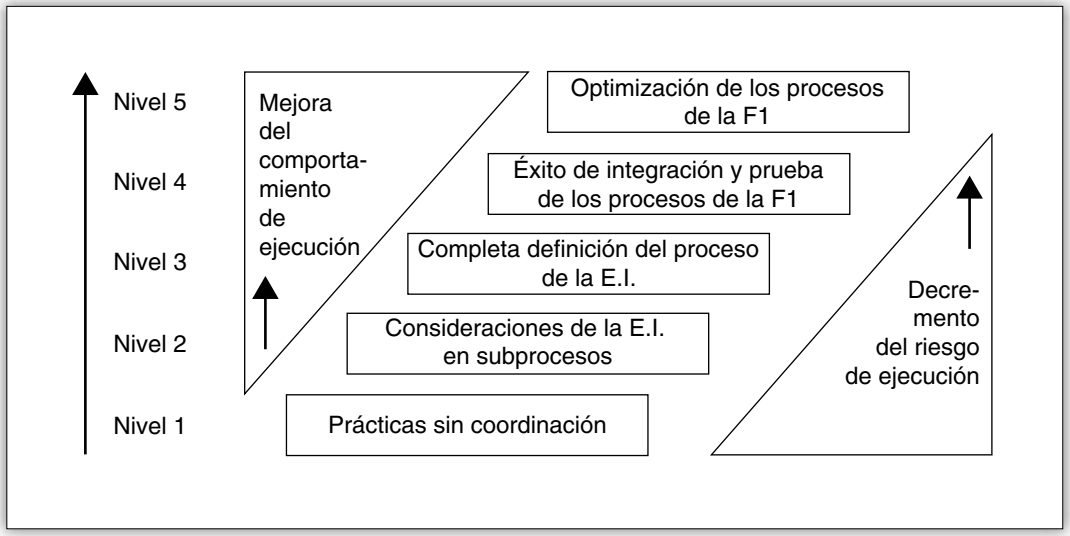

Figura 12.9 Modelo general PEMM.

ingeniería, el proyecto de la dirección y la tecnología. Esto se logra mediante encuestas expertas del método de métricas sobre preguntas y metas (en inglés, GQM—Goal Question Metric), identificando y midiendo los objetivos con preguntas y respuestas cuantificables (en la actualidad dichas respuestas sólo pueden ser sí o no).

12.5.5 *TickIt*

TickIt,[26] desarrollado por el Departamento de Comercio e Industria del Reino Unido, es primordialmente una guía con estrategias para lograr la certificación en la producción de software según los estándares ISO-9000. 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 la certificación es demostrar que existen, y son verificables, las prácticas necesarias para asegurar la calidad durante el desarrollo de software. La guía de auditoría provee la liga para evaluar la conformación del sistema auditado con respecto al modelo TickIt, de manera que pueda ser expresada en función de los criterios de la ISO 9001.

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 auditoría del sistema de calidad, (v) el proceso de certificación y (vi) guías complementarias.

12.5.6 *Mejoramiento del proceso de software y determinación de capacidad (ISO-15504/SPICE)*

El modelo de **mejoramiento del proceso de software y determinación de capacidad** (en inglés, SPICE—Software Process Improvement and Capability dEtermination)[27]

[26] http://www.tickit.org/

[27] Dorling, A., 1993, SPICE: "Software Process Improvement and Capability Determination", *Software quality journal*, 2, 209-224.

es un modelo de evaluación o determinación de la capacidad de los procesos, conocido también como la norma ISO-15504[28]. Es una familia de normas internacionales que tiene como objetivo el desarrollo de sistemas de calidad en el software. Combina los enfoques de CMM con los de ISO-9000, incorporando al marco de referencia de ISO 9000 con la evaluación de capacidad y madurez de proceso de CMM. Su objetivo es lograr ganancias significativas en productividad y calidad, además de ayudar a los compradores de productos de software a obtener un mejor valor por su dinero y reducir el riesgo asociado a los grandes proyectos de software. Este modelo busca mejorar la calidad del producto mediante una evaluación comprobada, sistemática y confiable del estado de los procesos de software de una organización, y usar los resultados de estas evaluaciones como parte de programas coherentes de mejoramiento. La norma establece un denominador común para una evaluación uniforme de los procesos de desarrollo. Ya que la tecnología evoluciona, ISO-15504 hace énfasis en la calidad, actualización, y vigencia del producto.

[28] http://www.sei.cmu.edu/iso-15504/

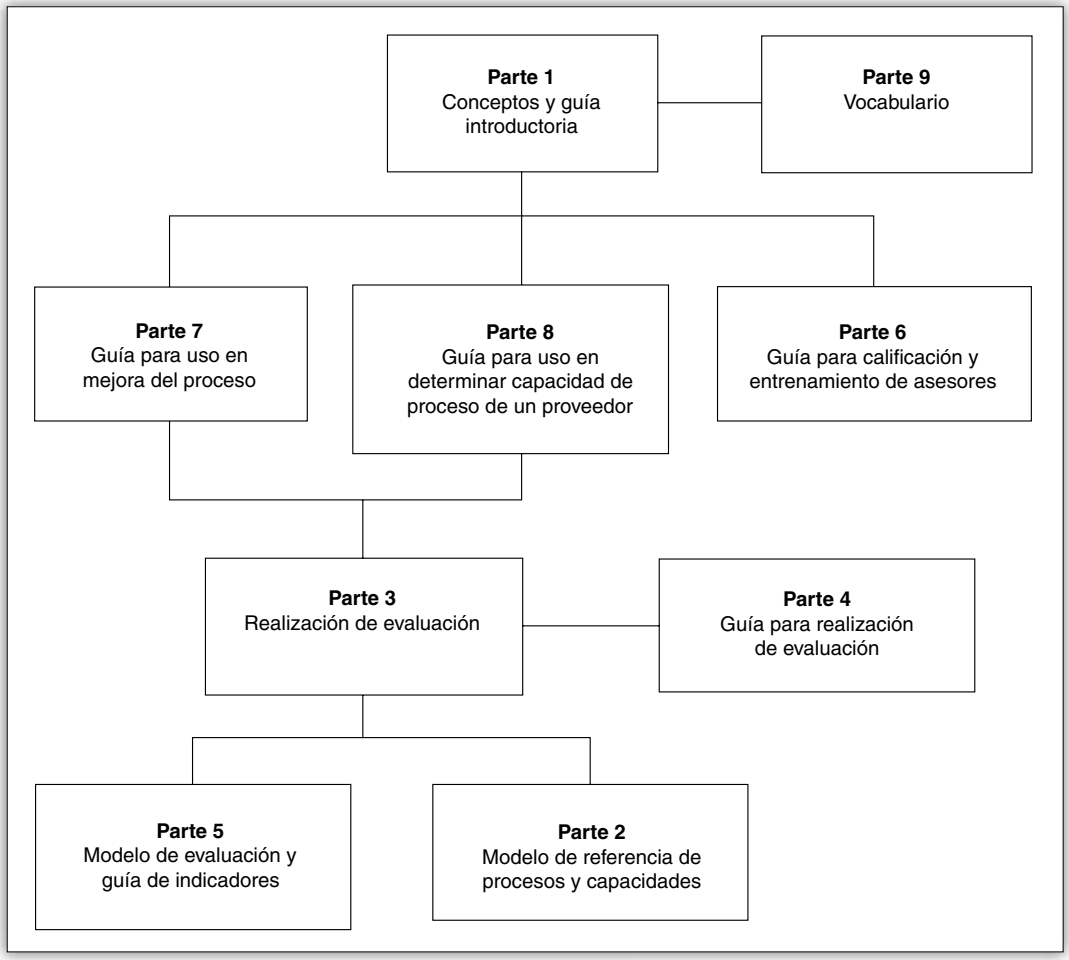

Figura 12.10 Componentes del modelo ISO-15504.

ISO-15504 está conformado por nueve documentos que permiten instrumentar paso a paso la norma con su correspondiente evaluación, como se muetra en la figura 12.10: parte 1 (conceptos y guía introductoria), parte 2 (modelo de referencia de procesos y capacidad), parte 3 (realización de evaluación), parte 4 (guía para realización de evaluación), parte 5 (modelo de evaluación y guía de indicadores), parte 6 (guía para calificación y entrenamiento de asesores), parte 7 (guía para uso en mejora del proceso), parte 8 (guía para uso en determinar capacidad del proceso de un proveedor) y parte 9 (vocabulario).

La arquitectura de evaluación de SPICE define las siguientes prácticas y procesos. Al igual que CMM, las prácticas de SPICE son tratadas de acuerdo con niveles de capacidad: nivel 0 (incompleto), nivel 1 (definido), nivel 2 (administrado), nivel 3 (establecido), nivel 4 (predecible) y nivel 5 (optimizado). Los procesos consideran cincos áreas de actividad: cliente-proveedor, ingeniería, soporte, administración y organización. Al igual que CMM, ISO-15504 integra una serie de niveles de madurez por los que deben pasar sus procesos.

Preguntas de repaso y ejercicios

1. Defina brevemente qué es PSP.
2. Enumere y explique brevemente cada uno de los procesos de PSP.
3. Defina brevemente qué es TSPi.
4. Explique brevemente cada uno de los procesos de TSPi.
5. Defina brevemente qué es XP.
6. Enumere y explique las variables que maneja XP.
7. Enumere y explique brevemente las prácticas de XP.

Referencias

(Beck, 2004) Beck, K. *Extreme programming explained: embrace change*. Addison-Wesley, 2004.

(Humphrey, 2004a) Humphrey, W.S. *Introduction to the personal software process*. SEI Series in Software Engineering. Addison-Wesley, 2004.

(Humphrey, 2004b) Humphrey, W.S. *Introduction to the team software process*. SEI Series in Software Engineering. Addison-Wesley, 2004.

(Humphrey, 2004c) Humphrey, W.S. *A discipline for software engineering*. SEI Series in Software Engineering. Addison-Wesley, 2004.