UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE INGENIERIA

SISTEMA DE PRODUCCION PARA LA INDUSTRIA FARMACEUTICA

T E S I S

QUE PARA OBTENER EL TITULO DE

INGENIERO MECANICO ELECTRICISTA

P R E S E N T A :

EZEQUIEL VICTOR VELAZQUEZ MENDEZ

FACULTAD DE INGENIERIA
U N A M

DIRECTOR DE TESIS:

ING. VICTOR HUGO TOVAR PEREZ

2 0 0 8

Dedicatoria

A mis padres Martha y Ezequiel, con profundo amor y respeto.
Quienes siempre me han apoyado incondicionalmente,
dándome mucho amor, apoyo, consejos y enseñándome a salir adelante
con su ejemplo a lo largo de mi vida.
A ellos, quienes sembraron la semilla y fertilizaron mi inquietud de conocer.

A mi hermana Martha, de la cual guardo tan buenos recuerdos
de la infancia y adolescencia;
a Arturo, su esposo y Sergio Arturo, su hijo,
por todo el cariño y apoyo que me han brindado los tres,
ya que siempre han estado muy cerca de mí a pesar de la distancia.

A la memoria de mi abuelita paterna María de Jesús
y a la de mi abuelito materno Germán,
ambos fallecidos pero que siempre me acompañan en mis pensamientos.
A mis abuelitos Guillermina y Luis con los que aún comparto su presencia en mi vida.

A mi esposa Marisol, con profundo amor,
en quien he depositado todas mis esperanzas
ya que siempre ha sido una gran compañera y amiga.

Agradecimientos

Expreso mi mayor agradecimiento a mis padres Martha y Ezequiel, por toda su dedicación, apoyo, esfuerzo, ejemplo de trabajo, de honradez y de lucha por la vida. Quiero agradecer doblemente a mi padre por su apoyo y colaboración directa en la elaboración de esta tesis.

Manifiesto mi gratitud a mi hermana Martha por el cariño que me ha brindado toda mi vida, a Arturo, su esposo, quien se ha ganado mi respeto y afecto, a ambos por el apoyo invaluable que siempre me han proporcionado y también a mi querido sobrino Sergio Arturo que vino a llenar de felicidad mi corazón, por su inteligencia y bondad.

A mis abuelitos, María de Jesús y Germán que ya no están con nosotros, y a mis abuelitos Guillermina y Luis, aún con vida, a todos porque tuve la dicha de conocerlos y compartir muchísimos momentos con ellos a lo largo de mi existencia llenándome de atenciones, enseñanzas positivas, cariño y apoyo en momentos difíciles sobre todo de mi infancia.

A mi esposa Marisol, que me ha brindando su cariño, comprensión y una enorme ayuda en las últimos esfuerzos para terminar este trabajo.

También a mis tías y tíos por la convivencia inolvidable, cuidados y consejos recibidos de ellos y el apoyo proporcionado en los momentos difíciles.

A mi amigo Jaime Osorio cuyo apoyo fue sustancial facilitándome la elaboración de esta tesis. Así como a mis amigos Mario Ota y Armando Suárez por los años de convivencia, tanto estudiantil como profesional y cuya amistad continúa hasta la fecha.

A mi director de Tesis, Ing. Víctor Hugo Tovar Pérez, por su atinada conducción en este trabajo. A los sinodales: Ing. Juan José Carreón Granados, Ing. Víctor Hugo Tovar Pérez, M. A. Víctor Damián Pinilla Morán, Ing. María del Rosario Barragán Paz y al Ing. Cruz Sergio Aguilar Díaz, por leer esta tesis y hacerme sus acertados comentarios y observaciones sobre la misma.

Por último, a la Facultad de Ingeniería de la UNAM, de la cual me siento muy orgulloso de ser egresado y a los maestros que compartieron conmigo sus conocimientos, ética profesional y calidad humana que conformaron la base de mi formación y me han permitido desempeñarme exitosa y satisfactoriamente en la vida profesional.

Indice

Introducción

El desarrollo de sistemas es una actividad que en los últimos años se ha generalizado y que provee de herramientas a los procesos productivos, nos sirven como entretenimiento y día a día se involucran más en nuestras vidas tanto profesionales como personales ya que últimamente las aplicaciones abarcan: juegos, agendas personales, aplicaciones de oficina, automatización de procesos, etcétera.

Para poder continuar avanzando en la generación de proyectos de software cada día más complejos y completos, el desarrollo del mismo se ha transformado de algo totalmente artesanal que dependía únicamente de un gran esfuerzo e intuición de los programadores a una serie de metodologías y mejores prácticas que nos orientan y ayudan a controlar un proyecto a lo largo de todas sus etapas, así como también dentro del desarrollo de sistemas se han creado nuevos roles de especialización que nos permiten segmentar el conocimiento e interactuar adecuadamente para llegar a nuestra meta de generar una aplicación útil y con la calidad requerida.

Esta rama de la ingeniería es muy joven todavía y su madurez aún no ha sido alcanzada pero se han logrado muchos avances y con el tiempo se ha pasado de desarrollar empírica y artesanalmente a desarrollar de manera metodológica y aunque todavía hay algunos puntos donde interviene la subjetividad y la experiencia, poco a poco se ha ido transformando el desarrollo de software en una actividad menos empírica, mucho más estable y más orientada a ser una rama de la ingeniería de rápido crecimiento, de grandes avances y de consolidación extraordinariamente rápida.

En general existen muchas metodologías para cada etapa del desarrollo y no es que exista una mejor para todos los casos, más bien son una serie de pasos a seguir que nos ayudarán a llegar a la solución y como tal, las formas de llegar a alcanzar los objetivos pueden ser varias. En este sentido lo principal es que de acuerdo a las características del proyecto se seleccionen las mejores formas de llegar a las metas propuestas y que también se aplique el sentido común ya que por ejemplo si se abusa de un método, lejos de obtener un beneficio, ello puede resultar contraproducente; por ejemplo, existen en las bases de datos relacionales un conjunto de reglas para optimizar (llamadas formas normales); sin embargo, no en absolutamente todos los casos es conveniente aplicar estas reglas ya que en algunos casos es mejor duplicar una columna aún dejando de cumplir estas formas normales si esto ayudara a hacer las consultas mucho más ágiles y por supuesto si el espacio no es algo en lo cual tengamos restricciones muy grandes. Y así como este ejemplo existen varios otros puntos donde tendremos que ser mesurados y no abusar de las metodologías.

El desarrollar sistemas es una tarea fascinante ya que es generar una solución a problemas muy diversos y complejos, aplicaciones que implican cálculos, reportes, procesos, integración de información y transformaciones de la misma, son partes comunes de una aplicación empresarial y cuando se inicia sólo se tiene una lista de requerimientos por cumplir o incluso una idea vaga de la problemática que existe y una necesidad imperiosa de una solución que resuelva de una manera eficiente y muchas veces simple (desde el punto de vista de el usuario porque en ocasiones las cosas que más simplifican la vida del usuario le dan más responsabilidad a la aplicación y por tanto generan soluciones más complejas desde el

punto de vista del desarrollo) a los requerimientos del negocio, por tanto es importante que se desarrollen de una forma ordenada y estructurada ya que de lo contrario lejos de solucionar un problema lo puede llegar a complicar más.

El presente trabajo es una muestra de cómo es el desarrollo de un sistema tratando de seguir los lineamientos que hay y las diferentes metodologías existentes, muestra una parte teórica donde podremos ver de una manera resumida algunas de estas reglas y digo de una manera resumida porque existen libros enteros de cada tema y también con el caso desarrollado en estas páginas se muestra la aplicación de un conjunto de métodos relacionados y el porqué se escogieron de entre los demás, al final están las conclusiones y un pequeño análisis de la experiencia de desarrollar de esta forma.

Objetivos

Objetivo General

El objetivo general es crear una aplicación que permita administrar el área de producción en una empresa farmacéutica; principalmente en la generación de órdenes. Esta aplicación deberá ser una herramienta útil, fácil de utilizar y que tenga un diseño que nos permita crecer la aplicación para adaptarla posteriormente y hacer que interactúe con otros sistemas de la empresa, así como también sea posible en algún momento generar algunos cambios en su funcionamiento ya que debido a la dinámica del área puede ser necesario modificar algunas reglas del negocio.

Se tienen como principales objetivos y metas las siguientes:

- Ofrecer una herramienta capaz de solucionar los problemas y satisfacer las necesidades que en este momento tiene una empresa farmacéutica en el área de producción.
- Proveer de un repositorio unificado de datos en donde se agregue toda la información transaccional de esta área y posiblemente de otras para su posterior explotación en otro tipo de sistemas.
- Proporcionar al área de producción con los elementos necesarios para poder generar y administrar sus órdenes de producción eficientemente.
- Aprovechar las tecnologías informáticas disponibles para ofrecer ventajas en cuanto a facilidad de captura de datos, manejo de la información, procesos automatizados, reportes de operaciones, etcétera.
- Emplear el conocimiento y técnicas de la Ingeniería de Software para desarrollar una aplicación con la calidad adecuada.
- Proveer de la infraestructura transaccional de datos, con esto me refiero a que se debe guardar en esta base todas las transacciones generadas por la empresa en las diferentes áreas y estas se deben generar en línea. Teniendo además como característica la consistencia de todos los datos

En conjunto, lo que se espera como alcance es tener por un lado una herramienta útil, que facilite la operación en el área de producción al cumplir con un grupo de requerimientos establecidos y por otra parte, que dicha herramienta sea desarrollada utilizando el conocimiento de Ingeniería de Software para asegurar la mejor forma de generar esta aplicación.

Capítulo 1. Antecedentes

1.1 Marco Histórico

Existe una parte de la Ingeniería que cubre los aspectos relacionados con el desarrollo de software y que es llamada Ingeniería de Software.

La Ingeniería de Software va a introducirse en la cuarta década de su existencia y sufre de muchos puntos fuertes y débiles. La Ingeniería de Software se va aproximando a su edad media con muchos logros a sus espaldas, pero con un trabajo significativo todavía por hacer. Hoy en día, está reconocida como una disciplina legítima, digna de tener una investigación seria, un estudio concienzudo y un grande y tumultuoso debate. En la industria el Ingeniero del Software ha sustituido al programador como título de trabajo preferente. Los modelos de procesos de software, métodos de Ingeniería de Software y herramientas se han adoptado con éxito en el amplio espectro de las aplicaciones industriales. Los gestores y usuarios reconocen la necesidad de un enfoque más disciplinado del software.

La búsqueda de técnicas que mejoren la calidad y permitan reducir los costos de las soluciones basadas en computadoras ha sido uno de los objetivos más perseguidos desde los inicios de la informática. A mediados de los 60, la creación de un producto de software se convertía en una tarea angustiosa, se hizo por tanto necesario introducir una serie de herramientas y procedimientos que facilitaran por un lado, la labor de creación de nuevo software y por otro, la comprensión y el manejo del mismo. Estos fueron los inicios de la Ingeniería de Software. Con el paso del tiempo, la evolución de estos métodos nos ha llevado a reconocer a la Ingeniería de Software como una verdadera ciencia, derivada de una investigación seria y de un estudio minucioso.

1.2 La evolución del Software

Durante los primeros años de la era de la computadora, el software se contemplaba como un añadido. La programación de computadoras era un "arte de andar por casa" para el que existían pocos métodos sistemáticos. El desarrollo del software se realizaba virtualmente sin ninguna planificación, hasta que los planes comenzaron a descalabrarse y los costos a correr. Los programadores trataban de hacer las cosas bien, y con un esfuerzo heroico, a menudo salían con éxito. El software se diseñaba a medida para cada aplicación y tenía una distribución relativamente pequeña.

La mayoría del software se desarrollaba y era utilizado por la misma persona u organización. La misma persona lo escribía, lo ejecutaba y si fallaba, lo depuraba. Debido a este entorno personalizado del software, el diseño era un proceso implícito, realizado en la mente de alguien y la documentación normalmente no existía.

La segunda era en la evolución de los sistemas de computadora se extiende desde la mitad de la década de los sesenta hasta finales de los setenta. La multiprogramación y los sistemas multiusuario introdujeron nuevos conceptos de interacción hombre-máquina. Las técnicas interactivas abrieron un nuevo mundo de aplicaciones y nuevos niveles de sofisticación del hardware y del software. Los sistemas de tiempo real podían recoger, analizar y transformar datos de múltiples fuentes, controlando así los procesos y produciendo salidas en milisegundos en lugar de minutos. Los avances en los dispositivos de almacenamiento en línea condujeron a la primera generación de sistemas de gestión de bases de datos.

La segunda era se caracterizó también por el establecimiento del software como producto y la llegada de las "casas del software". Los patronos de la industria, del gobierno y de la universidad se aprestaban a "desarrollar el mejor paquete de software" y ganar así mucho dinero.

Conforme crecía el número de sistemas informáticos, comenzaron a extenderse las bibliotecas de software de computadora. Las casas desarrollaban proyectos en los que se producían programas de decenas de miles de sentencias fuente. Todos esos programas, todas esas sentencias fuente tenían que ser corregidos cuando se detectaban fallos, modificados cuando cambiaban los requisitos de los usuarios o adaptados a nuevos dispositivos hardware que se hubieran adquirido. Estas actividades se llamaron colectivamente mantenimiento del software.

La tercera era en la evolución de los sistemas de computadora comenzó a mediados de los años setenta y continuó más allá de una década. El sistema distribuido, múltiples computadoras, cada una ejecutando funciones concurrentes y comunicándose con alguna otra, incrementó notablemente la complejidad de los sistemas informáticos. Las redes de área local y de área global, las comunicaciones digitales de alto ancho de banda y la creciente demanda de acceso "instantáneo" a los datos, supusieron una fuerte presión sobre los desarrolladores del software.

La conclusión de la tercera era se caracterizó por la llegada y amplio uso de los microprocesadores. El microprocesador ha producido un extenso grupo de productos inteligentes, desde automóviles hasta hornos de microondas, desde robots industriales hasta equipos de diagnóstico médico.

La cuarta era de la evolución de los sistemas informáticos se aleja de las computadoras individuales y de los programas de computadoras, dirigiéndose al impacto colectivo de las computadoras y del software. Potentes máquinas personales controladas por sistemas operativos sofisticados, en redes globales y locales, acompañadas por aplicaciones de software avanzadas se han convertido en la norma.

La industria del software ya es la cuna de la economía del mundo. Las técnicas de la cuarta generación para el desarrollo del software están cambiando en la forma en que la comunidad del software construye programas informáticos. Las tecnologías orientadas a objetos están desplazando rápidamente los enfoques de desarrollo de software más convencionales en muchas áreas de aplicaciones.

Sin embargo, un conjunto de problemas relacionados con el software ha persistido a través de la evolución de los sistemas basados en computadora, y estos problemas continúan aumentando.

- Los avances del hardware continúan dejando atrás nuestra habilidad de construir software para alcanzar el potencial del hardware.
- Nuestra habilidad de construir nuevos programas no pueden ir al mismo ritmo de la demanda de nuevos programas, ni podemos construir programas lo suficientemente rápido como para cumplir las necesidades del mercado y de los negocios.

- El uso extenso de computadoras ha hecho a la sociedad cada vez más dependiente de la operación fiable del software. Cuando el software falla, pueden ocurrir daños económicos enormes y ocasionar sufrimiento humano.
- Luchamos por construir software informático que tengan fiabilidad y alta calidad.
- Nuestra habilidad de soportar y mejorar los programas existentes se ve amenazada por diseños pobres y recursos inadecuados.

En respuesta a estos problemas, las prácticas de la Ingeniería de Software se están adoptando en toda la industria.

1.3 Características del Software

Para poder comprender lo que es el software (y consecuentemente la Ingeniería de Software), es importante examinar las características del software que lo diferencian de otras cosas que los hombres pueden construir.

El software es un elemento del sistema que es lógico, en lugar de físico. Por lo tanto el software tiene unas características considerablemente distintas a las del hardware:

El software se desarrolla, no se fabrica en un sentido clásico. Aunque existen similitudes entre el desarrollo del software y la construcción del hardware, ambas actividades son fundamentalmente diferentes. En ambas actividades la buena calidad se adquiere mediante un buen diseño, pero la fase de construcción del hardware puede introducir problemas de calidad que no existen (o son fácilmente corregibles) en el software. Ambas actividades dependen de las personas, pero la relación entre las personas dedicadas y el trabajo realizado es completamente diferente para el software. Ambas actividades requieren de la construcción de un producto, pero los métodos son diferentes.

Los costos del software se encuentran en la ingeniería. Esto significa que los proyectos de software no se pueden gestionar como si fueran proyectos de fabricación.

El software no se estropea. El software no es susceptible a los males del entorno que hacen que el hardware se estropee. Otro aspecto de ese deterioro ilustra la diferencia entre el hardware y el software. Cuando un componente se estropea, se sustituye por una pieza de repuesto. No hay pieza de repuesto para el software. Cada fallo en el software indica un error en el diseño o en el proceso mediante el que se tradujo el diseño a código maquina ejecutable. Por tanto, el mantenimiento del software tiene una complejidad considerablemente mayor que la del mantenimiento del hardware.

Aunque la industria tiende a ensamblar componentes, la mayoría del software se construye a la medida. No existen catálogos de componentes de software. Se puede comprar software ya desarrollado, pero solo como una unidad completa, no como componentes que pueden reensamblarse en nuevos programas.

Importante para un componente de software de alta calidad es que el componente debería diseñarse e implementarse para que pueda volver a ser utilizado.

La reutilización es una característica que implica que pueda volver a ser utilizado en muchos programas diferentes.

Los componentes de software se construyen mediante un lenguaje de programación que tiene un vocabulario limitado, una gramática definida explícitamente y reglas bien formadas de sintaxis y semántica.

Aplicaciones del Software

El software puede aplicarse en cualquier situación en la que se haya definido previamente un conjunto especifico de pasos procedimentales (es decir, un algoritmo). (Excepciones notables a esta regla son el software de los sistemas expertos y de redes neuronales).

Las siguientes áreas del software indican la amplitud de las aplicaciones potenciales:

Software de Sistemas: El Software de Sistemas es un conjunto de programas que han sido escritos para servir a otros programas. El área del Software de Sistemas se caracteriza por una fuerte interacción con el hardware de la computadora; una gran utilización por múltiples usuarios; una operación concurrente que requiere una planificación, el compartir recursos y una sofisticada gestión de procesos; unas estructuras de datos complejas y múltiples interfaces externas. (Por ejemplo: compiladores, editores, utilidades, ciertos componentes del sistema operativo, utilidades de manejo de periféricos, procesadores de telecomunicaciones).

Software de Tiempo Real: El software que mide/analiza/controla sucesos del mundo real conforme ocurren, se denomina de Tiempo Real. Entre los elementos del Software de Tiempo Real se incluyen: un componente de adquisición de datos que recolecta y da formato a la información recibida del entorno externo, un componente de análisis que transforma la información recibida del entorno externo, un componente de análisis que transforma la información según lo requiera la aplicación, un componente de control/salida que responda al entorno externo y un componente de monitorización que coordina todos los demás componentes, de forma tal que pueda mantenerse la respuesta en tiempo real.

Software de Sistemas de Información de Gestión: El procesamiento de información comercial constituye la mayor de las áreas de aplicación del software. Los sistemas discretos (p. Ej.: nominas, cuentas de haberes/débitos, inventarios, etc.), han evolucionado hacia el Software de Sistemas de Información de Gestión (SIG), que accede a una o más bases de datos grandes que contienen información comercial. Las aplicaciones en esta área reestructuran los datos existentes para facilitar las operaciones comerciales o gestionar la toma de decisiones. Además de las tareas convencionales de procesamiento de datos, las aplicaciones de software de gestión también realizan cálculo interactivo (p. Ej.: el procesamiento de transacciones en puntos de ventas).

Software de Ingeniería y Científico: El software de Ingeniería y Científico está caracterizado por los algoritmos de manejo de números. Las aplicaciones van desde la astronomía a la vulcanología, desde el análisis de la presión de los automotores a la dinámica orbital de los lanzadores espaciales y desde la biología molecular a la fabricación automática.

Software Empotrado: El software Empotrado reside en memoria de solo lectura y se utiliza para controlar productos y sistemas de los mercados industriales y de consumo. El software Empotrado puede ejecutar funciones muy limitadas y curiosas (p. Ej.: el control de las teclas de un horno de microondas) o suministrar una función significativa y con capacidad de control (p. Ej.: funciones digitales

en un automóvil, tales como control de la gasolina, indicaciones en el salpicadero, sistemas de frenado, etc.).

Software de Computadoras Personales: El mercado del Software de Computadoras Personales ha germinado en la pasada década. El procesamiento de textos, las hojas de cálculo, los gráficos por computadora, multimedia, entretenimientos, gestión de bases de datos, aplicaciones financieras de negocios y personales, y redes o acceso a bases de datos externas son algunos ejemplos de los cientos de aplicaciones.

Software de Inteligencia Artificial: El Software de Inteligencia Artificial (IA) hace uso de algoritmos no numéricos para resolver problemas complejos para los que no son adecuados el cálculo o el análisis directo. El área más activa de la IA es la de los sistemas expertos, también llamados sistemas basados en el conocimiento.

Hoy en día el software tiene un doble papel. Es un producto y al mismo tiempo el vehículo para hacer entrega de un producto. Como producto, hace entrega de la potencia informática del hardware informático. Si reside dentro de un teléfono celular u opera dentro de una computadora central, el software es un transformador de información, produciendo, gestionando, adquiriendo, modificando, mostrando o transmitiendo información que puede ser tan simple como un solo bit, o tan compleja como una simulación en multimedia. Como vehículo utilizado para hacer entrega del producto, el software actúa como la base de control de la computadora (sistemas operativos), la comunicación de información (redes), y la creación y control de otros programas (herramientas de software y entornos).

El software de computadora, se ha convertido en el alma mater. Es la máquina que conduce a la toma de decisiones comerciales. Sirve como la base de investigación científica moderna y de resolución de problemas de ingeniería. Es el factor clave que diferencia los productos y servicios modernos. Está inmerso en sistemas de todo tipo: de transportes, médicos, de telecomunicaciones, militares, procesos industriales, entretenimientos, productos de oficina, etc. El software será el que nos lleve de la mano en los avances en todo desde la educación elemental a la Ingeniería Genética.

1.4 ¿Qué es Ingeniería de Software?

La Ingeniería de software es una disciplina o área de la Informática o Ciencias de la Computación, que ofrece métodos y técnicas para desarrollar y mantener software de calidad que resuelven problemas de todo tipo. Hoy día es cada vez más frecuente la consideración de la Ingeniería de Software como una nueva área de la Ingeniería, y el Ingeniero del Software comienza a ser una profesión implantada en el mundo laboral internacional con derechos, deberes y responsabilidades que cumplir, junto a una ya reconocida consideración social en el mundo empresarial.

La Ingeniería de Software trata con áreas muy diversas de la Informática y de las Ciencias de la Computación, tales como construcción de compiladores, sistemas operativos o desarrollos de Intranet/Internet, abordando todas las fases del ciclo de vida del desarrollo de cualquier tipo de sistemas de información y aplicables a una infinidad de áreas tales como: negocios, investigación científica, medicina, producción, logística, banca, control de tráfico, meteorología, el mundo del derecho, la red de redes Internet, redes Intranet y Extranet, etc.

Definición del término Ingeniería de Software

El termino Ingeniería se define en el Diccionario de la Real Academia Española de la Lengua como: "1. Conjunto de conocimientos y técnicas que permiten aplicar el saber científico a la utilización de la materia y de las fuentes de energía. 2. Profesión y ejercicio del Ingeniero" y el término Ingeniero se define como: persona que profesa o ejerce la Ingeniería. De igual modo la Real Academia de Ciencias Exactas, Físicas y Naturales de España define el término Ingeniería como: " Un conjunto de conocimientos y técnicas cuya aplicación permite la utilización racional de los materiales y de los recursos naturales, mediante invenciones, construcciones u otras realizaciones provechosas para el hombre".

Evidentemente, si la Ingeniería de Software es una nueva Ingeniería, parece lógico que reúna las propiedades citadas en las definiciones anteriores. Sin embargo ni el DRAE (Diccionario de la Real Academia Española de la Lengua), ni la Real Academia Española de Ciencias han incluido todavía el término en sus últimas ediciones; en consecuencia vamos a recurrir para su definición más precisa a algunos de los autores más acreditados que comenzaron en su momento a utilizar el término o bien en las definiciones dadas por organismos internacionales profesionales de prestigio tales como IEEE (Institute of Electrical and Electronics Engineers) o ACM (Association for Computer Machinery), de los cuales se han seleccionado las siguientes definiciones de Ingeniería de Software**.**

Definición 1: Ingeniería de Software es el estudio de los principios y metodologías para desarrollo y mantenimiento de sistemas de software [Zelkovits, 1978].

Definición 2: Ingeniería de Software es la aplicación práctica del conocimiento científico en el diseño y construcción de programas de computadora y la documentación necesaria requerida para desarrollar, operar (funcionar) y mantenerlos [Bohem, 1976].

Definición 3: Ingeniería de Software trata del establecimiento de los principios y métodos de la Ingeniería a fin de obtener software de modo rentable que sea fiable y trabaje en máquinas reales [Bauer, 1972].

Definición 4: La aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, operación (funcionamiento) y mantenimiento del software; es decir, la aplicación de Ingeniería al software [IEEE, 1993].

Una perspectiva industrial

En los primeros días de la informática, los sistemas basados en computadora se desarrollaban usando técnicas de gestión orientadas a hardware. Los gestores del proyecto se centraban en el hardware, debido a que era el factor principal del presupuesto en el desarrollo del sistema. Para controlar los costos del hardware, los gestores instituyeron controles formales y estándares técnicos. Exigían un análisis y diseño completo antes de que algo se construyera. Medían el proceso para determinar dónde podían hacerse mejoras. Dicho sencillamente, aplicaban los controles, los métodos y las herramientas que reconocemos como Ingeniería del Hardware. Desgraciadamente, el software no era normalmente más que un añadido.

En los primeros días, la programación se veía como un arte. Existían pocos métodos formales y pocas personas los usaban.

Hoy, la distribución de costos en el desarrollo de sistemas informáticos ha cambiado drásticamente. El software, en lugar del hardware, es normalmente el elemento principal del costo.

En las décadas pasadas los ejecutivos y muchos aprendices técnicos se habían hecho las siguientes preguntas:

- ¿Por qué lleva tanto tiempo terminar los programas?
- ¿Por qué es tan elevado el costo?
- ¿Por qué no podemos encontrar todos los errores antes de entregar el software a nuestros clientes?
- ¿Por qué nos resulta difícil constatar el progreso conforme se desarrolla el software?

Estas y otras muchas cuestiones son una manifestación del carácter del software y de la forma en que se desarrolla, un problema que ha llevado a la adopción de la Ingeniería de Software como práctica.

Competitividad del Software

Durante muchos años, los desarrolladores de software empleados por grandes y pequeñas compañías eran los únicos en este campo. Como todos los programas se construían de forma personalizada, los desarrolladores de este software doméstico dictaban los costos, planificación y calidad. Hoy, todo esto ha cambiado.

El software ahora es una empresa extremadamente competitiva. El software que se construía internamente ahora se puede adquirir en tiendas. Muchas empresas que en su momento pagaban legiones de programadores para crear aplicaciones especializadas ahora ofrecen a un tercero mucho del trabajo del software.

El Software

La descripción de software en un libro de texto podría tomar la forma siguiente: el software es (1) instrucciones que cuando se ejecutan proporcionan la función y el rendimiento deseados, (2) estructuras de datos que permiten a los programas manipular adecuadamente la información y (3) documentos que describen la operación y el uso de programas.

Evolución de la Ingeniería de Software

Inicialmente la programación de las computadoras era un arte que no disponía de métodos sistemáticos en los cuales basarse para la realización de productos de software. Se realizaban sin ninguna planificación, evolución y perspectivas de la Ingeniería de Software. Posteriormente, desde mediados de los 60 hasta finales de los 70 se caracterizó por el establecimiento del software como un producto que se desarrollaba para una distribución general. En esta época nació lo que se conoce como el mantenimiento del software que se da cuando cambian los requisitos de los usuarios y se hace necesaria la modificación del software.

El esfuerzo requerido para este mantenimiento era en la mayoría de los casos tan elevado que se hacía imposible su mantenimiento. A continuación, surge una etapa que se caracteriza por la aparición de una serie de técnicas como la Programación Estructurada y las Metodologías de Diseño que solucionan los

problemas anteriores. A finales de esta etapa aparecen las herramientas CASE (Computer Aided Software Engineering), aunque como podemos imaginar eran muy rudimentarias.

En las décadas de 1980 y 1990, dos tendencias dominaron la Ingeniería de Software. Una fue el crecimiento explosivo de aplicaciones, incluidas las asociadas con Internet. La otra, el florecimiento de nuevas herramientas y paradigmas (como la orientación a objetos).

Sin embargo a pesar del advenimiento de nuevas tendencias, las actividades básicas requeridas para la construcción de software han permanecido estables. Estas actividades incluyen:

- Definición del proceso de desarrollo de software que se usará.
- Administración del proyecto de desarrollo.
- Descripción del producto de software que se desea.
- Diseño del producto.
- Implementación del producto (desarrollo o programación).
- Prueba de las partes del producto.
- Integración de las partes del producto.
- Integración de las partes del producto y pruebas del producto completo.
- Mantenimiento del producto.

El software se ha convertido en el elemento clave de la evolución de los sistemas y productos informáticos. En las pasadas cuatro décadas, el software ha pasado de ser una resolución especializada de problemas y una herramienta de análisis de información, a ser una industria por si misma. Pero la temprana cultura e historia de la programación ha creado un conjunto de problemas que persisten todavía. El software se ha convertido en un factor que limita la evolución de los sistemas informáticos.

La Ingeniería de Software es una tecnología multicapa, como se muestra en la siguiente figura.

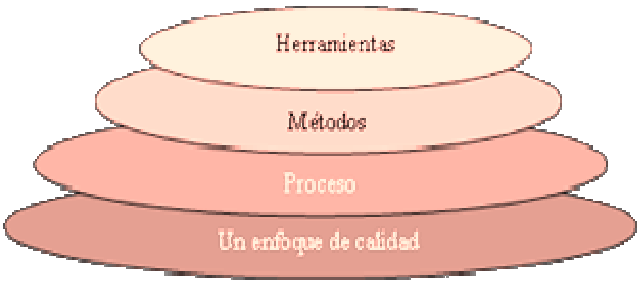


Fig. 1.- Capas de la Ingeniería de Software.

Cualquier enfoque de ingeniería (incluida Ingeniería de Software) deben apoyarse sobre un compromiso de calidad.

El fundamento de la Ingeniería de Software es la capa de proceso. El proceso de la Ingeniería de Software es la unión que mantiene juntas las capas de tecnología y que permite un desarrollo racional y oportuno de la Ingeniería de Software. El proceso define un marco de trabajo para un conjunto de áreas clave de proceso [PAU93]. Que se deben establecer para la entrega efectiva de la tecnología de la Ingeniería de Software. Las áreas claves del proceso forman la base del control de gestión de proyectos de software y establecen el contexto en el que se aplican los métodos técnicos, se obtienen productos del trabajo (modelos, documentos, datos, informes, formularios, etc.) se establecen hitos, se asegura la calidad y el cambio se gestiona adecuadamente.

Los métodos de la Ingeniería de Software indican *cómo* construir técnicamente el software. Los métodos abarcan una gran gama de tareas que incluyen análisis de requisitos, diseño, construcción de programas, pruebas y mantenimiento. Los métodos de la Ingeniería de Software dependen de un conjunto de principios básicos que gobiernan cada área de la tecnología e incluyen actividades de modelado y otras técnicas descriptivas.

Las herramientas de la Ingeniería de Software proporcionan un enfoque automático o semiautomático para el proceso y para los métodos. Cuando se integran herramientas para que la información creada por una herramienta la pueda utilizar otra, se establece un sistema de soporte para el desarrollo del software llamado Ingeniería de Software asistida por computadora (CASE).

La ingeniería es el análisis, diseño, construcción, verificación y gestión de entidades técnicas (o sociales).

Con independencia en la entidad a la que se va a aplicar ingeniería se deben cuestionar y responder las siguientes preguntas:

¿Cuál es el problema a resolver?

¿Cuáles son las características de la entidad que se utiliza para resolver el problema?

¿Cómo se realizará la entidad (y la solución)?

¿Cómo se construirá la entidad?

¿Cómo se apoyará la entidad cuando usuarios soliciten correcciones, adaptaciones y mejoras de la entidad?

El trabajo que se asocia a la Ingeniería de Software se puede dividir en tres fases genéricas con independencia del área de aplicación, tamaño o complejidad del proyecto. Cada fase se encuentra con una o varias cuestiones de las destacadas anteriormente.

La fase de definición se centra sobre el *qué*. Es decir durante la definición, el que desarrolla el software intenta identificar qué información ha de ser procesada, qué función y rendimiento se desea, qué comportamiento del sistema, qué interfaces van a ser establecidas, qué restricciones de diseño existen y qué criterios de validación se necesitan para definir un sistema correcto. Por tanto, han de identificarse los requisitos clave del sistema y del software. Aunque los métodos aplicados durante la fase de definición variarán dependiendo del paradigma de Ingeniería de Software (o combinación de paradigmas) que se aplique, de alguna manera tendrán lugar tres tareas principales: ingeniería de sistemas o de información, planificación del proyecto del software y análisis de los requisitos.

La fase de desarrollo se centra en el *cómo*. Es decir, durante el desarrollo un Ingeniero del Software intenta definir cómo han de diseñarse las estructuras de datos, cómo ha de implementarse la función dentro de una arquitectura de software, cómo han de implementarse los detalles procedimentales, cómo han de caracterizarse interfaces, cómo ha de traducirse el diseño en un lenguaje de programación (o lenguaje no procedimental) y cómo ha de realizarse la prueba. Los métodos aplicados durante la fase de desarrollo variarán, aunque las tres tareas específicas técnicas deberían ocurrir siempre: diseño del software, generación de código y prueba del software.

La fase de mantenimiento se centra en el cambio que va asociado a la corrección de errores, a las adaptaciones requeridas a medida que evoluciona el entorno del software y a cambios debidos a las mejoras producidas por los requisitos cambiantes del cliente.

Durante la fase de mantenimiento se encuentran cuatro tipos de cambios:

- **Corrección**. El mantenimiento correctivo cambia el software para corregir los defectos.
- **Adaptación.** El mantenimiento adaptable produce modificación en el software para acomodarlo a los cambios de su entorno externo.
- **Mejora**. Conforme se utilice el software, el cliente/usuario puede descubrir funciones adicionales que van a producir beneficios. El mantenimiento perfectivo lleva al software más allá de sus requisitos funcionales originales.
- **Prevención**. El software de computadora se deteriora debido al cambio, y por esto el mantenimiento preventivo también llamado reIngeniería de Software, se debe conducir a permitir que el software sirva para las necesidades de los usuarios finales. En esencia, el mantenimiento preventivo hace cambios en programas de computadora a fin de que se puedan corregir, adaptar y mejorar más fácilmente.

Las fases y los pasos relacionados descritos en esta visión genérica de la Ingeniería de Software se complementan con un número de actividades protectoras, las cuales se incluyen:

- Seguimiento y control del proyecto de software.
- Revisiones técnicas formales.
- Garantía de calidad del software.
- Preparación y producción de documentos.
- Gestión de reutilización.
- Mediciones.
- Gestión de riesgos.

El software se compone de programas, datos y documentos. Cada uno de estos elementos se compone de una configuración que se crea como parte del proceso de la Ingeniería de Software. El intento de la Ingeniería de Software es proporcionar un marco de trabajo para construir software con mayor calidad.

1.5 Análisis del problema

Antecedentes

En la empresa, anteriormente existían una serie de sistemas creados en ambiente MSDOS, principalmente creados en Clipper y Fox Pro que ayudaban a llevar tanto la operación como la administración de la empresa.

Estos sistemas tenían las siguientes desventajas:

- **Procesos aislados.** Es decir, los sistemas no se comunicaban entre si, lo que provocaba que cada sistema tuviera una base de datos, un entorno propio y manejaran su propia información por separado.
- **No existían catálogos estándares.** Como cada sistema usaba sus propios datos y estos estaban aislados, existía una diversidad de catálogos para cada sistema aunque estos se refirieran a entidades similares, como por ejemplo los catálogos de medicamentos y de suturas, a pesar de que ambos tenían características similares, su almacenamiento era en estructuras o tablas con características diferentes.
- **Existían muchos casos de reproceso de información.** En ocasiones la información de un sistema se tenía que complementar con los datos generados por otro, para poder generar nuevos procesos o nueva información, como por ejemplo se tenían que pasar los datos del sistema de compras para alimentar el sistema de contra recibos.
- **Trabajo manual de información en algunas áreas.** Los sistemas con que contaban, al no tener la información de otros sistemas no proveían de reportes que cruzaran información o que conjuntan la información para reportarla, por lo que manualmente se tenían que procesar y juntar mucha información para poder explotarla.
- **Islas de información.** Los sistemas anteriores eran como islas de información ya que al estar aislados no se podían ver los procesos como una parte de un todo, si no más bien procesos aislados unos de otros.
- **Sistemas no muy amigables.** Como se podrá recordar, los sistemas del sistema operativo MS-DOS por su misma naturaleza no son muy amigables ya que no se pueden agregar iconos, ni muchas imágenes; las ayudas son muy limitadas en general debido a que ya la mayoría de los programas que se usan en ambiente Windows tienen ciertos estándares que los de DOS no tienen y la mayoría de la gente trabaja ya en ambientes gráficos, esto dificulta tanto su uso como la curva de aprendizaje.
- **Reglas de negocio precarias.** Al no tener una visión general de la empresa estos sistemas, las reglas de negocio no estaban completas y en algunas ocasiones no se contemplaban muchos procesos integrales.
- **Infraestructura informática poco sólida.** Con la problemática antes mencionada se puede concluir que la infraestructura informática es poco sólida ya que si se quieren tener otro tipo de procesamientos como: Reportes Gerenciales, OLAP (On Line Analytical Processing o en español Procesamiento Analítico en Línea) o publicaciones de datos en Internet, Business to Business, la infraestructura con la que se cuenta no nos sirve de mucho.

Motivación

Como se vio anteriormente existe todavía mucha información aislada, grandes tiempos de reprocesamiento, subprocesos manuales o por lotes. Falta de información actualizada en determinados momentos porque algunos procesos son mensuales y se conocen los datos hasta que se hace el corte y se procesa la información.

Se está trabajando en una serie de sistemas entre los cuales se encuentra el sistema de producción en los cuales se tendrá una base de datos única y estos programas interactuarán entre ellos para generar toda la información transaccional de la empresa.

Al tener estos sistemas actuando conjuntamente unos con otros, se podrá tener la información en línea y se tendrán los datos necesarios y en el momento justo para poder tomar decisiones, aumentando también la productividad de las diferentes áreas que se ahorrarán mucho trabajo manual, dejarán de tener archivos en papel y podrán avocarse a tareas menos repetitivas y más tendientes a la mejora de las diferentes áreas que conforman la empresa.

Además, como se mencionó, es necesario robustecer la infraestructura con nuevos sistemas para con esto sentar las bases para otras tecnologías que permitan aprovechar los recursos de una mejor manera, así como para darle un valor agregado y competitividad a la empresa en general.

Este sistema que se piensa desarrollar es uno de una serie de programas que se han venido desarrollando y se tienen planeados para darle esta nueva infraestructura a la empresa, así como también una mayor competitividad al tener la información en línea y evitarse tanto trabajo manual en cuestión de información.

Además de que es muy importante tener cuantificadas algunas cifras para poder mejorar los procesos tanto productivos como administrativos de la empresa.

Necesidades del negocio

En este caso se trata de una empresa farmacéutica mediana que requiere aprovechar la tecnología para optimizar su desempeño y poder posicionarse mejor en el mercado, además dentro de estos nuevos sistemas en general y en el de producción en particular se incluyen varios parámetros a medir que nos permitirán hacer posteriores perfeccionamientos a los procesos actuales ya que lo que no se puede medir es poco factible de mejorar.

Por ejemplo es necesario que el negocio conozca tiempos y costos de producción para medir la eficiencia de los procesos productivos tanto en costo como en tiempo principalmente, aunque también se puede medir en calidad y otros parámetros.

Otra de las necesidades que es necesario cubrir, es el registro de las actividades de esta área de manera consistente y adecuada, generando información que nos sirva para conocer los diferentes aspectos operacionales de esta área.

1.6 Definición del problema

Por lo anteriormente expuesto, nuestro problema en resumen es la creación de un sistema dentro de serie de los mismos que sustituyan a los anteriores y que permitan tener procesos más eficientes. Que sea

también un eslabón más en la cadena de producción, mejore la administración así como para la adopción de nuevas tecnologías que renueven y consoliden los procesos productivos. Un sistema empresarial normalmente se debe ver como una herramienta que ayuda a resolver una serie de problemáticas o que viene a transformar los procesos actuales para hacerlos más eficientes.

Ahora bien, ¿Cómo vamos a crear un sistema?, ¿En qué consiste crear un sistema?, ¿Qué formas existen para crear un sistema?, ¿Qué metodologías hay? , ¿Cómo se si el sistema que se generará cumplirá con las metas?

Todas estas preguntas plantean un problema muy complejo que puede ser dividido en dos partes principalmente. La primera parte sería el cómo se va a resolver una problemática mediante un sistema y la segunda parte es el cómo crear un sistema que cumpla con una serie de requisitos y restricciones de una manera eficiente.

Esto plantea por un lado hacer un análisis del sistema desde el punto de vista de que deberá hacer, que características deberá de tener, cuales son las restricciones del sistema y por otra parte será tomar una serie de decisiones en cuanto a qué metodologías seguiremos para lograr crear el sistema de una manera eficiente y que cumpla con todos los requisitos obtenidos en el análisis.

Al final, deberemos obtener tanto un sistema que cumpla con los requerimientos de calidad, construido de una manera eficiente, dentro de los tiempos especificados, que cumpla con los estándares y que además sea útil al cubrir las funciones especificadas en el análisis, es decir, necesitamos planear generar una herramienta que tenga todas las cualidades para resolver nuestra problemática en la empresa y por otro lado tenemos que ver el problema de cómo crear dicho instrumento de la mejor forma.

En los siguientes subtemas veremos la manera en que se analizó el sistema y cómo se llegó a obtener el conjunto de funcionalidades que el mismo debería de tener, es decir, veremos la primera parte de nuestro problema.

En el capítulo siguiente veremos todo el marco teórico en que se fundamenta el desarrollo de software y en el cual se basan todas las decisiones que se tomaron para desarrollar este sistema y más adelante veremos en detalle las diferentes opciones que se tomaron y como se fue desarrollando el sistema (segunda parte de nuestro problema).

Este sistema se justifica debido a que el software que opera actualmente necesita renovarse ya que carece de muchas características importantes que le restan eficiencia a nuestros procesos productivos así como a los administrativos. Afortunadamente ya existe un sistema previo del que podemos aprender tanto de sus características positivas como de sus carencias y además el personal ya está acostumbrado a usar un sistema dentro de sus procesos productivos lo cual nos ayudará a que este nuevo programa sea aceptado más fácilmente, además, con estos sistemas anteriores se probó que efectivamente la solución es la creación de un sistema, sólo que estos deben ir cambiando conforme avanza la tecnología o las reglas del negocio los hacen obsoletos.

1.7 Definición de características y restricciones del sistema

El sistema de producción tendrá en esta su primera versión la posibilidad de realizar las siguientes cosas:

El sistema deberá permitir administrar, esto es, deberá manejar altas, bajas y cambios de los diferentes

catálogos que conforman la estructura básica o esqueleto del sistema ya que estas estructuras son las que tendrán de alguna manera los datos básicos que al combinarlos conforman una parte importante de la información que maneja el sistema.

Estos son los catálogos:

- Divisiones.
- Procesos.
- Genéricos.
- Forma Farmacéutica.
- Presentación.
- CBMSS.
- Concentración.
- Contenido.
- Hebra.
- Aguja.
- Productos.

Estos catálogos nos servirán también para clasificar los productos y tenerlos bien identificados dentro de este conjunto de categorizaciones. También, servirán estos catálogos para formar la clave única de los productos que maneja la empresa, tanto insumos como productos fabricados.

A continuación explicaré un poco más detalladamente a que se refiere cada una de estas clasificaciones.

La división se refiere a la sección denominada con el mismo nombre de división dentro de la empresa, que produce determinado producto. Actualmente sólo existen dos principales divisiones, la de producción de medicamentos y la de producción de suturas; pero realmente el sistema podría manejar cualquier cantidad de divisiones.

Los procesos se refieren más bien a una clasificación de tipos de materiales, como por ejemplo material de empaque, impreso, caja, granel, semiterminados, producto terminado, etcétera.

Genéricos es un catálogo que nos permite mostrar los nombres de los productos en su descripción general y su nombre comercial.

La Forma Farmacéutica es un catálogo que indica que tipo de presentación tiene el medicamento a producir, es decir, si está en tabletas, suspensión, inyectable, sutura, etcétera.

El catálogo de presentación nos indica las diversas formas de presentar un producto, es decir, puede ser para una muestra médica, para una marca propia o venta a mostrador o cualquier otra presentación.

El catálogo CBMSS permite identificar a los productos pertenecientes al Cuadro Básico de Medicamentos del Sector Salud, para cuando se producen medicamentos de este cuadro básico se tengan ubicados y estandarizados estos medicamentos.

El catálogo de Concentración contiene las diferentes concentraciones de los medicamentos.

El contenido es un catálogo que contiene todos los diversos contenidos que se tienen en los productos, por ejemplo envases con 30 grageas, tiras con 12 pastillas, etcétera.

Este catálogo (Hebra) es usado para suturas y define las diferentes longitudes y calibres de las suturas.

Aguja es un catálogo que define los diferentes calibres y tipos de agujas que se usan en las suturas y en los inyectables.

El catálogo productos contiene todos los productos usados para producir medicinas, desde los componentes activos de las medicinas hasta las etiquetas usadas para la presentación final.

En general los procesos productivos están divididos en tres subprocesos o etapas, el primero es la fabricación de producto a granel, la segunda etapa es la de semiterminados en donde el granel en general se agrupa en las cantidades en las que la presentación tendrá. Por ejemplo, en algunos productos el semiterminados consiste en empacar o agrupar el granel en tiras de celofán o en frascos y finalmente la etapa de acondicionamiento o terminado incluye el ponerlos en las cajas con la presentación final, así como pegarles las etiquetas etcétera.

Para cada una de estas etapas productivas se define una fórmula para cada producto generado, es decir, existe una fórmula para granel, otra para semiterminados y una más para producto terminado. En esta fórmula se indican qué materiales se van a necesitar y en qué cantidades. Al igual que en la sección de catálogos es necesario que dentro del sistema se puedan administrar estas fórmulas, es decir que mediante el mismo se den de alta nuevas fórmulas, se puedan hacer modificaciones y se den de baja en cada una de sus tres etapas de producción.

Dentro de esta formulación, se deberán manejar también los costos de cada una de las etapas productivas, así como se necesitará distinguir de alguna forma las fórmulas de los productos que están incompletas de las que están completas.

Deberá existir un proceso de aprobación de estas fórmulas ya que este proceso de formulación es muy delicado. Inclusive la formulación debe estar de acuerdo a ciertos certificados de la Secretaría de Salud. Por esta razón, el sistema no deberá permitirnos trabajar con fórmulas que no están completas, lo único que se podrá hacer con ellas es completarlas y aprobarlas, para que una vez hecho este paso se puedan utilizar para generar ordenes de producción.

Otra cosa que es importante mostrar es cuánto cuesta en promedio producir un producto ya que esta información nos servirá en algunos reportes y para otros procesos tanto contables como financieros. La administración de la información de los costos unitarios no se hará en este sistema, simplemente lo que deberá mostrar el sistema son los costos unitarios de cada producto, es decir al generarse una orden, esta tomará los costos unitarios de cada material y los multiplicará por las cantidades utilizadas.

En el caso de producto terminado habrá la opción de mostrar desglosado en los tres procesos los costos de cada etapa.

Además, el sistema capturará el tiempo utilizado en los diferentes procesos productivos esto con la finalidad de ir midiendo el tiempo que tarda cada uno de ellos y poder más adelante optimizar los subprocesos que componen el proceso de fabricación de un producto.

También el sistema deberá generar una orden de surtido o embarque para el almacén donde se citan todos los materiales a surtir para que esta área programe su entrega de todos los productos para poder generar la orden.

El sistema debe considerar lo que se llama una explosión de materiales, lo que significa que son todos los materiales que se necesitan para producir una cantidad dada de producto, pues bien, el sistema deberá calcular una explosión de materiales de uno o varios productos y deberá validar e informar la existencia de estos en el almacén. Esto quiere decir que a la hora de generar una explosión, el sistema deberá mostrar producto por producto la existencia de los materiales en el almacén y en caso de no haber suficientes materiales no deberá permitir generar órdenes de ese producto.

Al igual que con los catálogos deberá el sistema de permitir llevar una administración de los productos que se agregan o se quitan de las diferentes explosiones.

También deberá ser posible generar una explosión de materiales con costos, es decir deberá decirme el sistema que materiales necesito, en qué cantidades y cuanto me costará cada uno de estos para una serie de productos y cantidades dadas en la explosión.

Una vez generada una explosión el sistema debe permitir generar las órdenes de productos de los cuales si tenga existencias de todos sus insumos y por otro lado que cuando el almacén tenga estas existencias de las materias primas, el usuario recalcule la explosión y pueda generar las órdenes restantes.

En este punto cabe señalar que únicamente para la primera etapa que es la de granel existe lo que se le denomina un lote estándar que es un lote cuyas cantidades son las óptimas para producir, principalmente debido a que las mermas son las menores al producir esas cantidades, para los semiterminados y terminados no existe este lote estándar.

Esta es otra de las cosas que el sistema tiene que contemplar, para un granel el sistema tiene que generar para una cantidad dada, tantos lotes estándar como deba de generar para obtener la mayor eficiencia posible en la producción.

Para las otras dos fases no se aplicará esta regla. Por ejemplo supongamos que tenemos que producir el producto A, cuyo lote estándar son 100,000 grageas y se hace la explosión por una cantidad de 280 000 el sistema generará 3 ordenes con lote estándar. Para las otras fases de producción no será necesario generar lotes estándar, por ejemplo si se va a generar un semiterminados de un producto B que contenga como insumo el producto A mencionado anteriormente, se puede generar cualquier cantidad que se desee del mismo como 1000 tiras de 10 grageas que en total tomarán 10,000 grageas del producto A.

La siguiente funcionalidad que deberá tener este sistema es que una vez generado el número de orden se deberá poder generar la orden como tal, esto es, que el sistema tome los lotes adecuados y en las cantidades adecuadas de cada material que conforma la generación de un producto. Aquí es importante señalar dos cosas, la primera es que del componente activo no se pueden mezclar lotes, es decir, la sustancia marcada como activa tiene una fecha de caducidad y cuando se fabrica un medicamento la fecha de caducidad es la fecha que tiene la sustancia activa y no se pueden mezclar lotes diferentes de sustancias activas. La segunda regla es que el sistema tomará los lotes más antiguos para producir una orden de producción, esto aplica para las tres etapas de producción, es decir, no es posible mezclar lotes ni con diferentes lotes de sustancias activas, ni con diferentes lotes de granel, ni con diferentes lotes de semiterminados.

Se deberá contemplar también la cancelación de órdenes y la devolución de los materiales de las mismas.

Al hacer la elaboración de la orden el sistema tiene que hacer el descargo de los materiales existentes en el almacén y al devolver los materiales los mismos se regresan al almacén.

Otro paso importante es el cierre de órdenes que consiste en indicar las cantidades reales que se producen, así como registrar los tiempos en los que se hizo cada proceso dentro de la fabricación del producto.

Será necesario que el sistema cuente con algunos reportes básicos para darle seguimiento a las órdenes de producción y que el sistema provea con cierta información básica a los usuarios, esta información básica será reportarle al usuario la secuencia de las diferentes órdenes generadas ya sea en un periodo de tiempo, por producto, por fase de producción o por una combinación de estos criterios. También se podrá reportar las órdenes que ya han sido cerradas, bajo la misma combinación de criterios.

Debido a que las fórmulas deben estar aprobadas deberán existir las firmas electrónicas de las áreas que tendrán que aprobar las mismas, por lo que será necesario proveer un pequeño módulo donde puedan cambiar estas firmas y contraseñas para que estas áreas le den mantenimiento a sus contraseñas para poder firmar y aprobar las fórmulas.

También el sistema deberá generar una orden de embarque para surtir la orden de producción en cuanto esta última se genera. Esto lo hará automáticamente.

Se consideró demasiado extenso el mostrar todo el sistema, razón por la cual para cuestiones ilustrativas se mostrará sólo una parte representativa para mostrar ya sea algunas consideraciones, diagramas, diseños, etcétera. Por ejemplo en esta parte los Casos de Uso que se consideraron fueron los de generación de órdenes.

La siguiente tabla muestra las principales funciones de la parte de generación de órdenes de producción:

Normalmente esta lista de funciones nos es de mucha utilidad ya que nos guía a lo largo del proceso de análisis y diseño del sistema en cuanto que aquí es donde se resumen las principales funcionalidades y estas deberán de poder verificarse en los posteriores diagramas, finalmente la aplicación deberá de cumplir con todas estas funciones.

Referencia	Función	Categoría
R3.1	Mostrar las explosiones de materiales de una explosión dada	Evidente
R3.2	Genera un número de orden a partir de una explosión de materiales	Evidente
R3.3	Validar que se generen órdenes sólo cuando haya los materiales requeridos.	Evidente
R3.4	Genera lotes estándares para la primera etapa de producción (Granel)	Oculta
R3.5	Generar una orden de Surtido para el almacén con los lotes de materiales y cantidades adecuadas.	Evidente
R3.6	Reducir las cantidades del inventario cuando se realiza una orden	Oculta
R3.7	Generar un costo por orden	Oculta
R3.8	Generar un costo por lote	Oculta
R3.9	Devuelve los materiales de una orden	Oculta
R3.10	Cancelación de una orden	Evidente
R3.11	Generar el cierre de órdenes	Evidente
R3.12	Registra el tiempo utilizado en los diferentes procesos de fabricación	Evidente

Tabla 1.- Funciones de la generación de órdenes.

1.8 Propuesta y selección de una estrategia de solución

Como se dijo anteriormente, nuestro problema está dividido en dos partes y una vez establecidas las características que deberá tener nuestro sistema, la estrategia de construcción del sistema será el escoger ciertas metodologías de la Ingeniería de Software para construirlo de la mejor manera posible, de acuerdo a estándares y de una forma no artesanal que nos permitirá asegurarnos que el sistema cumplirá con la calidad que se requiere y que podremos repetir y mejorar el éxito de nuestro sistema en otros similares o en futuras versiones del mismo.

Al utilizar metodologías lo que hacemos es aprovechar todo el conocimiento que ha generado la Ingeniería de Software, por lo que en el siguiente capítulo se mostrarán las diferentes metodologías que existen en todo el proceso de desarrollo de software para posteriormente escoger algunas de las metodologías y en ello basarnos para construir el sistema, básicamente la estrategia es escoger y utilizar las metodologías que se consideren mejores para la construcción del sistema y que al final obtengamos un producto confiable, construido en el tiempo y la forma adecuada, como ya también se dijo para llegar a una solución existen muchos caminos, pero para este caso trataremos de seguir el que nos provee más certidumbre y el que es menos empírico, en ese sentido esa va a ser nuestra estrategia.

En el capítulo 3 se comentará específicamente que metodologías de todo el marco teórico se tomaron y el porqué se decidió seguir alguna sobre las demás, así como también en las conclusiones se plasmarán los resultados de la experiencia de construir el sistema haciendo uso de estas formas de trabajo.

La utilidad del sistema será en función de la veracidad y exactitud del análisis que se haga del mismo y también cabe resaltar que precisamente en los desarrollos iterativos se tienen fases de análisis a lo largo del proceso de desarrollo que nos ayudan a ir mejorando el conocimiento de los alcances del sistema y de esta forma generar una herramienta que cumpla con los objetivos para los cuales fue hecha, así como también es muy válido que exista un control de cambios que nos permitirán cambiar y agregar requerimientos cuando por alguna razón se hayan omitido o cuando las reglas del negocio hayan cambiado debido a la dinámica de las empresas, esto nos permite que a pesar de que las reglas que dirigen el funcionamiento del sistema se vayan descubriendo o cambiando la construcción del mismo pueda hacer frente a estos eventos. Si bien, en algunas ocasiones esto tendrá un costo ya sea en personal involucrado, o alcances modificados, o tiempo de desarrollo que al final se reflejará en dinero. Estas dos estrategias (desarrollo iterativo y control de cambios) nos permiten lograr que un sistema no sea obsoleto desde antes de salir a producción.

Capítulo 2. Metodología del Desarrollo de software

2.1 El Proceso del Software

Un proceso de software se puede caracterizar como se muestra en la siguiente figura.

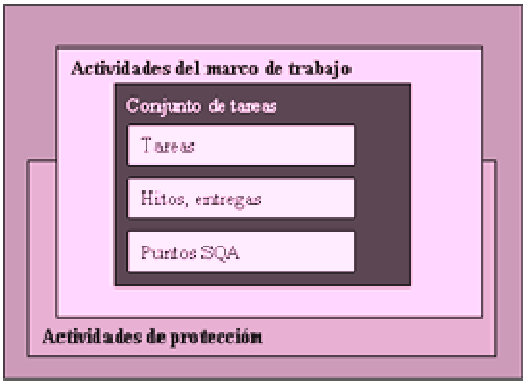


Fig. 2.- El proceso del software.

Se establece un marco común del proceso definiendo un pequeño número de actividades del marco de trabajo que son aplicables a todos los proyectos del software, con independencia de su tamaño o complejidad. Un número de conjuntos de tareas –cada uno es una colección de tareas de trabajo de Ingeniería de Software, hitos de proyectos, productos de trabajo y puntos de garantía de calidad- que permiten que las actividades del marco de trabajo se adapten a las características del proyecto del software y a los requisitos del equipo del proyecto. Finalmente, las actividades de protección –tales como la garantía de calidad del software, gestión de configuración del software y medición- abarcan el modelo de procesos. Las actividades de protección son independientes de cualquier actividad del marco de trabajo y aparecen durante todo el proceso.

En los últimos años, se ha hecho mucho énfasis en la "madurez del proceso". El Software Engineering Institute (SEI) ha desarrollado un modelo completo que se basa en un conjunto de funciones de Ingeniería de Software que deberían estar presentes conforme las organizaciones alcanzan diferentes niveles de madurez del proceso. Estos niveles son cinco y se definen de la forma siguiente:

Nivel 1: Inicial. Expresa el estado más primitivo. Sólo reconoce que la organización es capaz de producir productos de software. La organización no tiene un proceso reconocido para la producción de software, por lo que el éxito y la calidad de los productos y proyectos dependen por completo del esfuerzo individual. Por lo común los equipos dependen de los métodos proporcionados por un integrante del grupo que toma la iniciativa acerca del proceso que debe seguirse. El éxito de un proyecto tiene poca relación con el éxito de otro a menos que san similares y empleen ingenieros comunes. Cuando termina un proyecto, nada se registra de su costo, tiempos o calidad. El resultado es que los nuevos proyectos suelen realizarse de una manera no más competitiva que los anteriores.

Nivel 2: Repetible. El nivel "repetible" de CMM (Capability Mature Model), se aplica a las organizaciones capaces de rastrear sus proyectos hasta cierto grado. Mantienen registros de los costos y tiempos del proyecto. También describen la funcionalidad de cada producto por escrito. Así, es posible predecir el costo y los tiempos de proyectos muy similares que realiza el mismo equipo. Esto es una mejora respeto al nivel 1; lo que se necesita para mejorar es la capacidad para hacer predicciones independientes de las personas específicas que forman los equipos del proyecto.

Nivel 3: Definido. El nivel 3 se aplica a las organizaciones que reducen la dependencia excesiva en individuos específicos mediante la documentación del proceso e imponen un estándar. Algunas organizaciones adoptan estándares existentes como los del IEEE, mientras que otras definen los propios. En términos generales, siempre que la administración refuerce estándares profesionales coordinados y los ingenieros los implementen de manera uniforme, la organización está en el nivel 3. Esto suele requerir capacitación especial. Se permite que los equipos tengan flexibilidad para adaptar los estándares de la organización a las circunstancias especiales. Aunque las organizaciones de nivel 3 son capaces de producir aplicaciones con calidad consistente y aunque existen relativamente pocas organizaciones de este tipo, todavía les falta la capacidad de predecir. Pueden hacer pronósticos sólo para proyectos muy similares a los realizados en el pasado.

Nivel 4: Administrado. Se aplica a las organizaciones que pueden predecir el costo y la programación de tareas. Una manera de hacerlo es clasificar las tareas y sus componentes y medir y registrar el costo y tiempo para diseñar e implementar esas partes. Estas mediciones constituyen los datos métricos históricos que se usan para predecir el costo y los tiempos de las tareas subsecuentes. Esto requiere de una cantidad considerable de aptitudes de organización. Otra característica de este nivel es que se recopilan medidas detalladas del proceso del software y de la calidad del producto. Mediante la utilización de medidas detalladas, se comprenden y se controlan cuantitativamente tanto los productos como el proceso del Software.

El nivel 4 casi parece ser la capacidad máxima pero no lo es. Se sabe que la Ingeniería de Software cambia con rapidez. Por ejemplo, el paradigma de la orientación a objetos invadió la metodología: los conceptos de reutilización y componentes nuevas tienen un impacto creciente. Las mejoras y los paradigmas del futuro son impredecibles. Así, las capacidades de una organización de nivel 4 que no cambian de manera apropiada pueden disminuir.

Nivel 5: Optimizado. En lugar de intentar predecir los cambios futuros es preferible instituir procedimientos permanentes para buscar la explotación de métodos y herramientas nuevas y mejoradas. En otras palabras, su proceso incluye una manera sistemática de evaluar el proceso mismo de la organización.

Es importante destacar que cada nivel superior es la suma de los niveles anteriores, por ejemplo una

organización que esté en el nivel 4 requiere de todos los componentes del nivel 3, por ejemplo, que estén documentadas y bien definidas las actividades como la especificación del diseño de requisitos.

Proceso del Software

Para resolver los problemas reales de una industria, un Ingeniero del Software o un equipo de ingenieros deben incorporar una estrategia de desarrollo que acompañe al proceso, métodos y capas de herramientas como se mencionó anteriormente. Esta estrategia a menudo se llama modelo de proceso o paradigma de Ingeniería de Software. Se selecciona un modelo de proceso para la Ingeniería de Software según la naturaleza del proyecto y la aplicación, los métodos y las herramientas a utilizarse y los controles y entregas que se requieren.

2.1.1 Modelo Lineal Secuencial

También llamado ciclo de vida básico o modelo en cascada, el modelo lineal secuencial propone un enfoque sistemático, secuencial para el desarrollo del software que comienza en un nivel de sistemas y progresa con el análisis, diseño, codificación, pruebas y mantenimiento. La siguiente figura muestra este modelo.

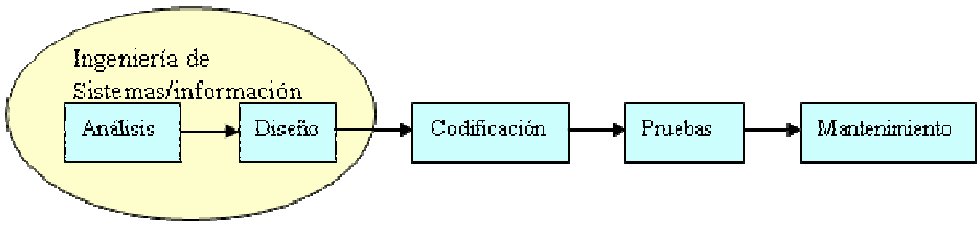


Fig. 3.- Modelo lineal secuencial.

Este modelo está estructurado según el ciclo de ingeniería convencional, el modelo comprende las siguientes actividades:

Ingeniería y modelado de Sistemas/Información. Como el software siempre forma parte de un sistema más grande (o empresa), el trabajo comienza estableciendo requisitos de todos los elementos del sistema y asignando al software algún subgrupo de estos requisitos. Esta visión del sistema es esencial cuando el software se debe interconectar con otros elementos como hardware, personas y bases de datos.

La ingeniería y el análisis de sistemas comprenden los requisitos que se recogen en el nivel del sistema con una pequeña parte del análisis y de diseño. La ingeniería de información abarca los requisitos que se recogen en el nivel de empresa estratégico y en el nivel del área de negocio.

Análisis de los requisitos del software. El proceso de reunión de requisitos se intensifica y se centra

especialmente en el software. Para comprender la naturaleza del (los) programa(s) a construirse, el Ingeniero (analista) del Software debe comprender el dominio de información del software, así como la función requerida, comportamiento, rendimiento e interconexión.

Diseño. El diseño del software es realmente un proceso de muchos pasos que se centra en cuatro atributos distintos de programa: estructura de datos, arquitectura de software, representaciones de interfaz y detalle procedimental (algoritmo). El proceso del diseño traduce requisitos en una representación del software donde se pueda evaluar su calidad antes de que comience la codificación.

Generación de código. El diseño se debe traducir en una forma legible por la máquina. El paso de generación de código lleva a cabo esta tarea. Si se lleva a cabo el diseño de una forma detallada, la generación de código se realiza mecánicamente.

Pruebas. Una vez que se ha generado el código, comienzan las pruebas del programa. El proceso de pruebas se centra en los procesos lógicos internos del software, asegurando que todas las sentencias se han comprobado y en los procesos externos funcionales; es decir, realizar las pruebas para la detección de errores y asegurar que la entrada definida produce resultados reales de acuerdo con los resultados requeridos.

Mantenimiento. El software indudablemente sufrirá cambios después de haber sido entregado al cliente (a excepción probablemente de software empotrado). Se producirán cambios porque se han encontrado errores, porque el software debe adaptarse para acoplarse a los cambios de su entorno externo (por ejemplo: se requiere un cambio debido a un sistema operativo o dispositivo periférico nuevo), o porque el cliente requiere mejoras funcionales o de rendimiento. El soporte y mantenimiento del software vuelve a aplicar cada una de las fases precedentes a un programa ya existente y no uno nuevo.

El modelo lineal secuencial es el paradigma más antiguo y más extensamente utilizado en la Ingeniería de Software. Sin embargo, este modelo tiene algunas desventajas. Entre los problemas que se encuentran al trabajar con este paradigma se encuentran:

- Los proyectos reales raras veces siguen el modelo secuencial que propone el modelo. Aunque el modelo lineal puede acoplar interacción, lo hace indirectamente. Como resultado, los cambios pueden causar confusión cuando el equipo del proyecto comienza.
- A menudo es difícil que el cliente exponga explícitamente todos los requisitos. El modelo lineal secuencial lo requiere y tiene dificultades a la hora de acomodar la incertidumbre natural al comienzo de muchos proyectos.
- El cliente debe tener paciencia. Una versión de trabajo del (los) programa(s) no estará disponible hasta que el proyecto esté muy avanzado. Un grave error puede ser desastroso si no se detecta hasta que se revisa el programa.

Cada uno de estos problemas es real. Sin embargo, el paradigma del ciclo de vida clásico tiene un lugar definido e importante en el trabajo de la Ingeniería de Software. Proporciona una plantilla en la que se encuentran métodos para el análisis, diseño, codificación, pruebas y mantenimiento. El ciclo de vida clásico sigue siendo el modelo de proceso más extensamente utilizado por la Ingeniería de Software. Pese a tener debilidades, es significativamente mejor que un enfoque hecho al azar para el desarrollo del software.

2.1.2 Modelo de construcción de prototipos

Un cliente, a menudo, define un conjunto de objetivos generales para el software, pero no identifica los requisitos detallados de entrada, proceso o salida. En otros casos, el responsable del desarrollo del software puede no estar seguro de la eficacia de un algoritmo, de la capacidad de adaptación de un sistema operativo, o de la forma en que debería tomarse la interacción hombre-máquina. En estas y en otras muchas situaciones, un paradigma de construcción de prototipos puede ofrecer el mejor enfoque.

El paradigma de construcción de prototipos comienza con la recolección de requisitos. El desarrollador y el cliente encuentran y definen los objetivos globales para el software, identifican los requisitos conocidos y las áreas del esquema en donde es obligatoria más definición. Entonces aparece un "diseño rápido". El diseño rápido se centra en una representación de esos aspectos del software que serán visibles para el usuario/cliente (por ejemplo: enfoques de entrada y formatos de salida). El diseño rápido lleva a la construcción de un prototipo. El prototipo lo evalúa el cliente/usuario y se utiliza para refinar los requisitos del software a desarrollar. La iteración ocurre cuando el prototipo se pone a punto para satisfacer las necesidades del cliente, permitiendo al mismo tiempo que el desarrollador comprenda mejor lo que se necesita hacer.

Lo ideal sería que el prototipo sirviera como un mecanismo para identificar los requisitos del software. Si se construye un prototipo de trabajo, el desarrollador intenta hacer uso de los fragmentos del programa ya existentes o aplica herramientas (por ejemplo: generadores de informes, gestores de ventanas, etc.) que permiten generar rápidamente programas de trabajo.

El modelo de construcción de prototipos se muestra en la siguiente figura:

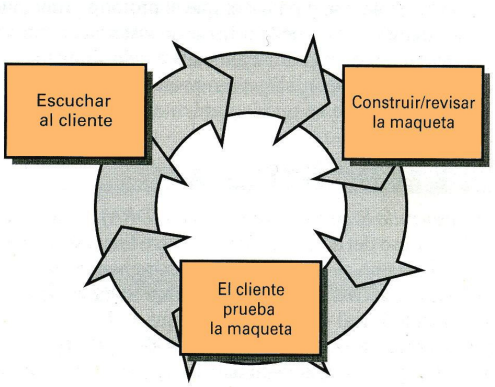


Fig. 4.- El paradigma de construcción de prototipos.

Pero ¿qué hacemos con el prototipo una vez que ha servido para el propósito descrito anteriormente?

El prototipo puede servir como "primer sistema" aunque será también cuestión de ver si el prototipo de

antemano está hecho como un desechable o como un prototipo hecho que pueda ser entregado al usuario y funcione. Es verdad que a los clientes y a los que desarrollan les gusta el paradigma de construcción de prototipos. A los usuarios les gusta el sistema real y a los que desarrollan les gusta construir algo inmediatamente. Sin embargo, la construcción de prototipos también puede ser problemática por las siguientes razones:

- El cliente ve lo que parece ser una versión de trabajo del software, sin tener conocimiento de que con la prisa de hacer que funcione no se ha tenido en cuenta la calidad del software global o la facilidad de mantenimiento a largo plazo. Cuando se informa que el producto se debe construir otra vez para que se puedan mantener los niveles altos de calidad, el cliente no lo entiende y pide que se apliquen "unos pequeños ajustes" para que se pueda hacer del prototipo un producto final. De forma demasiado frecuente la gestión de desarrollo del software es muy lenta.
- El desarrollador, a menudo, hace compromisos de implementación para hacer que el prototipo funcione rápidamente. Se puede utilizar un sistema operativo o lenguaje de programación inadecuado simplemente porque está disponible y porque es conocido; un algoritmo eficiente se puede implementar simplemente para demostrar la capacidad. Después de algún tiempo, el desarrollador debe familiarizarse con estas selecciones y olvidarse de las razones por las que son inadecuadas. La selección menos ideal ahora es una parte integral del sistema.
- Aunque pueden surgir problemas, la construcción de prototipos puede ser un paradigma efectivo para la Ingeniería de Software. La clave es definir las reglas del juego al comienzo; es decir, el cliente y el desarrollador se deben poner de acuerdo en que el prototipo se construya para servir como un mecanismo de definición de requisitos.

2.1.3 Modelo de Desarrollo Rápido de Aplicaciones

El Desarrollo Rápido de Aplicaciones (DRA) es un modelo de proceso de desarrollo del software lineal secuencial que enfatiza un ciclo de desarrollo extremadamente corto. El modelo DRA es una adaptación a "alta velocidad" del modelo lineal secuencial en el que se logra el desarrollo rápido utilizando una construcción basada en componentes. Si se comprenden bien los requisitos y se limita el ámbito del proyecto, el proceso DRA permite al equipo de desarrollo crear un "sistema completamente funcional" dentro de periodos cortos de tiempo (entre 60 a 90 días). Cuando se utiliza principalmente para aplicaciones de sistemas de información, el enfoque DRA comprende las siguientes fases [KER 94]:

Modelado de Gestión. El flujo de información entre las funciones de gestión se modela de forma que responda a las siguientes preguntas: ¿Qué información conduce el proceso de gestión?, ¿Qué información se genera?, ¿Quién la genera?, ¿A dónde va a la información?, ¿Quién la procesa?

Modelado de datos. El flujo de información definido como parte de la fase de modelado de gestión se refina como un conjunto de objetos de datos necesarios para apoyar a la empresa. Se definen las características (llamadas atributos) de cada uno de los objetos y las relaciones entre estos objetos.

Modelado del proceso. Los objetos de datos definidos en la fase de modelado de datos quedan transformados para lograr el flujo de información necesario para implementar una función de gestión. Las descripciones del proceso se crean para añadir, modificar, suprimir o recuperar un objeto de datos.

Generación de aplicaciones. El DRA asume la utilización de técnicas de cuarta generación. En lugar de crear software con lenguaje de programación de tercera generación, el proceso DRA trabaja para volver a utilizar componentes de programas ya existentes (cuando es posible) o a crear componentes reutilizables (cuando sea necesario). En todos los casos se utilizan herramientas para facilitar la construcción del software.

Pruebas y entrega. Como el proceso DRA enfatiza la reutilización, ya se han comprobado muchos de los componentes de los programas. Esto reduce tiempo de pruebas. Sin embargo, se deben probar todos los componentes nuevos y se deben ejercitar todas las interfaces a fondo.

El modelo de proceso DRA se ilustra en la siguiente figura. Obviamente, las limitaciones de tiempo impuestas en un proyecto DRA demandan "ámbito en escalas" [KER 94]. Si una aplicación de gestión puede modularse de forma que permita completarse cada una de las funciones principales en menos de tres meses (utilizando el enfoque descrito anteriormente), es un candidato del DRA. Cada una de las funciones puede ser afrontada por un equipo DRA separado y todas ellas pueden ser integradas en un solo conjunto.

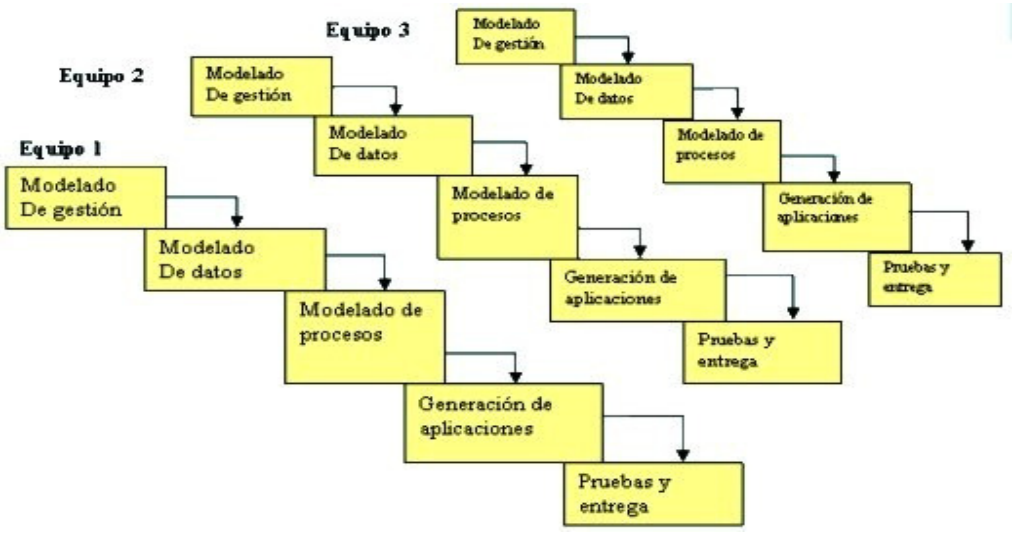


Fig. 5.- El modelo DRA.

Al igual que todos los modelos de proceso, el enfoque DRA tiene inconvenientes [BUT 94]:

- Para proyectos grandes aunque por escalas, el DRA requiere recursos humanos suficientes como para crear el número correcto de equipos DRA.
- DRA requiere clientes y desarrolladores comprometidos en las rápidas actividades necesarias para completar un sistema en un marco de tiempo abreviado. Si no hay compromiso por ninguna de las partes constituyentes, los proyectos DRA fracasarán.

- No todos los tipos de aplicaciones son apropiados para DRA. Si un sistema no se puede convertir en módulos adecuadamente, la construcción de los componentes necesarios para DRA será problemático. Si está en juego el alto rendimiento, y se va a conseguir el rendimiento convirtiendo interfaces en componentes de sistemas, el enfoque DRA puede que no funcione.
- DRA no es adecuado cuando los riesgos técnicos son altos. Esto ocurre cuando una nueva aplicación hace uso de tecnologías nuevas, o cuando el software nuevo requiere un alto grado de interoperabilidad con programas de computadora ya existente.

Normalmente los requisitos van cambiado conforme avanza el desarrollo haciendo que el camino que lleva al producto real no sea real; si a eso le agregamos las fechas tope de entrega hace que sea muy difícil finalizar un producto completo, por lo que normalmente se debe introducir una versión limitada para cumplir la presión de los tiempos: se comprende perfectamente el conjunto de requisitos de productos centrales o del sistema, pero todavía se tienen que definir los detalles de extensiones del producto o sistema. En estas y en otras situaciones similares, los Ingenieros del Software necesitan un modelo de proceso que se ha diseñado explícitamente para acomodarse a un producto que evolucione con el tiempo.

El modelo lineal secuencial se diseña para el desarrollo en línea recta. En esencia, este enfoque en cascada asume que se va a entregar un sistema completo una vez que la secuencia lineal se haya finalizado. El modelo de construcción de prototipos se diseña para ayudar al cliente (o al que desarrolla) a comprender los requisitos. En general, no se diseña para entregar un sistema de producción. En ninguno de los paradigmas anteriores se tienen en cuenta la naturaleza evolutiva del software.

Los modelos evolutivos son iterativos. Se caracterizan por la forma en que permiten a los Ingenieros del Software desarrollar versiones cada vez más completas del software, entre estos modelos se encuentran los siguientes:

2.1.4 Modelo Incremental

El modelo incremental combina elementos del modelo lineal secuencial (aplicados repetidamente) con la filosofía interactiva de construcción de prototipos. Como se muestra en la siguiente figura, el modelo incremental aplica secuencias lineales de forma escalonada mientras progresa el tiempo en el calendario. Cada secuencia lineal produce un "incremento" del software [MDE 93]. Por ejemplo, el software de tratamiento de textos desarrollado con el paradigma incremental podría extraer funciones de gestión de archivos básicos y de producción de documentos en el primer incremento; funciones de edición más sofisticadas y de producción de documentos en el segundo incremento; corrección ortográfica y gramatical en el tercero; y una función avanzada de esquema de página en el cuarto. Se debería tener en cuenta que el flujo del proceso de cualquier incremento puede incorporar el paradigma de construcción de prototipos.

Cuando se utiliza un modelo incremental, el primer incremento a menudo es un producto esencial. Es decir, se afrontan requisitos básicos, pero muchas funciones suplementarias (algunas conocidas, otras no) quedan sin extraer. El cliente utiliza el producto central (o realiza la revisión detallada). Como un resultado de utilización y/o de evaluación, se desarrolla un plan para el incremento siguiente. El plan afronta la modificación del producto central a fin de cumplir mejor las necesidades del cliente y la entrega de funciones y características adicionales. Este proceso se repite siguiendo la entrega de cada incremento, hasta que se elabore el producto completo. El modelo de proceso incremental, como la construcción de

prototipos y otros enfoques evolutivos, es iterativo por naturaleza. Pero a diferencia de la construcción de prototipos, el modelo incremental se centra en la entrega de un producto operacional con cada incremento. Los primeros incrementos son versiones "incompletas" del producto final, pero proporcionan al usuario la funcionalidad que precisa y también una plataforma para la evaluación.

El desarrollo incremental es particularmente útil cuando la dotación de personal no está disponible para una implementación completa en la fecha límite que se haya establecido para el proyecto. Los primeros incrementos se pueden implementar con menos personas.

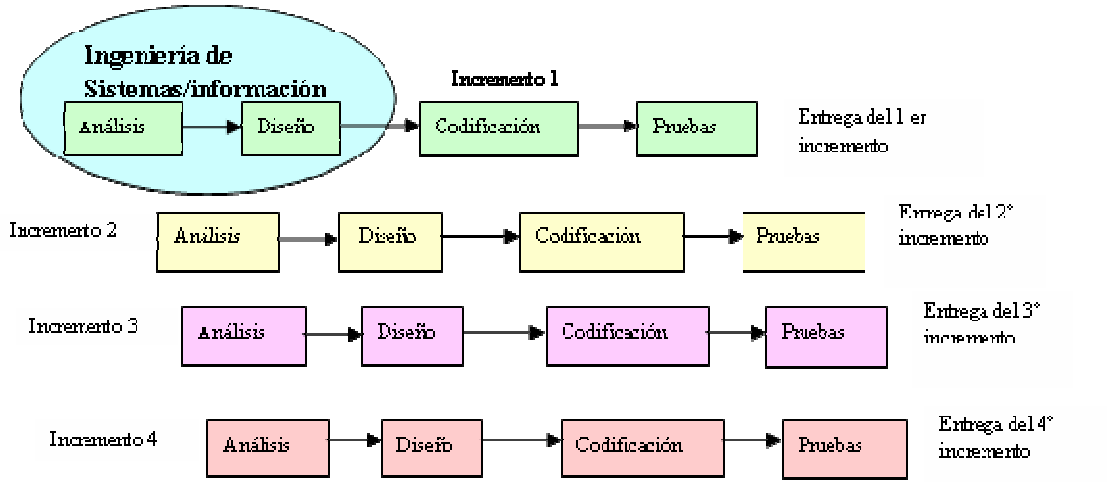


Fig. 6.- Modelo Incremental.

2.1.5 Modelo Espiral

El modelo en espiral, propuesto originalmente por Bohem [BOE 88], es un modelo de proceso de software evolutivo que conjuga la naturaleza iterativa de construcción de prototipos con los aspectos controlados y sistemáticos del modelo lineal secuencial. Proporciona el potencial para el desarrollo rápido de versiones incrementales del software. En el modelo espiral, el software se desarrolla en una serie de versiones incrementales. Durante las primeras iteraciones, la versión incremental podría ser un modelo en papel o un prototipo. Durante las últimas iteraciones, se producen versiones cada vez más completas del sistema diseñado.

El modelo en espiral se divide en un número de actividades de marco de trabajo, también llamadas regiones de tareas. Generalmente, existen entre tres y seis regiones de tareas:

- **Comunicación con el cliente.** Las tareas requeridas para establecer comunicación entre el desarrollador y el cliente.

- **Planificación.** Las tareas requeridas para definir recursos, el tiempo y otra información relacionadas con el proyecto.
- **Análisis de riesgos.** Las tareas requeridas para evaluar riesgos técnicos y de gestión.
- **Ingeniería. L**as tareas requeridas para construir una o más representaciones de la aplicación.
- **Construcción y acción.** Las tareas requeridas para construir, probar, instalar y proporcionar soporte al usuario (por ejemplo: documentación y práctica).
- **Evaluación del cliente. L**as tareas requeridas para obtener la reacción del cliente según la evaluación de las representaciones del software creadas durante la etapa de ingeniería e implementada durante la etapa de instalación**.**

El modelo en espiral se ilustra en la siguiente figura.

Cada una de las regiones está compuesta por un conjunto de tareas del trabajo, que se adaptan a las características del proyecto que va a emprenderse. Para proyectos pequeños, el número de tareas de trabajo y su formalidad es bajo. Para proyectos mayores y más críticos cada región de tareas contiene tareas de trabajo que se definen para lograr un nivel más alto de formalidad. En todos los casos, se aplican las actividades de protección (por ejemplo: gestión de configuración del software y garantía de calidad del software).

Cuando empieza este proceso evolutivo, el equipo de Ingeniería de Software gira alrededor de la espiral en dirección de las agujas del reloj, comenzando por el centro. El primer circuito de la espiral puede producir el desarrollo de una especificación de productos; los pasos siguientes en la espiral se podrían utilizar para desarrollar un prototipo y progresivamente versiones más sofisticadas del software. Cada paso por la región de planificación produce ajustes en el plan del proyecto. El costo y la planificación se ajustan con la realimentación ante la evaluación del cliente. Además el gestor del proyecto ajusta el número planificado de iteraciones requeridas para complementar el software.

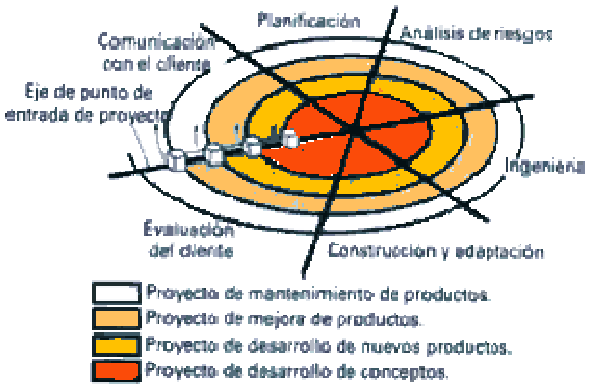


Fig. 7.- Un modelo espiral típico.

A diferencia del modelo de proceso clásico que termina cuando se entrega el software, el modelo en espiral puede adaptarse y aplicarse a lo largo de la vida del software de computadora.

El modelo en espiral es un enfoque realista del desarrollo de sistemas y de software a gran escala. Como el software evoluciona, a medida que progresa el proceso el desarrollador y el cliente comprenden y reaccionan mejor ante riesgos en cada uno de los niveles evolutivos. El modelo en espiral utiliza la construcción de prototipos como mecanismo de reducción de riesgos, pero, lo que es más importante, permite a quien lo desarrolla aplicar el enfoque de construcción de prototipos en cualquier etapa de evolución del producto. Mantiene el enfoque sistemático de los pasos sugeridos por el ciclo de vida clásico, pero lo incorpora al marco del trabajo iterativo que refleja de forma más realista el mundo real. El modelo en espiral demanda una consideración directa de los riesgos técnicos en todas las etapas del proyecto, y si se aplica adecuadamente, debe reducir los riesgos antes de que se conviertan en problemáticos.

Pero al igual que otros modelos, este paradigma en espiral no es la panacea. Puede resultar difícil convencer a grandes clientes (particularmente en situaciones bajo contrato) de que el enfoque evolutivo es controlable. Requiere una considerable habilidad para la evaluación del riesgo y cuenta con esta habilidad para el éxito. Si el riesgo importante no es descubierto y gestionado, indudablemente surgirán problemas. Finalmente, el modelo no se ha utilizado tanto como los modelos lineales secuenciales o de construcción de prototipos. Todavía tendrán que pasar muchos años antes de que se determine con absoluta certeza la eficacia de este paradigma.

2.1.6 Modelo WINWIN

El modelo en espiral sugiere una actividad del marco de trabajo que aborda la comunicación con el cliente. El objetivo de esta actividad es mostrar los requisitos del cliente. En un contexto ideal, el desarrollador simplemente pregunta al cliente lo que se necesita y el cliente proporciona detalles suficientes para continuar. Desgraciadamente, esto raramente ocurre. En realidad el cliente y el desarrollador entran en un proceso de negociación, donde el cliente puede ser preguntado para sopesar la funcionalidad, rendimiento y otros productos o características del sistema frente al costo y al tiempo de comercialización.

Las mejores negociaciones se esfuerzan para obtener "victoria-victoria". Esto es, el cliente gana obteniendo el producto o sistema que satisface la mayor parte de sus necesidades y el desarrollador gana trabajando para conseguir presupuestos y lograr una fecha de entrega realista.

El modelo en espiral WINWIN de Boehm [BOE 98] define un conjunto de actividades de negociación al principio de cada paso alrededor de la espiral. Más que una simple actividad de comunicación con el cliente, se definen las siguientes actividades:

- Identificación del sistema o subsistemas clave de los "directivos".
- Determinación de las "condiciones de victoria" de los directivos.
- Negociación de las condiciones de "victoria" de los directivos para reunirlas en un conjunto de condiciones "victoria-victoria" para todos los afectados (incluyendo el equipo del proyecto de software).

Conseguidos completamente estos pasos iniciales se logra un resultado "victoria-victoria". Que será el criterio clave para continuar con la definición del sistema y del software. El modelo en espiral WINWIN se ilustra en la siguiente figura.

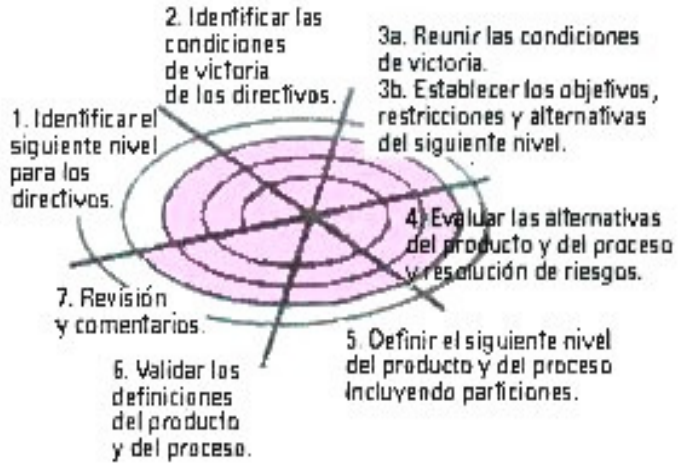


Fig. 8.- El modelo en espiral WINWIN [BOE 98].

Además del énfasis realizado en la negociación inicial, el modelo en espiral WINWIN introduce tres hitos en el proceso, llamados puntos de fijación [BOE 96], que ayudan a establecer un ciclo completo alrededor de la espiral y proporcionan hitos de decisión antes de continuar el proyecto de software.

En esencia, los puntos de fijación representan tres visiones diferentes del progreso mientras que el proyecto recorre la espiral. El primer punto de fijación, llamado objetivos del ciclo de vida (OCV), define un conjunto de objetivos para cada actividad principal de Ingeniería de Software. El segundo punto de fijación, llamado arquitectura del ciclo de vida (ACV), establece los objetivos que se deben conocer mientras que se define la arquitectura del software y del sistema. La capacidad operativa inicial (COI) es el tercer punto de fijación y representa un conjunto de objetivos asociados a la preparación del software para la instalación/distribución, preparación del lugar previamente a la instalación y la asistencia precisada de todas las partes que utilizará o mantendrá el software.

2.1.7 Desarrollo Basado en Componentes

Las tecnologías de objetos proporcionan el marco de trabajo técnico para un modelo de proceso basado en componentes para la Ingeniería de Software. El paradigma orientado a objetos enfatiza la creación de clases que encapsulan tanto los datos como los algoritmos que se utilizan para manejar los datos. Si se diseñan y se implementan adecuadamente, las clases orientadas a objetos son reutilizables por las diferentes aplicaciones y arquitecturas de sistemas basados en computadora.

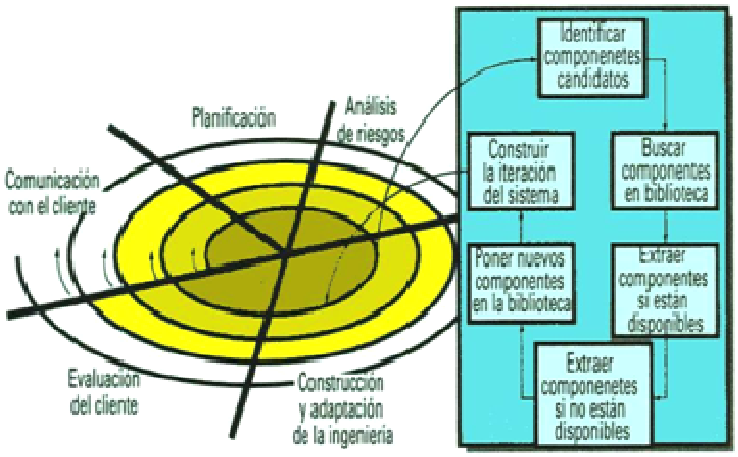


Fig. 9.- Modelo de desarrollo basado en componentes.

El modelo de desarrollo basado en componentes, (figura anterior) incorpora muchas de las características del modelo en espiral. Es evolutivo por naturaleza [NIE 92], y exige un enfoque iterativo para la creación del software. Sin embargo, el modelo de desarrollo basado en componentes configura aplicaciones desde componentes preparados de software llamados "clases".

La actividad de la ingeniería comienza con la identificación de clases candidatas. Esto se lleva a cabo examinando los datos que se van a manejar por parte de la aplicación y el algoritmo que se va a aplicar para conseguir el tratamiento. Los datos y los algoritmos correspondientes se empaquetan en una clase.

Las clases creadas en los proyectos de Ingeniería de Software anteriores, se almacenan en una biblioteca de clases o diccionario de datos (repository). Una vez identificadas las clases candidatas, la biblioteca de clases se examina para determinar si estas clases ya existen. En caso de que así fuera, se extraen de la biblioteca y se vuelven a utilizar. Si una clase candidata no reside en la biblioteca, se aplican los métodos orientados a objetos. Se compone así la primera iteración de la aplicación a construirse, mediante las clases extraídas de la biblioteca y las clases nuevas construidas para cumplir las necesidades únicas de la aplicación. El flujo del proceso vuelve a la espiral y volverá a introducir por último la iteración ensambladora de componentes a través de la actividad de ingeniería.

El modelo de desarrollo basado en componentes conduce a la reutilización del software, y la reutilización proporciona beneficios a los Ingenieros de Software. Según estudios de reutilización, QSM Associates, Inc. Informa que el ensamblaje de componentes lleva a una reducción del 70 por ciento del tiempo del ciclo de desarrollo, un 84 por ciento del costo del proyecto. Aunque estos resultados están en función de la robustez de la biblioteca de componentes, no hay duda de que el ensamblaje de componentes proporciona ventajas significativas para los Ingenieros de Software.

2.1.8 Modelo de métodos formales

El modelo de métodos formales, comprende un conjunto de actividades que conducen a la especificación matemática del software de computadora. Los métodos formales permiten que un Ingeniero del Software especifique, desarrolle y verifique un sistema basado en computadora aplicando una notación rigurosa y matemática. Algunas organizaciones de desarrollo del software actualmente aplican una variación de este enfoque, llamado Ingeniería de Software de Sala Limpia [MIL 87, DYE 92].

Los métodos formales ofrecen un fundamento para entornos de especificación que dan lugar a modelos de análisis más completos, consistentes y carentes de ambigüedad, que aquellos que se producen empleando métodos convencionales u orientados a objetos. Las capacidades descriptivas de la teoría de conjuntos y de la notación lógica capacitan al Ingeniero de Software para crear un enunciado claro de los hechos (requisitos). Los conceptos subyacentes que gobiernan los métodos formales son:

- **Los invariantes de datos.** Condiciones que son ciertas a lo largo de la ejecución del sistema que contiene una colección de datos.
- **El estado.** Los datos almacenados a los que accede el sistema y que son alterados por él.
- **La operación**. Una acción que tiene lugar en un sistema y que lee o escribe datos en un estado. Una operación queda asociada con dos condiciones: una precondición y una poscondición.
- **La matemática discreta**. La notación y práctica asociada a los conjuntos y a la especificación constructiva, a los operadores de conjuntos, a los operadores lógicos y a las sucesiones, constituyen la base de los métodos formales. Estas matemáticas están implementadas en el contexto de un lenguaje de especificación formal, tal como el lenguaje Z.

El lenguaje Z, al igual que todos los lenguajes de especificación formal, posee tanto un dominio semántico como un dominio sintáctico. El dominio sintáctico utiliza una simbología que sigue estrechamente a la notación de conjuntos y al cálculo de predicados. El dominio semántico capacita al lenguaje para expresar requisitos de forma concisa. La estructura Z contiene esquemas, estructuras en forma de cuadro que presentan las variables y que especifican las relaciones entre estas variables.

La decisión de utilizar métodos formales debe considerar los costos de arranque, así como los cambios puntuales asociados a una tecnología radicalmente distinta. En la mayoría de los casos, los métodos formales ofrecen los mayores beneficios para los sistemas de seguridad y para los sistemas críticos para los negocios.

Cuando se utilizan métodos formales durante el desarrollo, proporcionan un mecanismo para eliminar muchos de los problemas que son difíciles de superar con paradigmas de la Ingeniería de Software. La ambigüedad, lo incompleto y la inconsistencia se descubren y se corrigen más fácilmente –no mediante una revisión a propósito para el caso, sino mediante la aplicación del análisis matemático-. Cuando se utilizan métodos formales durante el diseño, sirven como base para la verificación de programas y por consiguiente permiten que el Ingeniero del Software descubra y corrija errores que no se pudieron detectar de otra manera.

Aunque todavía no hay un enfoque establecido, los modelos de métodos formales ofrecen la promesa de

un software libre de defectos. Sin embargo, se ha hablado de una gran preocupación sobre su aplicabilidad en un entorno de gestión:

- El desarrollo de modelos formales actualmente es bastante caro y lleva mucho tiempo.
- Se requiere un estudio detallado porque pocos responsables del desarrollo de software tienen los antecedentes necesarios para aplicar métodos formales.
- Es difícil utilizar los modelos como un mecanismo de comunicación con clientes que no tienen muchos conocimientos técnicos.

No obstante es posible que el enfoque a través de métodos formales tenga más partidarios entre los desarrolladores del software que deben construir software de mucha seguridad (por ejemplo: los desarrolladores de aviación y dispositivos médicos), y entre los desarrolladores que pasan grandes penurias económicas al aparecer errores de software.

2.1.9 Modelo de Sala limpia

La Ingeniería de Software de sala limpia es un enfoque formal para el desarrollo del software, que puede dar lugar a un software que posea una calidad notablemente alta. Emplea la especificación de estructura de cajas (o métodos formales) para el modelado de análisis y diseño y hace hincapié en la verificación de la corrección, más que en las pruebas, como mecanismo fundamental para hallar y eliminar errores. Se aplica una prueba estadística de utilización para desarrollar la información de tasa de fallos necesaria para certificar la finalidad del software proporcionado.

El enfoque de sala limpia comienza por unos modelos de análisis y diseño que hacen uso de una representación de estructura de cajas. Una "caja" encapsula el sistema (o algún aspecto del sistema) en un determinado nivel de abstracción. Se utilizan "cajas negras" para representar el comportamiento observable externamente de ese sistema. Las cajas de estado encapsulan los datos y operaciones de ese estado. Se utiliza una caja limpia para poder modelar el diseño de procedimientos que está implicado por los datos y operaciones de la caja de estados.

Se aplica la verificación de corrección una vez que está completo el diseño de estructura de cajas. El diseño de procedimientos para un componente de software se desglosará en una serie de subfunciones. Para demostrar la corrección de cada una de estas subfunciones, se definen condiciones de salida para cada una de las subfunciones y se aplican un conjunto de subpruebas. Si se satisfacen todas y cada una de las condiciones de salida, entonces el diseño debe ser correcto.

Una vez finalizada la verificación de corrección, comienza la prueba estadística de utilización. A diferencia de la comprobación condicional, la Ingeniería de Software de sala limpia no hace hincapié en la prueba unitaria o de integración. En su lugar, el software se comprueba mediante la definición de un conjunto de escenarios, mediante la determinación de las probabilidades de utilización de cada uno de estos escenarios y mediante la aplicación posterior de pruebas aleatorias que satisfagan estas probabilidades. Los registros de error resultantes se combinan con modelos de muestreo, de componentes y de certificación para hacer posible el cálculo matemático de la fiabilidad estimada de ese componente de software.

La filosofía de sala limpia es un enfoque riguroso de la Ingeniería de Software. Se trata de un modelo de

proceso del software que hace hincapié en la verificación matemática de la corrección y en la certificación de la fiabilidad del software. El resultado final son unas tasas de fallo extremadamente bajas, que sería difícil o imposible de conseguir empleando unos métodos menos formales.

2.1.10 Técnicas de cuarta generación

El término técnicas de cuarta generación (T4G) abarca un amplio espectro de herramientas de software que tienen algo en común: todas facilitan al Ingeniero del Software la especificación de algunas características del software a alto nivel. Luego, la herramienta genera automáticamente el código fuente basándose en la especificación del técnico. Cada vez parece más evidente que cuanto mayor sea el nivel en el que se especifique el software, más rápido se podrá construir el programa. El paradigma T4G para la Ingeniería de Software se orienta hacia la posibilidad de especificar el software usando formas de lenguaje especializado o notaciones gráficas que describan el problema que hay que resolver en términos que los entienda el cliente. Actualmente, un entorno para el desarrollo de software que soporte el paradigma T4G puede incluir todas o algunas de las siguientes herramientas: lenguajes no procedimentales de consulta a bases de datos, generación de informes, manejo de datos, interacción y definición de pantallas, generación de códigos, capacidades gráficas de alto nivel y capacidades de hoja de cálculo y generación automatizada de HTML y lenguajes similares utilizados para la creación de sitios Web usando herramientas de software avanzado. Inicialmente, muchas de estas herramientas estaban disponibles, pero sólo para ámbitos de aplicación muy específicos, pero actualmente los entornos T4G se han extendido a todas las categorías de aplicaciones de software.

Al igual que otros paradigmas, T4G comienza con el paso de reunión de requisitos. Idealmente, el cliente describe los requisitos, que son a continuación traducidos directamente a un prototipo operativo. Sin embargo, en la práctica no se puede hacer eso. El cliente puede que no esté seguro de lo que necesita; puede ser ambiguo en la especificación de hechos que le son conocidos y puede que no sea capaz o no esté dispuesto a especificar la información en la forma en que puede aceptar una herramienta T4G.

Para aplicaciones pequeñas, se puede ir directamente desde el paso de recolección de requisitos al paso de una implementación, usando un lenguaje de cuarta generación no procedimental (L4G) o un modelo comprimido de red de iconos gráficos. Sin embargo, es necesario un mayor esfuerzo para desarrollar estrategia de diseño para el sistema, incluso si se utiliza un L4G. El uso de T4G sin diseño (para grandes proyectos) causará las mismas dificultades (poca calidad, mantenimiento pobre, mala aceptación por el cliente) que se encuentran cuando se desarrolla software mediante los enfoques convencionales.

La implementación mediante L4G permite, al que desarrolla el software, centrarse en la representación de los resultados deseados, que es lo que se traduce automáticamente en un código fuente que produce dichos resultados. Obviamente, debe existir una estructura de datos con información relevante y a la que el L4G pueda acceder rápidamente.

Para transformar una implementación T4G en un producto, el que lo desarrolla debe dirigir una prueba completa, desarrollar con sentido una documentación y ejecutar el resto de las actividades de integración que son también requeridas por otros paradigmas de Ingeniería de Software. Además, el software desarrollado con T4G debe ser construido de forma que facilite la realización del mantenimiento de forma expedita.

Al igual que todos los paradigmas del software, el modelo T4G tiene ventajas e inconvenientes. Los defensores aducen reducciones drásticas en el tiempo de desarrollo del software y una mejora significativa en la productividad de la gente que construye el software. Los detractores argumentan que las herramientas actuales de T4G no son más fáciles de utilizar que los lenguajes de programación, que el código fuente producido por tales herramientas es "ineficiente" y que el mantenimiento de grandes sistemas de software desarrollado mediante T4G es cuestionable.

Hay algún mérito en lo que se refiere a indicaciones de ambos lados y es posible resumir el estado actual de los enfoques de T4G:

- El uso de T4G es un enfoque viable para muchas de las diferentes áreas de aplicación. Junto con las herramientas de Ingeniería de Software asistida por computadora (CASE) y los generadores de código, T4G ofrece una solución fiable a muchos problemas del software.
- Los datos recogidos en compañías que usan T4G parecen indicar que el tiempo requerido para producir software se reduce mucho para aplicaciones pequeñas y de tamaño medio, y que la cantidad de análisis y diseño para las aplicaciones pequeñas también se reduce.
- Sin embargo, el uso de T4G para grandes trabajos de desarrollo de software exige el mismo o más tiempo de análisis, diseño y prueba (actividades de Ingeniería de Software), para lograr un ahorro sustancial de tiempo que puede conseguirse mediante la eliminación de la codificación.

Resumiendo, las técnicas de cuarta generación ya se han convertido en una parte importante del desarrollo de software. Cuando se combinan con enfoques de ensamblaje de componentes, el paradigma T4G se puede convertir en el enfoque dominante hacia el desarrollo del software.

2.1.11 Métodos Agiles

Los Métodos Agiles, tales como Lean Development, eXtreme Programming y Adaptive Software Development, son estrategias de desarrollo de software que promueven prácticas que son adaptables en vez de predictivas, centradas en la gente o en los equipos, iterativas, orientadas hacia prestaciones y hacia la entrega, de comunicación intensiva y que requieren que el negocio se involucre en forma directa. Comparando esos atributos con los principios fundacionales de MSF (Microsoft Solution Framework), se encuentra que MSF y las metodologías ágiles están muy alineadas tanto en los principios como en las prácticas para el desarrollo de software en ambientes que requieren un alto grado de adaptabilidad.

Históricamente, a finales de la década de 1990 dos grandes temas irrumpieron en las prácticas de la Ingeniería de Software y en los métodos de desarrollo: el diseño basado en patrones y los métodos ágiles. De estos últimos, el más resonante ha sido la Programación Extrema (XP), que algunos consideran una innovación extraordinaria y otros la creen cínica [Rak 01], extremista [McC 02], falaz [Ber 03] o perniciosa para la salud de la profesión [Kee 03]. Patrones y XP se convirtieron de inmediato en temas de discusión masiva en la industria y de fuerte presencia en la Red. Al primero de esos temas el mundo académico lo está tratando como un asunto respetable desde hace un tiempo; al otro recientemente se está legitimando como tópico serio de investigación. La mayor parte de los documentos proviene todavía de los practicantes, los críticos y los consultores que impulsan o rechazan sus postulados. Pero el crecimiento de los métodos ágiles y su penetración ocurre a un ritmo pocas veces visto en la industria: en tres o cuatro

años, según el Cutter Consortium, el 50% de las empresas define como "ágiles" más de la mitad de los métodos empleados en sus proyectos [Cha 04].

Los métodos ágiles constituyen un movimiento heterodoxo que confronta con las metodologías consagradas, acordadas en organismos y apreciadas por consultores, analistas de industria y corporaciones. Contra semejante adversario, los MAs se expresaron a través de manifiestos y libros en tono de proclama, evitando (hasta hace poco) toda especificación formal.

Lo que los Métodos Agiles tienen en común es su modelo de desarrollo incremental (pequeñas entregas con ciclos rápidos), cooperativo (desarrolladores y usuarios trabajan juntos en estrecha comunicación), directo (el método es simple y fácil de aprender) y adaptable (capaz de incorporar los cambios). Las claves de estos métodos son la velocidad y su simplicidad. De acuerdo con ello, los equipos de trabajo se concentran en obtener lo antes posible una pieza útil que implemente sólo lo que sea más urgente; de inmediato requieren retroalimentación de lo que han hecho y lo tienen muy en cuenta. Luego prosiguen con ciclos igualmente breves, desarrollando de manera incremental. Estructuralmente, los Métodos Agiles se asemejan a los RADs (desarrollo rápido de aplicaciones) más clásicos y a otros modelos iterativos, pero sus énfasis son distintivos y su combinación de ideas es única.

El surgimiento de este nuevo paradigma surge de diversos estudios que habían revelado que la práctica metodológica fuerte, con sus exigencias de planeamiento y sus técnicas de control, en muchos casos no brindaba resultados que estuvieran a la altura de sus costos en tiempo, complejidad y dinero. Investigaciones como la de Joe Nandhakumar y David Avison [NA 99], en un trabajo de campo sobre "la ficción del desarrollo metodológico", denunciaban que las metodologías clásicas de sistemas de información "se tratan primariamente como una ficción necesaria para presentar una imagen de control o para proporcionar estatus simbólico" y que dichas metodologías son demasiado ideales, rígidas y mecanicistas para ser utilizadas al pie de la letra.

Pero también hay cientos de casos documentados de éxitos logrados con metodologías rigurosas. El otro paradigma (los métodos ágiles de desarrollo), sin embargo, tienen su buen catálogo de triunfos. Algunos nombres respetados (Martin Fowler con el respaldo de Cutter Consortium; Dee Hock con su visión caórdica, una combinación de caos y orden; Philippe Kruchten con su RUP adaptado a los tiempos que corren) e incluso Ivar Jacobson [Jac 02] consideran que los MAs constituyen un aporte que no sería sensato dejar de lado. No habría que tratarlo entonces como si fuera el capricho pasajero.

Los organismos y corporaciones han desarrollado una abundancia de estándares comprensivos que han ido marcando la historia y poblando los textos de metodologías e Ingeniería de Software: CMM, Spice, BootStrap, TickIt, derivaciones de ISO9000, SDCE, Trillium [Wie 98] [Wie 99] [She 97]. Algunos son verdaderamente métodos; otros, metodologías de evaluación o estimación de conformidad; otros más, estándares para metodologías o meta-modelos. Al lado de ellos se encuentra lo que se ha llamado una ciénaga de estándares generales o específicos de industria a los que se someten organizaciones que desean (o deben) articular sus métodos conforme a diversos criterios de evaluación, vehículos de selección de contratistas o marcos de referencia.

Los modelos de los métodos clásicos difieren bastante en su conformación y en su naturaleza, pero exaltan casi siempre las virtudes del planeamiento y poseen un espíritu normativo. Comienzan con las entrevistas al usuario para la obtención completa de sus requerimientos y el posterior análisis de ellos. Después de un largo período de intensa interacción con usuarios y clientes, los ingenieros establecen un conjunto definitivo y exhaustivo de rasgos, requerimientos funcionales y no funcionales. Esta información se documenta en forma de especificaciones para la segunda etapa, el diseño, en el que los arquitectos,

trabajando junto a otros expertos en temas puntuales (como pueden ser estructuras y bases de datos), generan la arquitectura del sistema. Luego los programadores implementan ese diseño bien documentado y finalmente el sistema completo se prueba y se despacha.

A fines del siglo XX había un abanico de tipos de procesos o modelos disponibles: el más convencional era el modelo en cascada o lineal-secuencial, pero al lado de él había otros como el modelo "V", el modelo de construcción de prototipos, el de desarrollo rápido o RAD, el modelo incremental, el modelo en espiral básico, el espiral WINWIN, el modelo de desarrollo concurrente y un conjunto de modelos iterativos o evolutivos (basados en componentes, por ejemplo) que en un primer momento convivían apaciblemente con los restantes, aunque cada uno de ellos se originaba en la crítica o en la percepción de las limitaciones de algún otro. Algunos eran normativo, otros descriptivos. Algunos de los modelos incluían iteraciones, incrementos, recursiones o bucles de retroalimentación; y algunos se preciaban también de rápidos, adaptables y expeditivos.

Modelo	Versión de origen	Características
Modelo en cascada	Secuencial: Bennington 1956 – Iterativo: Royce 1970 – Estándar DoD 2167-A	Secuencia de requerimiento, diseño del sistema, diseño de programa, codificación, pruebas, operación y mantenimiento
Modelo en cascada c/ fases superpuestas	Cf. McConnell 1996:143-144	Cascada con eventuales desarrollos en paralelo (Modelo Sashimi)
Modelo iterado con prototipado	Brooks 1975	Iterativo – Desarrollo incremental
Desarrollo rápido (RAD)	J. Martin 1991 – Kerr/Hunter 1994 – McConnell 1996	Modelo lineal secuencial con ciclos de desarrollo breves
Modelo V	Ministerio de Defensa de Alemania 1992	Coordinación de cascada con iteraciones
Modelo en espiral	Barry Boehm 1988	Iterativo – Desarrollo incremental. Cada fase no es lineal, pero el conjunto de fases sí lo es.
Modelo en espiral WINWIN	Barry Boehm 1998	Iterativo – Desarrollo incremental – Aplica teoría-W a cada etapa
Modelo de desarrollo	Davis y Sitaram 1994	Modelo cíclico con análisis de estado
Modelo de entrega incremental *(Staged delivery)*	McConnell 1996: 148	Fases tempranas en cascada – Fases ulteriores descompuestas en etapas

Tabla 2.- Comparativo entre procesos de desarrollo.

La tabla anterior, ilustra el repertorio de los métodos que podríamos llamar "de peso completo" o "de fuerza industrial", que en sus variantes más rigurosas imponen una prolija especificación de técnicas correspondientes a los diferentes momentos del ciclo de vida.

Los Métodos Agiles difieren en sus particularidades, pero coinciden en adoptar un modelo iterativo que la literatura más agresiva opone dialécticamente al modelo ortodoxo dominante. El modelo iterativo en realidad es bastante antiguo, y se remonta a ideas plasmadas por Tom Gilb en los 60 y por Vic Basili y Joe

Turner en 1975 [BT 75]. Como quiera que se les llame, la metodología de la corriente principal se estima opuesta en casi todos los respectos a las nuevas, radicales, heterodoxas, extremas, adaptables, minimalistas o marginales. El antagonismo entre ambas concepciones del mundo es, según los "agilistas", extrema y abismal.

Existen unos cuantos Métodos Agiles reconocidos por los especialistas. En este estudio se analizarán con algún detenimiento los que han alcanzado a figurar en la bibliografía mayor y cuentan en su haber con casos sustanciales documentados.

De acuerdo con el Cutter Consortium, el más popular entre los Métodos Agiles: 38% del mercado ágil contra 23% de su competidor más cercano, FDD. Luego vienen Adaptive Software Development con 22%, DSDM con 19%, Crystal con 8%, Lean Development con 7% y Scrum con 3%. Todos los demás juntos suman 9% [Cha 04].

2.1.12 eXtreme Programming (XP)

A mediados de la década de 1980, Kent Beck y Ward Cunningham trabajaban en un grupo de investigación de Tektronix; allí idearon las tarjetas CRC y sentaron las bases de lo que después serían los patrones de diseño y XP. Estas tarjetas se tratan de simples tarjetas de papel para fichado, de 4x6 pulgadas; CRC denota "Clase-Responsabilidad-Colaboración", y es una técnica que reemplaza a los diagramas en la representación de modelos. En las tarjetas se escriben las responsabilidades, una descripción de alto nivel del propósito de una clase y las dependencias primarias. En su economía sintáctica y en su abstracción, prefiguran a lo que más tarde serían las tarjetas de historias de XP. Beck y Cunningham prohibían escribir en las tarjetas más de lo que en ellas cabía [BC 89]. Hemos descrito ese patrón (o estilo) en el documento sobre estilos en arquitectura de software.

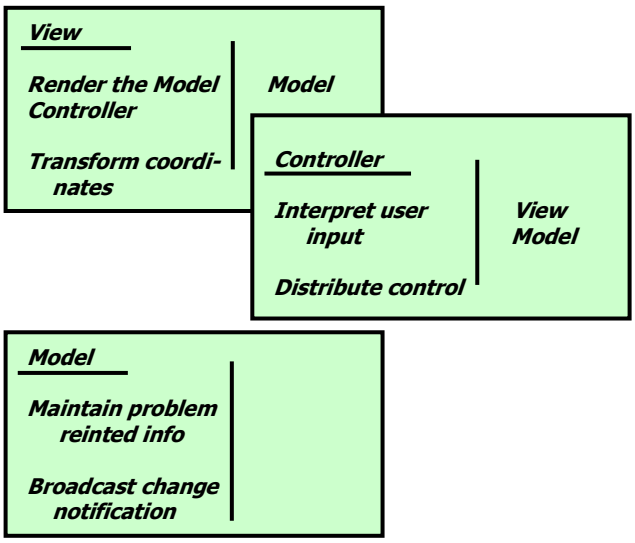


Fig. 10.- Tarjetas CRC del patrón MVC según [BC 89].

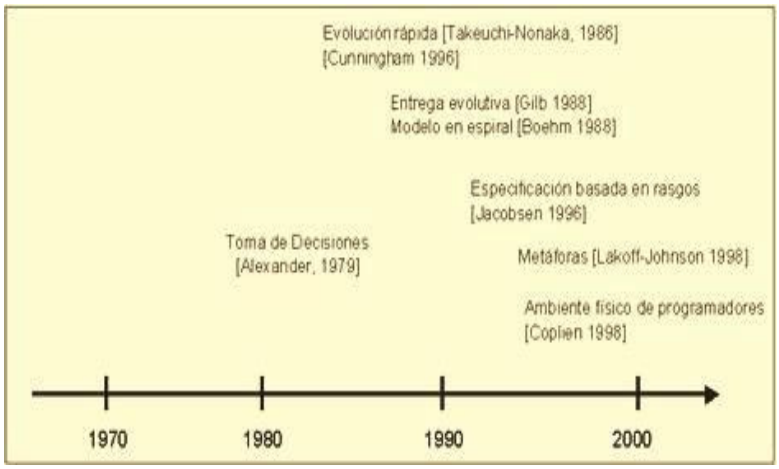


Fig. 11.- Gestación de XP, basado en [ASR 02].

XP se funda en cuatro valores: comunicación, simplicidad, retroalimentación y coraje. Pero tan conocidos como sus valores son sus prácticas. Beck sostiene que se trata más de lineamientos que de reglas:

- **Juego de planeamiento.** Busca determinar rápidamente el alcance de la versión siguiente, combinando prioridades de negocio definidas por el cliente con las estimaciones técnicas de los programadores. Éstos estiman el esfuerzo necesario para implementar las historias del cliente y éste decide sobre el alcance y la agenda de las entregas. Las historias se escriben en pequeñas fichas, que algunas veces se tiran. Cuando esto sucede, lo único restante que se parece a un requerimiento es una multitud de pruebas automatizadas, las pruebas de aceptación.
- **Entregas pequeñas y frecuentes**. Se "genera" [sic] un pequeño sistema rápidamente, al menos uno cada dos o tres meses. Pueden liberarse nuevas versiones diariamente (como es práctica en Microsoft), pero al menos se debe liberar una cada mes. Se agregan pocos rasgos cada vez.
- **Metáforas del sistema.** El sistema se define a través de una metáfora o un conjunto de metáforas, una "historia compartida" por clientes, administradores y programadores que orienta todo el sistema describiendo como funciona. Una metáfora puede interpretarse como una arquitectura simplificada. La concepción de metáfora que se aplica en XP deriva de los estudios de Lakoff y Johnson, bien conocidos en lingüística y psicología cognitiva.
- **Diseño simple.** El énfasis se deposita en diseñar la solución más simple susceptible de implementarse en el momento. Se eliminan complejidades innecesarias y código extra, y se define la menor cantidad de clases posible. No debe duplicarse código. En un oxímoron deliberado, se urge a "decir todo una vez y una sola vez". Nadie en XP llega a prescribir que no haya diseño concreto, pero el diseño se limita a algunas tarjetas elaboradas en sesiones de diseño de 10 a 30 minutos. Esta es la práctica donde se impone el minimalismo de YAGNI: no implementar nada que no se necesite ahora; o bien, nunca implementar algo que vaya a necesitarse más adelante; minimizar diagramas y documentos.
- **Prueba continua.** El desarrollo está orientado por las pruebas. Los clientes ayudan a escribir las

pruebas funcionales antes que se escriba el código. Eso es un test-driven development. El propósito del código real no es cumplir un requerimiento, sino pasar las pruebas. Las pruebas y el código son escritas por el mismo programador, pero la prueba debería realizarse sin intervención humana, y es a todo o nada. Hay dos clases de prueba: la prueba unitaria, que verifica una sola clase, o un pequeño conjunto de clases; la prueba de aceptación verifica todo el sistema, o una gran parte.

- **Refactorización[1] continua.** Se refactoriza el sistema eliminando duplicación, mejorando la comunicación y agregando flexibilidad sin cambiar la funcionalidad. El proceso consiste en una serie de pequeñas transformaciones que modifican la estructura interna preservando su conducta aparente. La práctica también se conoce como "Mejora Continua de Código" o Refactorización implacable. Se lo ha parafraseado diciendo: "Si funciona bien, arréglelo de todos modos". Se recomiendan herramientas automáticas.
- **Programación en pares.** Todo el código está escrito por pares de programadores. Dos personas escriben código en una computadora, turnándose en el uso del ratón y el teclado. El que no está escribiendo, piensa desde un punto de vista más estratégico y realiza lo que podría llamarse revisión de código en tiempo real. Los roles pueden cambiarse varias veces al día. Esta práctica no es en absoluto nueva. Hay antecedentes de programación en pares anteriores a XP [Hig 00b]. Steve McConnell la proponía en 1993 en su Code Complete [McC 93: 528].
- **Propiedad colectiva del código**. Cualquiera puede cambiar cualquier parte del código en cualquier momento, siempre que escriba antes la prueba correspondiente. Algunas veces los practicantes aplican el patrón organizacional CodeOwnership de Coplien [Cop 95].
- **Integración continua.** Cada pieza se integra a la base de código apenas está lista, varias veces al día. Debe ejecutarse la prueba antes y después de la integración. Existe una máquina (solamente) dedicada a este propósito.
- **Ritmo sostenible, trabajando un máximo de 8 horas por día**. Antes se llamaba a esta práctica "Semana de 40 horas". Mientras en RAD las horas extras eran una best practice [McC 96], en XP todo el mundo debe irse a casa a las cinco de la tarde. Dado que el desarrollo de software se considera un ejercicio creativo, se estima que hay que estar fresco y descansado para hacerlo eficientemente; con ello se motiva a los participantes, se evita la rotación del personal y se mejora la calidad del producto. Deben minimizarse los héroes y eliminar el "proceso neurótico". Aunque podrían admitirse excepciones, no se permiten dos semanas seguidas de tiempo adicional. Si esto sucede, se lo trata como problema a resolver.
- **Todo el equipo en el mismo lugar.** El cliente debe estar presente y disponible a tiempo completo para el equipo. También se llama "El Cliente en el Sitio". Como esto parecía no cumplirse (si el cliente era muy junior no servía para gran cosa, y si era muy senior no deseaba estar allí), se

[1] El término refactoring, introducido por W. F. Opdyke en su tesis doctoral [Opd92], se refiere "al proceso de cambiar un sistema de software [orientado a objetos] de tal manera que no se altere el comportamiento exterior del código, pero se mejore su estructura interna". Al igual que sucede con los patrones, existe un amplio catálogo de refactorizaciones más comunes: reemplazo de iteración por recursión; sustitución de un algoritmo por otro más claro; extracción de clase, interface o método; descomposición de condicionales; reemplazo de herencia por delegación, etcétera. La Biblia del método es Refactoring de Martin Fowler, Kent Beck, John Brant, William Opdyke y Don Roberts [FBB+99].

especificó que el representante del cliente debe ser preferentemente un analista. (Tampoco se aclara analista de qué; seguramente se definirá en una próxima versión).

- **Estándares de codificación.** Se deben seguir reglas de codificación y comunicarse a través del código. Según las discusiones en Wiki, algunos practicantes se desconciertan con esta regla, prefiriendo recurrir a la tradición oral. Otros la resuelven poniéndose de acuerdo en estilos de notación, indentación y nomenclatura, así como en un valor apreciado en la práctica, el llamado "código revelador de intenciones". Como en XP rige un cierto purismo de codificación, los comentarios no son bien vistos. Si el código es tan oscuro que necesita comentario, se lo debe reescribir o refactorizar.
- **Espacio abierto.** Es preferible una sala grande con pequeños cubículos o, mejor todavía, sin divisiones. Los pares de programadores deben estar en el centro. En la periferia se ubican las máquinas privadas. En un encuentro de espacio abierto la agenda no se establece verticalmente.
- **Reglas justas**. El equipo tiene sus propias reglas a seguir, pero se pueden cambiar en cualquier momento. En XP se piensa que no existe un proceso que sirva para todos los proyectos; lo que se hace habitualmente es adaptar un conjunto de prácticas simples a las características de cada proyecto.

Las prácticas se han ido modificando con el tiempo. Originariamente eran doce; de inmediato, trece. Las versiones más recientes enumeran diecinueve prácticas, agrupadas en cuatro clases. Como se puede ver en la siguiente tabla.

Prácticas conjuntas	Iteraciones Vocabulario Común – Reemplaza a Metáforas Espacio de trabajo abierto Retrospectivas
Prácticas de Programador	Desarrollo orientado a pruebas Programación en pares Refactorización Propiedad colectiva Integración continua YAGNI ("No habrás de necesitarlo") – Equivale a Diseño Simple
Prácticas de Management	Responsabilidad aceptada Cobertura aérea para el equipo Revisión trimestral Espejo – El manager debe comunicar un fiel reflejo del estado de cosas Ritmo sostenible
Prácticas de Cliente	Narración de historias Planeamiento de entrega Prueba de aceptación Entregas frecuentes

Tabla 3.- Esquema de prácticas en XP.

Beck [Beck 99a] sugiere adoptar XP paulatinamente: "Si quiere intentar con XP, por Dios le pido que no se lo trague todo junto. Tome el peor problema de su proceso actual y trate de resolverlo a la manera de XP". Las prácticas también pueden adaptarse a las características de los sistemas particulares. Pero no todo es negociable y la integridad del método se sabe frágil. La particularidad de XP radica en no requerir ninguna herramienta fuera del ambiente de programación y prueba. Al contrario de otros métodos, que permiten modelado, XP demanda comunicación oral tanto para los requerimientos como para el diseño.

El ciclo de vida es, naturalmente, iterativo. La siguiente figura describe su cuerpo principal:

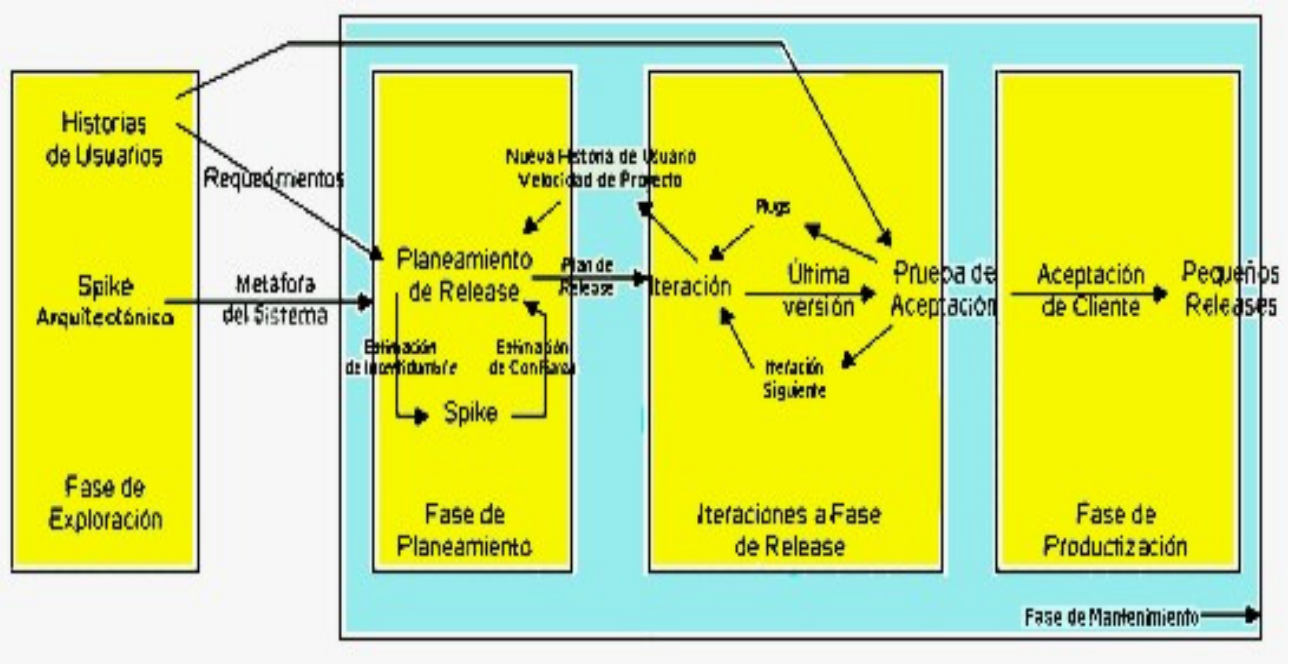


Fig. 12.- Ciclo de vida de XP, adaptado de [Beck 99a].

Algunos autores sugieren implementar spikes (es decir, "púas" o "astillas") para estimar la duración y dificultad de una tarea inmediata [JAH 01]. Un spike es un experimento dinámico de código, realizado para determinar cómo podría resolverse un problema. Es la versión ágil de la idea de prototipo. Se lo llama así porque "va de punta a punta, pero es muy fino" y porque en el recorrido de un árbol de opciones implementaría una opción de búsqueda depth-first (http://c2.com/cgi/wiki?SpikeSolution).

Entre los artefactos que se utilizan en XP vale la pena mencionar las tarjetas de historias (story cards); son tarjetas comunes de papel en que se escriben breves requerimientos de un rasgo, jamás Casos de Uso; pueden adoptar el esquema CRC. Las tarjetas tienen una granularidad de diez o veinte días. Las tarjetas se usan para estimar prioridades, alcance y tiempo de realización; en caso de discrepancia, gana la

estimación más optimista. Otros productos son listas de tareas en papel o en una pizarra (jamás en computadora) y gráficos visibles pegados en la pared.

Los roles de XP son pocos. Un cliente que escribe las historias y las pruebas de aceptación; programadores en pares; verificadores (que ayudan al cliente a desarrollar las pruebas); consultores técnicos; y como parte de la administración, un coach o consejero que es la conciencia del grupo, interviene y enseña, y un seguidor de rastros (tracker) que colecta las métricas y avisa cuando hay una estimación alarmante, además de un "Gran Jefe". La administración es la parte menos satisfactoriamente caracterizada en la bibliografía, como si fuera un mal necesario; Beck comenta, por ejemplo, que la función más relevante del coach es la adquisición de juguetes y comida [Hig 00b]; otros dicen que está para servir café. Lo importante es que el coach se vea como un rol para facilitar las cosas, en vez de como quien da las órdenes. Los equipos de XP son típicamente pequeños, de tres a veinte personas, y en general no se cree que su escala se avenga al desarrollo de grandes sistemas de misión crítica con tanta comodidad como FDD.

Algunas empresas han creado sus propias versiones de XP, como es el caso de Standard & Poor's, que ha elaborado plantillas para proyectos futuros. XP se ha combinado con Evo, a pesar que ambos difieren en su política de especificaciones; también se estima compatible con Scrum, al punto que existe una forma híbrida, inventada por Mike Beedle, que se llama XBreed [http://www.xbreed.net] y otra bautizada XP@Scrum [http://www.controlchaos.com/xpScrum.htm] en la que Scrum se usa como envoltorio de gestión alrededor de prácticas de ingeniería de XP. Se lo ha combinado muchas veces con UP, aunque éste recomienda pasar medio día de discusión de pizarra antes de programar y XP no admite más de 10 o 20 minutos. La diferencia mayor concierne a los Casos de Uso, que son norma en UP y que en XP se reemplazan por tarjetas de papel con historias simples que se refieren a rasgos. En las herramientas de RUP ahora hay plug-ins para XP.

Los obstáculos más comunes surgidos en proyectos XP son la "fantasía" [Lar 04] de pretender que el cliente se quede en el sitio y la resistencia de muchos programadores a trabajar en pares. Craig Larman señala como factores negativos la ausencia de énfasis en la arquitectura durante las primeras iteraciones (no hay arquitectos en XP) y la consiguiente falta de métodos de diseño arquitectónico. Un estudio de casos de Bernhard Rumpe y Astrid Schröder sobre 45 ejemplares de uso reveló que las prácticas menos satisfactorias de XP han sido la presencia del usuario en el lugar de ejecución y las metáforas, que el 40% de los encuestados no comprende para qué sirven o cómo usarlas [RS 01]. Las metáforas han sido reemplazadas por un vocabulario común (equivalente al "Sistema de Nombres" de Cunningham) en las últimas revisiones del método.

XP ha sido, de todos los MAs, el que más resistencia ha encontrado [Rak 01] [McB 02] [Baer 03] [Mel 03] [SR 03]. Algunos han sugerido que esa beligerancia es un efecto de su nombre, que debería ser menos intimidante. Jim Highsmith [Hig 02a] argumenta, sin embargo, que es improbable que muchos se entusiasmen por un libro que se titule, por ejemplo, Programación Moderada. Los nuevos mercados, tecnologías e ideas –piensa Highsmith– no se forjan a fuerza de moderación sino con giros radicales y con el coraje necesario para desafiar al status quo; XP, en todo caso, ha abierto el camino.

2.1.13 Feature Driven Development (FDD)

Feature Oriented Programming (FOP) es una técnica de programación guiada por rasgos o características (features) y centrada en el usuario, no en el programador; su objetivo es sintetizar un programa conforme

a los rasgos requeridos [Bat 03]. En un desarrollo en términos de FOP, los objetos se organizan en módulos o capas conforme a rasgos. FDD, en cambio, es un método ágil, iterativo y adaptable. A diferencia de otros MAs, no cubre todo el ciclo de vida sino sólo las fases de diseño y construcción y se considera adecuado para proyectos mayores y de misión crítica. FDD es, además, marca registrada de una empresa, Nebulon Pty. Aunque hay coincidencias entre la programación orientada por rasgos y el desarrollo guiado por rasgos, FDD no necesariamente implementa FOP.

FDD no requiere un modelo específico de proceso y se complementa con otras metodologías. Enfatiza cuestiones de calidad y define claramente entregas tangibles y formas de evaluación del progreso. Se lo reportó por primera vez en un libro de Peter Coad, Eric Lefebvre y Jeff DeLuca Java Modeling in Color with UML [CLD 00]; luego fue desarrollado con amplitud en un proyecto mayor de desarrollo por DeLuca, Coad y Stephen Palmer. Su implementación de referencia, análogo al C3 de XP, fue el Singapore Project; DeLuca había sido contratado para salvar un sistema muy complicado para el cual el contratista anterior había producido, luego de dos años, 3500 páginas de documentación y ninguna línea de código. Naturalmente, el proyecto basado en FDD fue todo un éxito, y permitió fundar el método en un caso real de misión crítica.

Los principios de FDD son pocos y simples:

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

Hay tres categorías de rol en FDD: roles claves, roles de soporte y roles adicionales. Los seis roles claves de un proyecto son: (1) administrador del proyecto, quien tiene la última palabra en materia de visión, cronograma y asignación del personal; (2) arquitecto jefe (puede dividirse en arquitecto de dominio y arquitecto técnico); (3) manager de desarrollo, que puede combinarse con arquitecto jefe o manager de proyecto; (4) programador jefe, que participa en el análisis del requerimiento y selecciona rasgos del conjunto a desarrollar en la siguiente iteración; (5) propietarios de clases, que trabajan bajo la guía del programador jefe en diseño, codificación, prueba y documentación, repartidos por rasgos y (6) experto de dominio, que puede ser un cliente, patrocinador, analista de negocios o una mezcla de todo eso.

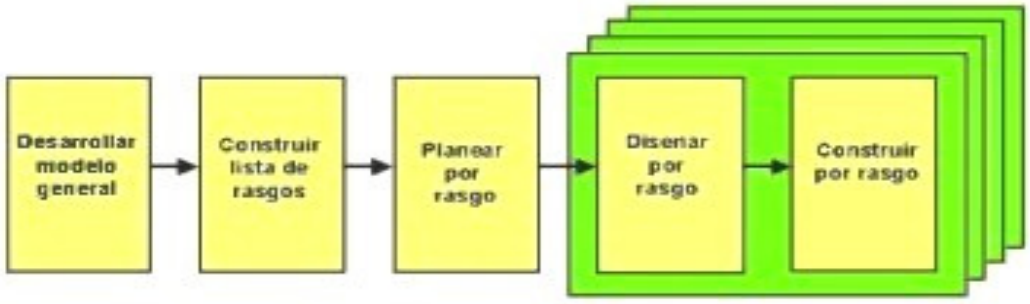


Fig. 13.- Proceso FDD, basado en [http://togethercommunities.com].

Los cinco roles de soporte comprenden: (1) administrador de entrega, que controla el progreso del proceso revisando los reportes del programador jefe y manteniendo reuniones breves con él; reporta al manager del proyecto; (2) abogado/gurú de lenguaje, que conoce a la perfección el lenguaje y la tecnología; (3) ingeniero de construcción, que se encarga del control de versiones de los builds y publica la documentación; (4) generador de herramientas "toolsmith", que construye herramientas ad hoc o mantiene bases de datos y sitios Web y (5) administrador del sistema, que controla el ambiente de trabajo o pone en producción el sistema cuando se lo entrega.

Los tres roles adicionales son los de verificadores, encargados del despliegue y escritores técnicos. Un miembro de un equipo puede tener otros roles a cargo, y un solo rol puede ser compartido por varias personas.

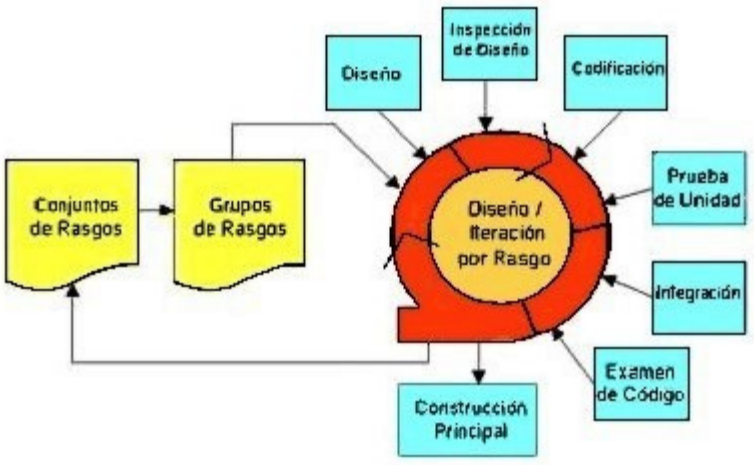


Fig. 14.- Ciclo de FDD, basado en [ASR 02].

FDD consiste en cinco procesos secuenciales durante los cuales se diseña y construye el sistema. La parte iterativa soporta desarrollo ágil con rápidas adaptaciones a cambios en requerimientos y necesidades del negocio. Cada fase del proceso tiene un criterio de entrada, tareas, pruebas y un criterio de salida. Típicamente, la iteración de un rasgo consume de una a tres semanas. Las fases son:

- **Desarrollo de un modelo general.** Cuando comienza este desarrollo, los expertos de dominio ya están al tanto de la visión, el contexto y los requerimientos del sistema a construir. A esta altura se espera que existan requerimientos tales como Casos de Uso o especificaciones funcionales. FDD, sin embargo, no cubre este aspecto. Los expertos de dominio presentan un ensayo (walkthrough) en el que los miembros del equipo y el arquitecto principal se informan de la descripción de alto nivel del sistema. El dominio general se subdivide en áreas más específicas y se define un ensayo más detallado para cada uno de los miembros del dominio. Luego de cada ensayo, un equipo de desarrollo trabaja en pequeños grupos para producir modelos de objeto de cada área de dominio. Simultáneamente, se construye un gran modelo general para todo el sistema.
- **Construcción de la lista de rasgos.** Los ensayos, modelos de objeto y documentación de requerimientos proporcionan la base para construir una amplia lista de rasgos. Los rasgos son pequeños ítems útiles a los ojos del cliente. Son similares a las tarjetas de historias de XP y se escriben en un lenguaje que todas las partes puedan entender. Las funciones se agrupan conforme a diversas actividades en áreas de dominio específicas. La lista de rasgos es revisada por los usuarios y patrocinadores para asegurar su validez y exhaustividad. Los rasgos que requieran más de diez días se descomponen en otros más pequeños.
- **Planeamiento por rasgo.** Incluye la creación de un plan de alto nivel, en el que los conjuntos de rasgos se ponen en secuencia conforme a su prioridad y dependencia, y se asigna a los programadores jefes. Las listas se priorizan en secciones que se llaman paquetes de diseño. Luego se asignan las clases definidas en la selección del modelo general a programadores individuales, o sea propietarios de clases. Se pone fecha para los conjuntos de rasgos.
- **Diseño por rasgo y construcción por rasgo.** Se selecciona un pequeño conjunto de rasgos del conjunto y los propietarios de clases seleccionan los correspondientes equipos dispuestos por rasgos. Se procede luego iterativamente hasta que se producen los rasgos seleccionados. Una iteración puede tomar de unos pocos días a un máximo de dos semanas. Puede haber varios grupos trabajando en paralelo. El proceso iterativo incluye inspección de diseño, codificación, prueba de unidad, integración e inspección de código. Luego de una iteración exitosa, los rasgos completos se promueven al "build" principal. Este proceso puede demorar una o dos semanas en implementarse.

FDD consiste en un conjunto de "mejores prácticas" que distan de ser nuevas pero se combinan de manera original. Las prácticas canónicas son:

- **Modelado de objetos del dominio, resultante en un framework cuando se agregan los rasgos.** Esta forma de modelado descompone un problema mayor en otros menores; el diseño y la implementación de cada clase u objeto es un problema pequeño a resolver. Cuando se combinan las clases completas, constituyen la solución al problema mayor. Una forma particular de la técnica es el modelado en colores [CLD 00], que agrega una dimensión adicional de visualización, si bien se puede modelar en blanco y negro. En FDD el modelado basado en objetos es imperativo.

- **Desarrollo por rasgo.** Hacer simplemente que las clases y objetos funcionen no refleja lo que el cliente pide. El seguimiento del progreso se realiza mediante examen de pequeñas funcionalidades descompuestas y funciones valoradas por el cliente. Un rasgo en FDD es una función pequeña expresada en la forma <acción> <resultado> <por | para | de | a> <objeto> con los operadores adecuados entre los términos. Por ejemplo, calcular el importe total de una venta; determinar la última operación de un cajero; validar la contraseña de un usuario.
- **Propiedad individual de clases (código).** Cada clase tiene una sola persona nominada como responsable por su consistencia, performance e integridad conceptual.
- **Equipos de rasgos, pequeños y dinámicamente formados.** La existencia de un equipo garantiza que un conjunto de mentes se apliquen a cada decisión y se tomen en cuenta múltiples alternativas.
- **Inspección.** Se refiere al uso de los mejores mecanismos de detección conocidos. FDD es tan escrupuloso en materia de inspección como lo es Evo.
- **Builds regulares.** Siempre se tiene un sistema disponible. Los builds forman la base a partir de la cual se van agregando nuevos rasgos.
- **Administración de configuración.** Permite realizar seguimiento histórico de las últimas versiones completas de código fuente.
- **Reporte de progreso.** Se comunica a todos los niveles organizacionales necesarios.

FDD suministra un rico conjunto de artefactos para la planificación y control de los proyectos. En http://www.nebulon.com/articles/fdd/fddimplementations.html se encuentran diversos formularios y tablas con información real de implementaciones de FDD: Vistas de desarrollo, Vistas de planificación, Reportes de progreso, Reportes de tendencia, Vista de plan (<acción><resultado><objeto>), etcétera. Se han desarrollado también algunas herramientas que generan vistas combinadas o específicas.

Id	Descripción	Prog. Jefe.	Prop. Clase	Ensayo		Diseño		Inspección de Diseño		Código		Inspección de Código		Promover a Build	
				Plan	Actual	Plan	Actual	Plan	Actual	Plan	Actual	Plan	Actual	Plan	Actual
MD125	Validar los límites transaccionales de un CAO contra una instrucción de implementación	CP	ABC	23/12/98	23/12/98	31/01/99	31/01/99	01/02/99	01/02/99	10/02/99		18/02/99		20/02/99	
MD126	Definir el estado de una instrucción de implementación	CP	ABC	23/12/98	23/12/98	31/01/99	31/01/99	01/02/99	01/02/99	10/02/99		18/02/99		20/02/99	
MD127	Especificar el oficial de autorización de una instrucción de implementación	CP	ABC	23/12/98	23/12/98	31/01/99	31/01/99	01/02/99	01/02/99	10/02/99		18/02/99		20/02/99	
MD128	Rechazar una instrucción de implementación para un conjunto de líneas	CP	ABC	STATUS: Inactivo NOTA: [agregado por CK: 3/2/99] No aplicable											
MD129	Confirmar una instrucción de implementación para un conjunto de líneas	CP	ABC	23/12/98	23/12/98	31/01/99	31/01/99	01/02/99	01/02/99	10/02/99		18/02/99		20/02/99	
MD130	Determinar si todos los documentos se han completado para un prestatario	CP	ABC	23/12/98	23/12/98	31/01/99	31/01/99	01/02/99	01/02/99	05/02/99		08/02/99		10/02/99	
				NOTA: [agregado por SL: 3/2/99] Bloqueado en AS											
MD131	Validar los límites transaccionales de un CAO contra una instrucción de desembolso	CP	ABC	23/12/98	23/12/98	31/01/99	31/01/99	01/02/99	01/03/99	05/02/99		08/02/99		10/02/99	
				NOTA: [agregado por: 3/2/99] Atrasado según estimaciones iniciales											
MD132	Enviar para autorización una instrucción de implementación	CP	ABC	23/12/98	23/12/98	31/01/99	31/01/99	01/02/99	01/02/99	05/02/99	05/02/99	06/02/99	06/02/99	08/02/99	08/02/99
MD133	Validar fecha de baja de una instrucción de implementación	CP	ABC	23/12/98	23/12/98	31/01/99	31/01/99	01/02/99	01/02/99	05/02/99	05/02/99	06/02/99	06/02/99	08/02/99	08/02/99

Fig. 15.- Plan de rasgo – Implementación – Basado en http://www.nebulon.com/articles/fdd/planview.html.

La matriz muestra un ejemplo de vista de un plan de rasgo, con la típica codificación en colores. En síntesis, FDD es un método de desarrollo de ciclos cortos que se concentra en la fase de diseño y construcción. En la primera fase, el modelo global de dominio es elaborado por expertos del dominio y

desarrolladores; el modelo de dominio consiste en Diagramas de Clases con clases, relaciones, métodos y atributos. Los métodos no reflejan conveniencias de programación sino rasgos funcionales.

Algunos "agilistas" sienten que FDD es demasiado jerárquico para ser un método ágil, porque demanda un programador jefe, quien dirige a los propietarios de clases, quienes dirigen equipos de rasgos. Otros críticos sienten que la ausencia de procedimientos detallados de prueba en FDD es llamativa e impropia. Los promotores del método aducen que las empresas ya tienen implementadas sus herramientas de prueba, pero subsiste el problema de su adecuación a FDD. Un rasgo llamativo de FDD es que no exige la presencia del cliente. FDD se utilizó por primera vez en grandes aplicaciones bancarias a fines de la década de 1990. Los autores sugieren su uso para proyectos nuevos o actualizaciones de sistemas existentes, y recomiendan adoptarlo en forma gradual. En [ASR 02] se asegura que aunque no hay evidencia amplia que documente sus éxitos, las grandes consultoras suelen recomendarlo incluso para delicados proyectos de misión crítica.

2.1.14 Adaptive Software Development

James Highsmith III, consultor de Cutter Consortium, desarrolló ASD hacia el año 2000 con la intención primaria de ofrecer una alternativa a la idea, propia de CMM Nivel 5, de que la optimización es la única solución para problemas de complejidad creciente. Este método ágil pretende abrir una tercera vía entre el "desarrollo monumental de software" y el "desarrollo accidental", o entre la burocracia y la adhocracia. Deberíamos buscar más bien, afirma Highsmith, "el rigor estrictamente necesario"; para ello hay que situarse en coordenadas apenas un poco fuera del caos y ejercer menos control que el que se cree necesario [Hig 00a].

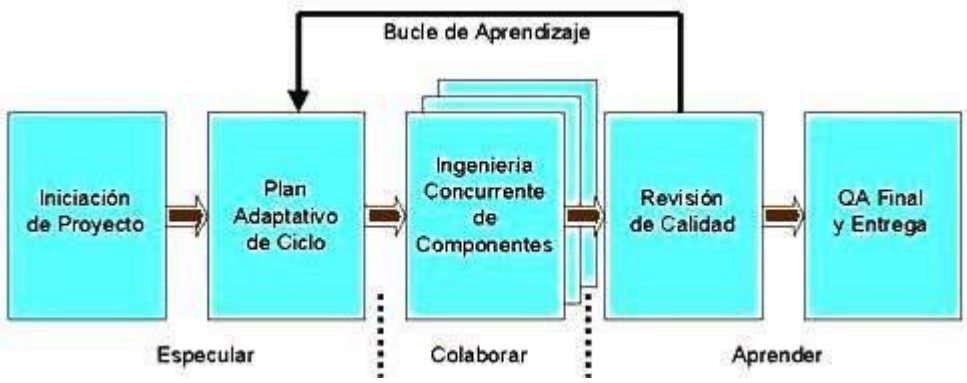


Fig. 16.- Fases del ciclo de vida de ASD, basado en Highsmith [Hig 00a: 84].

La estrategia entera se basa en el concepto de emergencia, una propiedad de los sistemas adaptables complejos que describe la forma en que la interacción de las partes genera una propiedad que no puede

ser explicada en función de los componentes individuales. El pensador de quien Highsmith toma estas ideas es John Holland, el creador del algoritmo genético y probablemente el investigador actual más importante en materia de procesos emergentes [Hol 95]. Holland se pregunta, entre otras cosas, cómo hace un macro-sistema extremadamente complejo, no controlado de arriba hacia abajo en todas las variables que intervienen (como por ejemplo la ciudad de Nueva York o la Web) para mantenerse funcionando en un aparente equilibrio sin colapsar.

La respuesta, que tiene que ver con la auto-organización, la adaptación al cambio y el orden que emerge de la interacción entre las partes, remite a examinar analogías con los sistemas adaptables complejos por excelencia, esto es: los organismos vivientes (o sus análogos digitales, como las redes neuronales auto-organizativas de Teuvo Kohonen y los autómatas celulares desde Von Neumann a Stephen Wolfram). Para Highsmith, los proyectos de software son sistemas adaptables complejos y la optimización no hace más que sofocar la emergencia necesaria para afrontar el cambio. Llevando más allá la analogía, Highsmith interpreta la organización empresarial que emprende un desarrollo como si fuera un ambiente, sus miembros como agentes y el producto como el resultado emergente de relaciones de competencia y cooperación. En los sistemas complejos no es aplicable el análisis, porque no puede deducirse el comportamiento del todo a partir de la conducta de las partes, ni sumarse las propiedades individuales para determinar las características del conjunto: el oxígeno es combustible, el hidrógeno también, pero cuando se combinan se obtiene agua, la cual no tiene esa propiedad.

Highsmith indaga además la economía del retorno creciente según Brian Arthur. Esta economía se caracteriza por la alta velocidad y la elevada tasa de cambio. Velocidad y cambio introducen una complejidad que no puede ser manipulada por las estrategias convencionales. En condiciones de complejidad el mercado es formalmente impredecible y el proceso de desarrollo deviene imposible de planificar bajo el supuesto de que después se tendrá control del proceso. El razonamiento simple de causa y efecto no puede dar cuenta de la situación y mucho menos controlarla. En este contexto, los sistemas adaptables complejos suministran los conceptos requeridos: agentes autónomos que compiten y cooperan, los ambientes mutables y la emergencia. Tomando como punto de partida estas ideas, Highsmith elabora un modelo de gestión que llama "Modelo de Liderazgo-Colaboración Adaptativo" (L-C), el cual tiene puntos en común con el modelado basado en agentes autónomos; en este modelo se considera que la adaptabilidad no puede ser comandada, sino que debe ser nutrida: nutriendo de conducta adaptable a cada agente, el sistema global deviene adaptable.

ASD presupone que las necesidades del cliente son siempre cambiantes. La iniciación de un proyecto involucra definir una misión para él, determinar las características y las fechas y descomponer el proyecto en una serie de pasos individuales, cada uno de los cuales puede abarcar entre cuatro y ocho semanas. Los pasos iniciales deben verificar el alcance del proyecto; los tardíos tienen que ver con el diseño de una arquitectura, la construcción del código, la ejecución de las pruebas finales y el despliegue.

Aspectos claves de ASD

- Un conjunto no estándar de "artefactos de misión" (documentos para ti y para mí), incluyendo una visión del proyecto, una hoja de datos, un perfil de misión del producto y un esquema de su especificación.
- Un ciclo de vida, inherentemente iterativo.
- Cajas de tiempo, con ciclos cortos de entrega orientados por riesgo.

Un ciclo de vida es una iteración; este ciclo se basa en componentes y no en tareas, es limitado en el tiempo, orientado por riesgos y tolerante al cambio. Que se base en componentes implica concentrarse en el desarrollo de software que trabaje, construyendo el sistema pieza por pieza. En este paradigma, el cambio es bienvenido y necesario, pues se concibe como la oportunidad de aprender y ganar así una ventaja competitiva; de ningún modo es algo que pueda ir en detrimento del proceso y sus resultados.

Highsmith piensa que los procesos rigurosos (repetibles, visibles, medibles) son encomiables porque proporcionan estabilidad en un entorno complejo, pero muchos procesos en el desarrollo (por ejemplo, el diseño del proyecto) deberían ser flexibles. La clave para mantener el control radica en los "estados de trabajo" (la colección de los productos de trabajo) y no en el flujo de trabajo (workflow). Demasiado rigor, por otra parte, acarrea rigor mortis, el cual impide cambiar el producto cuando se introducen las inevitables modificaciones. En la moderna teoría económica del retorno creciente, ser capaz de adaptarse es significativamente más importante que ser capaz de optimizar [Hig 00a].

La idea subyacente a ASD (y de ahí su particularidad) radica en que no proporciona un método para el desarrollo de software sino que más bien suministra la forma de implementar una cultura adaptable en la empresa, con capacidad para reconocer que la incertidumbre y el cambio son el estado natural. El problema inicial es que la empresa no sabe que no sabe, y por tal razón debe aprender. Los cuatro objetivos de este proceso de aprendizaje son entonces:

- Prestar soporte a una cultura adaptable o un conjunto mental para que se espere cambio e incertidumbre y no se tenga una falsa expectativa de orden.
- Introducir marcos de referencia para orientar el proceso iterativo de gestión del cambio.
- Establecer la colaboración y la interacción de la gente en tres niveles: interpersonal, cultural y estructural.
- Agregar rigor y disciplina a una estrategia RAD, haciéndola escalable a la complejidad de los intentos de la vida real.

ASD se concentra más en los componentes que en las tareas; en la práctica, esto se traduce en ocuparse más de la calidad que en los procesos usados para producir un resultado. En los ciclos adaptables de la fase de Colaboración, el planeamiento es parte del proceso iterativo, y las definiciones de los componentes se refinan continuamente, como se puede ver en la figura anterior. La base para los ciclos posteriores (el bucle de Aprendizaje) se obtiene a través de repetidas revisiones de calidad con presencia del cliente como experto, constituyendo un grupo de foco de cliente. Esto ocurre solamente al final de las fases, por lo que la presencia del cliente se suplementa con sesiones de desarrollo conjunto de aplicaciones (JAD). Hemos visto que una sesión JAD, común en el antiguo RAD, es un taller en el que programadores y representantes del cliente se encuentran para discutir rasgos del producto en términos no técnicos, sino de negocios.

El modelo de Highsmith es, naturalmente, complementario a cualquier concepción dinámica del método; no podría ser otra cosa que, después de todo, y por ello admite y promueve integración con otros modelos y marcos. En ASD la redundancia puede ser un subproducto táctico inevitable en un ambiente competitivo y debe aceptarse en tanto el producto sea "suficientemente bueno". En materia de técnicas, ASD las considera importantes pero no más que eso.

Hay ausencia de estudios de casos del método adaptable, aunque las referencias literarias a sus principios son abundantes. Como ASD no constituye un método de ingeniería de ciclo de vida sino una visión cultural o una epistemología, no califica como framework suficiente para articular un proyecto.

2.1.15 Dynamic System Development Method (DSDM)

Originado en los trabajos de Jennifer Stapleton, directora del DSDM Consortium, DSDM se ha convertido en el framework de desarrollo rápido de aplicaciones (RAD) más popular de Gran Bretaña [Sta 97] y se ha llegado a promover como el estándar de facto para desarrollo de soluciones de negocios sujetas a márgenes de tiempo estrechos. Se calcula que uno de cada cinco desarrolladores en Gran Bretaña utiliza DSDM y que más de 500 empresas mayores lo han adoptado.

Además de un método, DSDM proporciona un framework completo de controles para RAD y lineamientos para su uso. DSDM puede complementar metodologías de XP, RUP o Microsoft Solutions Framework, o combinaciones de todas ellas. DSDM es relativamente antiguo en el campo de los MAs y constituye una metodología madura, que ya va por su cuarta versión. Se dice que ahora las iniciales DSDM significan Dynamic Solutions Delivery Method. Ya no se habla de sistemas sino de soluciones, y en lugar de priorizar el desarrollo se prefiere enfatizar la entrega. El libro más reciente que sintetiza las prácticas se llama DSDM: Business Focused Development [DS 03].

La idea dominante detrás de DSDM es explícitamente inversa a la que se encuentra en otras partes, y al principio resulta contraria a la intuición; en lugar de ajustar tiempo y recursos para lograr cada funcionalidad, en esta metodología tiempo y recursos se mantienen como constantes y se ajusta la funcionalidad de acuerdo con ello. Esto se expresa a través de reglas que se conocen como "reglas MoSCoW" por las iniciales de su estipulación en inglés. Las reglas se refieren a rasgos del requerimiento:

- **Must have:** Debe tener. Son los requerimientos fundamentales del sistema. De éstos, el subconjunto mínimo ha de ser satisfecho por completo.
- **Should have:** Debería tener. Son requerimientos importantes para los que habrá una resolución en el corto plazo.
- **Could have:** Podría tener. Podrían quedar fuera del sistema si no hay más remedio.
- **Want to have but won't have this time around:** Se desea que tenga, pero no lo tendrá esta vuelta. Son requerimientos valorados, pero pueden esperar.

DSDM consiste en cinco fases:

- Estudio de viabilidad.
- Estudio del negocio.
- Iteración del modelo funcional.
- Iteración de diseño y versión.
- Implementación.

Las últimas tres fases son iterativas e incrementales. De acuerdo con la iniciativa de mantener el tiempo constante, las iteraciones de DSDM son cajas de tiempo. La iteración acaba cuando el tiempo se consume. Se supone que al cabo de la iteración los resultados están garantizados. Una caja de tiempo puede durar de unos pocos días a unas pocas semanas.

A diferencia de otros MAs, DSDM ha desarrollado sistemáticamente el problema de su propia implantación

en una empresa. El proceso de Examen de Salud (Health Check) de DSDM se divide en dos partes que se interrogan, sucesivamente, sobre la capacidad de una organización para adoptar el método y sobre la forma en que éste responde a las necesidades una vez que el proyecto está encaminado. Un Examen de Salud puede consumir entre tres días y un mes de trabajo de consultoría.

- **Estudio de factibilidad.** Se evalúa el uso de DSDM o de otra metodología conforme al tipo de proyecto, variables organizacionales y de personal. Si se opta por DSDM, se analizan las posibilidades técnicas y los riesgos. Se preparan como productos un reporte de viabilidad y un plan sumario para el desarrollo. Si la tecnología no se conoce bien, se hace un pequeño prototipo para ver qué pasa. No se espera que el estudio completo consuma más de unas pocas semanas. Es mucho para un método ágil, pero menos de lo que demandan algunos métodos clásicos.
- **Estudio del negocio.** Se analizan las características del negocio y la tecnología. La estrategia recomendada consiste en el desarrollo de talleres, donde se espera que los expertos del cliente consideren las facetas del sistema y acuerden sus prioridades de desarrollo. Se describen los procesos de negocio y las clases de usuario en una definición del área de negocios. Se espera así reconocer e involucrar a gente clave de la organización en una etapa temprana. La definición utiliza descripciones de alto nivel, como diagramas de entidad-relación o modelos de objetos de negocios. Otros productos son la definición de arquitectura del sistema y el plan de bosquejo de prototipos. La definición arquitectónica es un primer bosquejo y se admite que cambie en el curso del proyecto DSDM. El plan debe establecer la estrategia de prototipos de las siguientes etapas y un plan para la gestión de configuración.
- **Iteración del modelo funcional.** En cada iteración se planea el contenido y la estrategia, se realiza la iteración y se analizan los resultados pensando en las siguientes. Se lleva a cabo tanto el análisis como el código; se construyen los prototipos y en base a la experiencia se mejoran los modelos de análisis. Los prototipos no han de ser descartados por completo, sino gradualmente mejorados hacia la calidad que debe tener el producto final. Se produce como resultado un modelo funcional, conteniendo el código del prototipo y los modelos de análisis. También se realizan pruebas constantemente. Hay otros cuatro productos emergentes: (1) Funciones priorizadas en una lista de funciones entregadas al fin de cada iteración; (2) Los documentos de revisión del prototipo funcional reúnen los comentarios de los usuarios sobre el incremento actual para ser considerados en iteraciones posteriores; (3) Los requerimientos funcionales son listas que se construyen para ser tratadas en fases siguientes; (4) El análisis de riesgo de desarrollo ulterior es un documento importante en la fase de iteración del modelo, porque desde la fase siguiente en adelante los problemas que se encuentren serán más difíciles de tratar.
- **Iteración de diseño y construcción.** Aquí es donde se construye la mayor parte del sistema. El producto es un sistema probado que reúne por lo menos el conjunto mínimo de requerimientos acordados conforme a las reglas "MoSCoW". El diseño y la construcción son iterativos y el diseño y los prototipos funcionales son revisados por usuarios. El desarrollo ulterior se atiene a sus comentarios.
- **Despliegue.** El sistema se transfiere del ambiente de desarrollo al de producción. Se entrena a los usuarios, que ponen las manos en el sistema. Eventualmente la fase puede llegar a iterarse. Otros productos son el manual de usuario y el reporte de revisión del sistema. A partir de aquí hay cuatro cursos de acción posibles: (1) Si el sistema satisface todos los requerimientos, el desarrollo ha terminado. (2) Si quedan muchos requerimientos sin resolver, se puede correr el proceso nuevamente desde el comienzo. (3) Si se ha dejado de lado alguna prestación no crítica, el proceso se puede correr desde la iteración funcional del modelo en adelante. (4) Si algunas

cuestiones técnicas no pudieron resolverse por falta de tiempo se puede iterar desde la fase de diseño y construcción.

La configuración del ciclo de vida de DSDM se representa con un diagrama característico (del cual hay una evocación en el logotipo del consorcio) que vale la pena reproducir:

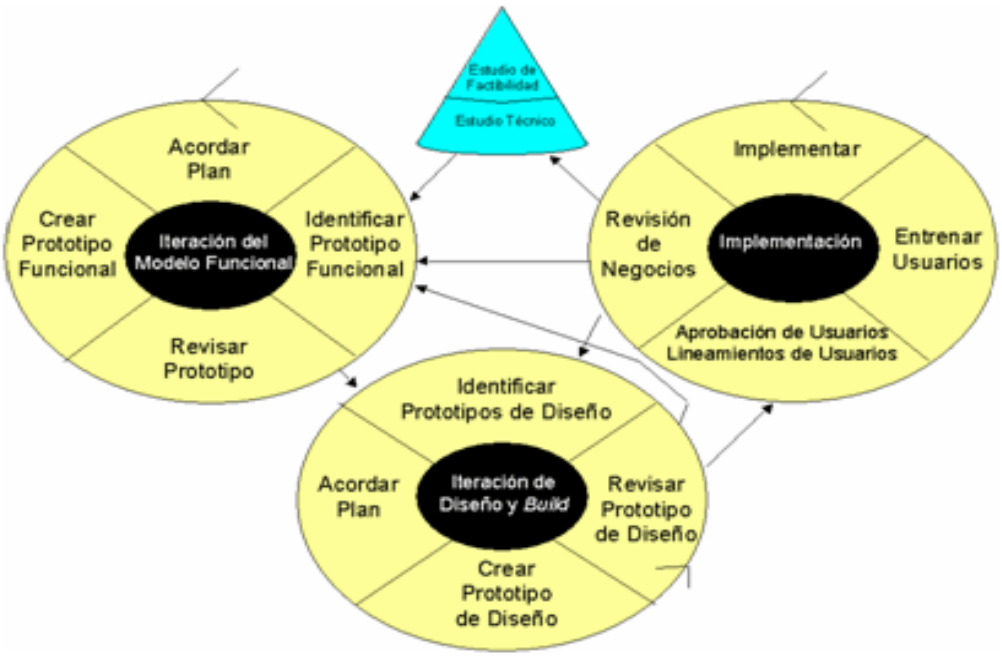


Fig. 17.- Proceso de desarrollo DSDM, basado en [http://www.dsdm.org].

DSDM define quince roles, algo más que el promedio de los MAs. Los más importantes son:

- Programadores y Programadores Senior. Son los únicos roles de desarrollo. El título de Senior indica también nivel de liderazgo dentro del equipo. Equivale a Nivel 3 de Cockburn. Ambos títulos cubren todos los roles de desarrollo, incluyendo analistas, diseñadores, programadores y verificadores.
- **Coordinador técnico.** Define la arquitectura del sistema y es responsable por la calidad técnica del proyecto, el control técnico y la configuración del sistema.
- **Usuario embajador.** Proporciona al proyecto conocimiento de la comunidad de usuarios y disemina información sobre el progreso del sistema hacia otros usuarios. Se define adicionalmente un rol de Usuario Asesor (Advisor) que representa otros puntos de vista importantes; puede ser alguien del personal de IT o un auditor funcional.
- **Visionario.** Es un usuario participante que tiene la percepción más exacta de los objetivos del

sistema y el proyecto. Asegura que los requerimientos esenciales se cumplan y que el proyecto vaya en la dirección adecuada desde el punto de vista de aquéllos.

- **Patrocinador Ejecutivo.** Es la persona de la organización que detenta autoridad y responsabilidad financiera, y es quien tiene la última palabra en las decisiones importantes.
- **Facilitador.** Es responsable de administrar el progreso del taller y el motor de la preparación y la comunicación.
- **Escriba.** Registra los requerimientos, acuerdos y decisiones alcanzadas en las reuniones, talleres y sesiones de generación de prototipos.

En DSDM las prácticas se llaman Principios, y son nueve:

- Es imperativo el compromiso activo del usuario.
- Los equipos de DSDM deben tener el poder de tomar decisiones.
- El foco radica en la frecuente entrega de productos.
- El criterio esencial para la aceptación de los entregables es la adecuación a los propósitos de negocios.
- Se requiere desarrollo iterativo e incremental.
- Todos los cambios durante el desarrollo son reversibles.
- La línea de base de los requerimientos es de alto nivel. Esto permite que los requerimientos de detalle se cambien según se necesite y que los esenciales se capten tempranamente.
- La prueba está integrada a través de todo el ciclo de vida. La prueba también es incremental. Se recomienda particularmente la prueba de regresión, de acuerdo con el estilo evolutivo de desarrollo.
- Es esencial una estrategia cooperativa entre todos los participantes. Las responsabilidades son compartidas y la colaboración entre usuario y desarrolladores no debe tener fisuras.

Desde mediados de la década de 1990 hay abundantes estudios de casos, sobre todo en Gran Bretaña, y la adecuación de DSDM para desarrollo rápido está suficientemente probada [ASR 02]. El equipo mínimo de DSDM es de dos personas y puede llegar a seis, pero puede haber varios equipos en un proyecto. El mínimo de dos personas involucra que un equipo consiste de un programador y un usuario. El máximo de seis es el valor que se encuentra en la práctica. DSDM se ha aplicado a proyectos grandes y pequeños. La precondición para su uso en sistemas grandes es su partición en componentes que pueden ser desarrollados por equipos normales.

Se ha elaborado en particular la combinación de DSDM con XP y se ha llamado a esta combinación "EnterpriseXP", término acuñado por Mike Griffiths de Quadrus Developments (http://www.enterprisexp.org). Se atribuye a Kent Beck haber afirmado que la comunidad de DSDM ha construido una imagen corporativa mejor que la del mundo XP y que sería conveniente aprender de esa experiencia. También hay documentos conjuntos de DSDM y Rational, con participación de Jennifer

Stapleton, que demuestran la compatibilidad del modelo DSDM con RUP, a despecho de sus fuertes diferencias terminológicas.

En el sitio del DSDM Consortium hay documentos específicos sobre la convivencia de la metodología con Microsoft Solutions Framework.

2.1.16 Rational Unified Process (RUP)

Uno podría preguntarse legítimamente qué hace RUP en el contexto de los MAs ¿No es más bien representativo de la filosofía a la que el "Manifiesto" se opone? ¿Están tratando los métodos clásicos de cooptar a los métodos nuevos? El hecho es que existe una polémica aún en curso respecto de si los métodos asociados al "Proceso Unificado", y en particular RUP, representan técnicas convencionales y pesadas o si por el contrario son adaptabless al programa de los MAs.

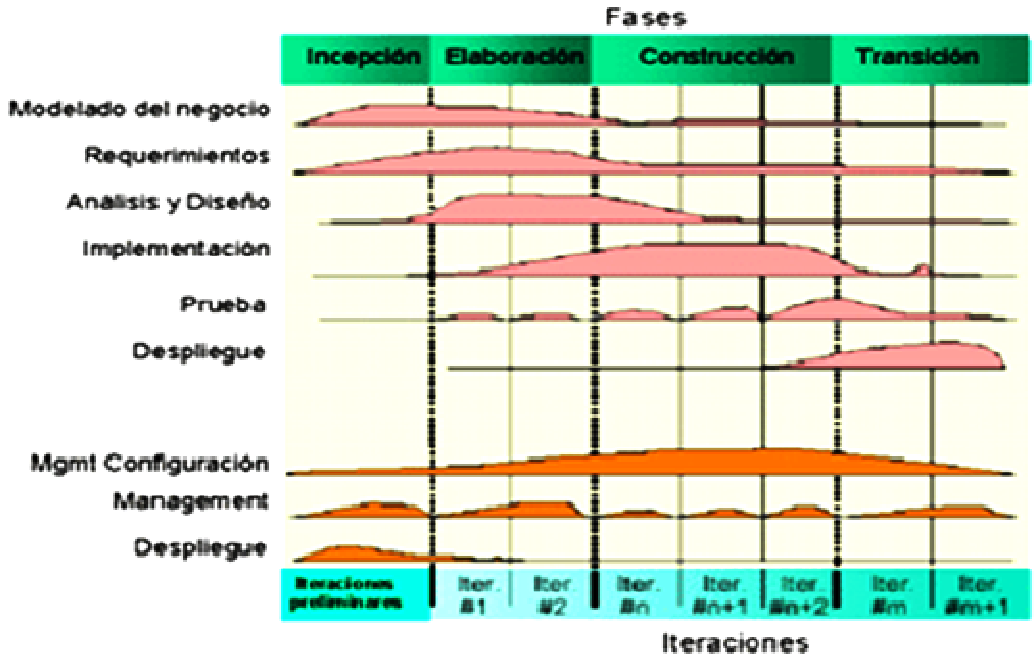


Fig. 18.- Fases y workflows de RUP, basado en [BMP 98].

Philippe Kruchten, impulsor tanto del famoso modelo de vistas 4+1 como de RUP, ha participado en el célebre "debate de los gurúes" [Hig 01] afirmando que RUP es particularmente apto para ambas clases de escenarios. Kruchten ha sido en general conciliador; aunque el diseño orientado a objetos suele otorgar a

la Arquitectura de Software académica un lugar modesto, el modelo de 4+1 [Kru 95] se inspira explícitamente en el modelo arquitectónico fundado por Dewayne Perry y Alexander Wolf [PW 92]. Recíprocamente, 4+1 tiene cabida en casi todos los métodos basados en arquitectura desarrollados recientemente por el SEI. Highsmith, comentando la movilidad de RUP de un campo a otro, señaló que nadie, en apariencia, está dispuesto a conceder "agilidad" a sus rivales; RUP pretende ser ágil, y tal vez lo sea. De hecho, los principales textos que analizan MAs acostumbran a incluir al RUP entre los más representativos, o como un método que no se puede ignorar [ASR 02] [Lar 04].

El RUP debido a Jacobson, Booch y Rumbaugh, se publicó en 1999.

Este proceso se deriva de metodologías anteriores desarrolladas por estos tres autores, a saber, la metodología "Objectory" de Jacobson, la "metodología de Booch" y la "técnica de modelado de objetos" de Rumbaugh.

Debido a que los enfoques iterativos repiten todas las partes del proceso en cascada, puede ser complicado describirlos.

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

- **Incepción**. Significa "comienzo", pero la palabra original (de origen latino y casi en desuso como sustantivo) es sugestiva y por ello la traducimos así. Se especifican los objetivos del ciclo de vida del proyecto y las necesidades de cada participante. Esto entraña establecer el alcance y las condiciones de límite y los criterios de aceptabilidad. Se identifican los Casos de Uso que orientarán la funcionalidad. Se diseñan las arquitecturas candidatas y se estima la agenda y el presupuesto de todo el proyecto, en particular para la siguiente fase de elaboración. Típicamente es una fase breve que puede durar unos pocos días o unas pocas semanas.
- **Elaboración.** Se analiza el dominio del problema y se define el plan del proyecto. RUP presupone que la fase de elaboración brinda una arquitectura suficientemente sólida junto con requerimientos y planes bastante estables. Se describen en detalle la infraestructura y el ambiente de desarrollo, así como el soporte de herramientas de automatización. Al cabo de esta fase, debe estar identificada la mayoría de los Casos de Uso y los actores, debe quedar descrita la arquitectura de software y se debe crear un prototipo de ella. Al final de la fase se realiza un análisis para determinar los riesgos y se evalúan los gastos hechos contra los originalmente planeados.
- **Construcción.** Se desarrollan, integran y verifican todos los componentes y rasgos de la aplicación. RUP considera que esta fase es un proceso de manufactura, en el que se debe poner énfasis en la administración de los recursos y el control de costos, agenda y calidad. Los resultados de esta fase (las versiones alfa, beta y otras versiones de prueba) se crean tan rápido como sea posible. Se debe compilar también una versión de entrega. Es la fase más prolongada de todas.
- **Transición.** Comienza cuando el producto está suficientemente maduro para ser entregado. Se corrigen los últimos errores y se agregan los rasgos pospuestos. La fase consiste en prueba beta, piloto, entrenamiento a usuarios y despacho del producto a mercadeo, distribución y ventas. Se produce también la documentación. Se llama transición porque se transfiere a las manos del usuario, pasando del entorno de desarrollo al de producción.

A través de las fases se desarrollan en paralelo nueve "workflows" o disciplinas: Modelado de Negocios, Requerimientos, Análisis y Diseño, Implementación, Prueba, Gestión de Configuración y Cambio, Gestión del Proyecto y Entorno. Además de estos "workflows", RUP define algunas prácticas comunes:

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

De esta forma se han mostrado los principales modelos de procesos de Software.

2.2 Administración de Proyectos de Software

Definición: Un proyecto es un esfuerzo temporal emprendido para crear un producto, servicio o resultado único [PMI 2004].

La administración de proyectos consiste en gestionar la producción de un producto dentro del tiempo dado y los límites de fondos. Como esto requiere recursos humanos, la administración del proyecto involucra no sólo la organización técnica y las habilidades organizacionales, sino también el arte de administrar personas. La administración de proyectos no es una actividad insignificante y en general comprende:

- **Estructura** (elementos organizacionales involucrados).
- **Proceso administrativo** (responsabilidades y supervisión de los participantes).
- **Proceso de desarrollo** (métodos, herramientas, lenguajes, documentación y apoyo).
- **Programa** (tiempos en los que deben realizarse las porciones del trabajo).

2.2.1 Variables principales

Quienes planean los proyectos pueden variar costos, capacidades, calidad y fechas de entrega. El grado en el que estos cuatro factores pueden controlarse depende del proyecto. Aunque con frecuencia los costos pueden estar fijos de antemano, muchas veces hay cierta flexibilidad. Por ejemplo, si queremos agregarle al sistema unas nuevas funciones (que genere gráficos o que sea más robusto en cuanto a seguridad), será necesario incrementar el costo del sistema. Las capacidades tampoco son las entidades fijas que parecen. Por ejemplo, el cliente puede estar de acuerdo con eliminar un requerimiento si hacerlo disminuye 15% la duración del proyecto (un trueque entre capacidades y fechas). Aunque puede parecer poco ortodoxo, incluso las metas de calidad pueden variar. Cuando las metas de calidad se establecen demasiado bajas, se crea un desequilibrio en los costos a corto y largo plazos debido al retrabajo e insatisfacción del cliente. Cuando las metas de calidad son demasiado altas, el costo de encontrar hasta el detalle más pequeño puede ser prohibitivo. Por ejemplo, la mayoría de las personas no estarían dispuestas a pagar el triple del precio por un procesador de palabras con e fin de tener una versión sin un defecto trivial reconocido. En ocasiones, se pueden negociar las fechas de terminación. Por ejemplo, un gerente puede estar dispuesto a cambiar una fecha de entrega si el producto resultante tiene tantas capacidades que es probable que capte el mercado.

La gestión eficaz de un proyecto de software se centra en un concepto que se denomina las cuatro "P's": personal, producto, proceso y proyecto. El orden no es arbitrario, el gestor que se olvida de que el trabajo de Ingeniería de Software es un esfuerzo humano intenso nunca tendrá éxito en la gestión de proyectos, un gestor que no fomenta una minuciosa comunicación con el cliente al principio de la evolución del proyecto se arriesga a construir una elegante solución para un problema equivocado. El administrador que presta poca atención al proceso corre el riesgo de arrojar métodos técnicos y herramientas eficaces al vacío. El gestor que emprende un proyecto sin un plan sólido arriesga el éxito del producto.

Personal

Desde los años 60 se viene discutiendo la necesidad de contar con personal altamente preparado y motivado para el desarrollo del software (por ejemplo [COU 80, WIT 94, DEM 98]). De hecho, el "factor humano" es tan importante que el Instituto de Ingeniería de Software ha desarrollado un modelo de madurez de la capacidad de gestión de personal (MMCGP) "para aumentar la preparación de organizaciones del software para llevar a cabo las cada vez más complicadas aplicaciones ayudando a atraer, aumentar, motivar, desplegar y retener el talento necesario para mejorar su capacidad de desarrollo de software [CUR 94].

El modelo de madurez de gestión de personal define las siguientes áreas clave prácticas para el personal que desarrolla software: reclutamiento, selección, gestión de rendimiento, entrenamiento, retribución, desarrollo de la carrera, diseño de la organización y del trabajo y desarrollo cultural y de espíritu de equipo.

El MMCGP es compañero del modelo de madurez de la capacidad de software, que guía a las organizaciones en la creación de un proceso de software maduro.

Producto

Antes de poder planificar un proyecto, se deberían establecer los objetivos y ámbito del producto, convendría considerar soluciones alternativas e identificar las dificultades técnicas y de gestión. Sin esta información es imposible definir unas estimaciones razonables (y exactas) del costo: una valoración efectiva del riesgo, una subdivisión realista de las tareas del proyecto o una planificación del proyecto asequible que proporcione una indicación fiable del progreso.

El desarrollador de software y el cliente deben reunirse para definir los objetivos del producto y su ámbito. En muchos casos, esta actividad empieza como parte del proceso de ingeniería del sistema o del negocio y continúa como el primer paso en el análisis de los requisitos del software. Los objetivos identifican las metas generales del proyecto sin considerar cómo se conseguirán (desde el punto de vista del cliente).

El ámbito identifica los datos primarios, funciones y comportamientos que caracterizan al producto, y más importante, intenta abordar estas características de una manera cuantitativa.

Una vez que se han entendido los objetivos y el ámbito del producto, se consideran soluciones alternativas.

Proceso

Un proceso de software proporciona la estructura desde la que se puede establecer un detallado plan para el desarrollo del software. Un pequeño número de actividades estructurales se puede aplicar a todos los proyectos de software, sin tener en cuenta su tamaño o complejidad. Diferentes conjuntos de -tareas, hitos, productos del trabajo y puntos de garantía de calidad- permiten a las actividades estructurales adaptarse a las características del proyecto de software y a los requisitos del equipo del proyecto. Finalmente, las actividades protectoras –tales como garantía de calidad del software, gestión de la configuración del software y medición cubren el modelo de proceso. Las actividades protectoras son independientes de las estructurales y tienen lugar a lo largo del proceso.

Proyecto

Dirigimos los proyectos de software planificados y controlados por una razón principal – es la única manera conocida de gestionarla complejidad-. En 1998, los datos de la industria del software indicaron que el 26% de proyectos de software fallaron completamente y que el 46% experimentaron un desbordamiento en la planificación y en el costo [REE 99]. Aunque la proporción de éxito para los proyectos de software ha mejorado un poco, nuestra proporción de fracaso de proyecto permanece más alto del que debería ser.

Para evitar el fracaso del proyecto, un gestor de proyectos de software y los ingenieros de software que construyeron el producto deben eludir un conjunto de señales de peligro comunes; comprender los factores del éxito críticos que conducen a la gestión correcta del proyecto y desarrollar un enfoque de sentido común para planificar, supervisar y controlar el proyecto.

2.2.2 Opciones para la organización del personal

Un aspecto muy importante en todo proyecto es la comunicación e interacción entre las personas, la

experiencia de algunos autores como Humprey [Hu 7] muestran que el número de desarrolladores con quienes cada desarrollador debe interactuar con regularidad debe ser entre tres y siete. Los estudios formales acerca del efecto del tamaño del equipo sobre el desempeño son raros, pero en la siguiente figura se ilustran los extremos que llevan a las recomendaciones del tamaño del equipo. En un extremo, el desarrollador trabaja de manera individual sin interacción habitual con otros. Aunque no gaste tiempo en comunicación, ese aislamiento suele repercutir en malos entendidos respecto a lo que se espera del desarrollador, y conducir a un a un nivel relativamente bajo de efectividad. En el otro extremo, el desarrollador tiene que interactuar por rutina con tantos individuos que no queda tiempo para realizar el desarrollo en sí, y de nuevo el resultado es inefectividad. En particular, "comunicación habitual" implica hablar con alguien para algo, alrededor de dos horas a la semana. Si un ingeniero tiene la comunicación habitual con otros diez, entonces una mitad de su tiempo estará dedicada a la comunicación, lo que le deja solo media semana para su contribución individual. Los organizadores del proyecto deben tener esto en cuenta cuando planeen proyectos, ya sea de diez o de cien personas.

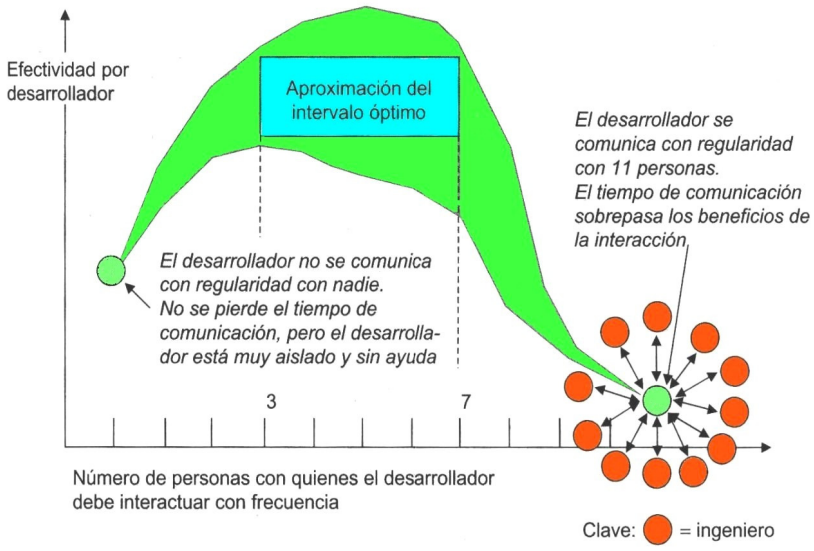


Fig. 19.- Tamaño óptimo aproximado para la interacción.

2.2.3 Opciones para la estructura de responsabilidades

La estructura jerárquica de la administración, como se muestra en la siguiente figura, es un extremo organizacional. Las ventajas de este esquema organizacional son que todos entienden las líneas de autoridad y decisión, y el número de personas con quienes cada uno debe interactuar con regularidad es aceptable. La desventaja es que los miembros del equipo tienden a participar menos en las decisiones porque es probable que las tareas se asignen desde arriba. Si todo lo demás es igual, ésta es

una manera segura de organizar un proyecto. Los proyectos más grandes organizados con este estilo jerárquico requieren organigramas más amplios.

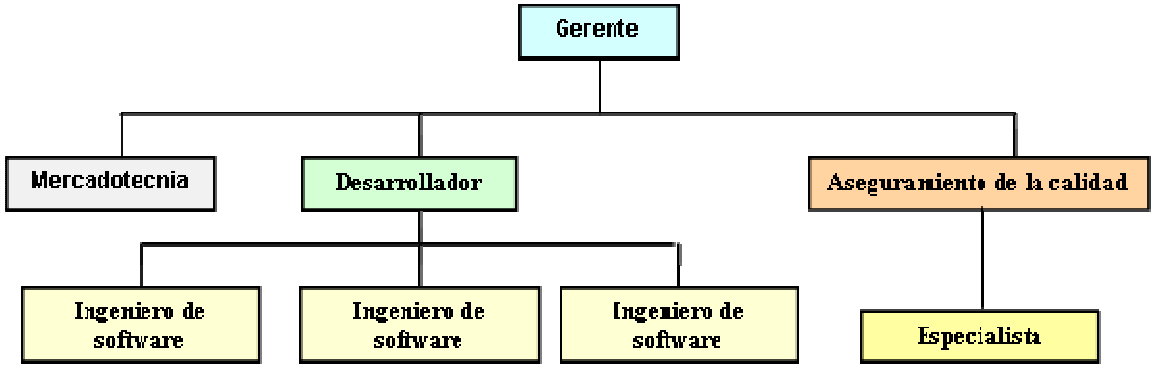


Fig. 20.- Organización jerárquica para la administración de proyectos.

En el otro extremo organizacional hay un equipo que consiste en una comunidad de colegas con la misma autoridad. La ventaja de esta organización es el potencial para la motivación que viene con la participación por igual en el proyecto. Esto funciona bien en especial si el grupo es pequeño, muy competente y está acostumbrado a trabajar en equipo. Las desventajas incluyen la dificultad para resolver las diferencias y el hecho de que "nadie está a cargo". El TSP de Humprey (vea [Hu 3]) es un conjunto específico de guías para este tipo de equipos. En última instancia, debe establecerse una mezcla de participación de colegas y responsabilidades de liderazgo adecuadas para el tamaño del proyecto, su naturaleza, su madurez y las personas involucradas.

Un punto medio es una estructura de organización horizontal, como se ilustra en la figura siguiente. La idea es que los miembros del equipo son iguales, excepto que uno de ellos es el líder designado. En la situación ideal, él debe estimular la participación de los miembros del equipo, pero debe tomar decisiones cuando se requiere.

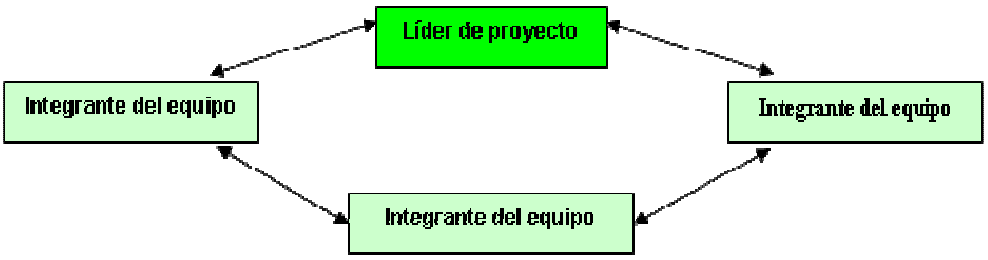


Fig. 21.- Organización horizontal para la administración de proyectos.

Conforme el número de participantes crece en un proyecto, la organización pura de los colegas se convierte en algo imposible debido a que el número de vínculos de comunicación (entre todos los pares) crece con el cuadrado del número de participantes.

Tres personas implican tres líneas de comunicación, cuatro personas implican seis, cinco personas 10, seis personas 15, n personas requieren (n-1) + (n-2)+…+1= n(n-1)/2. Entonces, 100 personas tendrán que participar en 4950 líneas de comunicación. Una alternativa para proyectos grandes es la organización que ilustra la siguiente figura, donde los grupos de colegas son pequeños y se designa un miembro de cada grupo como comunicador con los otros grupos. Este tipo de organización intenta preservar los beneficios de los equipos pequeños, pero cuenta con muchas personas para construir una aplicación extensa.

Es de conocimiento general que es raro el ingeniero con aptitudes para labores tanto de ingeniería como de administración. Sin embargo muchos ingenieros se han desempeñado bien dirigiendo grupos, incluso cuando muy pocos esperaban que pudieran.

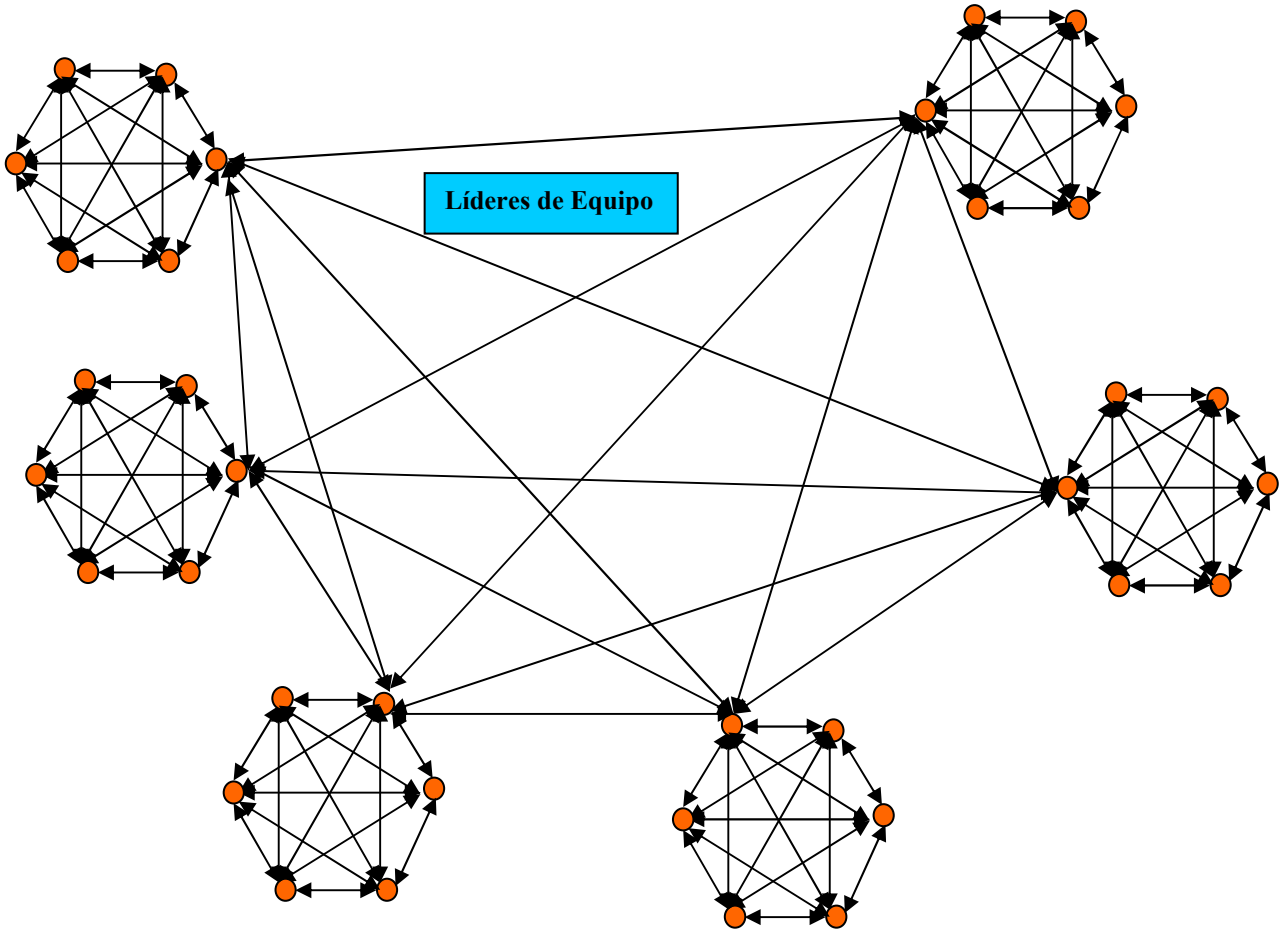


Fig. 22.- Organización de colegas para proyectos grandes.

2.2.4 Identificación y retiro del riesgo

Definición de "riesgos"

Un riesgo es algo que puede ocurrir en el curso de un proyecto que, según el peor resultado, lo afectaría de manera negativa y significativa. Por ejemplo, Rational Corporation asegura que "más de 70% de todos los proyectos de software tienen problemas o un deterioro severo". Los factores que a la larga ocasionan que un proyecto fracase aparecen como riesgos cuando se reconocen con prontitud, y al reconocerlos tal vez pueda prevenirse el fracaso mediante la acción adecuada. Existen dos tipos de riesgos.

- Riesgos que pueden evitarse o que se les puede sacar la vuelta (retirados).
- Riesgos que no pueden evitarse.

Un ejemplo del primer tipo es "¿qué pasa si el líder del proyecto en un equipo de 15 personas deja la compañía?" (se retira con la preparación de una persona de respaldo). Un ejemplo del segundo tipo es "2,100 datos de puntuales de vuelos deben recolectarse con el personal del aeropuerto antes de poder entregar el producto".

Si los riesgos del primer tipo se identifican con suficiente prontitud, su retiro convierte un proyecto fracasado en uno exitoso. También es de gran beneficio reconocer el riesgo del segundo tipo. Un proyecto puede detenerse antes de desperdiciar recursos (para aplicarlos de manera productiva en otro) o se puede cambiar el enfoque o agregar personal para minimizar el riesgo.

Los equipos efectivos adoptan una "mentalidad de riesgo" donde los riesgos se buscan todo el tiempo.

Las aplicaciones muy similares a trabajos realizados con anterioridad por los mismos ingenieros de software pueden no tener riesgos; sin embargo, el universo de las aplicaciones de software es vasto y crece con rapidez. Muchos trabajos consisten en nuevas formas de realizar las tareas, o son realizaciones de nuevas ideas. Por ello, es común que el desarrollo de aplicaciones de software incluya muchos riesgos.

Cada riesgo identificado debe ser bienvenido en el equipo del proyecto porque puede comenzar a prevenirlo. Los problemas reales son los riesgos que no se han identificado. Estos son como minas en el campo que esperan explotar. Como un gran porcentaje de proyectos nunca se termina (se llega a estimar en 80%), la atención constante a los riesgos aumenta la probabilidad de que un proyecto tenga 20% de éxito o hace que se cancele antes de desperdiciar una cantidad vergonzosa de dinero y dañe las carreras.

La "administración de riesgo" consiste en las actividades que se señalan a continuación:

- Identificar.

  Mentalidad: intentar identificar riesgos todo el tiempo.
- Planear el retiro.
- Dar prioridades.
- Retirar o atenuar

Estas actividades deben llevarse a cabo desde el principio del proyecto, y continuar de la manera disciplinada al menos durante la primera cuarta parte. Algunos equipos designan a un integrante el papel

de coordinador de riesgos, como responsable de impulsar a los miembros del equipo a detectar riesgos y supervisar su retiro.

Identificación de riesgos

La identificación de riesgos consiste en escribir todas las inquietudes o preocupaciones de quienes están relacionados con el proyecto, después presionar continuamente a los integrantes del equipo a pensar en más inquietudes. La identificación de riesgos requiere una mentalidad escéptica similar a la requerida para la inspección; una búsqueda global de defectos en el plan de desarrollo. Las categorías de riesgos incluyen subestimación del tamaño del trabajo, cambios demasiado rápidos en los requerimientos, falta de habilidad para encontrar una implantación con suficiente eficiencia, deficiencias en las aptitudes del personal, un lapso grande para aprender a usar las herramientas (como CASEtools), y deficiencias de los lenguajes (como una ejecución demasiado lenta).

Puede parecer extraño que los dos primeros riesgos tienen que ver con la falta de compromiso de los "dueños de intereses"; se supone que éstos son los más interesados en la aplicación. (Un "dueño de intereses" es cualquier persona que tiene algo que perder con el resultado del proyecto). Sin embargo, la comunidad de dueños es muy grande. Igual que muchos grupos, sus miembros tienen motivaciones que difieren y esto puede ser difícil de reconciliar. La falla en la reconciliación de motivaciones proporciona requerimientos inestables y destruye un proyecto. La siguiente lista enumera los factores de riesgo más comunes en los proyectos que Keil et al [Ke] identificaron al realizar estudios en Estados Unidos, Hong Kong y Finlandia.

- Falta de compromiso de la alta administración.
- Falta al obtener el compromiso del usuario.
- Error al entender los requerimientos.
- Participación inadecuada del usuario.
- Falla al manejar las expectativas del usuario final.
- Cambio de alcance y/o de objetivos.
- Falta de conocimiento o aptitudes requeridas del personal.

Observe que la mayoría de las categorías son de la misma naturaleza que las fuentes de riesgo número 1 y 2, excepto por el factor 7. Debe resaltarse que los aspectos técnicos constituyen sólo 20% de los diez factores principales y se consideran menos importantes que muchos otros. El resto de los factores son políticos y organizacionales. Estos resultados se pueden resumir diciendo que el líder del proyecto tiene la mayor responsabilidad de luchar por retirar las amenazas para el éxito de un proyecto.

Retiro de riesgo

Retirar el riesgo es el proceso mediante el cual los riesgos se reducen o incluso se anulan. Existen dos maneras de retirar un riesgo. Una es hacer cambios en los requerimientos del proyecto para retirar o "evitar" el aspecto que causa el riesgo. Otra manera es desarrollar técnicas y diseños que resuelvan el problema.

En un proyecto sano los riesgos se identifican en forma continua y no es raro tener riesgos en cola que esperan ser retirados. Deben darse prioridades en esta cola, porque con frecuencia no hay suficiente tiempo para retirar todos. En un proyecto bien administrado, los riesgos no retirados de antemano serán los menores, no serán problemas significativos cuando surjan en la secuencia normal de eventos. La siguiente tabla explica el esquema de prioridades. A cada riesgo se da: (1) nivel de impacto, (2) si es probable que sea real y (3) una evaluación del costo de retiro de ese riesgo. Cada una de estas medidas está en la misma escala (del 1 al 10). Los tres números u 11 menos el número, según el caso, se multiplican para obtener la prioridad del riesgo.

Manera de calcular las prioridades de los riesgos:

	Posibilidad 1-10	Impacto 1-10	Costo de retiro 1-10	Cálculo de prioridad	Prioridad resultante
	(1= menor posibilidad)	(1= menor impacto)	(1=menor costo de retiro)		(número menor se resuelve
La prioridad más alta	10 (muy posible)	10 (mayor impacto)	1 (menor costo de retiro)	(11-10)*(11-10)*1	1
La prioridad más baja	1 (poco posible)	1 (menor impacto)	10 (mayor costo de retiro)	(11-1)*(11-1)*10	1,000

Tabla 4.- Una manera de calcular las prioridades de los riesgos.

El uso de métricas como éstas para dar prioridades a los riesgos puede ser útil, pero siempre deben aumentarse con una sana dosis de sentido común. Por ejemplo, vale la pena observar por separado los riesgos que tienen una alta posibilidad de "detener el proyecto": suponiendo que un riesgo #1 tiene una prioridad más alta que el riesgo #2, pero retirar el riesgo #2 requiere más tiempo y por lo tanto, el trabajo para retirar el riesgo #2 debe iniciarse cuanto antes. Los equipos intentan obtener más de un punto de vista acerca de los riesgos. Si existen muchos riesgos serios, puede ser mejor retrasar el compromiso con el proyecto hasta que se hayan retirado.

2.2.5 Elecciones y decisiones

Elección de herramientas de desarrollo y soporte

La Ingeniería de Software es un mercado sustancial. Un gran número de proveedores venden herramientas y entornos para ayudar a los ingenieros a desarrollar aplicaciones de software. Estas con frecuencia reciben el nombre de herramientas de Ingeniería de Software asistida por computadora (CASE, Computer Aided Software Engineering). En ocasiones quienes respaldan las herramientas CASE prometen mucho y entregan menos. Los grandes proyectos, sencillamente no se pueden manejar sin al menos una de estas componentes CASE. Por ejemplo, en un proyecto grande, las herramientas de administración de configuración son indispensables.

Decisiones de construir o comprar

Existe un número creciente de herramientas y aplicaciones en el mercado que prometen ayudar en, o formar la base para, nuevas aplicaciones. Por ejemplo, al planear una aplicación de subasta por Internet, se puede comparar la compra de un marco de trabajo para subastas con el desarrollo de una aplicación propia. En general, estas decisiones se pueden retrasar hasta que se conocen los requerimientos, pero aquí se presentarán ya que son parte de la administración del proyecto.

Un enfoque común para este tipo de decisión es hacer una lista de gastos y estimar la magnitud de cada alternativa.

Selección del lenguaje

Debe identificarse el lenguaje o lenguajes para el desarrollo casi al principio del proyecto. En ocasiones esta decisión es directa, como cuando una organización impone un lenguaje es el único capaz de desarrollar los requerimientos. Sin embargo, algunas veces deben elegirse varias alternativas para el desarrollo.

Documentación

Durante la etapa de planeación, el equipo decide qué conjunto de documentación específico se producirá para el proyecto.

Servicio de apoyo

Los proyectos requieren apoyo para los administradores del sistema, de la red, de las bases de datos, los secretarios y otros. El administrador del proyecto debe asegurar que estas personas están disponibles. El TSP de Humphrey en realidad designa a un miembro del equipo como el "administrador de apoyo".

2.2.6 Planificación temporal y seguimiento del proyecto

La realidad de un proyecto técnico (tanto si implica la construcción de una planta hidroeléctrica o desarrollar un sistema operativo) es que hay que realizar cientos de pequeñas tareas antes de poder alcanzar el objetivo final. Algunas de estas tareas quedan fuera del camino principal y pueden complementarse sin preocuparse del impacto en la fecha de terminación del proyecto. Otras tareas se encuentran en el "camino crítico". Si estas tareas "críticas" se retrasan, la fecha de terminación del proyecto entero se pone en peligro.

El objetivo del gestor de proyecto es definir todas las tareas del proyecto, construir una red que describa sus interdependencias, identificar las tareas que son críticas dentro de la red y después hacerles un seguimiento para asegurarse de que el retraso se reconoce "de inmediato". Para conseguirlo, el gestor debe tener una planificación temporal que se haya definido con un grado de resolución que le permita supervisar el progreso y controlar el proyecto.

La planificación temporal de un proyecto de software es la actividad que distribuye el esfuerzo estimado a lo largo de la duración prevista del proyecto, asignando el esfuerzo a las tareas específicas de la Ingeniería de Software. Es importante resaltar, sin embargo que la planificación temporal evoluciona con el tiempo.

Durante las primeras etapas de la planificación del proyecto, se desarrolla una planificación temporal macroscópica. Este tipo de planificación temporal identifica las principales actividades de la Ingeniería de Software y las funciones del producto a las que se aplican. A medida que el proyecto va progresando, cada entrada en la planificación temporal macroscópica se refina en una planificación temporal detallada. Aquí, se identifican y programan las tareas del software específicas (requeridas para realizar una actividad).

La planificación temporal para proyectos de desarrollo de software puede verse desde dos perspectivas bastante diferentes. En la primera se ha establecido ya (irrevocablemente) una fecha final de entrega de un sistema basado en computadora. La organización del software está limitada a distribuir el esfuerzo dentro del marco de tiempo previsto. El segundo punto de vista de la planificación temporal asume que se han estudiado unos límites cronológicos aproximados pero que la fecha final será establecida por la organización de la Ingeniería del Software. El esfuerzo se distribuye para conseguir el mejor empleo de los recursos, y se define una fecha final después de un cuidadoso análisis del software. Desgraciadamente, la primera situación es más frecuente que la segunda.

Como todas las áreas de la Ingeniería de Software, la planificación temporal de proyectos de software se guía por unos principios básicos:

Compartimentación. El proyecto debe dividirse en un número de actividades y tareas manejables. Para llevar a cabo esta compartimentación, se descomponen tanto el producto como el proceso.

Interdependencia. Se deben determinar las interdependencias de cada actividad o tarea compartimentada. Algunas tareas deben ocurrir en una secuencia determinada; otras pueden darse en paralelo. Algunas actividades no pueden comenzar hasta que el resultado de otras no esté disponible. Otras actividades pueden ocurrir independientemente.

Asignación de tiempo. A cada tarea que se vaya a programar se le debe asignar cierto número de unidades de trabajo (por ejemplo, personas-día de esfuerzo). Además, a cada tarea se le debe asignar una fecha de inicio y otra de finalización que son función de las interdependencias y de que el trabajo se haga ya sea a tiempo total o tiempo parcial.

Validación de esfuerzo. Todos los proyectos tienen un número definido de miembros de la plantilla. A medida que se hace la asignación de tiempo, el gestor del proyecto debe asegurarse de que no se ha asignado un número de personas mayor que el de la plantilla en ese momento. Por ejemplo, considere un proyecto que tiene una plantilla asignada de tres miembros (3 personas-día están disponibles por día de esfuerzo asignado). Un día cualquiera, se deben realizar siete tareas concurrentemente. Cada tarea requiere .5 personas-día esfuerzo. Se ha asignado más esfuerzo del que pueden realizar las personas disponibles.

Responsabilidades definidas. Cada tarea que se programe debe asignarse a un miembro del equipo específico.

Resultados definidos. Cada tarea programada debería tener un resultado definido. Para los proyectos de software, el resultado es normalmente un producto (por ejemplo, el diseño de un módulo) o una parte de un producto. Los productos se combinan frecuentemente en entregas.

Hitos definidos. Todas las tareas o grupos de tareas deberían asociarse con un hito del proyecto. Se consigue un hito cuando se ha revisado la calidad de uno o más productos y se han aceptado.

Cada uno de los principios anteriores se aplica a medida que evoluciona la planificación temporal del proyecto.

2.2.7 Selección de las tareas de Ingeniería de Software

Para desarrollar una planificación temporal del proyecto, se debe distribuir un conjunto de tareas a lo largo de la duración del proyecto. El conjunto de tareas variará dependiendo del tipo de proyecto y del grado de rigor. Cada uno de los tipos de proyectos puede enfocarse usando un modelo de proceso lineal secuencial e iterativo (por ejemplo, el modelo incremental o de creación de prototipos) o evolutivo (por ejemplo, el modelo en espiral). En algunos casos, un tipo de proyecto fluye suavemente hacia el siguiente. Por ejemplo, los proyectos de desarrollo de concepto que tienen éxito evolucionan a menudo en nuevos proyectos de desarrollo de aplicación. Cuando termina un proyecto de desarrollo de una nueva aplicación, empieza a veces un proyecto de mejora de la aplicación. Esta progresión es natural y predecible y ocurrirá sea cual sea el modelo de proceso que adopte una organización. Por consiguiente, las principales tareas de Ingeniería de Software descritas en las secciones siguientes son aplicables a todos los flujos del modelo de proceso.

Definir una red de tareas

Las tareas y subtareas individuales tienen interdependencias basadas en su secuencia. Además cuando hay más de una persona implicada en un proyecto de Ingeniería de Software, es probable que las actividades de desarrollo y tareas se realicen en paralelo. Cuando ocurre esto, las tareas concurrentes deben coordinarse de manera que estén finalizadas cuando tareas posteriores requieran sus resultados. Una red de tareas, también llamada red de actividades, es una representación gráfica de flujo de tareas de un proyecto. Se emplea a veces como el mecanismo a través del cual se introduce la secuencia de tareas y las dependencias en una herramienta de programación automática de la planificación temporal de un proyecto. En su forma más sencilla (la que se emplea en una planificación temporal macroscópica), la red de tareas muestra las tareas principales de Ingeniería de Software. La siguiente figura muestra una red de tareas esquemática para un proyecto de desarrollo de concepto.

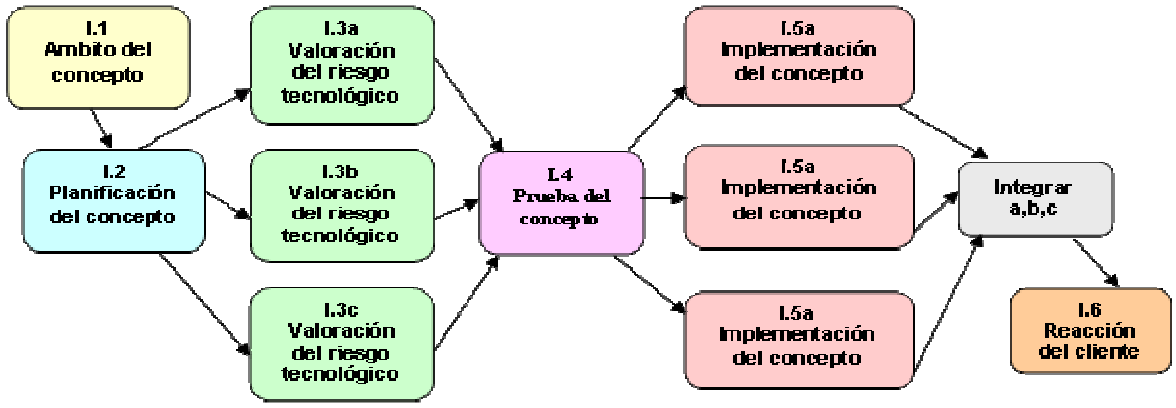


Fig. 23.- Una red de tareas para un proyecto de desarrollo de concepto.

La naturaleza concurrente de las actividades de Ingeniería del Software lleva a varios requisitos importantes de la planificación temporal. Como las tareas paralelas ocurren asíncronamente, el planificador debe determinar las dependencias entre las tareas para garantizar un progreso continuo hasta su finalización. Además, el gestor del proyecto debería estar al tanto de las tareas que pertenecen al camino crítico. Es decir, tareas que deben finalizarse según la planificación temporal si se quiere que el proyecto en general se termine a tiempo.

Es importante resaltar que la red de tareas mostrada en la figura anterior es macroscópica. En una red de tareas detallada cada actividad mostrada en la figura se expandiría.

Planificación Temporal

La planificación temporal de un proyecto de software no difiere mucho del de cualquier esfuerzo de ingeniería multitarea. Por tanto, se pueden aplicar herramientas de planificación temporal de proyectos y técnicas generales al software con una pequeña modificación en los proyectos de software.

La técnica de evaluación y revisión de programa (PERT) y el método del camino crítico (CPM) son dos métodos de la planificación temporal de un proyecto que pueden aplicarse al desarrollo de software. Ambas técnicas son dirigidas por la información ya desarrollada en actividades anteriores de la planificación del proyecto:

- Estimaciones de esfuerzo.
- Una descomposición de la función del producto.
- La selección del modelo de proceso adecuado y del conjunto de tareas.
- La descomposición de tareas.

Las interdependencias entre las tareas deben definirse empleando una red de tareas. Las tareas, a veces denominadas estructura de descomposición del trabajo del proyecto (en inglés, WBS), se definen para el producto como un todo o para las funciones individuales.

Tanto PERT como CPM proporcionan herramientas cuantitativas que permiten al planificador del software: (1) determinar el camino crítico, la cadena de tareas que determina la duración del proyecto; (2) establecer las dimensiones de tiempo "más probables" para las tareas individuales aplicando modelos estadísticos, y (3) calcular las limitaciones de período que definen una "ventana" de tiempo de una tarea determinada.

Los cálculos de las limitaciones de tiempo pueden ser muy útiles en la planificación temporal de proyectos de software. El retraso en el diseño de una función, por ejemplo, puede retardar el posterior diseño de otras funciones. Rigg [RIG 81] describe importantes limitaciones de tiempo que pueden discernirse de una red PERT o CPM: (1) lo antes posible que puede empezar una tarea cuando las tareas precedentes se completen también lo antes posible; (2) lo más tarde que puede empezar una tarea antes de que se retrase el tiempo mínimo para finalizar el proyecto; (3) la fecha más temprana de finalización –la suma de la fecha más temprana de inicio y la duración de la tarea-; (4) la fecha límite de finalización –la fecha más tardía de inicio sumada a la duración de la tarea-, y (5) el margen total –la cantidad de tiempo extra o atrasos permitidos en la planificación temporal de las tareas de manera que el camino crítico de la red se mantenga conforme a la planificación temporal-. Los cálculos de los tiempos límite llevan a la determinación del mínimo crítico y proporcionan al gestor un método cuantitativo para evaluar el progreso a medida que se completan las tareas.

Tanto PERT como CPM se han implementado en varias herramientas automáticas disponibles virtualmente en todos los ordenadores personales.

Tales herramientas son fáciles de usar y hacen asequibles los métodos de la planificación temporal descritos anteriormente a todos los gestores de proyectos de software.

2.2.8 Gráficos de tiempo

Cuando se crea una planificación temporal de un proyecto de software, el planificador empieza un conjunto de tareas (la estructura de descomposición del trabajo). Si se emplean herramientas automáticas, la descomposición del trabajo es introducida como una red de tareas o esquema de tareas. El esfuerzo, duración y fecha de inicio son las entradas de cada tarea. Además, se asignan las tareas a individuos específicos.

Como consecuencia de esta entrada, se genera un gráfico de tiempo, también denominado gráfico Gantt. Se puede desarrollar un gráfico de tiempo para todo el proyecto. Alternativamente, se pueden desarrollar diferentes gráficos para cada función del proyecto o para cada individuo que trabaje en el proyecto. La siguiente figura ilustra el formato de un gráfico de tiempo. Muestra una parte de la planificación temporal de un proyecto de software que enfatiza la tarea de ámbito del concepto para un nuevo producto de software de procesador de texto. Todas las tareas del proyecto (para ámbito del concepto) se listan en la columna de la izquierda. Las barras horizontales indican la duración de cada tarea. Cuando aparecen múltiples barras al mismo tiempo en la planificación temporal, implican concurrencia de tareas. Los rombos indican hitos.

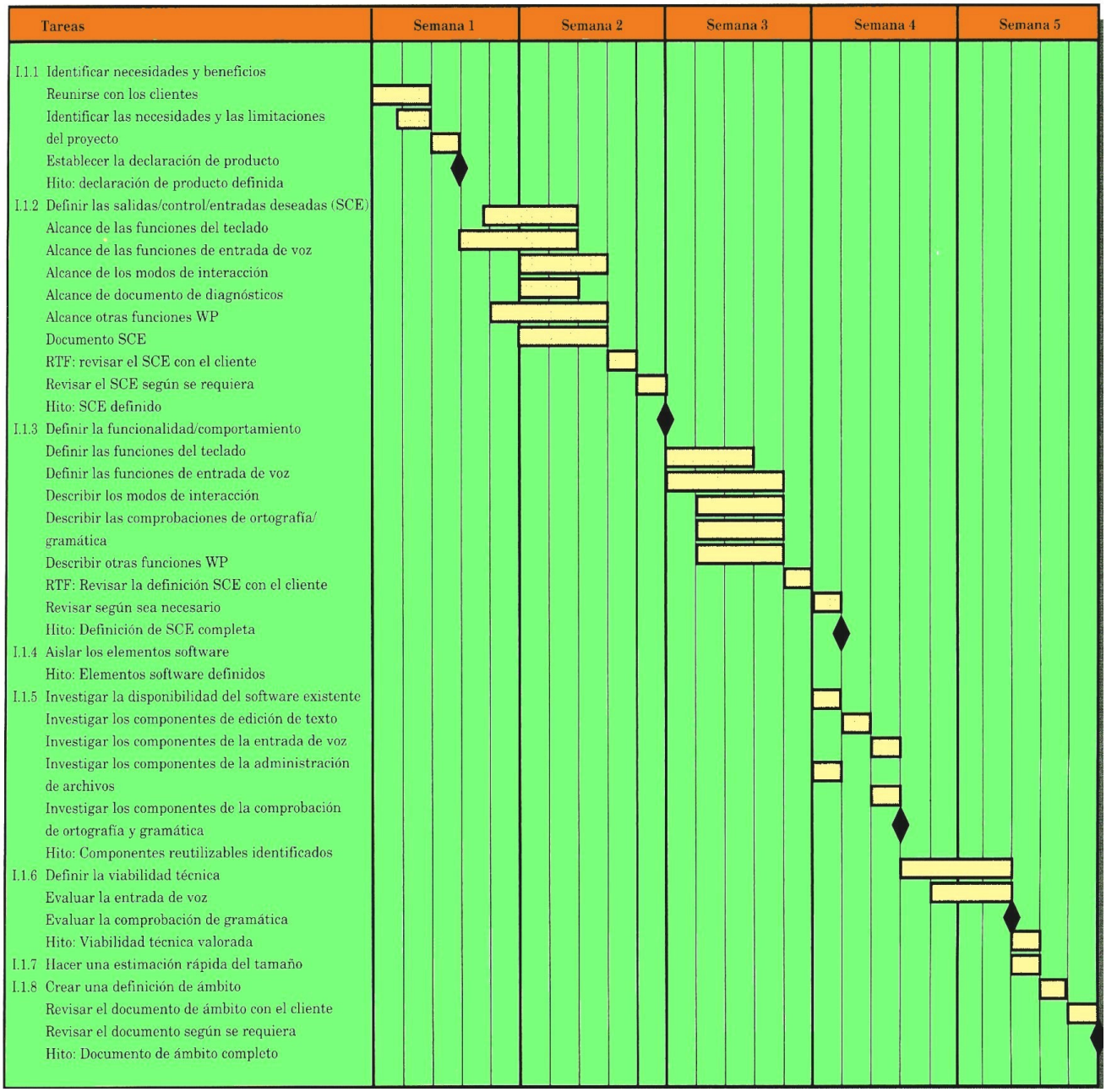


Fig. 24.- Un ejemplo de gráfico de tiempo.

Una vez que se ha introducido la información necesaria para generar el gráfico de tiempo, la mayoría de las herramientas de la planificación temporal de proyectos de software producen tablas de proyecto -un

listado tabular de todas las tareas del proyecto, sus fechas previstas y reales de inicio y finalización, e información varia relativa al tema-. Empleando las tablas junto con los gráficos de tiempo, le permiten al gestor del proyecto hacer un seguimiento del progreso.

2.2.9 Seguimiento de la planificación temporal

La planificación temporal del proyecto le proporciona al gestor un mapa de carreteras. Si se ha desarrollado apropiadamente, define las tareas e hitos que deben seguirse y controlarse a medida que progresa el proyecto. El seguimiento se puede hacer de diferentes maneras:

- Realizando reuniones periódicas del estado del proyecto en las que todos los miembros del equipo informan el progreso y de los problemas.
- Evaluando los resultados de todas las revisiones realizadas a lo largo del proceso de Ingeniería de Software.
- Determinando si se han conseguido los hitos formales del proyecto (los rombos mostrados en la figura anterior).
- Reuniéndose informalmente con los profesionales del software para obtener sus valoraciones subjetivas del progreso hasta la fecha y los problemas que se avecinan.

El control lo usa el gestor para administrar los recursos del proyecto, enfrentarse a los problemas y dirigir al personal del proyecto. Si las cosas van bien (es decir, el proyecto va según la planificación temporal y dentro del presupuesto, las revisiones indican que se está haciendo un progreso real y que se están alcanzando los hitos), el control es liviano. Pero cuando aparecen los problemas, el gestor debe ejercer el control para solucionarlos tan pronto como sea posible. Una vez que el problema se ha diagnosticado, se pueden concentrar recursos adicionales en el área del problema: se puede redistribuir la plantilla, o se puede redefinir la planificación temporal del proyecto.

Cuando se enfrenta a la presión de una fecha de entrega muy ajustada, los gestores de proyecto utilizan a veces una planificación temporal de proyecto y una técnica de control denominada time-boxing (tiempo encajonado) [ZAH 95]. Esta estrategia reconoce que quizás no se pueda entregar el producto completo para la fecha límite predefinida. Por tanto, se elige un paradigma incremental del software y se crea una planificación temporal para cada entrega de un incremento.

Las tareas asociadas con cada incremento se encajonan en el tiempo. Esto significa que la planificación temporal para cada tarea se ajusta trabajando hacia atrás desde la fecha de entrega para cada incremento. Se pone una "caja" alrededor de cada tarea. Cuando una tarea alcanza el límite de su caja de tiempo (más o menos un 10 por 100), se termina el trabajo y se empieza la siguiente tarea.

La primera reacción frente al enfoque de encajonamiento de tiempo es a menudo negativa: "Si no se ha terminado el trabajo, ¿cómo podemos proseguir?". La respuesta se encuentra en la manera en que se realiza el trabajo. Cuando se llega al límite de la caja de tiempo, es probable que se haya completado el 90 por 100 de la tarea. El restante 10 por 100, aunque importante, puede: (1) retrasarse hasta el siguiente incremento o (2) completarse más tarde si es necesario. En vez de estar "estancado" en una tarea, el proyecto progresa hacia la fecha de entrega.

2.2.10 Plan del Proyecto

Cada paso en el proceso de Ingeniería de Software debería obtener una entrega que pueda revisarse y que pueda hacer de fundamento para los siguientes pasos. El "Plan del Proyecto de Software" se produce a la culminación de las tareas de planificación. Proporciona información básica de costos y planificación temporal que será empleada a lo largo del proceso de software.

El Plan de Proyecto de Software es un documento relativamente breve dirigido a una audiencia diversa. Debe: (1) comunicar el ámbito y recursos a los gestores del software, personal técnico y al cliente; (2) definir los riesgos y sugerir técnicas de aversión al riesgo; (3) definir los costos y planificación temporal para la revisión de la gestión; (4) proporcionar un enfoque general del desarrollo del software para todo el personal relacionado con el proyecto y (5) describir cómo se garantizará la calidad y se gestionan los cambios.

Es importante señalar que el Plan de Proyecto de Software no es un documento estático. Esto es, el equipo del proyecto consulta el plan repetidamente –actualizando riesgos, estimaciones, planificaciones e información relacionada- a la vez que el proyecto avanza y es más conocido.

2.2.11 Estimación de Costos: Cálculos Preliminares

Introducción

El costo de un proyecto es de interés continuo y vital para los copartícipes. Con el precio de producción equivocado, incluso el producto más fantástico puede ser un desastre. La estimación de costo más sencilla es la que proporciona un costo fijo desde el principio, sin permitir desviaciones en ninguna circunstancia. Aunque las organizaciones muy competentes tienen suficientes aptitudes para cambiar las variables restantes (capacidad, programa de tiempos y calidad) para cumplir con un costo predeterminado, la rigidez absoluta en el costo no siempre se adopta. Suponga, por ejemplo, que un proyecto que desarrolla un producto que se vende muy bien se queda sin fondos cuando lleva el 90%. En lugar de abandonar todo el proyecto, lo común es que la organización haga todo lo posible para encontrar fondos para ese último 10%. Aún cuando el costo del proyecto sea rígido, es necesario estimar el costo de un conjunto dado de requerimientos y/o del diseño para asegurar que cumple con el costo acordado y, si no lo hace, cambiarlos y después hacer la estimación de nuevo.

El proceso de estimar los costos (esto es, para capacidades, control de calidad y programación fijas) con frecuencia comienza desde la concepción del proyecto y continúa aún después de iniciada la codificación. Cuando se inicia un proyecto, tal vez el equipo tenga sólo una idea vaga de su costo. Si la estimación del costo se pude posponer hasta que el proyecto tome forma, sin duda deben esperar, pero siempre existe necesidad de estimar por lo menos un "intervalo burdo" a partir de un resumen de requerimientos. Cuanto más se sepa de los requerimientos para el producto y más diseño se realice, más preciso será el costo. Lo anterior se ilustra en al siguiente figura.

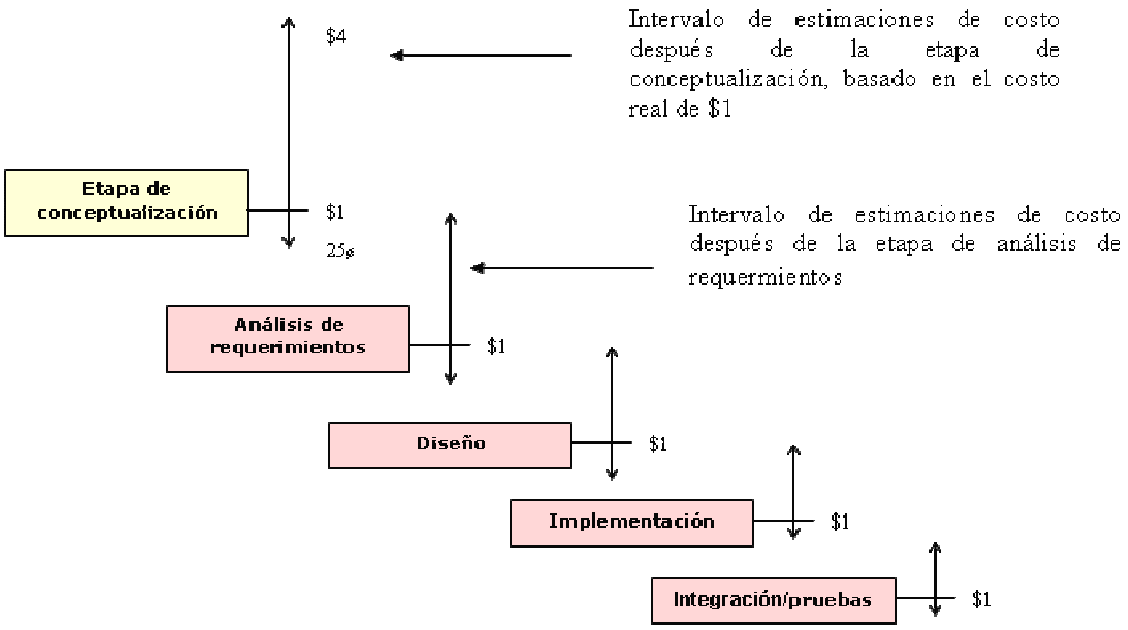


Fig. 25.- Intervalo de errores en la estimación del costo.

El error de estimación tan grande que indica la figura anterior se debe a un estudio reportado por Boehm [Bo]. Por ejemplo, para una aplicación que con el tiempo costará $100 000, las estimaciones hechas después de desarrollar el concepto de la aplicación pueden ser tan bajas como $ 25,000 y tan altas como $400 000. Para perfeccionar la estimación del costo de un proyecto se usan varias técnicas lo antes posible, lo que significa reducir la altura de las líneas verticales de la figura. Sólo hasta el final del desarrollo se puede tener una confianza completa en las estimaciones. (Sin embargo, las estimaciones son menos útiles en ese momento, ¡ya que la mayor parte del dinero está gastado!). Como la precisión es prácticamente imposible, un intervalo es una buena manera de proyectar los costos y esto se aplica al ejemplo anterior.

Sorprende a muchas personas que podamos siquiera pensar en los costos sin un diseño y los requerimientos detallados, pero ésta es una práctica común en otros campos. Se puede obtener una estimación burda del costo de construir una casa, por ejemplo, sin un diseño o los requerimientos detallados. Se pueden usar reglas cortas como "las casas en esta área cuestan alrededor de $100 por pie cuadrado de construcción", y así una casa de 100 pies cuadrados costará alrededor de $100 000.

Una buena manera de enfocar la estimación de costos del proyecto durante las primeras etapas es desarrollar estimaciones de varias maneras independientes y después combinar los resultados. Incluso se pueden ponderar las estimaciones obtenidas de acuerdo con el nivel de confianza personal en cada una de ellas.

Una máquina de coser o un torno es una herramienta compleja que no sirve sin un usuario capacitado. De

igual manera, la primera vez que se usan las medidas de aproximación de costos es poco probable que los resultados sean confiables. El uso preciso de estas herramientas se aprende con el tiempo, la retroalimentación y la comprobación.

Estimación de líneas de código sin el proceso de puntos de función

En esta se sección presenta la estimación de líneas de código en la etapa inicial, mucho antes de iniciar el trabajo de diseño. Una vez realizado éste, los métodos se basan en las partes del diseño y se vuelven mucho más precisos, como se indica en la figura anterior.

Varios métodos de estimación, como el modelo COCOMO, dependen del número de líneas de código (LoC). "COCOMO" son las siglas abreviadas de modelo de costos constructivo en inglés (Constructive Cost Model) de Boehm [Bo]. En las primeras etapas de un proyecto es posible que COCOMO no parezca muy útil debido a que falta mucho para llegar a la codificación. Sin embargo, cuando un producto se puede comparar con otros, es factible estimar las líneas. Por ejemplo, se podría estimar que el trabajo actual de control por satélite se puede comparar con el último trabajo, que requirió 3 millones de líneas de código de FORTRAN; no obstante, el trabajo actual tiene el requerimiento adicional de poder detectar y supervisar huracanes. Es posible hacer una estimación burda del tamaño de esta parte adicional con base en otros rastreadores de huracanes (100 000 líneas de FORTAN, por ejemplo). Cuando cambia el lenguaje de la aplicación se usan los factores de conversión estándar de la industria.

Las organizaciones que trabajan arriba del nivel 1 en los modelos de madurez de capacidades deben poder registrar las horas-persona y la duración de las partes de los trabajos. En ausencia de este tipo de datos, se tendrá que comparar la aplicación a construir con aplicaciones del mismo tipo, por ejemplo en la construcción de un videojuego se tendrá que comparar con otros videojuegos de la industria o incluso con proyectos relacionados, por ejemplo, un proyecto de simulaciones.

La página www.construx.com proporciona herramientas de estimación gratis.

El Personal Software Process incluye una recolección intensiva de medidas personales. Esta práctica esencial proporciona a los individuos y organizaciones datos históricos que pueden usar en las estimaciones de líneas de código.

Puntos de función y líneas de código

A partir de 1979 con Albrecht [Al] se desarrolló el concepto fundamental de puntos de función para evaluar el tamaño de un proyecto sin tener que conocer su diseño. La técnica de puntos de función es un medio para calibrar las capacidades de una aplicación de manera uniforme, como un solo número. Este número se puede usar para estimar las líneas de código, los costos y la duración. Los puntos de función es un concepto atractivo, ya que intenta llegar al corazón de la capacidad de un producto futuro; sin embargo, requiere mucha práctica aplicarla con exactitud y consistencia.

El cálculo de los puntos de función comprende los siguientes pasos:

Puntos de Función, paso 1. Identifique las funciones (como "recuperar", "desplegar") que debe tener la aplicación. El grupo Internacional Function Point Users Group (IFPUG; vea [IF]) ha publicado criterios de lo que constituye una "función" de una aplicación en este sentido. Consideran la funcionalidad a nivel del

usuario, en lugar de respecto a la programación en algún lenguaje. En general, una función es el equivalente a procesar una pantalla o forma en el monitor.

Puntos de Función, paso 2. Para cada función calcule su contribución de puntos de función a partir de las fuentes que muestra la siguiente figura. Lo siguiente resume el sentido de cada factor que contribuye. Las guías deben seguirse con cuidado, de otra manera es difícil obtener estimaciones consistentes.

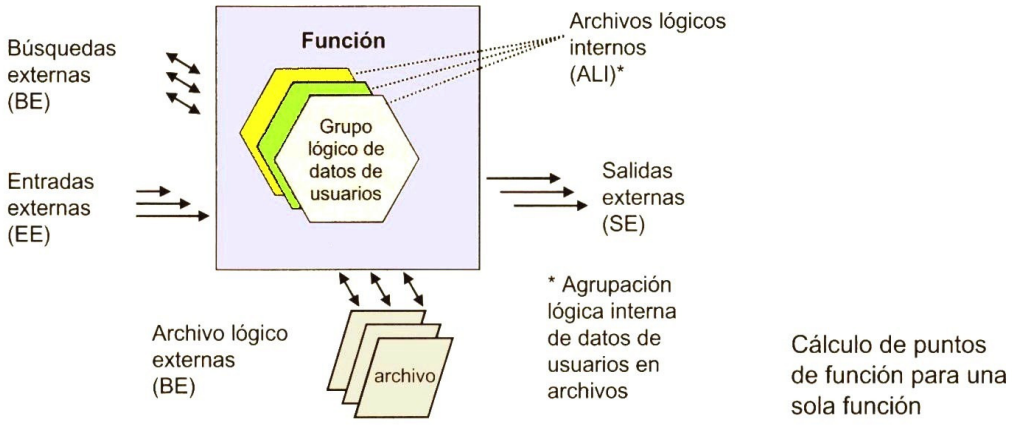


Fig. 26.- Cálculo de puntos de .función para una sola función.

- Entradas externas: sólo se cuentan separadas las entradas que afectan la función en forma diferente a las otras. Entonces, una función de una aplicación que resta dos números tendrá EE=1 y no EE=2. Por otro lado, si el carácter A puede introducirse para solicitar una suma y S para una resta, esto contribuiría 2 a EE.
- Salidas externas: deben contarse sólo salidas responsables de algoritmos verdaderas o funcionalidades no triviales. Por ejemplo, un proceso que tiene como salida un carácter en varias fuentes debe contarse como 1; los mensajes de error no cuentan. Las representaciones gráficas de las gráficas cuentan como 2 (1 por los datos y l por el formato) y los datos enviados a destinos separados no triviales (como impresora o monitor) se cuentan por separado.
- Búsqueda interna: cada búsqueda independiente cuenta como 1.
- Archivos lógicos internos: cuenta cada grupo lógico único de datos del usuario creados por o mantenidos por la aplicación. Las combinaciones de estos grupos lógicos no cuentan; cada área funcional de la aplicación que maneja un grupo lógico único aumenta la cuenta en 1.
- Archivos lógicos externos: cuenta cada grupo único de datos en archivos externos a la aplicación.

Puntos de función, paso 3. Como se muestra en la siguiente figura, cada uno de estos valores de los parámetros se factorizan por un número, según el grado de complejidad del parámetro en la aplicación. El IFPUG [IF] publicó descripciones detalladas del significado de "simple" y "complejo" en este contexto.

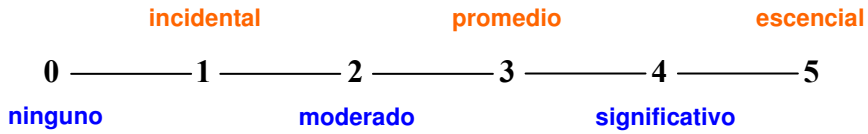


Fig. 27.- Valores de factorización.

Puntos de función, paso 4. Después se calculan ponderaciones para las 14 características generales del proyecto, cada una entre 0 y 5. Como se muestra en la siguiente figura. Una vez más, se necesita una experiencia consistente para evaluar los valores adecuados de estas variables.

1. ¿Requiere respaldo/recuperación?
2. ¿Requiere comunicación de datos?
3. ¿Tiene distribución de funciones de procesamiento?
4. ¿El desempeño es crítico?
5. ¿Corre en entorno existente con uso pesado?
6. ¿Requiere entrada de datos en línea?
7. ¿Tiene ventanas de entrada múltiples?
8. ¿Campos maestros actualizados en línea?
9. ¿Son complejas entradas, salidas, búsquedas de archivos?
10. ¿El procesamiento interno es complejo?
11. ¿Se diseñó el código para reuso?
12. ¿Incluye conversión e instalación?
13. ¿Se hacen instalaciones múltiples en diferentes organizaciones?
14. ¿Debe simplificarse el cambio y facilidad de uso para el usuario?

Fig. 28.- Características generales del proyecto a ponderar.

Puntos de función, paso 5. Por último, los puntos de función ajustados totales se calculan con la fórmula:

Puntos de función (ajustados) = [puntos de función no ajustados] * [0.65 + 0.01 * (características generales totales)]

Esta ecuación establece que si no hay demandas especiales sobre la aplicación (características generales totales = 0), entonces la medida de puntos de función debe disminuir de la calificación (directa) no ajustada en 35% (lo que explica el 0.65). De otra manera, la medida debe aumentar de la cantidad no ajustada en un punto porcentual por cada unidad de característica general.

Conversión de puntos de función en líneas de código

Una vez obtenidos con precisión, los puntos de función son muy útiles. Por ejemplo, se pueden aprovechar como medidas comparativas que permitirán a la organización estimar sus trabajos con base en las medidas de puntos de función de trabajos anteriores. Se pueden convertir en líneas de código usando tablas estándar. Después, las líneas de código se pueden utilizar para estimar el esfuerzo total en personas-mes, lo mismo que la duración.

Las hojas de cálculos para los puntos de función libres están disponibles en Internet en:

www.ifpug.org/home/docs/freebies.html

Estimación del Esfuerzo y la duración a partir de las líneas de código

Una vez estimadas las líneas de código (ya sea por el método de puntos de función o algún otro) se pueden usar para estimar los requerimientos de mano de obra y duración del proyecto. Barry Boehm [Bo] observó que, en términos generales, la mano de obra requerida para desarrollar las aplicaciones aumenta más rápido que el tamaño de la aplicación. La función experimental, con exponente cercano a 1.12, se usa para expresar esta relación. El modelo de Boehm también dice que la duración aumenta de manera exponencial con el esfuerzo, pero con un exponente menor que 1 (el exponente usado en ese caso es cercano a .35). Esto refleja la observación de que después de cierto tamaño (el "doblez" de la curva (2)), el esfuerzo adicional requerido tiene solo un efecto de aumento gradual con el tiempo que lleva completar el proyecto. Eso se ilustra en la siguiente figura donde LOC significa "líneas de código".

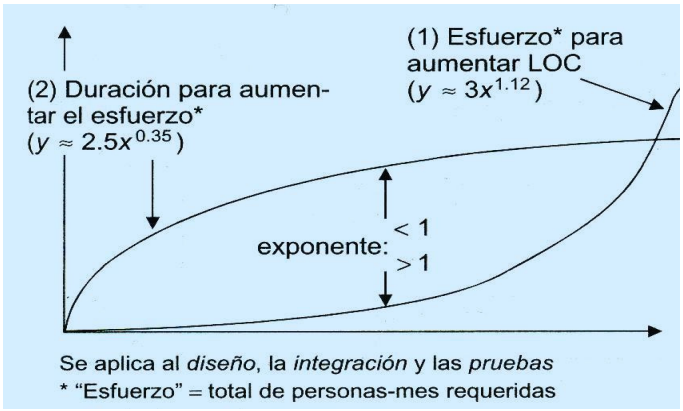


Fig. 29.- Significado de las fórmulas de COCOMO.

Con datos de numerosos proyectos, Boehm estimó los parámetros para esas relaciones, suponiendo una relación exponencial. Sus fórmulas se ilustran en la siguiente figura. Las aplicaciones orgánicas son aplicaciones independientes, como los procesadores de palabras clásicos (no activados en Internet). Las aplicaciones inmersas están integradas a sistemas de software-hardware (como un sistema de protección no vulnerable). Las aplicaciones semiaisladas son intermedias. Por ejemplo, un juego activado en la red es semiaislado, ya que no es orgánico, pero no tiene una inmersión fuerte como el código en un sistema de protección. El juego se comunicaría con el Internet por medio de señales que sólo son ocasionales, si se comparan con la frecuencia de ejecución de instrucciones del CPU.

$$\text{Esfuerzo en personas-mes} = a \times \text{KLOC}^b$$
$$\text{Duración} = c \times \text{Esfuerzo}^d$$

Proyecto de software	*a*	*b*	*c*	*d*
Orgánico	2.4	1.05	2.5	0.38
Semiaislado	3.0	1.12	2.5	0.35
Inmerso	3.6	1.20	2.5	0.32

Fig. 30.- Fórmulas básicas COCOMO.

El modelo de Boehm dice que el esfuerzo requerido y la duración tienen modelos separados (fórmulas) para cada tipo de aplicación (que difieren en los factores a y b). Por ejemplo, un trabajo individual con 20 000 líneas de código tomaría $2.4 * 20^{1.05} \approx 51$ personas–mes de duración si es orgánico (independiente), pero alrededor de $3.6 * 20^{1.2} \approx 76$ personas–mes si está inmerso.

La fórmula para la duración se puede expresar directamente en términos de miles de líneas de código (KLOC) como sigue:

Duración = $c * \text{esfuerzo}^d = c * (a * \text{KLOC}^b)^d = c * a^d * \text{KLOC}^{bd}$

A primera vista, la fórmula para la duración de Boehm puede parecer rara porque la relación entre el esfuerzo y la duración parece mucho más sencilla. Por ejemplo, si se sabe que un trabajo requiere 120 personas–mes y se le asignan 10 personas, ¿no quedaría terminado en 12 meses? Sin duda así sería si las 10 personas se pudieran emplear de manera útil y consistente desde el día 1 hasta el día 365, pero esto no suele ser posible. Piense, por ejemplo, en el día uno: todavía no se sabe nada del proyecto, entonces, ¿Qué actividades útiles pueden realizar los ingenieros ese día? Así, se sabe que si se asignan 10 ingenieros desde el primer día, entonces 120 personas-mes en realidad necesitarán más de 12 meses.

La fórmula de duración de Boehm tiene la rara propiedad de ser independiente del número de personas que se asignan al trabajo. Depende sólo del tamaño del trabajo. En realidad, la fórmula supone que el proyecto tendrá un número aproximado adecuado de personas disponibles asignadas en cualquier tiempo (por ejemplo, uno en el día uno, 30 en el día 100, si eso es lo que se necesita).

El modelo de Boehm, que está bien probado y tiene amplia aceptación, se ha mejorado con el tiempo. Sin embargo, se requiere mucha práctica para usarlo de manera efectiva, y es mejor cuando se usa con un método independiente y mucho sentido común (que suele llamarse "verificación de salud").

2.2.12 Proceso de software en equipo

No es deseable tener una preocupación continua por aspectos comunes a todos los proyectos de desarrollo de software, que pueden resolverse antes de iniciar. El proceso de software en equipo (TSP) estipula un gran número de aspectos y prácticas de administración de proyectos para todas las actividades en equipo. El TSP es una guía para los grupos, en cada una de las etapas de desarrollo del proyecto después de realizar el análisis de requerimientos. Se pide que los participantes en TSP tengan capacitación en el desarrollo personal de software (PSP). El método se organiza alrededor de las iteraciones de la secuencia en cascada y requiere que el equipo "ejecute" cada iteración en una reunión donde se estudian varios aspectos determinados con anterioridad. Humphrey proporciona numerosos escritos detallados. Las etapas pueden realizarse varias veces, lo que requiere varias corridas. Los aspectos que deben establecerse con estas corridas se listan a continuación:

- Proceso a usar.
- Metas de calidad.
- Manera de dar seguimiento a metas de calidad.
- Cómo toma decisiones el equipo.
- Qué hacer si no se logran las metas de calidad.
    - Posiciones de respaldo.
- Qué hacer si no se aprueba el plan.
    - Posiciones de respaldo.
- Definir papeles en el equipo.
- Asignar papeles en el equipo.

Humprey recomienda que los aspectos enumerados en el siguiente listado, se produzcan en cada etapa.

- Metas escritas del equipo.
- Papeles definidos en el equipo.
- Plan de desarrollo del proceso.
- Plan de calidad.
- Plan de apoyo del proyecto.
    - Computadoras, software, personal, etc.
- Plan y programa de desarrollo global.

- Planes detallados para cada ingeniero.
- Evaluación de riesgo para el proyecto.
- Informe del estado del proyecto.

2.2.13 Plan de administración del proyecto de software

El plan del proyecto se documenta de modo que todos sepan qué hacer y cuándo hacerlo. Existen muchos formatos para el plan: uno de ellos es el estándar de IEEE, 1058.1-1987 (confirmado en 1993). La tabla de contenido para 1058.1-1987, el plan de administración del proyecto de software (PAPS), se muestra en la figura siguiente.

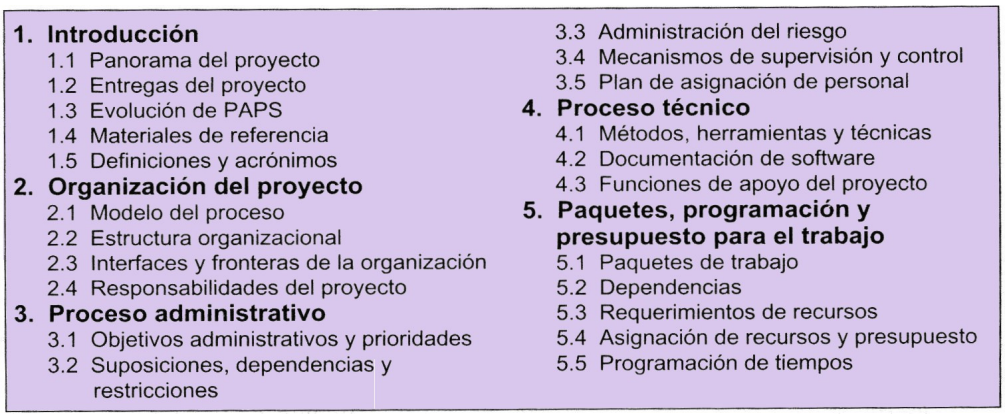


Fig. 31.- Contenido de IEEE 1058. 1-1987 PAPS.

La sección 1.1 de la figura anterior, el panorama del proyecto, debe identificar el proyecto pero no debe intentar cubrir sus requerimientos (es decir, las descripciones de su comportamiento). Estas se cubren en la especificación de requerimientos de software descrita más adelante. No es necesario repetir el material en el PAPS y violaría el objetivo de documentación de una sola fuente. La sección "entregas" (1.2) enumera todos los documentos, el código fuente y el código objeto que deben producirse. La sección 1.3 describe cómo se espera que el PAPS se desarrolle y cambie. En este punto debe haberse desarrollado el plan de administración de configuración de software (PACS) para tener un control adecuado de las versiones del PAPS.

En la sección 2.1 se refiere al proceso que se usará (cascada, espiral, por incrementos). En la sección 2.2 se estudian las "estructuras organizacionales" posibles. La sección 2.3, "interfaces y fronteras de la organización", describe las formas de comunicación entre las organizaciones. Esto depende de los interesados (las personas que tienen intereses en el proyecto). Por ejemplo ¿cómo interactúan la ingeniería y la comercialización?, ¿realizan juntas periódicas?, ¿usan correo electrónico? La sección 2.4 establece quién es responsable y de qué.

"Objetivos y prioridades de la administración", se puede establecer la filosofía de operación del proyecto. No todos los proyectos tienen la misma prioridad. Por ejemplo, en un videojuego, la primera prioridad quizá sea la "creación de un entorno verdaderamente fascinante para el jugador". Después de todo, si nadie compra el juego, el resto de los aspectos es irrelevante. Por otro lado, para las aplicaciones médicas críticas, la "seguridad" es la prioridad fundamental. Con otras aplicaciones, la posibilidad de reuso de las partes pude ser una prioridad de la administración.

La administración del riesgo (sección 3.3 de la figura anterior) se describió en la sección "identificación y retiro de riesgo". Los "mecanismos de supervisión y control" (3.4) especifican quién administrará, controlará y revisará el proyecto, lo mismo que cómo y cuándo lo hará. Por ejemplo, la alta administración necesita conocer el avance de los proyectos, por lo que se describe el proceso para mantenerlos informados. El "plan de asignación de personal" establece justo quién estará en cada puesto. Así, por ejemplo, la sección 2 puede establecer que el proyecto tendrá un administrador con responsabilidades dadas, mientras que la sección 3.5 establecerá quién ocupará el puesto.

El "proceso técnico" en la sección 4 proporciona las restricciones sobre los lenguajes y las herramientas (por ejemplo, "este proyecto debe usar Java de Sun, versión 1.2.1 y Rational Rose versión 1"). La sección 4 puede incluir información de los requerimientos de reuso y de técnicas como patrones de diseño. Las "funciones de apoyo al proyecto" (sección 4.3) hacen referencia o describen la actividades que apoyan el desarrollo del proceso, como la administración de la configuración y el aseguramiento de la calidad. Si la función de apoyo se describe en un documento separado (como el PACS o el de aseguramiento de calidad), se hace referencia a esos documentos. De otra manera debe describirse la función de apoyo en su totalidad.

La sección 5.3 "requerimientos de recursos", estima la mano de obra, el hardware y software requeridos para construir y mantener la aplicación. Los resultados de las técnicas de estimación de costos se pueden establecer aquí. Conforme avanza el proyecto se regresa a esta sección para incluir estimaciones más detalladas y precisas.

La sección 5.4 "asignación de recursos y presupuesto", describe cómo asignar los recursos (más que nada dinero) a las partes del proyecto durante su vida. Esto consiste en los gastos persona-día, pero también incluye software y hardware.

Por último, la sección 5.5 concluye el PAPS con un calendario que establece cómo y cuándo deben realizarse las partes del proceso.

2.2.14 Calidad en la administración del proyecto

Todos desean participar en los proyectos "buenos", pero no será posible a menos que se pueda definir qué significa "buenos". Para esto es necesario definir las métricas, hacer las mediciones de nuestros proyectos y compararlas con las métricas definidas, y después mejorarlos hasta que se conviertan en "buenos".

Métricas del proceso

Introducción a las métricas del proceso. Recuerde que el nivel 5 de CMM requiere mejora continua en el proceso mismo. Aunque pocas organizaciones trabajan en este nivel, sus metas deben tenerse en

mente. A fin de mejorar el proceso de administración de un proyecto debemos prepararnos para medir su efectividad mediante las métricas del proceso. Estas métricas, que pueden medir la efectividad de la organización del proceso, incluyen la secuencia de los pasos. También se mide por separado la efectividad del análisis, diseño, codificación y pruebas de los requerimientos.

Ejemplos de métricas del proceso. Una métrica común es la tasa de detección de defectos para una etapa de detección y una etapa de inyección dadas. Por ejemplo, se obtendría una "tasa de detección de defectos de 0.2 por cada 100 requerimientos en la etapa de implementación" si se detectara un defecto en los requerimientos en el momento de la implantación, como parte del implementar 500 requerimientos.

Cuando las tasas de detección de defectos se comparan con las normas de la organización, se mide el proceso y no sólo el proyecto. La siguiente tabla muestra un proyecto en el que se han recolectado estos datos de defectos. Para mantener la sencillez se omitieron las etapas de prueba y después de la entrega, que completarían el panorama.

	Etapa en la que se detectan los defectos (este proyecto/norma)		
Etapa que contiene defectos	**Requerimientos detallados**	**Diseño**	**Implantación**
Requerimientos detallados	2/5		
Diseño	0.5/1.5	3/1	
Implantación	0.1/0.3	1/3	2/2

Tabla 5.- Ejemplo de recolección de defectos por etapa.

La atención se centrará en la parte de requerimientos detallados en la tabla anterior. De las inspecciones se observa que se detectaron dos defectos por cada 100 durante la etapa de requerimientos. Esto se compara de modo favorable con la norma de la organización de cinco por cada 100. Observe en el renglón de "requerimientos detallados" que nuestro proceso detectó menos defectos de requerimientos que la tasa normal durante las etapas subsecuentes. Esto dice que nuestro proyecto, y tal vez el proceso que se usa, es comparable en efectividad respecto a la producción de requerimientos de calidad.

Los resultados para los defectos de diseño son los siguientes. Se detectó más del número usual de defectos de diseño durante las inspecciones cuando se produjeron, pero se encontraron menos defectos de diseño en una etapa posterior del proceso, lo cual indica que el proyecto y tal vez el proceso son superiores a las normas de la organización.

Para completar la tabla, se incluyen datos de defectos similares recolectados durante las pruebas, en un tiempo específico (como tres meses) después de entregar el producto.

Las métricas de la siguiente figura, resumen las métricas apropiadas del proceso e incluyen las métricas de defectos descritas. Note que sólo los números comparados con las normas de la compañía o la industria constituyen las métricas del proceso: los números en sí no son suficientes para evaluar el proceso usado.

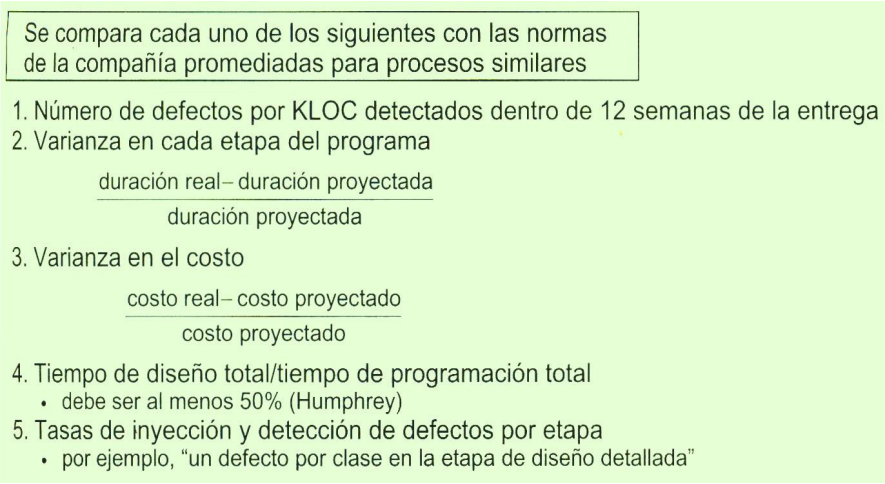


Fig. 32.- Cinco ejemplos de métricas de proceso.

Por ejemplo, si nuestro proyecto tiene un defecto por KLOC después de la entrega, detectado dentro de los primeros seis meses, y la norma de la compañía es de 1.3 defectos (por KLOC para el mismo periodo), entonces nuestro proceso bien puede ser una mejora. Para establecer esto será necesario ejecutar varios proyectos con un proceso dado y comparar los datos promedio.

Otras métricas de procesos se encuentran en [IEEE 982]. La siguiente sección estudia cómo usar los datos de métricas a fin de mejorar el proceso.

IEEE 739-1989 PAPS: parte 2

Las consideraciones de calidad para un proyecto se pueden detallar en un documento de calidad como el PAQS. La segunda mitad del PAQS tiene la apariencia de la figura siguiente. En ocasiones es preferible que las partes de este documento hagan referencia a otros documentos. Si se intenta duplicarlos completos o en partes se viola la regla de documentación de "una sola fuente".

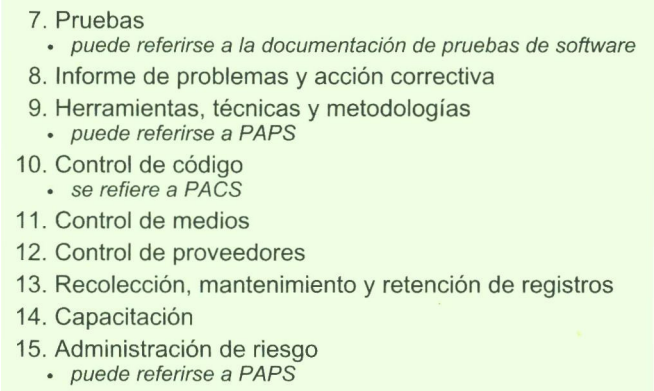


Fig. 33.- IEEE739, contenido de "1989 Software Quality Assurance".

La lista siguiente contiene una secuencia de acciones que se pueden tomar durante la vida del proyecto con el fin de mejorar el proceso de manera continua.

Una manera de reunir métricas de proceso:

1. Identifique y defina las métricas que usará el equipo en cada etapa; incluidas:
   - Tiempo gastado en: (1) investigación, (2) ejecución y (3) revisión.
   - Tamaño (como líneas de código).
   - Número de defectos detectados por unidad (líneas de código, incluye fuente).
   - Autoevaluación de la calidad en una escala de 1 a 10 (debe mantenerse la distribución de campana).
2. Documente esto en el PAQS.
3. Acumule los datos históricos por etapas.
4. Decida dónde se colocarán los datos de mediciones.
   - Conforme avanza el proyecto.
   - ¿En PAQS, PAPS o como apéndice?
5. Designe ingenieros para administrar la recolección por etapa.
   - Líder de QA o líderes de etapas (como líder de diseño).
6. Programe revisiones de los datos para aprender de la experiencia.
   - Especifique cuándo y cómo retroalimentar las mejoras.

La tabla siguiente es un ejemplo de los tipos de datos que se pueden recolectar acerca del proceso. Los números son sólo ilustrativos y no deben verse como estándares de la industria. Una comparación con los datos normativos de la organización revela deficiencias en el proceso de reuniones del equipo, y en sus

ejecuciones individuales (es decir, el proceso de escribirlo). Esto expone problemas, por ejemplo, en las juntas, que el mismo equipo había evaluado en 2 de 10 puntos. Se determinó (no visible en los datos) que el proceso de reuniones mejoraría si el bosquejo propuesto en la junta estuviera más terminado.

Documentos de requerimientos: 200 requerimientos detallados	Reunión	Investigación	Ejecución	Revisión personal	Inspección
Horas gastadas	0.5 * 4	4	5	3	6
% del tiempo total	10%	20%	25%	15%	30%
% tiempo total: norma de la organización	15%	15%	30%	15%	25%
Auto evaluación de calidad, 1-10	2	8	5	4	6
Defectos por cada 100	N/D	N/D	N/D	5	6
Defectos por cada 100: Norma de la organización	N/D	N/D	N/D	3	4
Horas gastadas por requerimiento detallado	0.01	0.02	0.025	0.015	0.03
Horas gastadas por requerimiento detallado: Norma de la organización	0.02	0.02	0.04	0.01	0.03
Mejora del proceso	Mejora del "bosquejo" traído a la reunión		10% más tiempo de ejecución		
Resumen:	Productividad: 200/22=9.9 requerimientos detallados por hora Tasa de defectos restante probable 6/4 * [norma de la org. De 0.8 por cada 100] = 1.2 por cada 100				

Tabla 6.- Recolección de mediciones del proyecto para las etapas.

El otro problema observado fue en el paso de ejecución, donde se realiza el trabajo de escribir los requerimientos. La tasa de defectos es más alta que la normal (5 contra 3) y la calidad autoevaluada es un poco menor que el promedio (4). Al comparar con las normas de la compañía parece que hay lugar para dedicar más tiempo a la ejecución del trabajo (esto es, como individuos) y reducir con esto la cuenta de defecto y mejorar la autoevaluación subjetiva. El lector puede ver, a partir de este proceso, que tener un estándar para contar las partes de la etapa es fundamental para nuestra habilidad de medir. En este caso, se cuentan los "requerimientos detallados".

La "tasa de defectos restantes" se refiere al número de requerimientos detallados defectuosos por cada 100 restantes en el documento de requerimientos después que termina esta etapa. Esto mide la efectividad de esta actividad. La tasa de defectos restantes se determina con el cálculo de una proporción de la tasa de defectos restantes histórica de la organización. Esta última se conoce al contar los defectos por cada 100 requerimientos detallados que se encontraron en proyectos anteriores después de completar el documento de requerimientos. Estos se encontraron durante el diseño, la implantación y las pruebas. La proporción, usada es 6/4, es la "tasa de defectos de requerimientos encontrada en este proyecto" dividida entre la "tasa promedio de defectos de requerimientos". El principio es que es probable que las tasas de defectos para un paso sean comparables a las del paso anterior. Un pronosticador más confiable tomaría en cuenta la tasa de defectos para pasos anteriores (durante la inspección personal en este caso). Esto requeriría una regresión lineal.

Aún en ausencia de datos históricos, el equipo predice los valores de las métricas. Con estas predicciones, los equipos tienden a trabajar mejor y recordar los resultados. Los datos recolectados se convierten en la base de futuros datos históricos. La administración de todo esto no es técnicamente difícil, pero se ha hecho al mismo tiempo que muchas otras actividades urgentes. Por esto, la asignación de responsabilidades claras y la revisión periódica de los datos de métricas comienza desde esta primera etapa del proceso. Como se verá en la siguiente sección, el proceso de mejora mediante la retroalimentación es lo que separa las grandes organizaciones de desarrollo de las que sólo son buenas.

2.3 Análisis de Requerimientos

El análisis global de los requerimientos de una aplicación, es un proceso de conceptualización y expresión de los conceptos en forma concreta. La mayor parte de los defectos encontrados en el software entregado se originan durante el análisis de requerimientos. En general, esos defectos también son más caros para reparar.

Significado de análisis de requerimientos

Para construir algo primero debe entenderse lo que debe ser este "algo". El proceso de entender y documentar este algo se llama "análisis de requerimientos". En general, los requerimientos expresan "qué" se supone debe hacer una aplicación: por lo común, no intentan expresar "cómo" lograr estas funciones. Por ejemplo, la siguiente afirmación es un requerimiento para una aplicación de contabilidad.

- El sistema debe permitir al usuario acceso a sus saldos.

En términos generales, la siguiente afirmación no es un requerimiento para una aplicación.

- Los estados de cuenta del cliente se almacenarán en una tabla llamada "saldos" en una base de datos de Access ®

Esta última afirmación se refiere a "cómo" debe construirse la aplicación, y no a "qué" debe hacer una aplicación.

Un requerimiento en un nivel con frecuencia se traduce en más de un requerimiento específico en el siguiente nivel más detallado. Para entender esto imagine que su requerimiento para una casa es que tenga "una vista de 180° de las montañas". Esto podría traducirse como la afirmación que tenga "una terraza en el lado derecho con dimensiones de 20 por 50 pies". Este es un requerimiento más específico a un nivel más detallado. De manera similar, la segunda afirmación puede en realidad ser un requerimiento en un nivel subsecuente dentro del proceso de desarrollo.

Además, existen excepciones a la regla de que en los requerimientos se evite especificar cómo debe hacerse algo. Por ejemplo, el cliente del ejemplo anterior podría, por alguna razón, querer que los estados de cuenta se almacenen en la base de datos de Access® con el nombre indicado. En ese caso, la segunda afirmación sería un requerimiento.

La salida del análisis de requerimientos es un documento que se conoce como especificación de requerimientos o especificación de requerimientos de software (ERS).

Requerimientos C y requerimientos D

Durante algún tiempo se ha discutido quién es el "dueño" de los requerimientos: el cliente o el desarrollador [Be 1]. Para manejar este aspecto, el análisis de requerimientos se divide en dos niveles ([Ro 1], [Br]). El primer nivel documenta los deseos y necesidades del cliente y se expresa en lenguaje claro para él. Los resultados suelen llamarse "requerimientos del cliente" o "requerimientos C". La audiencia primaria para los requerimientos C es la comunidad del cliente y la secundaria es la comunidad del desarrollador. El segundo nivel documenta los requerimientos de manera específica y estructurada. Estos se llaman "requerimientos del desarrollador" o "requerimientos D". La audiencia primordial de éstos es la comunidad del desarrollador y la secundaria la del cliente.

La distinción entre los tipos de requerimientos C y D en los títulos principales de la plantilla del documento estándar del IEEE se ilustra en la simiente figura:

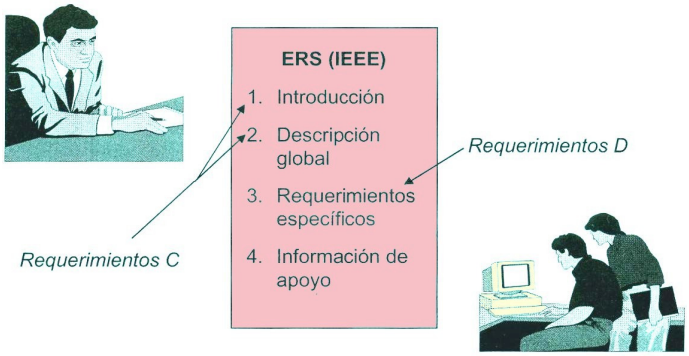


Fig. 34.- Requerimientos del cliente comparados con los requerimientos detallados.

Aunque las audiencias principales para los requerimientos C y D son diferentes, los clientes y desarrolladores trabajan juntos para crear productos exitosos. Una manera de asegurar una buena comunicación es hacer que representantes de los clientes trabajen junto con los desarrolladores. Algunas organizaciones de desarrollo incluso se rehúsan a aceptar trabajos si no se hace esto. Esta es la base del método de programación extrema que se menciona en la sección de procesos de desarrollo de software.

En la actualidad, son muchos los procesos de desarrollo de software que existen. Con el pasar de los años, la Ingeniería de Software ha introducido y popularizado una serie de estándares para medir y certificar la calidad, tanto del sistema a desarrollar, como del proceso de desarrollo en sí. Se han publicado muchos libros y artículos relacionados con este tema, con el modelado de procesos del negocio y la reingeniería. Un número creciente de herramientas automatizadas han surgido para ayudar a definir y aplicar un proceso de desarrollo de software efectivo. Hoy en día la economía global depende más de sistemas automatizados que en épocas pasadas; esto ha llevado a los equipos de desarrollo a enfrentarse con una nueva década de procesos y estándares de calidad.

Sin embargo, ¿cómo explicamos la alta incidencia de fallos en los proyectos de software? ¿Por qué existen tantos proyectos de software víctimas de retrasos, presupuestos sobregirados y con problemas de calidad? ¿Cómo podemos tener una producción o una economía de calidad, cuando nuestras actividades diarias dependen de la calidad del sistema?

Tal vez suene ilógico pero, a pesar de los avances que ha dado la tecnología, aún existen procesos de producción informales, parciales y en algunos casos no confiables.

La Ingeniería de Requerimientos cumple un papel primordial en el proceso de producción de software, ya que enfoca un área fundamental: la definición de lo que se desea producir. Su principal tarea consiste en la generación de especificaciones correctas que describan con claridad, sin ambigüedades, en forma consistente y compacta, el comportamiento del sistema; de esta manera, se pretende minimizar los problemas relacionados al desarrollo de sistemas.

El reemplazo de plataformas y tecnologías obsoletas, la compra de sistemas completamente nuevos, las modificaciones de todos o de casi todos los programas que forman un sistema, entre otras razones, llevan a desarrollar proyectos en calendarios sumamente ajustados y en algunos casos irreales; esto ocasiona que se omitan muchos pasos importantes en el ciclo de vida de desarrollo, entre estos, la definición de los requerimientos.

Estudios realizados muestran que más del 53% de los proyectos de software fracasan por no realizar un estudio previo de requisitos. Otros factores como falta de participación del usuario, requerimientos incompletos y el cambio a los requerimientos, también ocupan sitios altos en los motivos de fracasos.

2.3.1 La ingeniería de requerimientos y sus principales actividades

¿Qué son Requerimientos?

Normalmente, un tema de la Ingeniería de Software tiene diferentes significados. De las muchas definiciones que existen para requerimiento, a continuación se presenta la definición que aparece en el glosario de la IEEE[1]:

(1) Una condición o necesidad de un usuario para resolver un problema o alcanzar un objetivo. (2) Una condición o capacidad que debe estar presente en un sistema o componentes de sistema para satisfacer un contrato, estándar, especificación u otro documento formal. (3) Una representación documentada de una condición o capacidad como en (1) o (2).

Los requerimientos[2] puedes dividirse en requerimientos funcionales y requerimientos no funcionales.

Los requerimientos funcionales definen las funciones que el sistema será capaz de realizar. Describen las transformaciones que el sistema realiza sobre las entradas para producir salidas.

Los requerimientos no funcionales tienen que ver con características que de una u otra forma puedan limitar el sistema, como por ejemplo, el rendimiento (en tiempo y espacio), interfaces de usuario, fiabilidad (robustez del sistema, disponibilidad de equipo), mantenimiento, seguridad, portabilidad, estándares, etc.

[1] IEEE Std. 610.12-1990

[2] En el documento se usa la palabra requisito como sinónimo de requerimiento.

Características de los requerimientos

Las características de un requerimiento son sus propiedades principales. Un conjunto de requerimientos en estado de madurez, deben presentar una serie de características tanto individualmente como en grupo.

A continuación se presentan las más importantes.

Necesario: Un requerimiento es necesario si su omisión provoca una deficiencia en el sistema a construir, y además su capacidad, características físicas o factor de calidad no pueden ser reemplazados por otras capacidades del producto o del proceso.

Conciso: Un requerimiento es conciso si es fácil de leer y entender. Su redacción debe ser simple y clara para aquellos que vayan a consultarlo en un futuro.

Completo: Un requerimiento está completo si no necesita ampliar detalles en su redacción, es decir, si se proporciona la información suficiente para su comprensión.

Consistente: Un requerimiento es consistente si no es contradictorio con otro requerimiento.

No ambiguo: Un requerimiento no es ambiguo cuando tiene una sola interpretación. El lenguaje usado en su definición, no debe causar confusiones al lector.

Verificable: Un requerimiento es verificable cuando puede ser cuantificado de manera que permita hacer uso de los siguientes métodos de verificación: inspección, análisis, demostración o pruebas.

Dificultades para definir los requerimientos:

- Los requerimientos no son obvios y vienen de muchas fuentes.
- Son difíciles de expresar en palabras (el lenguaje es ambiguo).
- Existen muchos tipos de requerimientos y diferentes niveles de detalle.
- La cantidad de requerimientos en un proyecto puede ser difícil de manejar.
- Nunca son iguales. Algunos son más difíciles, más riesgosos, más importantes o más estables que otros.
- Los requerimientos están relacionados unos con otros, y a su vez se relacionan con otras partes del proceso.
- Cada requerimiento tiene propiedades únicas y abarcan áreas funcionales específicas.
- Un requerimiento puede cambiar a lo largo del ciclo de desarrollo.
- Son difíciles de cuantificar, ya que cada conjunto de requerimientos es particular para cada proyecto.

Ingeniería de Requerimientos vs. Administración de Requerimientos

El proceso de recopilar, analizar y verificar las necesidades del cliente para un sistema es llamado Ingeniería de Requerimientos. La meta de la Ingeniería de Requerimientos (IR) es entregar una especificación de requisitos de software correcta y completa.

A continuación se darán algunas definiciones para Ingeniería de Requerimientos.

"Ingeniería de Requerimientos es la disciplina para desarrollar una especificación completa, consistente y no ambigua, la cual servirá como base para acuerdos comunes entre todas las partes involucradas y en donde se describen las funciones que realizará el sistema" Boehm 1979.

"Ingeniería de Requerimientos es el proceso por el cual se transforman los requerimientos declarados por los clientes[3], ya sean hablados o escritos, a especificaciones precisas, no ambiguas, consistentes y completas del comportamiento del sistema, incluyendo funciones, interfaces, rendimiento y limitaciones". STARTS Guide 1987.

"Es el proceso mediante el cual se intercambian diferentes puntos de vista para recopilar y modelar lo que el sistema va a realizar. Este proceso utiliza una combinación de métodos, herramientas y actores, cuyo producto es un modelo del cual se genera un documento de requerimientos" Leite 1987.

"Ingeniería de Requerimientos es un enfoque sistémico para recolectar, organizar y documentar los requerimientos del sistema; es también el proceso que establece y mantiene acuerdos sobre los cambios de requerimientos, entre los clientes y el equipo del proyecto" Rational Software.

Importancia de la Ingeniería de Requerimientos

Los principales beneficios que se obtienen de la Ingeniería de Requerimientos son:

- **Permite gestionar las necesidades del proyecto en forma estructurada:** Cada actividad de la IR consiste de una serie de pasos organizados y bien definidos.
- **Mejora la capacidad de predecir cronogramas de proyectos, así como sus resultados:** La IR proporciona un punto de partida para controles subsecuentes y actividades de mantenimiento, tales como estimación de costos, tiempo y recursos necesarios.
- **Disminuye los costos y retrasos del proyecto:** Muchos estudios han demostrado que reparar errores por un mal desarrollo no descubierto a tiempo, es sumamente caro; especialmente aquellas decisiones tomadas durante la RE.
- **Mejora la calidad del software:** La calidad en el software tiene que ver con cumplir un conjunto de requerimientos (funcionalidad, facilidad de uso, confiabilidad, desempeño, etc.).
- **Mejora la comunicación entre equipos:** La especificación de requerimientos representa una forma de consenso entre clientes y desarrolladores. Si este consenso no ocurre, el proyecto no será exitoso.
- **Evita rechazos de usuarios finales:** La Ingeniería de Requerimientos obliga al cliente a considerar sus requerimientos cuidadosamente y revisarlos dentro del marco del problema, por lo que se le involucra durante todo el desarrollo del proyecto.

Personal involucrado en la Ingeniería de Requerimientos

Realmente, son muchas las personas involucradas en el desarrollo de los requerimientos de un sistema. Es importante saber que cada una de esas personas tienen diversos intereses y juegan roles específicos

[3] El "cliente" puede ser interpretado como un usuario de contabilidad, el grupo de mercadeo, otra organización interna o un cliente externo.

dentro de la planificación del proyecto; el conocimiento de cada papel desempeñado, asegura que se involucren a las personas correctas en las diferentes fases del ciclo de vida, y en las diferentes actividades de la IR.

No conocer estos intereses puede ocasionar una comunicación poco efectiva entre clientes y desarrolladores, que a la vez traería impactos negativos tanto en tiempo como en presupuesto. Los roles más importantes pueden clasificarse como sigue:

- **Usuario Final:** Son las personas que usarán el sistema desarrollado. Ellos están relacionados con la utilidad, la disponibilidad y la fiabilidad del sistema; están familiarizados con los procesos específicos que debe realizar el software, dentro de los parámetros de su ambiente laboral. Serán quienes utilicen las interfaces y los manuales de usuario.
- **Usuario Líder:** Son los individuos que comprenden el ambiente del sistema o el dominio del problema en donde será empleado el software desarrollado. Ellos proporcionan al equipo técnico los detalles y requerimientos de las interfaces del sistema.
- **Personal de Mantenimiento:** Para proyectos que requieran un mantenimiento eventual, estas personas son las responsables de la administración de cambios, de la implementación y resolución de anomalías. Su trabajo consiste en revisar y mejorar los procesos del producto ya finalizado.
- **Analistas y Programadores:** Son los responsables del desarrollo del producto en sí; ellos interactúan directamente con el cliente.
- **Personal de Pruebas:** Se encargan de elaborar y ejecutar el plan de pruebas para asegurar que las condiciones presentadas por el sistema son las adecuadas. Son quienes van a validar si los requerimientos satisfacen las necesidades del cliente.

Otras personas que pueden estar involucradas, dependiendo de la magnitud del proyecto, pueden ser: administradores de proyecto, documentadores, diseñadores de base de datos, entre otros.

Puntos a considerar durante la Ingeniería de Requerimientos

En esta lista, se enumeran los puntos más importantes siguientes:

Objetivos del negocio y ambiente de trabajo: Aunque los objetivos del negocio están definidos frecuentemente en términos generales, son usados para descomponer el trabajo en tareas específicas. En ciertas situaciones IR se enfoca en la descripción de las tareas y en el análisis de sistemas similares. Esta información proporciona la base para especificar el sistema que será construido; aunque frecuentemente se añadan al sistema tareas que no encajan con el ambiente de trabajo planificado.

El nuevo sistema cambiará el ambiente de trabajo, sin embargo, es muy difícil anticipar los efectos actuales sobre la organización. Los cambios no ocurren solamente cuando un nuevo software es implementado y puesto en producción; también ocurren cuando cambia el ambiente que lo rodea (nuevas soluciones a problemas, nuevo equipo para instalar, etc.). La necesidad de cambio es sustentada por el enorme costo de mantenimiento; aunque existen diversas razones que dificultan el mantenimiento del software, la falta de atención a la IR es la principal.

Frecuentemente la especificación inicial es también la especificación final, lo que obstaculiza la

comunicación y el proceso de aprendizaje de las personas involucradas; esta es una de las razones por las cuales existen sistemas inadecuados.

Punto de vista de los clientes: Muchos sistemas tienen diferentes tipos de clientes. Cada grupo de clientes tiene necesidades diferentes y, diferentes requerimientos tienen diferentes grados de importancia para ellos. Por otro lado, escasas veces tenemos que los clientes son los mismos usuarios; trayendo como consecuencia que los clientes soliciten procesos que causan conflictos con los solicitados por el usuario.

Diferentes puntos de vistas también pueden tener consecuencias negativas, tales como datos redundantes, inconsistentes y ambiguos.

El tamaño y complejidad de los requerimientos ocasiona desentendimiento, dificultad para enfocarse en un solo aspecto a la vez y dificultad para visualizar relaciones existentes entre requerimientos.

Barreras de comunicación: La Ingeniería de Requerimientos depende de una intensa comunicación entre clientes y analistas de requerimientos; sin embargo, existen problemas que no pueden ser resueltos mediante la comunicación.

Para remediar esto, se deben abordar nuevas técnicas operacionales que ayuden a superar estas barreras y así ganar experiencia dentro del marco del sistema propuesto.

Evolución e integración del sistema: Pocos sistemas son construidos desde cero. En la práctica, los proyectos se derivan de sistemas ya existentes. Por lo tanto, los analistas de requerimientos deben comprender esos sistemas, que por lo general son una integración de componentes de varios proveedores.

Para encontrar una solución a problemas de este tipo, es muy importante hacer planeamientos entre los requerimientos y la fase de diseño; esto minimizará la cantidad de fallas directas en el código.

Documentación de requerimientos: Los documentos de Ingeniería de Requerimientos son largos. La mayoría están compuestos de cientos o miles de páginas; cada página contiene muchos detalles que pueden tener efectos profundos en el resto del sistema.

Normalmente, las personas se encuentran con dificultades para comprender documentos de este tamaño, sobre todo si lo leen cuidadosamente. Es casi imposible leer un documento de especificación de gran tamaño, pues difícilmente una persona puede memorizar los detalles del documento. Esto causa problemas y errores que no son detectados hasta después de haberse construido el sistema.

Actividades de la Ingeniería de Requerimientos

En el proceso de IR son esenciales diversas actividades. En este documento serán presentadas secuencialmente, sin embargo, en un proceso de Ingeniería de Requerimientos efectivo, estas actividades son aplicadas de manera continua y en orden variado.

Dependiendo del tamaño del proyecto y del modelo de proceso de software utilizado para el ciclo de desarrollo, las actividades de la IR varían tanto en número como en nombres. La siguiente tabla muestra algunos ejemplos de las actividades identificadas para cada proceso.

MODELO	Oliver and Steiner 1996	EIA / IS-632	IEEE Std 1220- 1994	CMM nivel Repetitivo (2)	RUP
Actividades	Evaluar la información disponible	Análisis de requerimientos		Identificación de requerimientos	Análisis del Problema
	Definir métricas efectivas	Análisis funcional	Estudio de los requerimientos	Identificación de restricciones del sistema a desarrollar	Comprender las necesidades de los involucrados
	Crear un modelo del comportamiento del sistema	Síntesis	Validación de requerimientos	Análisis de los requerimientos	Definir el sistema
	Crear un modelo de los objetos	Análisis y control del sistema	Análisis funcional	Representación de los requerimientos	Analizar el alcance del proyecto
	Ejecutar el análisis		Evaluación y estudio de funciones	Comunicación de los requerimientos	Modificar la definición del sistema
	Crear un plan secuencial de construcción y pruebas		Verificación de funciones	Validación de requerimientos	Administrar los cambios de requerimientos
			Síntesis		
			Estudio y evaluación del diseño		
			Verificación física		
			Control		

Tabla 7.- Actividades de la IR para diferentes modelos de procesos de Ingeniería de Software.

A pesar de las diferentes interpretaciones que cada desarrollador tenga sobre el conjunto de actividades mostradas en la figura anterior, podemos identificar y extraer cinco actividades principales que son:

- Análisis del Problema.
- Evaluación y Negociación.
- Especificación.
- Validación.
- Evolución.

A continuación se explicará el Análisis del Problema:

Análisis del Problema: El objetivo de esta actividad es entender las verdaderas necesidades del negocio.

Antes de describir qué pasos deben cumplirse en esta actividad, debemos tener una definición clara del término "Problema".

Un problema puede ser definido como la diferencia entre las cosas como se perciben y las cosas como se desean"[4]. Aquí vemos nuevamente la importancia que tiene una buena comunicación entre desarrolladores y clientes; de esta comunicación con el cliente depende que entendamos sus necesidades.

A través de la definición de problema, podemos ver entonces que la actividad de "Análisis del Problema"[5] tiene por objetivo que se comprendan los problemas del negocio, se evalúen las necesidades iniciales de todos los involucrados en el proyecto y que se proponga una solución de alto nivel para resolverlo.

Durante el análisis del problema, se realizan una serie de pasos para garantizar un acuerdo entre los involucrados, basados en los problemas reales del negocio.

Estos pasos son los siguientes:

- **Comprender el problema que se está resolviendo:** Es importante determinar quién tiene el problema realmente, considerar dicho problema desde una variedad de perspectivas y explorar muchas soluciones desde diferentes puntos de vista. Veamos la siguiente necesidad: "El cliente se queja mucho por la enorme fila que debe formar para realizar una transacción bancaria".

  Perspectiva del cliente = Pérdida de tiempo.

  Perspectiva del banco = Posibles pérdidas de clientes.

  Posibles soluciones pueden ser, determinar por qué demoran los cajeros, colocar una nueva caja (implica contratación de nuevos cajeros), abrir una nueva sucursal (involucra personal nuevo y estudio de mercado), realizar transacciones por otros medios (teléfono, Internet, mediante cajeros automáticos, autobancos, etc.).

  Como puede verse, múltiples soluciones aplican para el mismo problema, sin embargo, sólo una de ellas será la más factible. Las soluciones iniciales, deben ser definidas tomando en cuenta tanto la perspectiva técnica como la del negocio.

- **Construir un vocabulario común:** Debe confeccionarse un glosario en dónde se definan todos los términos que tengan significados comunes (sinónimos) y que serán utilizados durante el proyecto. Por ejemplo, las palabras pignoración, retención, valor en suspenso, custodia, garantía, entre otras, son utilizadas para referirse a la acción de dejar una prenda (puede ser cualquier forma de ahorros) como garantía de una deuda adquirida.

  La creación de un glosario es sumamente beneficiosa ya que reduce los términos ambiguos desde el principio, ahorra tiempo, asegura que todos los participantes de una reunión están en la misma página, además de ser reutilizable en otros proyectos.

- **Identificar a los afectados por el sistema:** Identificar a todos los afectados evita que existan sorpresas a medida que avanza el proyecto. Las necesidades de cada afectado, son discutidas y sometidas a debate durante la Ingeniería de Requerimientos, aunque esto no garantiza que vaya a estar disponible toda la información necesaria para especificar un sistema adecuado.

[4] Gause & Weinberg. Turn your lights on, 1989.

[5] A esta actividad también se le conoce como "Identificación de Requerimientos", "Elaboración de necesidades y objetivos", "Elicitation Requirements", entre otras, pero en todas se realizan los mismo pasos.

Para saber quiénes son las personas, departamentos, organizaciones internas o externas que se verán afectadas por el sistema, debemos realizar algunas preguntas.

- ¿Quién usará el sistema que se va a construir?
- ¿Quién desarrollará el sistema?
- ¿Quién probará el sistema?
- ¿Quién documentará el sistema?
- ¿Quién dará soporte al sistema?
- ¿Quién dará mantenimiento al sistema?
- ¿Quién comercializará, venderá, y/o distribuirá el sistema?
- ¿Quién se beneficiará por el retorno de inversión del sistema?

Como vemos, debe conocerse la opinión de todo aquél que de una u otra forma está involucrado con el sistema, ya sea directa o indirectamente.

Definir los límites y restricciones del sistema: Este punto es importante pues debemos saber lo que se está construyendo, y lo que no se está construyendo, para así entender la estrategia del producto a corto y largo plazo. Debe determinarse cualquier restricción ambiental, presupuestaria, de tiempo, técnica y de factibilidad que limite el sistema que se va a construir.

2.3.2 Evaluación y negociación de los requerimientos

La diversa gama de fuentes de las cuales provienen los requerimientos, hacen necesaria una evaluación de los mismos antes de definir si son adecuados para el cliente. El término "adecuado" significa que ha sido percibido a un nivel aceptable de riesgo tomando en cuenta las factibilidades técnicas y económicas, a la vez que se buscan resultados completos, correctos y sin ambigüedades.

En esta etapa se pretende limitar las expectativas del cliente apropiadamente, tomando como referencia los niveles de abstracción y descomposición de cada problema presentado.

Los principales pasos de esta actividad son:

Descubrir problemas potenciales: En este paso se asegura que todas las características descritas en el punto 1.1 estén presentes en cada uno de los requerimientos, es decir, se identifican aquellos requerimientos ambiguos, incompletos, inconsistentes, etc.

Clasificar los requerimientos: En este paso se busca identificar la importancia que tiene un requerimiento en términos de implementación. A esta característica se le conoce como prioridad y debe ser usada para establecer la secuencia en que ocurrirán las actividades de diseño y prueba de cada requisito. La prioridad de cada requerimiento dependerá de las necesidades que tenga el negocio.

En base a la prioridad, cada requerimiento puede ser clasificado como obligatorio, deseable o innecesario u opcional. Un requerimiento es obligatorio si afecta una operación crítica del negocio. Si existe algún proceso que se quiera incluir para mejorar los procesos actuales, estamos ante un requerimiento deseable;

y si se trata de un requerimiento informativo o que puede esperar para fases posteriores, el requerimiento es catalogado como innecesario.

Una vez hecha esta categorización de los requerimientos, se puede tomar como estrategia general el incluir los obligatorios, discutir los deseables y descartar los innecesarios. Antes de decidir la inclusión de un requerimiento, también debe analizarse su costo, complejidad, y una cantidad de otros factores. Por ejemplo, si un requerimiento fuera trivial de implementar, puede ser una buena idea incluirlo por más que éste sea sólo deseable.

Evaluar factibilidades y riesgos: Involucra la evaluación de factibilidades técnicas (¿pueden implementarse los requerimientos con la tecnología actual?); factibilidades operacionales (¿puede ser el sistema utilizado sin alterar el organigrama actual?); factibilidades económicas (¿ha sido aprobado por los clientes el presupuesto?).

En la actividad de evaluación y negociación, se incrementa la comunicación entre el equipo de desarrollo y los afectados. Para que los requerimientos puedan ser comunicados de manera efectiva, hay una serie de consideraciones que deben tenerse en cuenta; entre las principales tenemos:

- Documentar todos los requerimientos a un nivel de detalle apropiado.
- Mostrar todos los requerimientos a los involucrados en el sistema.
- Analizar el impacto que causen los cambios a requerimientos antes de aceptarlos.
- Establecer las relaciones entre requerimientos que indiquen dependencias.
- Negociar con flexibilidad para que exista un beneficio mutuo.
- Enfocarse en intereses y no en posiciones.

2.3.3 Especificación de Requisitos de Software (SRS)

La especificación de requisitos de software es la actividad en la cual se genera el documento, con el mismo nombre, que contiene una descripción completa de las necesidades y funcionalidades del sistema que será desarrollado; describe el alcance del sistema y la forma en cómo hará sus funciones, definiendo los requerimientos funcionales y los no funcionales.

En la SRS se definen todos los requerimientos de hardware y software, diagramas, modelos de sistemas y cualquier otra información que sirva de soporte y guía para fases posteriores.

Es importante destacar que la especificación de requisitos es el resultado final de las actividades de análisis y evaluación de requerimientos; este documento resultante será utilizado como fuente básica de comunicación entre los clientes, usuarios finales, analistas de sistema, personal de pruebas, y todo aquel involucrado en la implementación del sistema.

Los clientes y usuarios utilizan la SRS para comparar si lo que se está proponiendo, coincide con las necesidades de la empresa. Los analistas y programadores la utilizan para determinar el producto que

debe desarrollarse. El personal de pruebas elaborará las pruebas funcionales y de sistemas en base a este documento. Para el administrador del proyecto sirve como referencia y control de la evolución del sistema.

La SRS posee las mismas características de los requerimientos: completa, consistente, verificable, no ambigua, factible, modificable, rastreable, precisa, entre otras. Para que cada característica de la SRS sea considerada, cada uno de los requerimientos debe cumplirlas; por ejemplo, para que una SRS se considere verificable, cada requerimiento definido en ella debe ser verificable; para que una SRS se considere modificable, cada requerimiento debe ser modificable y así sucesivamente. Las características de la SRS son verificadas en la actividad de validación, descrita en el punto siguiente.

La estandarización de la SRS es fundamental pues ayudará, entre otras cosas, a facilitar la lectura y escritura de la misma. Será un documento familiar para todos los involucrados, además de asegurar que se cubren todos los tópicos importantes.

2.3.4 Validación de Requisitos

La validación es la actividad de la IR que permite demostrar que los requerimientos definidos en el sistema son los que realmente quiere el cliente; además revisa que no se haya omitido ninguno, que no sean ambiguos, inconsistentes o redundantes.

En este punto es necesario recordar que la SRS debe estar libre de errores, por lo tanto, la validación garantiza que todos los requerimientos presentes en el documento de especificación sigan los estándares de calidad. No debe confundirse la actividad de evaluación de requerimientos con la validación de requerimientos. La evaluación verifica las propiedades de cada requerimiento, mientras que la validación revisa el cumplimiento de las características de la especificación de requisitos.

Durante la actividad de validación pueden hacerse preguntas en base a cada una de las características que se desean revisar. A continuación se presentan varios ejemplos:

- ¿Están incluidas todas las funciones requeridas por el cliente? (completa).
- ¿Existen conflictos en los requerimientos? (consistencia).
- ¿Tiene alguno de los requerimientos más de una interpretación? (no ambigua).
- ¿Está cada requerimiento claramente representado? (entendible).
- ¿Pueden los requerimientos ser implementados con la tecnología y el presupuesto disponible? (factible).
- ¿Está la SRS escrita en un lenguaje apropiado? (clara).
- ¿Existe facilidad para hacer cambios en los requerimientos? (modificable).
- ¿Está claramente definido el origen de cada requisito? (rastreable).
- ¿Pueden los requerimientos ser sometidos a medidas cuantitativas? (verificable).

La validación de requerimientos es importante pues de ella depende que no existan elevados costos de mantenimiento para el software desarrollado.

2.3.5 Evolución de los requerimientos

Los requerimientos son una manera de comprender mejor el desarrollo de las necesidades de los usuarios y cómo los objetivos de la organización pueden cambiar, por lo tanto, es esencial planear posibles cambios a los requerimientos cuando el sistema sea desarrollado y utilizado. La actividad de evolución es un proceso externo que ocurre a lo largo del ciclo de vida del proyecto. Los requerimientos cambian por diferentes razones. Las más frecuentes son:

- Porque al analizar el problema, no se hacen las preguntas correctas a las personas correctas.
- Porque cambió el problema que se estaba resolviendo.
- Porque los usuarios cambiaron su forma de pensar o sus percepciones.
- Porque cambió el ambiente de negocios.
- Porque cambió el mercado en el cual se desenvuelve el negocio.

Cambios a los requisitos involucra modificar el tiempo en el que se va a implementar una característica en particular, modificación que a la vez puede tener impacto en otros requerimientos. Por esto, la administración de cambios involucra actividades como establecer políticas, guardar históricos de cada requerimiento, identificar dependencias entre ellos y mantener un control de versiones.

Tener versiones de los requerimientos es tan importante como tener versiones del código, ya que evita tener requerimientos emparchados [6] en un proyecto.

Entre algunos de los beneficios que proporciona el control de versiones están:

- Prevenir cambios no autorizados.
- Guardar revisiones de los documentos de requerimientos.
- Recuperar versiones previas de los documentos.
- Administrar una estrategia de "releases".
- Prevenir la modificación simultánea a los requisitos.

En vista que las peticiones de cambios provienen de muchas fuentes, las mismas deben ser enrutadas en un solo proceso. Esto se hace con la finalidad de evitar problemas y conseguir estabilidad en los requerimientos.

[6] Se le llama requerimiento emparchado a aquél que ha sufrido cambios excesivos en la semántica.

2.3.6 Técnicas y herramientas utilizadas en la Ingeniería de Requerimientos

Técnicas utilizadas en las actividades de IR

Existen varias técnicas para la IR. Cada técnica puede aplicarse en una o más actividades de la IR; en la práctica, la técnica más apropiada para cada actividad dependerá del proyecto que esté desarrollándose.

Entrevistas y Cuestionarios

Las entrevistas y cuestionarios se emplean para reunir información proveniente de personas o de grupos. Durante la entrevista, el analista conversa con el encuestado; el cuestionario consiste en una serie de preguntas relacionadas con varios aspectos de un sistema.

Por lo común, los encuestados son usuarios de los sistemas existentes o usuarios en potencia del sistema propuesto. En algunos casos, son gerentes o empleados que proporcionan datos para el sistema propuesto o que serán afectados por él.

Las preguntas que deben realizarse en esta técnica, deben ser preguntas de alto nivel y abstractas que pueden realizarse al inicio del proyecto para obtener información sobre aspectos globales del problema del usuario y soluciones potenciales.

Con frecuencia, se utilizan preguntas abiertas para descubrir sentimientos, opiniones y experiencias generales, o para explorar un proceso o problema. Este tipo de preguntas son siempre apropiadas, además que ayudan a entender la perspectiva del afectado y no están influenciadas por el conocimiento de la solución.

Las preguntas pueden ser enfocadas a un elemento del sistema, tales como usuarios, procesos, etc. El siguiente ejemplo muestra algunos tipos de preguntas abiertas.

Del Usuario:

- ¿Quién es el cliente?
- ¿Quién es el usuario?
- ¿Son sus necesidades diferentes?
- ¿Cuáles son sus habilidades, capacidades, ambiente?

Del Proceso:

- ¿Cuál es la razón por la que se quiere resolver este problema?
- ¿Cuál es el valor de una solución exitosa?
- ¿Cómo usted resuelve el problema actualmente?
- ¿Qué retrasos ocurren o pueden ocurrir?

Del Producto:

- ¿Qué problemas podría causar este producto en el negocio?

- ¿En qué ambiente se usará el producto?
- ¿Cuáles son sus expectativas para los conceptos fácil de usar, confiable, rendimiento?
- ¿Qué obstáculos afectan la eficiencia del sistema?

El éxito de esta técnica combinada, depende de la habilidad del entrevistador y de su preparación para la misma. Los analistas necesitan ser sensibles a las dificultades que algunos entrevistados crean durante la entrevista y saber cómo tratar con problemas potenciales. Asimismo, necesitan considerar no sólo la información que adquieren a través del cuestionario y la entrevista, sino también, su significado.

Lluvia de Ideas (Brainstorm)

Este método comenzó en el ámbito de las empresas, aplicándose a temas tan variados como la productividad, la necesidad de encontrar nuevas ideas y soluciones para los productos del mercado, encontrar nuevos métodos que desarrollen el pensamiento creativo a todos los niveles, etc. Pero pronto se extendió a otros ámbitos, incluyendo el mundo del desarrollo de sistemas; básicamente se busca que los involucrados en un proyecto desarrollen su creatividad, promoviendo la introducción de los principios creativos.

A esta técnica se le conoce también como torbellino de ideas, tormenta de ideas, desencadenamiento de ideas, movilización verbal, bombardeo de ideas, sacudidas de cerebros, promoción de ideas, tormenta cerebral, avalancha de ideas, tempestad en el cerebro y tempestad de ideas, entre otras.

Principios de la lluvia de ideas

- Aplazar el juicio y no realizar críticas, hasta que no se agoten las ideas, ya que actuaría como un inhibidor. Se ha de crear una atmósfera de trabajo en la que nadie se sienta amenazado.
- Cuantas más ideas se sugieren, mejores resultados se conseguirán: "la cantidad produce la calidad". Las mejores ideas aparecen tarde en el periodo de producción de ideas, será más fácil que encontremos las soluciones y tendremos más variedad sobre las cuales elegir.
- La producción de ideas en grupos puede ser más efectiva que la individual.
- Tampoco debemos olvidar que durante las sesiones, las ideas de una persona, serán asociadas de manera distinta por cada miembro, y hará que aparezcan otras por contacto.

El equipo en una lluvia de ideas debe estar formado por:

El Director: Es la figura principal y el encargado de dirigir la sesión. Debe ser un experto en pensamiento creador. Su función es formular claramente el problema y que todos se familiaricen con él. Cuando lo haga, debe estimular ideas y hacer que se rompa el hielo en el grupo. Es el encargado de que se cumplan las normas, no permitiendo las críticas. Debe permanecer callado e intervenir cuando se corte la afluencia de ideas, por lo que le será útil llevar ya un listado de ideas. Debe hacer que todos participen y den ideas. Además, es la persona que concede la palabra y da por finalizada la sesión. Posteriormente, clasificará las ideas de la lista que le proporciona el secretario.

El secretario: registra por escrito las ideas según van surgiendo. Las enumera, las reproduce fielmente, las redacta y se asegura que todos están de acuerdo con lo escrito. Por último realizará una lista de ideas.

Los participantes: pueden ser habituales o invitados; cualquier involucrado en el proyecto entra en esta categoría. Su función es producir ideas. Conviene que entre ellos no haya diferencias jerárquicas.

Las personas que componen el grupo deben estar motivadas para solucionar el problema, y con un ambiente que propicie la participación de todos. Todos pueden sentirse confiados y con la sensación de que pueden hablar sin que se produzcan críticas. Todas las ideas en principio deben tener el mismo valor, pues cualquiera de ellas puede ser la clave para la solución. Es necesario prestar mucha atención a las frases que pueden coartar la producción de ideas. Además durante la celebración no deben asistir espectadores.

Debemos evitar todos los bloqueos que paralizan la generación de ideas: como son nuestros hábitos o ideas preconcebidas, el desánimo o falta de confianza en sí mismo, el temor y la timidez.

Fases de aplicación en el Brainstorm

Descubrir hechos: Al menos con un día de antelación, el director comunica por escrito a los miembros del grupo sobre los temas a tratar. El director explica los principios de la "Tormenta de Ideas" e insiste en la importancia de tenerlos en cuenta. La sesión comienza con una ambientación de unos 10 minutos, tratando un tema sencillo y no comprometido. Es una fase especialmente importante para los miembros sin experiencia.

Se determina el problema, delimitándolo, precisándolo y clarificándolo. A continuación se plantea el problema, recogiendo las experiencias que se poseen o consultando documentación. Cuando es complejo, conviene dividirlo en partes. Aquí es importante la utilización del análisis, desmenuzando el problema en pequeñas partes para conectar lo nuevo y lo desconocido.

Producir ideas (es la fase de Tormenta de Ideas propiamente dicha).

Se van aplicando alternativas. Se busca producir una gran cantidad de ideas, aplicando los principios que hemos visto. Además, es útil cuando se ha trabajado mucho alejarse del problema, pues es un buen momento para que se produzcan asociaciones. Muchas de las nuevas ideas serán ideas antiguas, mejoradas o combinadas con varias ya conocidas.

Al final de la reunión, el director da las gracias a los asistentes y les ruega que no abandonen el problema, ya que al día siguiente se le pedirá una lista de ideas que les puedan haber surgido. Se incorporan las ideas surgidas después de la reunión.

Descubrir soluciones: Se elabora una lista definitiva de ideas, para seleccionar las más interesantes. La selección se realiza desechando las ideas que no tienen valor y se estudia si son válidas las que se consideran interesantes. Lo mejor es establecer una lista de criterios de conveniencia para cada idea. Se seleccionan las ideas más útiles y si es necesario se ponderarán. Pueden realizarlo los mismos miembros del grupo o crear otros para esta tarea; la clasificación debe hacerse por categorías (tarea que corresponde al director). Se presentan las ideas de forma atractiva, haciendo uso de soportes visuales.

Prototipos

Los prototipos permiten al desarrollador crear un modelo del software que debe ser construido.

Al igual que todos los enfoques al proceso de desarrollo del software, el crear prototipos comienza con la captura de requerimientos. Desarrolladores y clientes se reúnen y definen los objetivos globales del software, identifican todos los requerimientos que son conocidos y señalan áreas en las que será necesaria la profundización en las definiciones. Luego de esto, tiene lugar un "diseño rápido". El diseño rápido se centra en una representación de aquellos aspectos del software que serán visibles al usuario (por ejemplo, entradas y formatos de las salidas). El diseño rápido lleva a la construcción de un prototipo. El prototipo es evaluado por el cliente y el usuario y utilizado para refinar los requerimientos del software a ser desarrollado. Un proceso de iteración tiene lugar a medida que el prototipo es "puesto a punto" para satisfacer las necesidades del cliente y permitiendo al mismo tiempo una mejor comprensión del problema por parte del desarrollador.

Existen principalmente dos tipos de prototipos:

Prototipo rápido (o concept prototype): La generación de prototipos rápidos es un mecanismo para lograr la validación precompromiso. Se utiliza para validar requerimientos en una etapa previa al diseño específico. En este sentido, el prototipo puede ser visto como una aceptación tácita de que los requerimientos no son totalmente conocidos o entendidos antes del diseño y la implementación. El prototipo rápido puede ser usado como un medio para explorar nuevos requerimientos y así ayudar a "controlar" su constante evolución.

Prototipo evolutivo: Desde una perspectiva diferente, todo el ciclo de vida de un producto puede ser visto como una serie incremental de detallados prototipos acumulativos. Tradicionalmente, el ciclo de vida está dividido en dos fases distintas: desarrollo y mantenimiento. La experiencia ha demostrado que esta distinción es arbitraria y va en contra de la realidad ya que la mayor parte del costo del software ocurre después de que el producto se ha entregado. El punto de vista evolutivo del ciclo de vida del software considera a la primera entrega como un prototipo inicial en el campo. Modificaciones y mejoras subsecuentes resultan en nuevas entregas de prototipos más maduros. Este proceso continúa hasta que se haya desarrollado el producto final. La adopción de esta óptica elimina la distinción arbitraria entre desarrollo y mantenimiento, resultando en un importante cambio de mentalidad que afecta las estrategias para la estimación de costos, enfoques de desarrollo y adquisición de productos.

Proceso de Análisis Jerárquico (AHP)

Esta técnica tiene por objetivo resolver problemas cuantitativos, para facilitar el pensamiento analítico y las métricas. Consiste en una serie de pasos a saber:

- Encontrar los requerimientos que van a ser priorizados.
- Combinar los requerimientos en las filas y columnas de la matriz n x n de AHP.
- Hacer algunas comparaciones de los requerimientos en la matriz.
- Sumar las columnas.
- Normalizar la suma de las filas.
- Calcular los promedios.

Estos pasos pueden aplicarse fácilmente a una cantidad pequeña de requerimientos, sin embargo, para un volumen grande, esta técnica no es la más adecuada.

Administración de Requerimientos con Casos de Uso

¿Qué son los Casos de Uso?

Los Casos de Uso son una técnica para especificar el comportamiento de un sistema: "Un Caso de Uso es una secuencia de interacciones entre un sistema y alguien o algo que usa alguno de sus servicios".

Todo sistema de software ofrece a su entorno[7] una serie de servicios. Un Caso de Uso es una forma de expresar cómo alguien o algo externo a un sistema lo usa. Cuando decimos "alguien o algo" hacemos referencia a que los sistemas son usados no sólo por personas, sino también por otros sistemas de hardware y software.

Por ejemplo, un sistema de ventas, si pretende tener éxito, debe ofrecer un servicio para ingresar un nuevo pedido de un cliente. Cuando un usuario accede a este servicio, podemos decir que está "ejecutando" el Caso de Uso ingresando pedido.

Los Casos de Uso fueron introducidos por Jacobson en 1992 [Jacobson 92]. Sin embargo, la idea de especificar un sistema a partir de su interacción con el entorno es original de McMenamin y Palmer, dos precursores del análisis estructurado, que en 1984 definieron un concepto muy parecido al del Caso de Uso: el evento. Para McMenamin y Palmer, un evento es algo que ocurre fuera de los límites del sistema, ante lo cual el sistema debe responder. Siguiendo con nuestro ejemplo anterior, nuestro sistema de ventas tendrá un evento "Cliente hace Pedido". En este caso el sistema deberá responder al estimulo que recibe.

Sin embargo, existen algunas diferencias entre los Casos de Uso y los eventos. Las principales son:

- Los eventos se centran en describir qué hace el sistema cuando el evento ocurre, mientras que los Casos de Uso se centran en describir cómo es el diálogo entre el usuario y el sistema.
- Los eventos son "atómicos": se recibe una entrada, se le procesa y se genera una salida, mientras que los Casos de Uso se prolongan a lo largo del tiempo mientras dure la interacción del usuario con el sistema. De esta forma, un Caso de Uso puede agrupar a varios eventos.
- Para los eventos, lo importante es qué datos ingresan al sistema o salen de él cuando ocurre el evento (estos datos se llaman datos esenciales), mientras que para los Casos de Uso la importancia del detalle sobre la información que se intercambia es secundaria. Según esta técnica, ya habrá tiempo más adelante en el desarrollo del sistema para ocuparse de este tema.

Los Casos de Uso combinan el concepto de evento del análisis estructurado con otra técnica de especificación de requerimientos bastante poco difundida: aquella que dice que una buena forma de expresar los requerimientos de un sistema es escribir su manual de usuario antes de construirlo. Esta técnica, si bien ganó pocos adeptos, se basa en un concepto muy interesante: al definir requerimientos, es importante describir al sistema desde el punto de vista de aquél que lo va a usar y no desde el punto

[7] Se conoce como entorno todos aquellos que usen el sistema.

de vista del que lo va a construir. De esta forma, es más fácil validar que los requerimientos documentados son los verdaderos requerimientos de los usuarios, ya que éstos comprenderán fácilmente la forma en la que están expresados.

Los Casos de Uso y UML

A partir de la publicación del libro de Jacobson, gran parte de los más reconocidos especialistas en métodos Orientados a Objetos coincidieron en considerar a los Casos de Uso como una excelente forma de especificar el comportamiento externo de un sistema. De esta forma, la notación de los Casos de Uso fue incorporada al lenguaje estándar de modelado UML (Unified Modeling Language), propuesto por Ivar Jacobson, James Rumbaugh y Grady Booch, tres de los precursores de las metodologías de Análisis y Diseño Orientado a Objetos, y avalado por las principales empresas que desarrollan software en el mundo. UML va en camino de convertirse en un estándar para modelado de sistemas de software de amplia difusión.

A pesar de ser considerada una técnica de Análisis Orientado a Objetos, es importante destacar que los Casos de Uso poco tienen que ver con entender a un sistema como un conjunto de objetos que interactúan, que es la premisa básica del Análisis Orientado a Objetos "clásico". En este sentido, el éxito de los Casos de Uso no hace más que dar la razón al análisis estructurado, que propone que la mejor forma de empezar a entender un sistema es a partir de los servicios o funciones que ofrece a su entorno, independientemente de los objetos que interactúan dentro del sistema para proveerlos.

Como era de esperar, es probable que en el futuro los métodos de análisis y diseño que prevalezcan hayan adoptado las principales ventajas de todos los métodos disponibles en la actualidad (estructurados, métodos formales, métodos orientados a objetos, etc.).

Definiciones y Notaciones

Actores

Un actor es una agrupación uniforme de personas, sistemas o máquinas que interactúan con el sistema que estamos construyendo de la misma forma. Por ejemplo, para una empresa que recibe pedidos en forma telefónica, todos los operadores que reciban pedidos y los ingresen en un sistema de ventas, si pueden hacer las mismas cosas con el sistema, son considerados un único actor: "Empleado de Ventas".

Los actores son externos al sistema que vamos a desarrollar. Por lo tanto, al identificar actores estamos empezando a delimitar el sistema y a definir su alcance. Definir el alcance del sistema debe ser el primer objetivo de todo analista, ya que un proyecto sin alcance definido nunca podrá alcanzar sus objetivos.

Es importante tener clara la diferencia entre usuario y actor. Un actor es una clase de rol, mientras que un usuario es una persona que cuando usa el sistema asume un rol. De esta forma, un usuario puede acceder al sistema como distintos actores. La forma más simple de entender esto es pensar en perfiles de usuario de un sistema operativo. Una misma persona puede acceder al sistema con distintos perfiles, que le permiten hacer cosas distintas. Los perfiles son en este caso equivalentes a los actores.

Otro sistema que interactúa con el que estamos construyendo también es un actor. Por ejemplo, si

nuestro sistema debe generar asientos contables para ser procesados por el sistema de contabilidad, este último sistema será un actor, que usa los servicios de nuestro sistema.

También puede ocurrir que el actor sea una máquina, en el caso en que el software controle sus movimientos, o sea operado por una máquina. Por ejemplo, si estamos construyendo un sistema para mover el brazo de un robot, el hardware del robot será un actor, asumiendo que dentro de nuestro sistema están las rutinas de bajo nivel que controlan al hardware.

Los actores se representan con dibujos simplificados de personas, llamados en inglés "stick man" (hombres de palo).

Si bien en UML los actores siempre se representan con "hombres de palo", a veces resulta útil representar a otros sistemas con alguna representación más clara.

Las flechas, que existían en la propuesta original de Jacobson, pero que desaparecieron del modelo semántico de UML, pueden usarse para indicar el flujo de información entre el sistema y el actor. Si la flecha apunta desde el actor hacia el sistema, esto indica que el actor está ingresando información en el sistema. Si la flecha apunta desde el sistema hacia el actor, el sistema está generando información para el actor.

Identificar a los actores es el primer paso para usar la técnica de Casos de Uso. Por ejemplo, en el sistema de pedidos nombrado anteriormente, sin conocer prácticamente ningún detalle sobre cómo funcionará, podemos decir que:

- El grupo de usuarios que ingrese pedidos al sistema será un actor.
- El grupo de usuarios que haga otras operaciones con los pedidos, como por ejemplo autorizarlos, cancelarlos y modificar sus plazos de entrega, será un actor.
- Todo grupo de usuarios que reciba ciertos informes del sistema, como por ejemplo estadísticas de ventas, será un actor.

Es común que los distintos actores coincidan con distintas áreas de la empresa en la que se implementará el sistema o con jerarquías dentro de la organización (empleado, supervisor y gerente son distintos actores, si realizan tareas distintas).

Todos los actores participan de los Casos de Uso. Ahora bien, es lógico que existan intersecciones entre lo que hacen los distintos actores. Por ejemplo, un supervisor puede autorizar pedidos, pero también puede ingresarlos. Veremos más adelante cómo definiendo actores abstractos, podemos especificar este comportamiento común para evitar redundancia.

Casos de Uso

Como mencionamos anteriormente, un Caso de Uso es una secuencia de interacciones entre un sistema y alguien o algo que usa alguno de sus servicios. Un Caso de Uso es iniciado por un actor. A partir de ese momento, ese actor junto con otros actores, intercambia datos o control con el sistema, participando de ese Caso de Uso.

El nombre de un Caso de Uso se expresa con un verbo en gerundio, seguido generalmente por el principal

objeto o entidad del sistema que es afectado por el caso. Gráficamente los Casos de Uso se representan con un óvalo, con el nombre del caso en su interior.

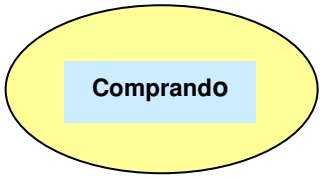


Fig. 35.- Ejemplo de un Caso de Uso.

Es importante notar que el nombre del caso siempre está expresado desde el punto de vista del actor y no desde el punto de vista del sistema. Por eso, un segundo Caso de Uso se debería llamar "Recibiendo Información de Pedidos" y no "Generando Información de Pedidos".

Los Casos de Uso tienen las siguientes características:

- Están expresados desde el punto de vista del actor.
- Se documentan con texto informal.
- Describen tanto lo que hace el actor como lo que hace el sistema cuando interactúa con él, aunque el énfasis está puesto en la interacción.
- Son iniciados por un único actor.
- Están acotados al uso de una determinada funcionalidad del sistema claramente diferenciada.

El último punto es tal vez el más difícil de definir. Uno podría después de todo, decir que todo sistema tiene un único Caso de Uso denominado "Usando el Sistema". Sin embargo, la especificación resultante sería de poca utilidad para entenderlo; sería como implementar un gran sistema escribiendo un único programa.

La pregunta importante es: ¿Qué es una "funcionalidad claramente diferenciada"? Por ejemplo, ¿ingresar pedidos es un Caso de Uso y autorizarlos es otro? ¿Cancelar los pedidos, es otro Caso de Uso, o es parte del Caso de Uso referido al ingreso de pedidos?

Si bien se pueden encontrar argumentos válidos para cualquiera de las dos alternativas, en principio la respuesta a todas estas preguntas es que todos son Casos de Uso distintos. Lamentablemente, si en la programación los criterios para dividir la funcionalidad en programas suelen ser difusos, los criterios para dividir la funcionalidad de un sistema en Casos de Uso son aún más difusos y por esto se hace importante usar el sentido común en estas decisiones.

En principio podríamos decir que la regla general es: una función del sistema es un Caso de Uso si se debe indicar explícitamente al sistema que uno quiere acceder a esa función. Por ejemplo, si

uno quiere dar de alta un pedido, accederá a la funcionalidad de "alta de pedidos del sistema". Sin embargo, si uno quiere dar de alta un campo del pedido, no debe indicar al sistema que quiere acceder a esa función. Dar de alta un campo de un pedido es una función que forma parte de un Caso de Uso mayor: "dar de alta un pedido".

Cuando pensamos en el grado de detalle de la división de los Casos de Uso también resulta útil imaginar que uno está escribiendo el manual del usuario del sistema. A nadie se le ocurriría escribir un manual de usuario con un solo capítulo en el que se describe toda su funcionalidad. De la misma forma, no se debe escribir una especificación con un solo Caso de Uso.

Extensiones (Extends)

Muchas veces, la funcionalidad de un Caso de Uso incluye un conjunto de pasos que ocurren sólo en algunas oportunidades. Supongamos que estamos especificando un sistema en el cual los clientes pueden ingresar pedidos interactivamente y que dentro de la funcionalidad del ingreso de pedidos, el usuario puede solicitar al sistema que le haga una presentación sobre los nuevos productos disponibles, sus características y sus precios. En este caso, tengo una excepción dentro del Caso de Uso "Ingresando Pedido". La excepción consiste en interrumpir el Caso de Uso y pasar a ejecutar el Caso de Uso "Revisando Presentación de Nuevos Productos". En este caso decimos que el Caso de Uso "Revisando Presentación de Nuevos Productos", extiende el Caso de Uso "Ingresando Pedido" y se representa por una línea de trazos desde el caso que "extiende a" al caso que es "extendido".

Las extensiones tienen las siguientes características:

- Representan una parte de la funcionalidad del caso que no siempre ocurre.
- Son un Caso de Uso en sí mismas.
- No necesariamente provienen de un error o excepción.

Usos (Uses)

Es común que la misma funcionalidad del sistema sea accedida a partir de varios Casos de Uso. Por ejemplo, la funcionalidad de buscar un producto puede ser accedida desde el ingreso de pedidos, desde las consultas de productos, o desde los reportes de ventas por producto. ¿Cómo hago para no repetir el texto de esta funcionalidad en todos los Casos de Uso que la acceden? La respuesta es simple: sacando esta funcionalidad a un nuevo Caso de Uso, que es usado por los casos de los cuales fue sacada. Este tipo de relaciones se llama "Relaciones de Uso" y se representa por una línea punteada desde el caso que "usa a" al caso que es "usado".

Decimos por ejemplo, que el Caso de Uso "Obteniendo reporte de ventas por producto", usa al Caso de Uso "Buscando producto".

Este concepto no es novedoso, es simplemente el concepto de la subrutina o subprograma usado en un nivel más alto de abstracción.

Las características de las relaciones de uso son:

- Aparecen como funcionalidad común, luego de haber especificado varios Casos de Uso.
- Los casos usados son Casos de Uso en sí mismos.
- El caso es usado siempre que el caso que lo usa es ejecutado. Esto marca la diferencia con las extensiones, que son opcionales.

Abstracción

Al identificar relaciones de uso y extensión, puede ser que extraigamos Casos de Uso que son accedidos por varios actores. Por ejemplo, el Caso de Uso "Buscando datos de producto" es accedido por muchos actores (el empleado de ventas que ingresa un pedido, el gerente que quiere obtener estadísticas por producto, el supervisor que quiere consultar la información de algún producto, etc.). Ahora bien, como el Caso de Uso nunca se ejecuta fuera del contexto de otro Caso de Uso, decimos que es un Caso de Uso abstracto. Lo llamamos abstracto porque no es implementable por sí mismo; sólo tiene sentido como parte de otros casos.

De la misma forma, el actor que participa de este Caso de Uso, que reúne características comunes a todos los actores de los Casos de Uso que lo usan, es un actor abstracto. En nuestro ejemplo, podemos decir que tenemos un actor abstracto "Buscador de Datos de Producto". Los actores abstractos entonces, son necesarios para no dejar sin actores a los Casos de Uso abstractos.

Herencia

La duda ahora es cómo relacionar este actor abstracto con los actores concretos: los que sí existen en la realidad y ejecutan Casos de Uso concretos, como "Ingresando Pedido" y "Obteniendo Estadísticas de Ventas".

Para esto podemos usar el concepto de herencia, uno de los conceptos básicos de la orientación a objetos. Como todos los actores concretos también ejecutan el caso "Buscando Datos de Producto", a través de la relación de uso, podemos decir que los actores concretos heredan al actor abstracto.

La relación de herencia no necesariamente implica la existencia de un caso abstracto. Puede ocurrir que un actor ejecute todos los casos que ejecuta otro actor y algunos más. Por ejemplo, el supervisor de ventas puede hacer todo lo que hace el empleado de ventas, pero además puede autorizar pedidos. En este caso, podemos decir que el Supervisor de Ventas hereda al Empleado de Ventas, aunque el Empleado de Ventas no sea un actor abstracto. De esta forma, toda la funcionalidad que está habilitada para el Empleado de Ventas también lo está para el Supervisor.

2.3.7 El Proceso de Ingeniería de Requerimientos con Casos de Uso

A continuación, se describen los pasos a seguir para aplicar la técnica con Casos de Uso a la IR.

Identificar los Actores

Si la primera pregunta que un analista debe hacer a sus usuarios es ¿Para qué es este sistema?, la segunda es claramente ¿Para quiénes es este sistema? Como mencionamos al hablar sobre los actores, identificar a todos ellos es crítico para un buen análisis de requerimientos. Por lo tanto, antes de avanzar con los Casos de Uso, debo tratar de identificar todos los tipos de usuario diferentes que tiene el sistema. Si el sistema será implementado en una empresa, debo preguntar cuáles de las áreas afectadas usarán o actualizarán su información.

A pesar de hacer una identificación inicial de los actores, también debo repetirla a medida que empiezo a describir los Casos de Uso, ya que al conocer más detalles del sistema, pueden aparecer nuevos tipos de usuarios.

Identificar los Principales Casos de Uso de cada Actor

El siguiente paso es enunciar los nombres de los principales Casos de Uso de cada uno de los actores que se identificaron en el paso anterior. No es necesario especificar cuáles son las acciones dentro del Caso de Uso. Tampoco debo preocuparme si no aparecen muchos casos, ya que existen técnicas para encontrar nuevos Casos de Uso a partir de los existentes.

Identificar Nuevos Casos a Partir de los Existentes

Uno de los principales errores que se pueden cometer al identificar requerimientos es algo que parece obvio, pero que muchas veces ocurre: ¡olvidarse de algún requerimiento! Como los requerimientos están en la cabeza de los usuarios, el éxito de esta tarea depende de la habilidad del analista. Para ayudarnos a identificar nuevos Casos de Uso a partir de los casos existentes, podemos aplicar las mismas técnicas utilizadas para identificar eventos según el análisis estructurado. Algunas de las preguntas que debemos hacernos son:

- ¿Cuáles son las tareas de un actor?
- ¿Necesita el actor estar informado de ciertas ocurrencias del sistema?
- ¿Necesita el actor informar al sistema de cambios externos súbitos?
- ¿Proporciona el sistema el comportamiento correcto al negocio?
- ¿Pueden ser todos los requerimientos funcionales, desarrollados por los Casos de Uso?
- ¿Qué Casos de Uso soportarán y mantendrán al sistema?
- ¿Qué información debe ser modificada o creada?

Documentación de Casos de Uso

Una vez que identificamos todos los Casos de Uso, empezamos a documentar sus pasos; este documento se crea para cada Caso de Uso, detallando lo que el sistema debe proporcionar al actor cuando el caso

de uso es ejecutado. Esta tarea no es estrictamente secuencial de la anterior: es posible que mientras empezamos a documentar los casos, sigamos buscando otros nuevos.

Un contenido típico de un documento de Caso de Uso sería:

- Describir cómo comienza el Caso de Uso y cómo termina.
- Realizar un flujo normal de eventos.
- Realizar un flujo alterno de eventos.
- Detallar las excepciones al flujo de eventos.

Definir Prioridades

Una vez documentados los Casos de Uso, es conveniente definir las prioridades de los distintos requerimientos, expresados como Casos de Uso. Para los escenarios claves y los Casos de Uso que serán analizados en su iteración, se debe:

- Representar la funcionalidad central de cada Caso de Uso.
- Cubrir varios aspectos de arquitectura.
- Poner bajo estrés un punto delicado de la arquitectura.

Gráficos a Utilizar

Dependiendo del tamaño del sistema, es probable que un único gráfico con todos los Casos de Uso nos quede chico. No olvidemos que los modelos gráficos son para aclarar el texto y no para confundir. Si el gráfico de Casos de Uso es una maraña indescifrable, no está cumpliendo su objetivo. Por lo tanto, podemos usar las siguientes reglas para incluir gráficos de Casos de Uso dentro de la SRS.

- Un gráfico de Casos de Uso no debe mostrar más de 15 casos.
- Si es necesario particionar el gráfico, debe hacerse por actor. La primera partición debe separar los casos centrales de los casos auxiliares, ya que probablemente les interesen a personas distintas.
- Si las relaciones de uso y las extensiones entran en el diagrama principal, sin dejar de cumplir con la regla 1, debo dejarlas ahí. Lo mismo se aplica a los actores abstractos.
- Si las relaciones de uso no entran en el diagrama principal, debo mostrarlas en gráficos teniendo en cuenta que siempre debo mostrar todos los Casos de Uso que usan a otro en un mismo diagrama.
- Si tengo un Caso de Uso que es usado por gran parte de los otros casos, como por ejemplo el Caso de Uso "Identificándose ante el sistema", debo evitar mostrarlo en el gráfico principal, ya que las flechas serán imposibles de organizar. Es probable que no haga falta mostrar esta relación de uso en un gráfico.

2.3.8 Herramientas automatizadas para la Administración de Requerimientos

Hoy en día, la Ingeniería de Software cuenta con una serie de herramientas automatizadas destinadas a diferentes propósitos. Dentro de las herramientas CASE que sirven de apoyo a los procesos de Ingeniería de Software, están las especializadas en la administración de requisitos. Estas herramientas se concentran en capturar requerimientos, administrarlos y producir una especificación de requisitos.

Las ventajas que nos proporcionan las herramientas automatizadas para IR son:

- Permiten un mayor control de proyectos complejos.
- Permiten reducir costos y retrasos en la liberación de un proyecto.
- Permiten una mayor comunicación en equipos de trabajo.
- Ayudan a determinar la complejidad del proyecto y esfuerzos necesarios.

En este capítulo veremos dos de las herramientas más utilizadas para este propósito: RequisitePro y DOORS.

RequisitePro

RequisitePro(R) es la herramienta que ofrece Rational Software para tener un mayor control sobre los requerimientos planteados por el usuario y todos aquellos requerimientos técnicos o nuevos requerimientos de usuario que surjan durante el ciclo de vida del proyecto.

Con RequisitePro los requerimientos se encuentran documentados bajo un esquema organizado de documentos; estos esquemas, cumplen completamente con los estándares requeridos por IEEE, ISO, SEI, CMM y por el Rational Unified Process.

RequisitePro se integra con aplicaciones para la administración de cambios, herramientas de modelado de sistemas y con herramientas de pruebas. Esta integración asegura que los diseñadores conocen los requerimientos del usuario, del sistema y del software en el momento de su desarrollo. RequisitePro permite el desarrollo en equipo, vía el check-in y check-out de los documentos involucrados en el proyecto. Con esta integración, se pueden conservar juntos todos los requerimientos y ser manipulados por todos y cada uno de los miembros del equipo.

Todos los requerimientos tienen atributos y estos son la principal fuente de información para ayudarle a planear, comunicar y rastrear las actividades del proyecto a través del ciclo de vida. Cada proyecto tiene necesidades únicas y se deberán seleccionar los atributos que sean críticos para asegurar su éxito: prioridad de desarrollo, status, autor, responsable, relaciones, fecha de registro, fecha última modificación, versión, etc.

RequisitePro permite la asignación de prioridades a los requerimientos en base a:

- **Beneficios al cliente:** Todos los requerimientos no son desarrollados de igual forma. Se les da prioridad en base a la importancia relativa del usuario final basado en un análisis previo de los analistas y desarrolladores del equipo.

- **Esfuerzo:** Claramente, algunos requerimientos o cambios a éstos demandan más tiempo y recursos que otros. Estimar el número de semanas-personas o líneas de código requeridas en base a requerimientos, es la mejor forma de determinar "que" y "que no" puede ser desarrollado en el tiempo estipulado.

Una característica importante es que la curva de aprendizaje de RequisitePro es pequeña, este aprendizaje puede ser basado en el uso de los tutoriales, los cuales guían por los puntos principales en el uso de la herramienta.

Beneficios de RequisitePro

- Permite el trabajo en equipo por medio de un repositorio compartido de información.
- Permite la clasificación de requerimientos, en base a las necesidades de cada empresa: usuario, técnicos, comunicación, pruebas.
- Define atributos para todos los tipos de requerimientos especificados.
- Ayuda a manipular el alcance del proyecto mediante la asignación de prioridad de desarrollo a cada uno de los requerimientos planteados.
- Características avanzadas de rastreo por medio de matrices, que permiten visualizar las dependencias entre requerimientos dentro de un proyecto o en diferentes proyectos.
- Marcas que automáticamente indican cuándo un requerimiento es impactado por cambios a otro requerimiento o a atributos asociados.
- Administración de cambios mediante el rastreo y la visualización histórica de los cambios efectuados al requerimiento, cuándo y quién los realizó.
- Manejo de plantillas creadas por el usuario o creadas por otras empresas.
- Recolección de requerimientos mediante su importación por medio de Wizards para obtenerlos automáticamente de fuentes externas, incluyendo archivos o bases de datos.
- Interactúa con los demás productos Rational para el ciclo de vida, así como con herramientas de Microsoft Office.
- Ayuda a determinar en forma automatizada cuántos requerimientos tiene el proyecto.
- Ayuda a determinar responsables y actores en cada uno de los requerimientos.
- RequisitePro, le permite organizar sus requerimientos, establecer y mantener relaciones padre/hijo entre ellos.

Doors.

DOORS(R) es la herramienta de administración de requisitos creada por Quality Systems and Software. Esta herramienta permite capturar, relacionar, analizar y administrar un rango de información para asegurar el cumplimiento del proyecto en materia de requerimientos.

DOORS permite el acceso de un gran número de usuarios concurrentes en la red, manteniendo en línea un gran número de requerimientos así como su información asociada.

DOORS ayuda al usuario a procesar las solicitudes de cambios de requerimientos en línea. Permite realizar cualquier modificación vía remota cuando la base de datos está off-line, incorporando sus actualizaciones a la base de datos maestra. Esto hace más fácil la comunicación del equipo con otras organizaciones, subcontratistas y proveedores.

Esta herramienta proporciona rastreabilidad multi-nivel para aquellas relaciones entre requerimientos que poseen gran tamaño. DOORS cuenta con un wizard que le permite generar enlaces a reportes de muchos niveles, para desplegarlos en la misma vista.

Beneficios de DOORS

- Análisis y comparación de requerimientos.
- Clasificación de requerimientos.
- Interpretación manual de cada requerimiento.
- Identificación de inconsistencias.
- Operación vía batch.
- Permite compartir requerimientos entre proyectos.
- Permite crear relaciones entre requerimientos mediante la táctica drag-and-drop.
- Envía una notificación vía e-mail cuando los cambios son revisados.
- Permite visualizar los cambios pendientes de otros usuarios para anticipar el impacto que ocasionará.
- Despliega estadísticas y métricas a través de gráficas.
- Los documentos están escritos en lenguaje claro, lo que proporciona una comprensión inmediata de cada requerimiento.
- Permite importar sus documentos a formatos de herramientas de Microsoft Office, RTF, HTML, texto, entre otros.
- Las plantillas presentan la información de manera estandarizada.

2.3.9 Análisis comparativo de las técnicas de Ingeniería de Requerimientos

En este capítulo se presentan las principales ventajas y desventajas de cada una de las técnicas utilizadas en las etapas de la Ingeniería de Requerimientos.

Técnica	Ventajas	Desventajas
Entrevistas y Cuestionarios	• Mediante ellas se obtiene una gran cantidad de información correcta a través del usuario. • Pueden ser usadas para obtener una visión rápida del dominio del problema. • Son flexibles. • Permiten combinarse con otras técnicas.	• La información obtenida al principio puede ser redundante o incompleta. • Si el volumen de información manejado es alto, requiere mucha organización de parte del analista, así como la habilidad para tratar y comprender el comportamiento de todos los involucrados.
Lluvia de Ideas	• Los diferentes puntos de vista y las confusiones en cuento a terminología, son aclaradas por expertos. • Ayuda a desarrollar ideas unificadas basadas en la experiencia de un experto.	• Es necesaria una buena compenetración del grupo participante.
Prototipos	• Ayudan a validar y desarrollar nuevos requerimientos. • Permite comprender aquellos requerimientos que no están muy claros y que son de alta volatilidad.	• El cliente puede llegar a pensar que el prototipo es una versión del software que será desarrollado. • A menudo, el desarrollador hace compromisos de implementación con el objetivo de acelerar la puesta en funcionamiento del prototipo
Análisis Jerárquico	• Permite determinar el grado de importancia de cada requerimiento. • Ayuda a identificar conflictos en los requerimientos. • Muestra el orden en que deben ser implementados los requerimientos.	• Debe construirse un estándar claro de evaluación, que incluya la participación del cliente.
Casos de Uso	• Representan los requerimientos desde el punto de vista del usuario. • Permiten representar más de un rol para cada afectado. • Identifica requerimientos estancados, dentro de un conjunto de requerimientos.	• En sistemas grandes, toma mucho tiempo definir todos los Casos de Uso. • El análisis de calidad depende de la calidad con que se haya hecho la descripción inicial.

Tabla 8.- Comparativo de las técnicas de la Ingeniería de Requerimientos.

En base a las ventajas y desventajas mostradas anteriormente, se hace una comparación entre algunas de las técnicas.

Entrevistas vs. Casos de Uso: Un alto porcentaje de la información recolectada durante una entrevista, puede ser usada para construir Casos de Uso. Mediante esto, el equipo de desarrollo puede entender mejor el ambiente de trabajo de los involucrados. Cuando el analista sienta que tiene dificultades para entender una tarea, pueden recurrir al uso de un cuestionario y mostrar los detalles recabados en un Caso de Uso. De hecho, durante las entrevistas cualquier usuario puede utilizar diagramas de Casos de Uso para explicar su entorno de trabajo.

Entrevistas vs. Lluvia de Ideas: Muchas de las ideas planteadas en el grupo, provienen de información recopilada en entrevistas o cuestionarios previos. Realmente la lluvia de ideas trata de encontrar las dificultades que existen para la comprensión de términos y conceptos por parte de los participantes; de esta forma se llega a un consenso.

Casos de Uso vs. Lluvia de Ideas: La lista de ideas proveniente del Brainstorm puede ser representada gráficamente mediante Casos de Uso.

La siguiente tabla muestra las técnicas que pueden ser utilizadas en las diferentes actividades de la IR.

	Análisis del Problema	Evaluación y negociación	Especificación de Requisitos	Validación	Evolución
Entrevistas y Cuestionarios	X				X
Lluvia de Ideas		X			X
Prototipos				X	
Análisis Jerárquico		X			X
Casos de Uso	X		X		X

Tabla 9.- Técnicas usadas en las diferentes fases de la Ingeniería de Requerimientos.

2.3.10 Requisitos de ISO 9001

La Norma internacional ISO-9001 especifica los requisitos que debe cumplir un sistema de calidad cuando es necesario demostrar la capacidad de un proveedor para diseñar y suministrar productos conformes. La lista de esos 20 requisitos o direcciones evaluadas es:

1) Responsabilidad de la dirección

- Política de Calidad.
- Organización.
- Responsabilidad y autoridad.
- Recursos.
- Representante de la Dirección.
- Revisión por la Dirección.

2) Sistema de Calidad

- Generalidades.
- Procedimientos.
- Planificación de calidad.

3) Revisión de contratos

- Revisión.
- Modificación de contratos.
- Registros.

4) Control de diseño

- Planificación del diseño y del desarrollo.
- Interfaces organizativas y técnicas.
- Elementos de entrada.
- Elementos de salida.
- Revisión del diseño.
- Verificación del diseño.
- Validación del diseño.
- Control de cambios.

5) Control de documentación y de los datos

- Aprobación y distribución de documentos y de datos.
- Cambios en documentos y datos.

6) Compras

- Evaluación de subcontratistas.
- Datos sobre las compras.
- Verificación de los productos comprados.
- Verificación en la casa del proveedor realizadas por el comprador.
- Verificación realizada por el cliente sobre los productos que subcontrata el proveedor.

7) Control sobre los productos que suministramos.

8) Identificación y capacidad de rastrear del producto.

9) Control de procesos.

10) Inspección y ensayos.

- En la recepción.
- En el proceso.
- Finales.
- Registros, inspección y ensayos.

11) Control de equipos de inspección, medición y ensayos.

- Procedimientos de control.

12) Estado de inspección y ensayo.

13) Control de productos no-conformes.

- Revisión y tratamiento.

14) Acciones correctivas y preventivas.

15) Manipulación, almacenamiento, embalaje, preservación y entrega.

16) Control de registros de calidad.

17) Auditorías internas.

18) Entrenamiento.

19) Servicio posventa.

20) Técnicas estadísticas.

- Identificación de la necesidad.
- Procedimientos.

* La norma UNIT-ISO 9004 contempla otros dos elementos adicionales a los 20 anteriores:

21) Consideraciones financieras de los sistemas de calidad.

22) Seguridad de los productos.

2.4 Diseño

Después de varias décadas en las que los Ingenieros de Software crearon sus propios diseños desde el principio, o reusaron diseños que por azar habían visto, ahora surge la disciplina de Diseño y Arquitectura de Software. Cada vez más es posible expresar diseño de alto y bajo nivel con términos que son comunes a todos los Ingenieros de Software profesionales.

Panorama Global

Las aplicaciones requieren de manera invariable, componentes tanto de software como hardware. Un ejemplo es un sistema de seguridad de frenos con partes mecánicas, electrónicas y de software, que trabajan juntas para maximizar el frenado sin que el vehículo patine. Un sistema de conversación interactiva en Internet es otro ejemplo de un sistema con componentes de hardware y software.

La Ingeniería de Software es el proceso de diseño y análisis que desglosa una aplicación en hardware y

software. Algunos aspectos de esta descomposición pueden ser requerimientos del cliente y otros de los Ingenieros de Software.

El proceso de Ingeniería de Sistemas comienza con los requerimientos totales del sistema, hace trueques entre hardware y software y determina un desglose de éstos. Después, el proceso de Ingeniería de Software se aplica a las partes de software, comenzando con el análisis de requerimientos, etcétera.

Las aplicaciones inmersas presentan el mayor reto para la Ingeniería de Sistemas, porque su tiempo de respuesta casi siempre es crítico. El Ministerio de Defensa de Estados Unidos es un usuario antiguo de la Ingeniería de Sistemas avanzada debido a la extensa integración de software/hardware requerida por los sistemas de defensa. Los contratistas en esta área emplean numerosos Ingenieros en Sistemas. Estos ingenieros desarrollan los requerimientos del sistema y realizan estudios para evaluar las configuraciones apropiadas de hardware y software.

El IEEE y otras organizaciones de normas han publicado estándares para Ingeniería de Sistemas como IEEE P1233 "Guía para el desarrollo de especificaciones de requerimientos de sistemas". Esta incluye amplias consideraciones como las dimensiones y el peso de componentes mecánicas, limitaciones de recursos, restricciones ambientales, desempeño, funcionalidad, compatibilidad, confiabilidad, mantenimiento y producibilidad.

2.4.1 Significado de Arquitectura de Software

Si se compara la Ingeniería de Software con la construcción de un puente, el proceso de análisis de requerimientos es como decidir dónde debe comenzar el puente, dónde debe terminar y qué tipo de carga debe soportar. Siguiendo la analogía, el diseñador del puente tendrá que decidir si elige un puente suspendido, de vigas voladas, de cables o algún otro tipo para satisfacer los requerimientos. Estas son las arquitecturas del puente. Los Ingenieros de Software se enfrentan a procesos de decisión similares. Aquí se describirán las opciones de las arquitecturas para los Ingenieros de Software.

"Arquitectura" es equivalente a "diseño en el más alto nivel". Se hará referencia al resto del proceso de diseño como "diseño detallado".

La especificación clara de las arquitecturas de software, importantes para todas las aplicaciones, es indispensable en trabajos de desarrollo con más de una persona. Esto se debe a que las aplicaciones grandes deben diseñarse e implementarse en partes ("módulos") y después ensamblarse. La selección de la arquitectura proporciona estos módulos. Los ingenieros encargados de la tarea de desarrollar la arquitectura, los "arquitectos técnicos o de software", por lo común son los ingenieros con más experiencia.

Metas de la selección de la arquitectura

Para un proyecto de desarrollo de software dado pueden existir varias arquitecturas adecuadas para elegir; decidir cuál es la mejor depende de las metas. Suele ser difícil satisfacer todas las metas, ya que un diseño que satisface una puede no satisfacer otra. Por esto se les asignan prioridades. La siguiente lista enumera las metas de diseño más importantes:

- **Extensión.**
  - Facilitar la adición de nuevas características.
- **Cambio.**
  - Facilitar los cambios de requerimientos.
- **Sencillez.**
  - Hacerlo de fácil comprensión.
  - Hacerlo de fácil implementación.
- **Eficiencia.**
  - Lograr alta velocidad: ejecución y/o compilación.
  - Lograr un tamaño pequeño: tiempo de corrida y/o código base.

La meta "extensión" describe el grado en el que se desea introducir nuevas características. Con frecuencia, si la arquitectura debe permitir la adición sencilla de nuevas características, entonces se requiere más trabajo de diseño y el diseño obtenido es más complejo. Casi siempre esto implica introducir más abstracción en el proceso. Existen muchas ventajas en la generalidad, pero requiere una inversión de tiempo. Una tarea importante al establecer la generalidad es evaluar los tipos de extensiones que surgirán. No se puede diseñar para todas las extensiones posibles. Aquí es donde los requerimientos "opcionales" y "deseables" son útiles porque señalan hacia donde se dirige la aplicación.

Diseñar para el cambio es diferente de diseñar para la extensión, aunque las técnicas de diseño suelen ser similares.

La sencillez es una meta de diseño en todas las circunstancias. Las arquitecturas sencillas que permiten extensiones y cambios son raras y muy anheladas. Otras metas para elegir la arquitectura incluyen el uso eficiente de ciclos de CPU y/o espacio.

Descomposición

Con suficiente práctica no es difícil escribir programas pequeños; sin embargo, las aplicaciones grandes presentan problemas muy diferentes y se ha encontrado que es difícil crearlos en la práctica. El problema principal de los sistemas de software es la complejidad; no el número de líneas de código en sí, si no sus interrelaciones. Un arma muy buena contra la complejidad es desglosar el problema para que tenga las características de un programa pequeño. Entonces, la descomposición (o "modularización") del problema tiene una importancia crítica y es uno de los mayores retos. El diseñador debe formar un modelo mental de cómo funcionará la aplicación a un alto nivel, después desarrollar un desglose que se ajuste a este modelo mental. Por ejemplo, ¿qué cuatro o cinco módulos englobarán un juego de video? O ¿qué cinco o seis módulos deben usarse para descomponer una aplicación de finanzas personales? El siguiente problema se convierte en el desglose de las componentes resultantes, etcétera. Algunas veces, este proceso recibe el nombre de "diseño recursivo".

Se inicia con las metas de descomposición del software.

"Cohesión" dentro de un módulo es el grado de comunicación entre los elementos del módulo.

"Acoplamiento" es el grado en el que los módulos se comunican con otros módulos. La modularización efectiva se logra al maximizar la cohesión y minimizar el acoplamiento. Esto hace posible descomponer tareas complejas en otras más sencillas. La siguiente figura sugiere las metas de acoplamiento/cohesión con una arquitectura idealizada para un puente en el que cada una de las seis componentes tiene una gran cohesión y el acoplamiento entre ellas es bajo.

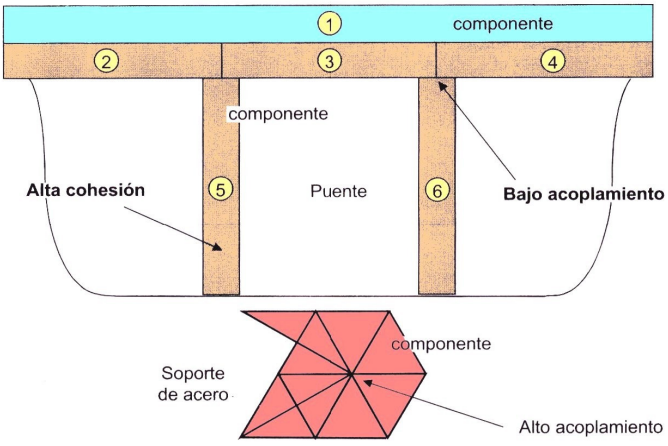


Fig. 36.- Cohesión y acoplamiento.

Las partes (como los ladrillos) de cada componente del puente (como las columnas) son mutuamente dependientes. Esta es la cohesión. Cada componente depende justo de algunas otras. Por ejemplo, cada columna está acoplada sólo con dos piezas transversales. Eso es acoplamiento bajo. El "soporte de acero" en la figura muestra ocho componentes que dependen entre sí en algún lugar. Este es un alto grado relativo de acoplamiento.

Bajo acoplamiento y alta cohesión son en particular importantes en el diseño de software debido a la necesidad continua de modificación de las aplicaciones. Compare el ciclo de vida de una aplicación de software típica con el de un puente. La necesidad de modificación suele ser más probable en el software que en el puente. Es más sencillo modificar las arquitecturas de bajo acoplamiento y alta cohesión, ya que los cambios tienden a tener efectos locales sobres ellos. Sin embargo, no es tan fácil lograr estos tipos de arquitecturas.

El número de paquetes de alto nivel en una arquitectura debe ser pequeño. Un número de "7$\pm$2" es una guía útil, aunque los proyectos específicos pueden variar mucho de este intervalo por razones especiales. La diferencia entre proyectos de gran escala y pequeña escala es la cantidad de módulos o paquetes anidados o inmersos. Los proyectos de gran escala organizan cada paquete de alto nivel en subpaquetes, éstos en sub-subpaquetes, etcétera. La guía de "7$\pm$2" se aplica a cada una de las descomposiciones.

La descomposición perfecta es una meta valiosa pero difícil de lograr. El software no es el único tipo de ingeniería en la que es difícil lograr módulos limpios y claros. Shnayerson [Sh1] demostró que a pesar de los mejores intentos de General Motors para modularizar el diseño de su primer auto eléctrico de

producción EVI, los factores de forma y ajuste (el requerimiento de ajustar las partes funcionales del auto en espacios pequeños) forzaron un acoplamiento demasiado alto entre los componentes.

Descomponer un diseño en sus componentes es un paso esencial, pero se requiere mucho más trabajo de arquitectura. Primero es necesario coordinar los Casos de Uso, las clases, las transiciones de estados y la descomposición. Todo esto se conoce como modelos de perspectivas.

Al idear el modelo de clases con frecuencia es aconsejable desarrollar o usar la colección de software preexistente que forma la base de una familia de aplicaciones similares. Esta familia, llamada marco de trabajo, se describe más adelante.

El diseño detallado consiste en todo el trabajo de diseño que excluye la arquitectura por un lado y la implementación por el otro. Incluye definir las clases que conectan las clases de dominio con las clases de arquitectura.

En lugar de "reinventar la rueda" se intenta reusar diseños probados como efectivos en aplicaciones anteriores. Los patrones de diseño son patrones de clases y métodos interrelacionados que han demostrado su valor en muchas aplicaciones. Un ejemplo es una colección de clases que forma un árbol de objetos. Los patrones de diseño se aplican tanto a nivel arquitectura como a nivel de diseño detallado.

2.4.2 Uso de "Modelos"

Por lo común es necesario describir las aplicaciones desde varias perspectivas. Esto se puede comparar con la arquitectura de una casa, que requiere múltiples perspectivas como planos, vista vertical, vista frontal, plano de plomería, etcétera. En el mundo del software, las perspectivas se llaman Modelos. Hubo un gran desarrollo en este campo durante los últimos años del siglo XX. A continuación se muestran varios Modelos, muchos tomados del método Proceso Unificado de Desarrollo de Software (USDM) (Jacobson, [Ja1] y Kruchten [Kr]):

- **Modelo de Casos de Uso:** Es una colección de Casos de Uso. Estos dicen mucho de lo que la aplicación intenta hacer. Las versiones iniciales de los Casos de Uso son adecuadas como requerimientos C (en ocasiones llamados "Casos de Uso del negocio"). Conforme avanza el proyecto, se expresan como diagramas de secuencia en formas cada vez más específicas. Se realizan como escenarios específicos, que se usan para las pruebas.
- **Modelo de Clases**: Se ha visto una buena parte del modelo de clases (Diagramas de Clases). Estos modelos explican los bloques de construcción con los que se construirá la aplicación. Los modelos de clases con frecuencia se denominan "Modelos de Objetos". Dentro de los Modelos de Clases se pueden mostrar los métodos y atributos.
- **Modelo de Componentes**: Es una colección de diagramas de flujo de datos. Describen la manera en que la aplicación debe realizar su trabajo en términos de mover datos.
- **Modelo de Estados:** Es la colección de diagramas estado/transición. Estos modelos describen cuándo la aplicación hace el trabajo.

Dentro de estos modelos se proporcionan niveles cada vez mayores de detalle. Según el tamaño del trabajo puede ser que se aplique, de una manera recursiva, uno o más niveles de detalle dentro de cada modelo. La arquitectura de una aplicación a menudo se expresa en términos de uno de los modelos y está

apoyada por uno o más de los modelos restantes. El Proceso Unificado de Desarrollo de Software (USDP) incluye el modelo de "implementación", que tiene que ver con la manera en que se organiza el código. Cada arquitectura tiene al menos un modelo de clases capaz de implementarlo.

2.4.3 Lenguaje de Modelado Unificado (UML)

Lenguaje Unificado de Modelado (**UML**, por sus siglas en inglés, "Unified Modeling Language") es el lenguaje de modelado de sistemas de software conocido y utilizado en la actualidad; aún cuando todavía no es un estándar oficial, está apoyado en gran medida por el OMG (Object Management Group). Es un lenguaje gráfico para visualizar, especificar, construir y documentar un sistema de software. UML ofrece un estándar para describir un "plano" del sistema (modelo), incluyendo aspectos conceptuales tales como procesos de negocios y funciones del sistema y aspectos concretos como expresiones de lenguajes de programación, esquemas de bases de datos y componentes de software reutilizables.

El punto importante para notar aquí es que UML es un "lenguaje" para especificar y no un método o un proceso. UML se usa para definir un sistema de software; para detallar los artefactos en el sistema; para documentar y construir -es el lenguaje en el que está descrito el modelo. UML se puede usar en una gran variedad de formas para soportar una metodología de desarrollo de software (tal como el Proceso Unificado de Rational) pero no especifica en sí mismo qué metodología o proceso usar.

UML cuenta con varios tipos de diagramas, los cuales muestran diferentes aspectos de las entidades representadas.

En UML 2.0 hay 13 tipos de diagramas. Para comprenderlos, a veces es útil catalogarlos jerárquicamente, como se muestra en la figura siguiente.

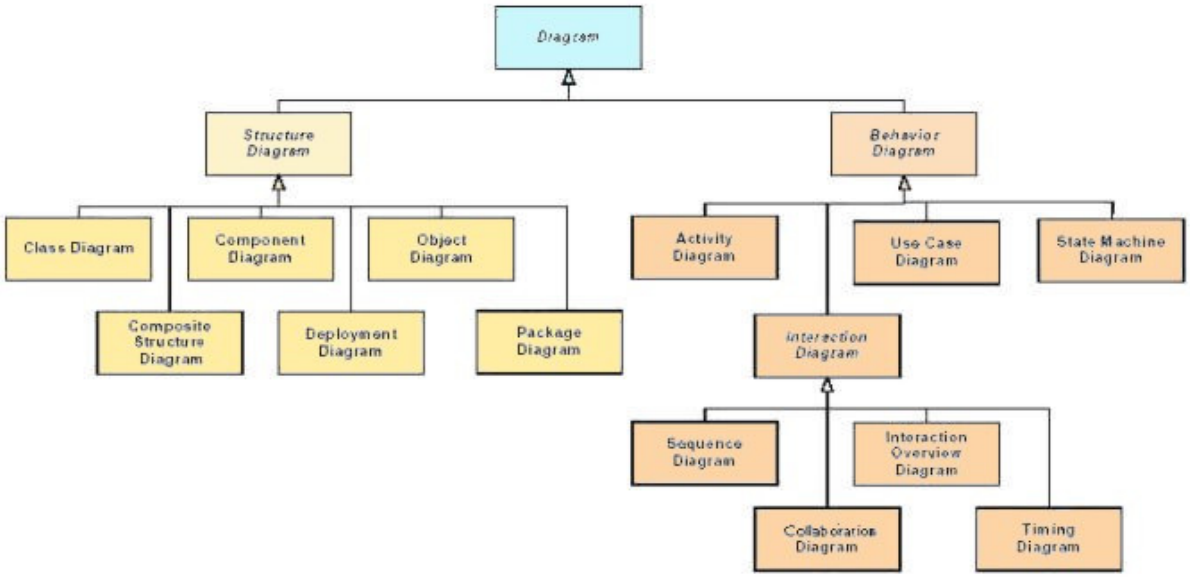


Fig. 37.- Diagramas usados en UML.

2.4.3.1 Diagramas

Los diagramas se pueden clasificar en los siguientes tipos:

Diagramas de estructura: Enfatizan en los elementos que deben existir en el sistema modelado.

- Diagrama de Clases.
- Diagrama de Componentes.
- Diagrama de Objetos.
- Diagrama de Estructura Compuesta (UML 2.0).
- Diagrama de Despliegue.
- Diagrama de Paquetes.

Diagramas de comportamiento: Enfatizan en lo que debe suceder en el sistema modelado.

- Diagrama de Actividades.
- Diagrama de Casos de Uso.
- Diagrama de Estados.

Diagramas de Interacción: Un subtipo de diagramas de comportamiento, que enfatiza sobre el flujo de control y de datos entre los elementos del sistema modelado:

- Diagrama de Secuencia.
- Diagrama de Colaboración.
- Diagrama de Tiempos (UML 2.0).
- Diagrama de Vista de Interacción (UML 2.0).

Diagramas de Clases

Los Diagramas de Clases son utilizados durante el proceso de análisis y diseño de los sistemas informáticos, donde se crea el diseño conceptual de la información que se manejara en el sistema, los componentes que se encargaran del funcionamiento y la relación entre uno y otro.

Algunas definiciones

Propiedades: Valores que corresponden a un objeto, como color, material, cantidad, ubicación. Generalmente se conoce como la información detallada del objeto.

Operaciones: Son aquellas actividades o verbos que se pueden realizar con/para este objeto, como por ejemplo abrir, cerrar, buscar, cancelar, acreditar, cargar.

Interfaz: Es un conjunto de operaciones y/o propiedades que permiten a un objeto comportarse de cierta manera, por lo que define los requerimientos mínimos del objeto.

Herencia: Se define como la reutilización de un objeto padre ya definido para poder extender la funcionalidad en un objeto hijo. Los objetos hijos heredan todas las operaciones y/o propiedades de un objeto padre. Por ejemplo: Una persona puede subdividirse en Proveedores, Acreedores, Clientes, Accionistas, Empleados; todos comparten datos básicos como una persona, pero además tendrán información adicional que depende del tipo de persona, como saldo del cliente, total de inversión del accionista, salario del empleado, etc.

Al diseñar una clase debemos pensar en cómo podemos identificar un objeto real, como una persona, un transporte, un documento o un paquete. Estos ejemplos de clases de objetos reales, es sobre lo que un sistema se diseña. Durante el proceso del diseño de las clases tomamos las propiedades que identifican como único al objeto y otras propiedades adicionales como datos que corresponden al objeto. Con los siguientes ejemplos formaremos tres objetos que se incluyen en un diagrama de clases:

Ejemplo 1

Una persona tiene un número de documento de identificación, nombres, apellidos, fecha de nacimiento, género, dirección postal, posiblemente también tenga número de teléfono de casa, del móvil, fax y correo electrónico.

Ejemplo 2

Un sistema informático puede permitir el manejo de la cuenta bancaria de una persona, por lo que tendrá un número de cuenta, número de identificación del propietario de la cuenta, saldo actual, moneda en la que se maneja la cuenta.

Ejemplo 3

Otro objeto puede ser "Manejo de Cuenta", donde las operaciones bancarias de una cuenta (como en el ejemplo 2) se manejaran realizando diferentes operaciones que en el diagrama de clases solo se representan como operaciones, que pueden ser:

- Abrir.
- Cerrar.
- Depósito.
- Retiro.
- Acreditar Intereses

Estos ejemplos constituyen diferentes clases de objetos que tienen propiedades y/u operaciones que contienen un contexto y un dominio, los primeros dos ejemplos son clases de datos y el tercero clase de lógica de negocio, dependiendo de quién diseñe el sistema se pueden unir los datos con las operaciones.

El diagrama de clases incluye mucha más información como la relación entre un objeto y otro, la herencia de propiedades de otro objeto, conjuntos de operaciones/propiedades que son implementadas para una interfaz, etc.

Diagrama de Componentes

Un Diagrama de Componentes es un diagrama tipo del Lenguaje Unificado de Modelado.

Los componentes pertenecen al mundo material de los bits y por lo tanto, son un bloque de construcción importante cuando se modelan los aspectos físicos de un sistema. Un componente es una parte física y reemplazable de un sistema que se conforma con un conjunto de interfaces y proporciona la realización de dicho conjunto.

Los componentes se utilizan para modelar los elementos físicos que pueden hallarse en un nodo, tales como ejecutables, bibliotecas, tablas, archivos y documentos. Normalmente, un componente presenta el empaquetamiento físico de elementos que por el contrario son lógicos, tales como clases, interfaces y colaboraciones.

Los buenos componentes definen abstracciones precisas con interfaces bien definidas, permitiendo reemplazar fácilmente los componentes más viejos con otros más nuevos y compatibles.

El producto final de una constructora es un edificio tangible existente en el mundo real. Los modelos lógicos se construyen para visualizar, especificar y documentar las decisiones sobre la construcción: la localización de las paredes, puertas y ventanas; la distribución de los sistemas eléctricos y de plomería; y el estilo arquitectónico global. A la hora de construir el edificio, esas paredes, puertas, ventanas y demás elementos conceptuales se convierten en cosas reales, físicas.

Tanto la vista lógica como la física son necesarias. Si se va a construir un edificio desechable para el que el costo de destruir y reconstruir es prácticamente cero (por ejemplo, si se construye una casa para un perro), probablemente se pueda abordar la construcción física sin hacer ningún modelado lógico. Si por otro lado, se está construyendo algo duradero y para lo cual el costo de cambiar o fallar es alto, entonces lo más práctico para manejar el riesgo es crear tanto los modelos lógicos como los físicos.

Lo mismo ocurre cuando se construye un sistema con gran cantidad de software. El modelado lógico se hace para visualizar, especificar y documentar las decisiones acerca del vocabulario del dominio y sobre cómo colaboran estos elementos tanto estructuralmente como desde el punto de vista del comportamiento. El modelado físico se hace para construir el sistema ejecutable. Mientras los elementos lógicos pertenecen al mundo conceptual, los elementos físicos pertenecen al mundo de los bits (es decir, en última instancia se encuentran en nodos físicos y pueden ser ejecutados directamente o formar parte de alguna forma indirecta, de un sistema ejecutable).

En UML, todos estos elementos físicos se modelan como componentes. Un componente es un elemento físico que conforma con un conjunto de interfaces y proporciona la realización de esas interfaces. Por lo tanto, las interfaces enlazan los modelos lógico y físico. Por ejemplo, se puede especificar una interfaz por una clase en un modelo lógico y esa misma interfaz se trasladará a algún componente físico que la realice.

En el terreno del software, muchos sistemas operativos y lenguajes de programación soportan directamente el concepto de componente. Las bibliotecas de código objeto, los ejecutables, los componentes COM+ y los Enterprise Java Beans son todos ejemplos de componentes que se pueden representar directamente en UML mediante componentes. No sólo se pueden utilizar los componentes para modelar estos tipos de elementos, sino también pueden utilizarse para representar otros elementos que participan en un sistema en ejecución, tales como tablas, archivos y documentos.

UML proporciona una representación gráfica de un componente, como se muestra en la siguiente figura.

Esta notación canónica permite visualizar un componente de forma independiente de cualquier sistema

operativo o lenguaje de programación. Con los estereotipos, uno de los mecanismos de extensibilidad de UML, se puede particularizar esta notación para representar tipos específicos de componentes.

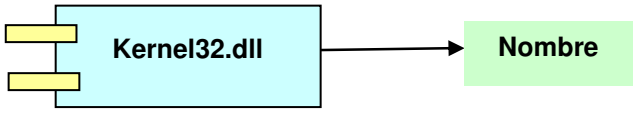


Fig. 38.- Componentes.

Un componente es una parte física y reemplazable de un sistema que se conforma con un conjunto de interfaces y proporciona la realización de estas interfaces. Gráficamente, un componente se representa como un rectángulo con pestañas.

Un diagrama de componentes representa la separación de un sistema de software en componentes físicos (por ejemplo archivos, cabeceras, módulos, paquetes, etc.) y muestra las dependencias entre estos componentes.

Muestra la organización y las dependencias entre un conjunto de componentes. Los diagramas de componentes cubren la vista de implementación estática de un sistema. Se relacionan con los Diagramas de Clases en que un componente se corresponde por lo común, con una o más clases, interfases o colaboraciones.

Diagrama de Objetos

Muestra un conjunto de objetos y sus relaciones. Los Diagramas de Objetos representan instantáneas de instancias de los elementos encontrados en los Diagramas de Clases. Estos diagramas cubren la vista de diseño estática o la vista estática de procesos de un sistema como lo hacen los Diagramas de Clases, pero desde la perspectiva de casos reales o prototípicos.

Los Diagramas de Objetos modelan las instancias de los elementos contenidos en los Diagramas de Clases. Un Diagrama de Objetos muestra un conjunto de objetos y sus relaciones en un momento concreto.

Los Diagramas de Objetos se utilizan para modelar la vista de diseño estática o la vista de procesos estática de un sistema. Esto conlleva el modelado de una instantánea del sistema en un momento concreto y la representación de un conjunto de objetos, su estado y sus relaciones.

Los Diagramas de Objetos no sólo son importantes para visualizar, especificar y documentar modelos estructurales, sino también para construir los aspectos estáticos de sistemas a través de ingeniería directa e inversa.

Si uno no conoce bien el juego, el fútbol parece un deporte muy sencillo (un grupo incontrolado de personas corriendo locamente sobre un terreno de juego persiguiendo una pelota). Al mirar la difusa imagen de cuerpos en movimiento, difícilmente parece que haya alguna sutileza o estilo en ello.

Si se congela el movimiento un momento y se clasifican los jugadores individuales, aparece una imagen muy diferente del juego. Ya no es una masa de gente, sino que se pueden distinguir delanteros, centrocampistas y defensas. Profundizando un poco se entenderá como colaboran los jugadores, siguiendo estrategias para conseguir el gol, mover el balón, robar el balón y atacar. En un equipo ganador no se encontraran jugadores colocados al azar en el campo.

En realidad, en cualquier momento del juego, su colocación en el campo y sus relaciones con otros jugadores están bien definidas.

Algo parecido ocurre al intentar visualizar, especificar, construir o documentar un sistema con gran cantidad de software. Si se fuera a trazar el flujo de control de un sistema en ejecución, rápidamente se perdería la visión de conjunto de cómo se organizan las partes del sistema, especialmente si se tienen varios hilos de control. Análogamente, si se tiene una estructura de datos compleja, no ayuda mucho mirar simplemente el estado de un objeto en un momento dado. En vez de ello, es necesario estudiar una instantánea del objeto, sus vecinos y las relaciones con estos vecinos. En todos los sistemas orientados a objetos, excepto en los más simples, existirá una multitud de objetos, manteniendo cada uno de ellos una relación precisa con los demás. De hecho, cuando un sistema orientado a objetos falla, no suele ser por un fallo en la lógica sino porque se rompen conexiones entre objetos o por un estado no válido en objetos individuales.

Con UML, los Diagramas de Clases se utilizan para visualizar los aspectos estáticos de bloques de construcción del sistema. Los Diagramas de Interacción se utilizan para ver los aspectos dinámicos del sistema y constan de instancias de estos bloques y mensajes enviados entre ellos. Un Diagrama de Objetos contiene un conjunto de instancias de los elementos encontrados en un Diagrama de Clases. Por tanto, un Diagrama de Objetos expresa la parte estática de una interacción, consistiendo en los objetos que colaboran, pero sin ninguno de los mensajes enviados entre ellos.

En ambos casos, un Diagrama de Objetos e congela un instante en el tiempo, como se muestra en la figura.

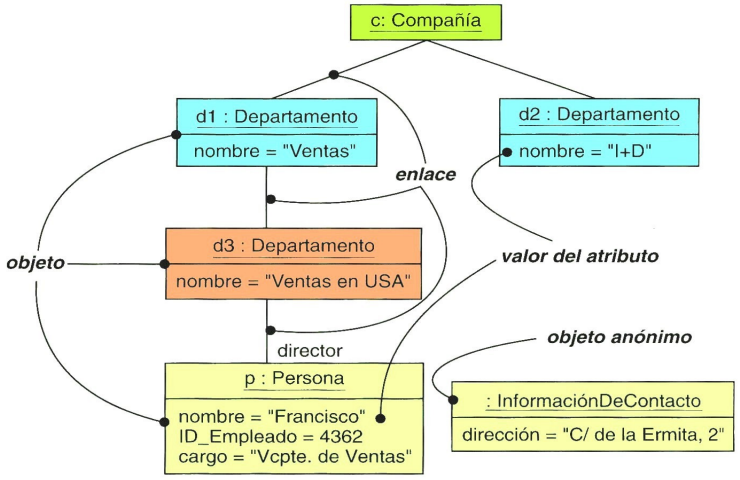


Fig. 39.- Un Diagrama de Objetos.

Diagrama de Estructura Compuesta

UML permite poner Diagramas de Clase dentro de una clase. Cuando hablamos acerca de composiciones, esto no es una limitación como puede parecer. Dado que el segundo compartimiento de una clase muestra su estructura, y una composición tiene una estructura compleja dentro de sí misma, entonces se puede mostrar las partes de la composición interna como un mini diagrama de clases.

UML 2 tiene un nuevo diagrama para esta notación alternativa: Diagrama de Estructura Compuesta.

La notación UML tiene tres componentes principales:

- La primera parte nombra la clase, describe su estereotipo y lista sus propiedades.
- La segunda parte muestra la estructura de la clase como una lista de atributos.
- La tercera parte es donde se pone la especificación del funcionamiento de la clase.

Esta partición fue considerada como una idea interesante en la versión de UML 1.4. La mayoría de las herramientas CASE, sin embargo, no acogieron esta idea. Pero está cambiando esto con la versión 2 de UML.

Mostrando partes como clases

Al modelar una fuerte forma de agregación, la composición, a menudo resulta en un diagrama de clases con bastantes líneas confusas. Se tienen además líneas entre las clases jugando el rol del todo y clases jugando el rol de las partes. Además se tienen líneas mostrando las asociaciones entre las partes internas individuales y los compuestos. Con todas estas líneas, el diagrama puede ser difícil de leer. La versión 2 de UML permite modelar compuestos y sus partes como un diagrama de clases dentro de una clase (diagrama de estructura compuesta). Esto permite ser más claro en lo que se desea expresar y reduce el desorden.

Se pueden mostrar las partes de un compuesto dentro de la parte de la estructura de una clase colocando una caja alrededor de la parte y suministrando el nombre de la parte: después dos puntos y por último el nombre de la clase de la parte. Si se tiene más de una parte para el mismo tipo en una composición, entonces se puede mostrar su multiplicidad entre corchetes. Por ejemplo, la parte del cuerpo de un reporte genérico, debería estar rodeada por una caja con detalle: Cuerpo[1..*] dentro, como se muestra en la siguiente figura.

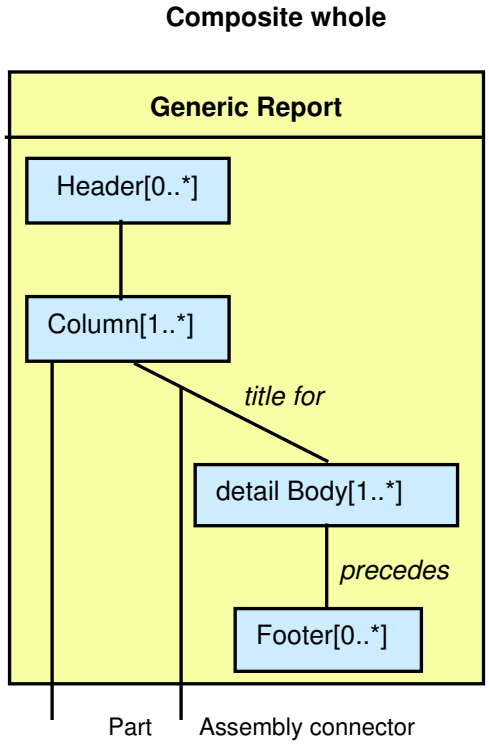


Fig. 40.- Partes Compuestas mostradas dentro de una clase.

Las partes pueden estar conectadas (si así se desea) por conectores -líneas que indican las ligas entre las instancias de partes dentro de un compuesto- de tal forma que estas partes pueden comunicarse con otras. UML 2.0 provee dos tipos de conectores-ensamblado y delegado. Un conector de ensamblado permite a una parte de la composición proveer servicios que otra parte necesita. Por el otro lado, use un conector de delegación para mostrar la composición completa presentando algunas solicitudes externas de comportamiento de una de sus partes internas. El conector de ensamblado conecta dos partes como en una asociación. El conector de delegación conecta el todo con una de sus partes. El conector de delegación es mostrado como una línea desde la frontera de la clase compuesta a una de las partes de la clase compuesta.

La figura anterior ilustra el diagrama. La clase Reporte genérico está jugando el rol del todo o compuesto. Las partes son anónimas con el encabezado nombrando la clase, columna y pie. Una de estas partes es nombrada detalle, la cual es el cuerpo de la clase. Las partes son conectadas usando líneas que pueden ser nombradas como asociaciones. Además, se puede agregar multiplicidad, nombre de roles y modificadores en estas conexiones. Cada una de las conexiones mostradas en la figura anterior son conexiones ensambladas. Por ejemplo, el encabezado invocará el servicio impreso de columnas.

Mostrando partes como atributos

Esta sección muestra como se unen y juntan los compuestos, diagramas de partes (aquellos diagramas de

clases dentro de una clase) y atributos. La siguiente figura muestra la clase ReporteGenerico y sus atributos. Note la correspondencia entre los atributos en la figura siguiente y las clases en la anterior. La definición de clases en la figura siguiente, esconde la estructura interna de la clase ReporteGenerico al simplemente listar las partes principales como atributos. La propiedad SqlStatement no es una parte, en lugar de ello, es uno de los atributos de la clase ReporteGenerico.

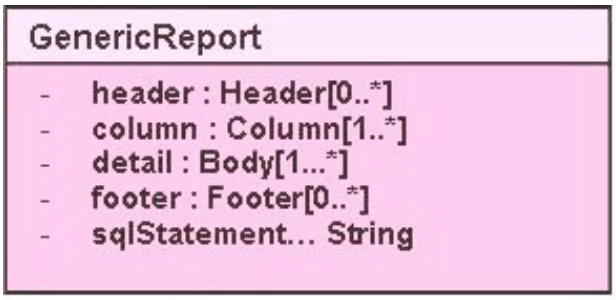


Fig. 41.- Ejemplos de clases mostradas como atributos.

Si usted desea convertir una clase simple en un diagrama de estructura compuesto, puede usar la siguiente tabla como guía. La tabla muestra la correspondencia entre atributos en un Diagrama de Clase simple y los elementos de un Diagrama de Partes dentro de una Clase Compuesta. Por ejemplo, el atributo detalle del ReporteGenerico se convierte en una parte con el mismo nombre en el Diagrama de Estructura Compuesta. El tipo de datos Body se convierte en el nombre de la clase de la parte de detail. La multiplicidad [1..*] es arrastrada a la multiplicidad de la parte detail.

Atributo	Característica de estructura compuesta
Nombre del Atributo	Nombre de la Parte
Tipo	Nombre de la clase parte
Multiplicidad	Número de conexiones permisibles entre instancias parte.

Tabla 10.- Guía para convertir una clase en un diagrama de estructura compuesta.

Finalmente diremos que el Diagrama de Estructura Compuesta, se emplea para visualizar de manera gráfica las partes que definen la estructura interna de un clasificador. Cuando se utiliza en el marco de una

clase, este diagrama permite elaborar un diagrama de clases donde se muestran los diferentes atributos (partes) y las clases, a partir de las cuales se definen los atributos, indicando principalmente las asociaciones de agregación o de composición de la clase a la que se le elabora el diagrama.

Diagrama de Despliegue

Un Diagrama de Despliegue muestra la configuración de nodos de procesamiento en tiempo de ejecución y los componentes que residen en ellos. Los diagramas de despliegue cubren la vista de despliegue estática de una arquitectura. Se relacionan con los diagramas de componentes en que un nodo incluye, por lo común, uno o más componentes.

Los Diagramas de Despliegue son uno de los dos tipos de diagramas que aparecen cuando se modelan los aspectos físicos de los sistemas orientados a objetos. Un Diagrama de Despliegue muestra la configuración de nodos que participan en la ejecución y de los componentes que residen en ellos.

Los Diagramas de Despliegue se utilizan para modelar la vista de despliegue estática de un sistema. La mayoría de las veces, esto implica modelar la topología del hardware sobre la que se ejecuta el sistema. Los Diagramas de Despliegue son fundamentalmente Diagramas de Clases que se ocupan de modelar los nodos de un sistema.

Los Diagramas de Despliegue no sólo son importantes para visualizar, especificar y documentar sistemas empotrados, sistemas cliente/servidor y sistemas distribuidos, sino también para gestionar sistemas ejecutables mediante ingeniería directa e inversa.

Cuando se construye un sistema con gran cantidad de software, la intención principal del desarrollador se centra en diseñar y desplegar el software. Sin embargo, para un Ingeniero de Sistemas, la atención principal está en el hardware y el software del sistema y en el manejo de los compromisos entre ambos. Mientras que los desarrolladores de software trabajan con artefactos en cierto modo intangibles, tales como modelos y código, los desarrolladores de sistemas trabajan a su vez con un hardware bastante tangible.

UML se centra principalmente en ofrecer facilidades para visualizar, especificar, construir y documentar artefactos de software, pero también ha sido diseñado para cubrir los artefactos hardware. Esto no equivale a decir que UML sea un lenguaje de descripción de hardware de propósito general, como VHDL. Más bien, UML ha sido diseñado para modelar muchos de los aspectos hardware de un sistema a nivel suficiente para que un Ingeniero de Software pueda especificar la plataforma sobre la que se ejecutará el software del sistema y para que un Ingeniero de Sistemas pueda manejar la frontera entre el hardware y el software del sistema. En UML, los Diagramas de Clases y los Diagramas de Componentes se utilizan para razonar sobre la estructura del software. Los Diagramas de Secuencia, los Diagramas de Colaboración, los Diagramas de Estados y los Diagramas de Actividades se utilizan para especificar el comportamiento del software. Cuando se trata del hardware y el software del sistema, se utilizan los diagramas de despliegue para razonar sobre la topología de procesadores y dispositivos sobre los que se ejecuta el software.

Con UML, los Diagramas de Despliegue se utilizan para visualizar los aspectos estáticos de estos nodos físicos y sus relaciones y para especificar sus detalles para la construcción, como se muestra en la figura siguiente.

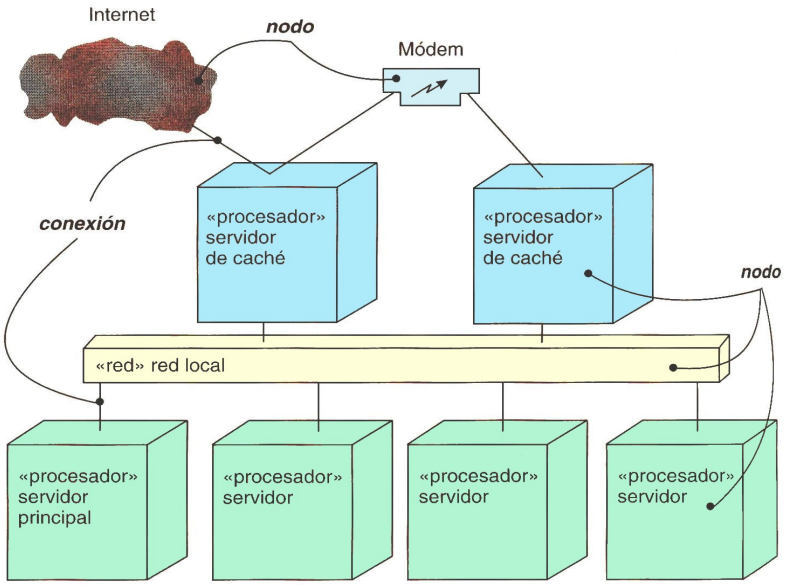


Fig. 42.- Diagrama de Despliegue.

El Diagrama de Despliegue es un tipo de diagrama del Lenguaje Unificado de Modelado que sirve para modelar el hardware utilizado en las implementaciones de sistemas y las relaciones entre sus componentes.

Los elementos usados por este tipo de diagrama son nodos (representados como un prisma), componentes (representados como una caja rectangular con dos protuberancias del lado izquierdo) y asociaciones.

En el UML 2.0 los componentes ya no están dentro de nodos. En cambio, puede haber artefactos u otros nodos dentro de un nodo.

Un artefacto puede ser algo como un archivo, un programa, una biblioteca, o una base de datos construida o modificada en un proyecto. Estos artefactos implementan colecciones de componentes. Los nodos internos indican ambientes, un concepto más amplio que el hardware propiamente dicho, ya que un ambiente puede incluir al lenguaje de programación y al sistema operativo.

Diagrama de paquetes

Los elementos de agrupación son las partes organizativas de los modelos UML. Estos son las cajas en las que puede descomponerse un modelo. En total, hay un elemento de agrupación principal, los paquetes.

Un paquete es un mecanismo de propósito general para organizar elementos en grupos. Los elementos estructurales, los elementos de comportamiento, e incluso otros elementos de agrupación pueden incluirse en un paquete. Al contrario que los componentes (que existen en tiempo de ejecución), un paquete es puramente conceptual (sólo existe en tiempo de desarrollo). Gráficamente, un paquete se visualiza como una carpeta, incluyendo normalmente sólo su nombre y a veces, su contenido.

Visualizar, especificar, construir y documentar grandes sistemas conlleva manejar una cantidad de clases, interfaces, componentes, nodos, diagramas y otros elementos que puede ser muy elevada. Conforme va creciendo el sistema hasta alcanzar un gran tamaño, se hace necesario organizar estos elementos en bloques mayores. En UML el paquete es el mecanismo de propósito general para organizar elementos de modelado en grupos.

Los paquetes se utilizan para organizar los elementos de modelado en partes mayores que se pueden manipular como grupo. La visibilidad de estos elementos puede controlarse para que algunos sean visibles fuera del paquete mientras otros permanezcan ocultos. También se pueden emplear los paquetes para presentar diferentes vistas de la arquitectura del sistema.

Los paquetes bien diseñados agrupan elementos cercanos semánticamente y que suelen cambiar juntos. Por tanto, los paquetes bien estructurados son cohesivos y poco acoplados, estando muy controlado el acceso a su contenido.

Las casas de perro no son complejas: se tienen cuatro paredes, una de ellas con un agujero del tamaño de un perro, y un techo. Al construir una caseta de perro sólo se necesitan unas cuantas tablas. No hay mucha más estructura.

Las casas son más complejas. Paredes, techos y suelos se combinan en abstracciones mayores que llamamos habitaciones. Incluso esas habitaciones se organizan en abstracciones mayores: la sala de estar, la sala de entretenimiento, etc. Estos grupos mayores no tienen por qué manifestarse como algo que construir en la propia casa, sino que pueden ser simplemente los nombres que se dan a habitaciones lógicamente relacionadas y se aplican cuando se habla sobre el uso que se hará de la casa.

Los grandes edificios son muy complejos. No sólo existen estructuras elementales, tales como paredes, techos y suelos, sino que hay estructuras más complejas, tales como zonas públicas, el área comercial y las oficinas. Estas estructuras probablemente se agruparon en otras aún más complejas, tales como las zonas de alquileres y las zonas de servicios del edificio. Puede que estas estructuras más complejas no tengan nada que ver con el edificio final, sino que sean simplemente artefactos que se utilizan para organizar los planos del edificio.

Todos los sistemas grandes se jerarquizan en niveles de esta forma. De hecho, quizás la única forma de comprender un sistema complejo sea agrupando las abstracciones en grupos cada vez mayores. La mayoría de las abstracciones básicas (como las habitaciones) son, por derecho propio, abstracciones de la misma naturaleza que las clases, para las que puede haber muchas instancias. La mayoría de las abstracciones mayores son puramente conceptuales (como el área comercial), para las que no existen instancias reales. Ellas nunca se manifiestan en el sistema real, sino más bien existen con el único objetivo de comprender el sistema. Estos últimos tipos de abstracciones no tienen identidad en el sistema desplegado; sólo tienen identidad en el modelo del sistema.

En UML las abstracciones que organizan un modelo se llaman paquetes. Un paquete es un mecanismo de propósito general para organizar elementos en grupos. Los paquetes ayudan a organizar los elementos en los modelos con el fin de comprenderlos más fácilmente. Los paquetes también permiten controlar el acceso a sus contenidos para controlar las líneas de separación de la arquitectura del sistema.

UML proporciona una representación gráfica de los paquetes, como se muestra en la figura siguiente. Esta notación permite visualizar grupos de elementos que se pueden manipular como un todo y en una forma que permite controlar la visibilidad y el acceso a elementos individuales.

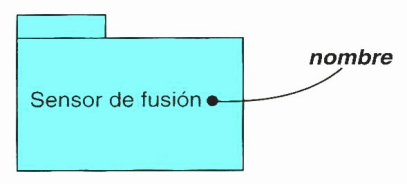


Fig. 43.- Paquetes.

Diagrama de Actividades

Un Diagrama de Actividades es un tipo especial de Diagrama de Estados que muestra el flujo de actividades dentro de un sistema. Los Diagramas de Actividades cubren la vista dinámica de un sistema. Son especialmente importantes al modelar el funcionamiento de un sistema y resaltan el flujo de control entre objetos.

Los Diagramas de Actividades son uno de los cinco tipos de diagramas de UML que se utilizan para el modelado de los aspectos dinámicos de los sistemas. Un Diagrama de Actividades es fundamentalmente un diagrama de flujo que muestra el flujo de control entre actividades.

Los Diagramas de Actividades se utilizan para modelar los aspectos dinámicos de un sistema. La mayoría de las veces, esto implica modelar los pasos secuenciales (y posiblemente concurrentes) de un proceso computacional. Con un Diagrama de Actividades también se puede modelar el flujo de un objeto conforme pasa de estado a estado en diferentes puntos del flujo de control. Los Diagramas de Actividades pueden utilizarse para visualizar, especificar, construir y documentar la dinámica de una sociedad de objetos, o pueden emplearse para modelar el flujo de control de una operación. Mientras que los Diagramas de Interacción destacan el flujo de control entre objetos, los Diagramas de Actividades destacan el flujo de control entre actividades. Una actividad es una ejecución no atómica en curso, dentro de una máquina de estados. Las actividades producen alguna acción, compuesta de computaciones atómicas ejecutables que producen un cambio en el estado del sistema o el retorno de un valor.

Los Diagramas de Actividades no son sólo importantes para modelar los aspectos dinámicos de un sistema, sino también para construir sistemas ejecutables a través de ingeniería directa e inversa.

Considérese un flujo de trabajo asociado a la construcción de una casa. En primer lugar se selecciona un sitio. A continuación, se contrata a un arquitecto para diseñar la casa. Después de llegar a un acuerdo en el plano, el constructor consulta las ofertas para establecer el precio de la casa. Una vez acordados un precio y un plano, puede comenzar la construcción. Se obtienen los permisos, se mueven tierras, se echan cimientos, se erige la estructura, etcétera, hasta que todo queda hecho. Entonces se entregan las llaves y un certificado de vivienda y el propietario toma posesión de la casa.

Aunque ésta es una tremenda simplificación de lo que realmente ocurre en un proceso de construcción, captura el camino crítico del flujo de trabajo. En un proyecto real, hay muchas actividades paralelas entre varios profesionales. Por ejemplo, los electricistas pueden trabajar a la vez que los fontaneros y los carpinteros. También aparecerán condiciones y bifurcaciones. Por ejemplo, dependiendo del resultado de las pruebas de las tierras, se tendrá que excavar, usar explosivos o colocar una estructura que permita oscilaciones. Incluso podría haber ciclos. Por ejemplo, una inspección podría revelar violaciones de las leyes que produjesen como resultado escombro y repetición de trabajo.

En la industria de la construcción se utilizan frecuentemente técnicas como los diagramas de Gantt y los diagramas de Pert para visualizar, especificar, construir y documentar el flujo de trabajo del proyecto.

Cuando se modelan sistemas con gran cantidad de software aparece un problema similar. ¿Cuál es la mejor forma de modelar un flujo de trabajo o una operación, que son ambos aspectos de la dinámica del sistema? La respuesta es que existen dos elecciones básicas, similares al uso de diagramas de Gantt y diagramas de Pert.

Por un lado, se pueden construir representaciones gráficas de escenarios que involucren la interacción de ciertos objetos interesantes y los mensajes que se pueden enviar entre ellos. En UML se pueden modelar estas representaciones de dos formas: destacando la ordenación temporal de los mensajes (con Diagramas de Secuencia) o destacando las relaciones estructurales entre los objetos que interactúan (con Diagramas de Colaboración). Los Diagramas de Interacción son similares a los diagramas de Gantt, los cuales se centran en los objetos (recursos) que juegan alguna actividad a lo largo del tiempo.

Por otro lado, estos aspectos dinámicos se pueden modelar con diagramas de actividades, que se centran en las actividades que tienen lugar entre los objetos, como se muestra en la siguiente figura. En este sentido, los Diagramas de Actividades son similares a los Diagramas de Pert. Un Diagrama de Actividades es esencialmente un Diagrama de Flujo que destaca la actividad que tiene lugar a lo largo del tiempo. Se puede pensar en un Diagrama de Actividades como en un Diagrama de Interacción al que se le ha dado la vuelta. Un Diagrama de Interacción muestra objetos que pasan mensajes; un Diagrama de Actividades muestra las operaciones que se pasan entre los objetos. La diferencia semántica es sutil, pero tiene como resultado una forma muy diferente de mirar el mundo.

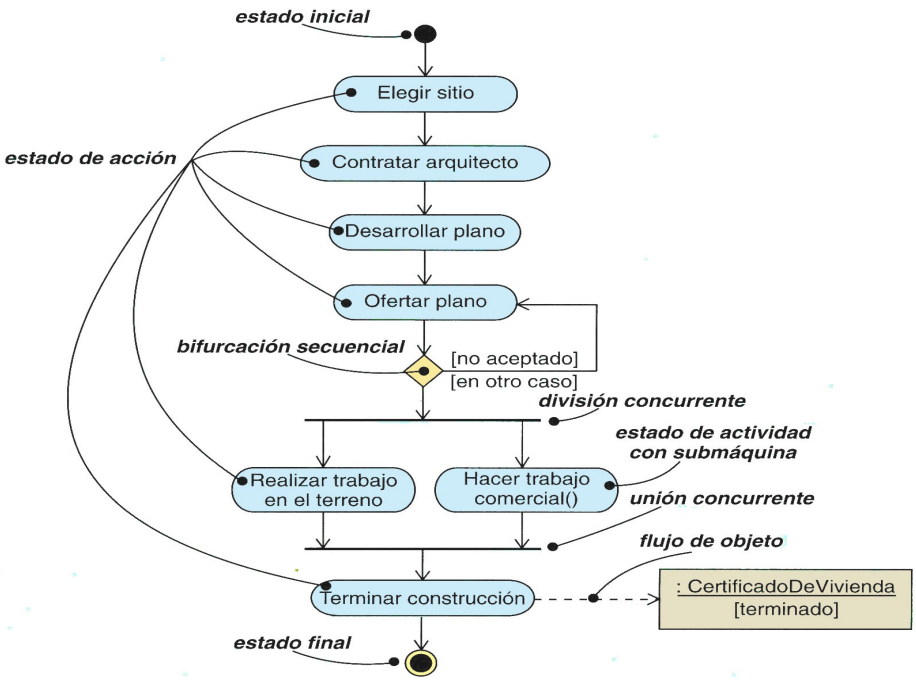


Fig. 44.- Diagrama de Actividades.

Diagrama de Casos de Uso

Un Diagrama de Casos de Uso muestra un conjunto de Casos de Uso y actores (un tipo especial de clases) y sus relaciones. Los Diagramas de Casos de Uso cubren la vista de Casos de Uso estática de un sistema. Estos diagramas son especialmente importantes en el modelado y organización del comportamiento de un sistema.

Los Diagramas de Casos de Uso son uno de los cinco tipos de diagramas de UML que se utilizan para remodelado de los aspectos dinámicos de un sistema (los otros cuatro tipos son los Diagramas de Actividades, de Estados, de Secuencia y de Colaboración). Los Diagramas de Casos de Uso son importantes para modelar el comportamiento de un sistema, un subsistema o una clase. Cada uno muestra un conjunto de Casos de Uso, actores y sus relaciones.

Los Diagramas de Casos de Uso se emplean para modelar la vista de Casos de Uso de un sistema. La mayoría de las veces, esto implica modelar el contexto del sistema, subsistema o clase, o el modelado de los requisitos de comportamiento de esos elementos.

Los Diagramas de Casos de Uso son importantes para visualizar, especificar y documentar el comportamiento de un elemento. Estos diagramas facilitan que los sistemas, subsistemas y clases sean abordables y comprensibles, al presentar una vista externa de cómo pueden utilizarse estos elementos en un contexto dado. Los Diagramas de Casos de Uso también son importantes, para probar sistemas ejecutables a través de ingeniería directa y para comprender sistemas ejecutables a través de ingeniería inversa.

Supongamos que alguien nos da una caja. En un lado hay algunos botones y una pequeña pantalla de cristal líquido. Aparte de esto, no viene ninguna descripción con la caja; ni siquiera se dispone de ningún indicio sobre cómo usarla. Podríamos pulsar aleatoriamente los botones y ver qué ocurre; pero nos veríamos obligados a imaginar qué hace la caja o cómo utilizarla correctamente, a menos que dedicásemos mucho tiempo a hacer pruebas de ensayo y error.

Los sistemas con gran cantidad de software pueden ser parecidos. A un usuario se le podría dar una aplicación y pedirle que la utilizara. Si la aplicación sigue las convenciones normales del sistema operativo al que el usuario está acostumbrado, podría ser capaz de que hiciera algo útil después de un rato, pero de esta forma nunca llegaría a entender su comportamiento más complejo y sutil. Análogamente, a un desarrollador se le podría dar una aplicación ya existente o un conjunto de componentes y decirle que los use. El desarrollador se vería presionado ante la necesidad de conocer cómo usar estos elementos hasta que se formara un modelo conceptual para su uso.

Con UML, los Diagramas de Casos de Uso se emplean para visualizar el comportamiento de un sistema, subsistema o una clase, de forma que los usuarios pueden comprender cómo utilizar ese elemento y de forma que los desarrolladores puedan implementarlo. Como se muestra en la figura siguiente, se pude proporcionar un Diagrama de Casos de Uso para modelar el comportamiento de esa caja (que la mayoría de la gente llamaría un teléfono móvil).

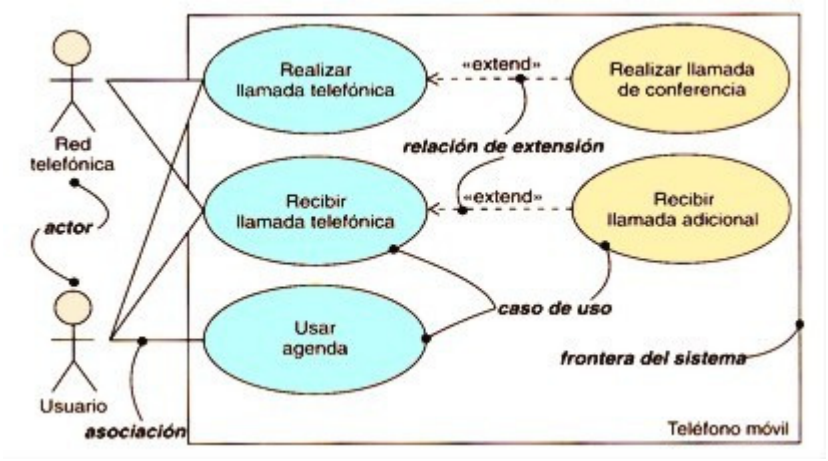


Fig. 45.- Diagrama de Casos de Uso.

Diagrama de Estados

Un Diagrama de Estados muestra una máquina de estados, que consta de estados, transiciones, eventos y actividades. Los Diagramas de Estados cubren la vista dinámica de un sistema. Son especialmente importantes en el modelado del comportamiento de una interfaz, una clase o una colaboración y resaltan el comportamiento dirigido por eventos de un objeto, lo cual es especialmente útil en el modelado de sistemas reactivos.

Los Diagramas de Estados (statechart) son uno de los cinco tipos de diagramas de UML que se utilizan para el modelado de los aspectos dinámicos de un sistema. Un Diagrama de Estados muestra una máquina de estados. Un Diagrama de Actividades es un caso especial de Diagrama de Estados en el cual todos o la mayoría de los estados son estados de actividad y todas o la mayoría de las transiciones se disparan por la terminación de las actividades en el estado origen. Así, tanto los Diagramas de Actividades como los Diagramas de Estados son útiles para modelar la vida de un objeto. Sin embargo, mientras un Diagrama de Actividades muestra el flujo de control entre actividades, un Diagrama de Estados muestra el flujo de control entre estados.

Los Diagramas de Estados se utilizan para modelar los aspectos dinámicos de un sistema. La mayoría de las veces, esto supone el modelado del comportamiento de objetos reactivos. Un objeto reactivo es aquél para el que la mejor forma de caracterizar su comportamiento es señalar cuál es su respuesta a los eventos lanzados desde fuera de su contexto. Un objeto reactivo tiene un ciclo de vida bien definido, cuyo comportamiento se ve afectado por su pasado. Los Diagramas de Estados pueden asociarse a las clases, los Casos de Uso, o a sistemas completos para visualizar, especificar, construir y documentar la dinámica de un objeto individual.

Los Diagramas de Estados no sólo son importantes para modelar los aspectos dinámicos de un sistema, sino también para construir sistemas ejecutables a través de ingeniería directa e inversa.

Considérese al inversor que financia la construcción de un rascacielos. Es poco probable que esté

interesado en los detalles del proceso de construcción. La selección de materiales, la planificación de los trabajos y las reuniones sobre detalles de ingeniería son actividades importantes para el constructor, pero mucho menos importantes para la persona que financia el proyecto.

El inversor está interesado en obtener unos importantes beneficios de la inversión y esto significa proteger la inversión frente al riesgo. Un inversor realmente confiado entregará al constructor una gran cantidad de dinero, se marchará durante un tiempo y regresará sólo cuando el constructor esté en condiciones de entregarle las llaves del edificio. Un inversor como éste está interesado únicamente en el estado final del edificio.

Un inversor más práctico también confiará en el constructor, pero al mismo tiempo se preocupará de verificar que el proyecto está en marcha antes de entregar el dinero. Así, en vez de darle al constructor una gran cantidad de dinero y despreocuparse, el inversor prudente establecerá unos hitos claros en el proyecto, cada uno de los cuales irá asociado a la terminación de ciertas actividades, tras los cuales irá entregando el dinero al constructor para la siguiente fase del proyecto. Por ejemplo, al comenzar el proyecto se entregaría una modesta cantidad de fondos, para financiar el trabajo de arquitectura. Después de haber sido aprobada la visión arquitectónica, podrían aportarse más fondos para pagar los trabajos de ingeniería. Tras completar este trabajo y dejar satisfechos a las personas interesadas, podría entregarse una gran cantidad de dinero para que el constructor proceda a remover tierras. A lo largo del camino, desde el movimiento de tierras hasta la emisión del certificado de habitabilidad, habrá otros hitos.

Cada uno de estos hitos representa un estado estable del proyecto: arquitectura terminada, ingeniería hecha, tierras removidas, infraestructura terminada, edificio cerrado, etcétera. Para el inversor, es más importante seguir el cambio de estado del edificio que seguir el flujo de actividades, que es lo que el constructor podría estar haciendo utilizando diagramas de Pert para modelar el flujo de trabajo del proyecto.

Cuando se modelan sistemas con gran cantidad de software, se encuentra que la forma más natural de visualizar, especificar, construir y documentar el comportamiento de ciertos tipos de objetos es centrarse en el flujo de control entre estados en vez del flujo de actividades. Esto último se hace utilizando un diagrama de flujo (y en UML, con un Diagrama de Actividades). Imagínese por un momento, el modelado del comportamiento de un sistema de seguridad de una vivienda. Un sistema de esa naturaleza funciona de forma continua, reaccionando a ciertos eventos externos, tales como la rotura de una ventana. Además, el orden de los eventos cambia la forma en que se comporta un sistema. Por ejemplo, la detección de la rotura de una ventana sólo disparará una alarma si el sistema antes está activado. La mejor forma de especificar el comportamiento de un sistema de estas características es modelar sus estados estables (por ejemplo, Inactivo, Montado, Activo, Comprobando, etcétera), los eventos que producen un cambio de estado y las acciones que ocurran en cada cambio de estado.

En UML, el comportamiento dirigido por eventos de un objeto se modela utilizando Diagramas de Estados. Como se muestra en la siguiente figura, un Diagrama de Estados es simplemente la representación de una máquina de estados, que destaca el flujo de control entre estados.

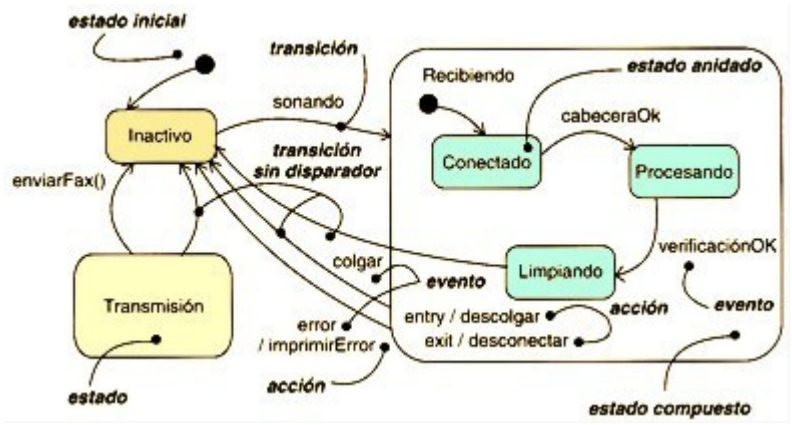


Fig. 46.- Diagrama de Estados.

Diagrama de secuencia

Un Diagrama de Secuencia es un Diagrama de Interacción que resalta la ordenación temporal de los mensajes.

Un Diagrama de Secuencia destaca la ordenación temporal de los mensajes. Como se muestra en la figura siguiente, un Diagrama de Secuencia se forma colocando en primer lugar los objetos que participan en la interacción en la parte superior del diagrama, a lo largo del eje X. Normalmente, se coloca a la izquierda el objeto que inicia la interacción y los objetos subordinados a la derecha. A continuación se colocan los mensajes que estos objetos envían y reciben a lo largo del eje Y, en orden de sucesión en el tiempo, desde arriba hasta abajo. Esto ofrece al lector una señal visual clara del flujo de control a lo largo del tiempo.

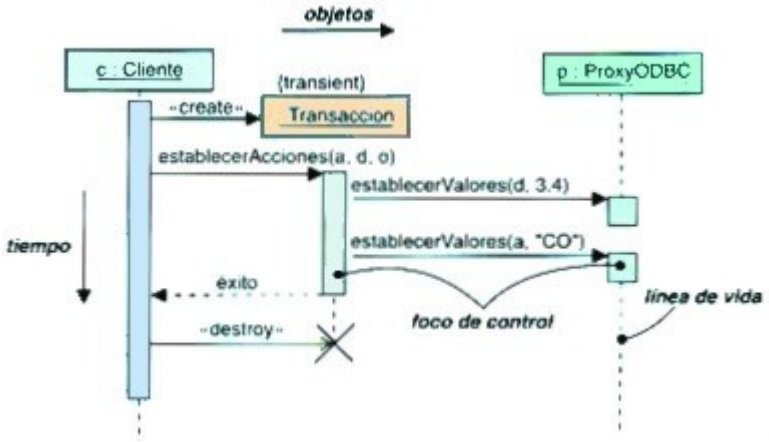


Fig. 47.- Diagramas de Secuencia.

Los Diagramas de Secuencia tienen dos características que los distinguen de los Diagramas de Colaboración.

En primer lugar, está la línea de vida. La línea de vida de un objeto es la línea discontinua vertical que representa la existencia de un objeto a lo largo de un período de tiempo. La mayoría de los objetos que aparecen en un diagrama de interacción existirán mientras dure la interacción, así que los objetos se colocan en la parte superior del diagrama, con sus líneas de vida dibujadas desde arriba hasta abajo. Pueden crearse objetos durante la interacción. Sus líneas de vida comienzan con la recepción del mensaje estereotipado como "create". Los objetos pueden destruirse durante la interacción. Sus líneas de vida acaban con la recepción del mensaje estereotipado como destroy (además se muestra la señal visual de una gran X que marca el final de sus vidas).

Nota: Si un objeto cambia el valor de sus atributos, su estado o sus roles, se pude colocar una copia del ícono del objeto sobre su línea de vida en el punto en el que ocurre el cambio, mostrando estas modificaciones.

En segundo lugar, está el foco de control. El foco de control es un rectángulo delgado y estrecho que representa el período de tiempo durante el cual un objeto ejecuta una acción, bien sea directamente o a través de un procedimiento subordinado. La parte superior del rectángulo se alinea con el comienzo de la acción; la inferior se alinea con su terminación (y puede marcarse con un mensaje de retorno). También puede mostrarse el anidamiento de un foco de control (que puede estar causado por recursión, una llamada a una operación propia, o una llamada desde otro objeto) colocando otro foco de control ligeramente a la derecha de su foco padre (esto se puede hacer a cualquier nivel de profundidad). Si se quiere ser especialmente preciso acerca de dónde se encuentra el foco de control, también se puede sombrear la región del rectángulo durante la cual el método del objeto está ejecutándose (y el control no ha pasado a otro objeto).

Diagrama de Colaboración

Tanto los Diagramas de Secuencia como los Diagramas de Colaboración son un tipo de Diagramas de Interacción. Un diagrama de este tipo muestra una interacción, que consta de un conjunto de objetos y sus relaciones, incluyendo los mensajes que pueden ser enviados entre ellos. Los Diagramas de Interacción cubren la vista dinámica de un sistema. Un Diagrama de Secuencia es un Diagrama de Interacción que resalta la ordenación temporal de los mensajes; un Diagrama de Colaboración es un Diagrama de Interacción que resalta la organización estructural de los objetos que envían y reciben mensajes. Los Diagramas de Secuencia y los Diagramas de Colaboración son isomorfos, es decir, que se puede tomar uno y transformarlo en el otro.

Un Diagrama de Colaboración destaca la organización de los objetos que participan en una interacción. Como se muestra en la siguiente figura, un Diagrama de Colaboración se construye colocando en primer lugar los objetos que participan en la colaboración como nodos en el grafo. A continuación se representan los enlaces que conectan esos objetos como arcos del grafo. Por último estos enlaces se adornan con los mensajes que envían y reciben los objetos. Esto da al lector una señal visual clara del flujo de control en el contexto de la organización estructural de los objetos que colaboran.

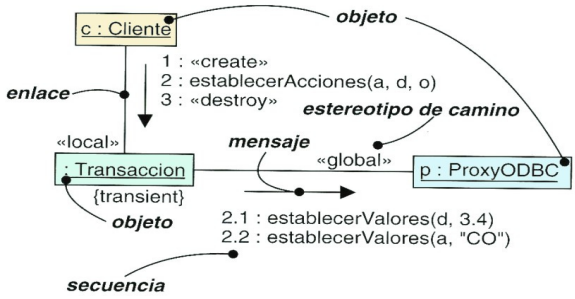


Fig. 48.- Diagrama de Colaboración.

Los Diagramas de Colaboración tienen dos características que los distinguen de los diagramas de secuencia.

En primer lugar, el camino. Para indicar cómo se enlaza un objeto a otro, se puede asociar un estereotipo de camino al extremo más lejano de un enlace (como <<local>>, que indica que el objeto designado es local al emisor). Normalmente sólo se necesita representar explícitamente el camino del enlace para los caminos local, parameter, global y self (pero no association).

En segundo lugar, está el número de secuencia. Para indicar la ordenación temporal de un mensaje, se precede de un número (comenzando con el mensaje número 1), que se incrementará secuencialmente por cada nuevo mensaje en el flujo de control (2, 3, etc.). Para representar el anidamiento, se utiliza la numeración decimal de Dewey (1 es el primer mensaje; 1.1 es el primer mensaje dentro del mensaje 1; 1.2 es el segundo mensaje dentro del mensaje 1; etc.). El anidamiento se puede representar a cualquier nivel de profundidad. Nótese también que a través del mismo enlace, se pueden mostrar varios mensajes (posiblemente enviados desde distintas direcciones) y cada uno tendrá un número de secuencia único.

La mayoría de las veces se modelarán flujos de control simples y secuenciales. Sin embargo, también se pueden modelar flujos más complejos, que impliquen iteración y bifurcación. Una iteración representa una secuencia repetida de mensajes. Para modelar una iteración, el número de secuencias de un mensaje se precede de una expresión de iteración, como en * [i := 1..n] (o sólo * si se quiere indicar iteración pero no se desea especificar los detalles). Una iteración indica que el mensaje (y cualquier mensaje anidado) se repetirá de acuerdo con la expresión dada. Análogamente, una condición representa un mensaje cuya ejecución depende de la evaluación de una expresión booleana. Para modelar una condición, el número de secuencia de un mensaje se precede de una cláusula de condición como [x>0]. Los distintos caminos alternativos de una bifurcación tendrán el mismo número de secuencia, pero cada camino debe ser distinguible de forma única por una condición que no se solape con las otras.

Tanto para la iteración como para la bifurcación, UML no impone el formato de la expresión entre corchetes; se puede utilizar pseudocódigo o la sintaxis de un lenguaje de programación específico.

Diagrama de Tiempo

Los Diagramas de Tiempo son un tipo de Diagrama de Interacción que enfatiza la especificación detallada de tiempo para mensajes. Son usados a menudo para modelar sistemas de tiempo real

tal como comunicaciones de satélites o conexiones de comunicaciones entre hardware. Existe una notación específica para indicar cuanto tiempo tiene un sistema para procesar o responder a mensajes y cómo las interrupciones externas son factores dentro de la ejecución.

Un Diagrama de Tiempo permite mostrar la interacción de objetos y cambios de estados en los mismos a lo largo del eje del tiempo. Un diagrama de este tipo, provee una manera conveniente para mostrar objetos activos y sus cambios de estados durante sus interacciones con otros objetos activos y recursos del sistema. El eje X del diagrama tiene unidades de tiempo, mientras que el eje Y muestra los objetos y sus estados.

La siguiente figura muestra un caso donde dos objetos activos comparten un recurso común. En este caso, ambos objetos muestran una venta que requiere un motor de ejecución por algún tiempo. Un objeto es para un usuario "platinum" a quien se tiene garantizada la venta dentro de 10 unidades de tiempo, y el otro es un usuario "gold" quien no tiene garantía de ejecución. Los objetos usuario tienen estados de espera y ejecución y además un estado de inactivo. El motor de ejecución tiene un estado de procesando que toma 5 unidades de tiempo por cada venta y un estado de ejecución que toma dos unidades de tiempo. En este caso, el estado de ejecución es una operación secuenciada estrictamente que no puede ser interrumpida, de esta forma un usuario con prioridad "platinum" aún tiene que esperar en algún punto cuando el motor de ejecución esté en estado de ejecución, aún si un objeto de menor prioridad mantiene al motor ocupado. Si el motor de ejecución está solo procesando, el objeto de mayor prioridad asegura el recurso y el objeto de menor prioridad tendrá que esperar. Los Diagramas de Tiempo pueden además ser expresados en una forma compacta donde un valor es escrito como texto entre dos líneas paralelas. Cuando las líneas paralelas cruzan, esto indica un evento que cambia aquel valor.

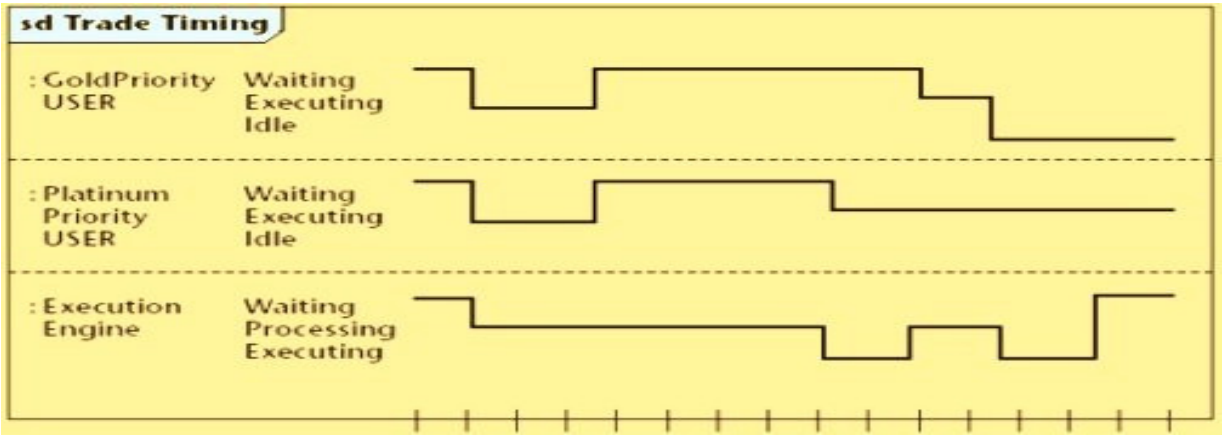


Fig. 49.- Diferentes prioridades de objetos mostrados en un Diagrama de Tiempo.

Los Diagramas de Tiempo proveen una herramienta útil para ordenar condiciones de carreras, inversiones con prioridad, puntos muertos y otras cuestiones de comunicaciones. Cuando se estén tratando de clasificar estas ejecuciones complejas, usa el Diagrama de Tiempo.

Diagrama de Vista de Interacción

Un Diagrama de Interacción es realmente un tipo de Diagrama de Funcionamiento. Se usan Diagramas de Interacción para representar el intercambio de mensajes dentro de una colaboración (un grupo de objetos que cooperan) en línea para lograr su objetivo.

El Diagrama de Vista de Interacción provee al modelador con la oportunidad de revisar el flujo principal de interacciones a un nivel alto. Esta característica puede probar su utilidad cuando se está tratando de asegurarse de que el diseño ha sido capturado con todos los elementos definidos en un Caso de Uso. Un Diagrama de Vista de Interacción es básicamente un Diagrama de Actividad con los principales nodos reemplazados por los fragmentos de interacción, o partes de Diagramas de Secuencia, puestos en un orden específico. El diagrama además provee otro método para mostrar el flujo de control durante una interacción. El punto de los Diagramas de Vista de Interacción es mostrar en un lugar las opciones que existen para la interacción.

Este diagrama es una variante de los Diagramas de Actividad. Varios nodos de flujo de control de los Diagramas de Actividad, pueden ser combinados con los fragmentos de secuencia para crear un Diagrama de Vista de Interacción.

Estos diagramas muestran cómo un conjunto de fragmentos pueden ser iniciados en varios escenarios. Los Diagramas de Interacción se enfocan en la vista del flujo de control donde los nodos son interacciones u ocurrencias de interacciones. Las líneas de vida y los mensajes no aparecen a este nivel de vista. Están esquematizados por el mismo tipo de marco que encierra otros Diagramas de Interacción. El texto de encabezado puede además incluir una lista de líneas de vida contenidas (las cuales no aparecen gráficamente). La siguiente figura muestra una interacción que consta de referencias a otras dos interacciones con algún flujo de control puesto alrededor.

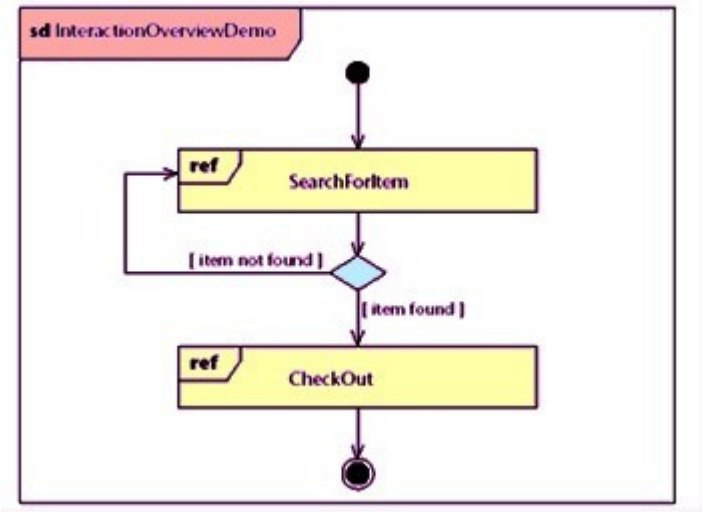


Fig. 50.- Diagrama de Interacción.

2.4.3.2 UML 2.0

Además de haberse convertido en un estándar de facto, UML es un estándar industrial promovido por el grupo OMG al mismo nivel que el estándar CORBA para intercambio de objetos distribuidos. Para la revisión de UML se formaron dos "corrientes" que promovían la aparición de la nueva versión desde distintos puntos de vista. Finalmente se impuso la visión más industrial frente a la académica. Recientemente se ha publicado la versión 2.0 en la que aparecen muchas novedades y cambios que fundamentalmente se centran en resolver carencias prácticas. Además esta versión recibe diversas mejoras que provienen del lenguaje SDL.

2.4.4 Críticas a UML

A pesar de su status de estándar ampliamente reconocido y utilizado, UML siempre ha sido muy criticado por su carencia de una semántica precisa, lo que ha dado lugar a que la interpretación de un modelo UML no pueda ser objetiva. Otro problema de UML es que no se presta con facilidad al diseño de sistemas distribuidos. En tales sistemas cobran importancia factores como transmisión, serialización, persistencia, etc. UML no cuenta con maneras de describir tales factores. No se puede por ejemplo, usar UML para señalar que un objeto es persistente o remoto o que existe en un servidor que corre continuamente y que es compartido entre varias instancias de ejecución del sistema analizado.

Marcos de trabajo

Un marco de trabajo es una colección de clases que pueden usar varias aplicaciones. Con frecuencia las clases de un marco de trabajo se relacionan. También pueden ser abstractas o pretender que se usen a través de la herencia.

La interfaz de programación de aplicaciones (API, Application Programming Interface), un ejemplo de paquetes de marcos de trabajo útiles, ha obtenido el entusiasmo de la comunidad de desarrollo por marcos de trabajo enriquecedores con los cuales hacer su tarea. La parte central de los paquetes Java API tiene el propósito de servir a una amplia variedad de aplicaciones, mientras que nuestra intención al desarrollar una aplicación es servir sólo a aplicaciones de alguna manera similares a la que se diseña. Los paquetes de la aplicación se relacionan con los paquetes de los marcos de trabajo por medio del agregado y/o la herencia. Por ejemplo, considere cómo se puede usar el paquete de Java Abstract Windowing Toolkit (awt). No se modifica awt sino que se crean las clases de GUI para la aplicación, que hereden de las clases awt, o agreguen objetos de awt como atributos.

Algunos piensan que los marcos de trabajo deben diseñarse sólo si un número grande de aplicaciones los van a usar, como el Java API. Sin embargo, a menudo se tienen ventajas significativas al desarrollar un marco de trabajo parcial en paralelo con la aplicación, aunque no se asegure un número grande de aplicaciones para él. Este marco de trabajo sirve como una capa abstracta invaluable que heredan las clases de muchas aplicaciones.

2.4.5 Patrones de Diseño

Un patrón de diseño es:

- Una solución estándar para un problema común de programación.
- Una técnica para flexibilizar el código haciéndolo satisfacer ciertos criterios.
- Un proyecto o estructura de implementación que logra una finalidad determinada.
- Un lenguaje de programación de alto nivel.
- Una manera más práctica de describir ciertos aspectos de la organización de un programa.
- Una forma de conectar componentes y programas.
- La forma de un Diagrama de Objeto o de un Modelo de Objeto.

Ejemplos

Vamos a presentar algunos ejemplos de patrones de diseño: A cada diseño de proyecto le sigue el problema que trata de resolver, la solución que aporta y las posibles desventajas asociadas. Un desarrollador debe buscar un equilibrio entre las ventajas y las desventajas a la hora de decidir que patrón utilizar. Lo normal es, como observará a menudo en la ciencia computacional y en otros campos, buscar el balance entre flexibilidad y rendimiento.

Encapsulación (ocultación de datos)

Problema: los campos externos pueden ser manipulados directamente a partir del código externo, lo que conduce a violaciones del invariante de representación o a dependencias indeseables que impiden modificaciones en la implementación.

Solución: esconda algunos componentes, permitiendo sólo accesos estilizados al objeto.

Desventajas: la interfaz no puede, eficientemente, facilitar todas las operaciones deseadas. El acceso indirecto puede reducir el rendimiento.

Subclase (herencia)

Problema: abstracciones similares poseen miembros similares (campos y métodos). Esta repetición es tediosa, propensa a errores y un quebradero de cabeza durante el mantenimiento.

Solución: herede miembros por defecto de una superclase, seleccione la implementación correcta a través de resoluciones sobre qué implementación debe ser ejecutada.

Desventajas: el código para una clase está muy dividido, con lo que potencialmente se reduce la comprensión. La introducción de resoluciones en tiempo de ejecución introduce overhead (procesamiento extra).

Iteración

Problema: Los clientes que desean acceder a todos los miembros de una colección deben realizar un transversal especializado para cada estructura de datos, lo que introduce dependencias indeseables que impiden la ampliación del código a otras colecciones.

Solución: Las implementaciones, realizadas con conocimiento de la representación, realizan transversales y registran el proceso de iteración. El cliente recibe los resultados a través de una interfaz estándar.

Desventajas: La implementación fija la orden de iteración, esto es, no está controlada en absoluto por el cliente.

Excepciones

Problema: Los problemas que ocurren en una parte del código normalmente han de ser manipulados en otro lugar. El código no debe desordenarse con rutinas de manipulación de error, ni con valores de retorno para identificación de errores.

Solución: Introducir estructuras de lenguaje para arrojar e interceptar excepciones.

Desventajas: Es posible que el código pueda continuar aún desordenado. Puede ser difícil saber dónde será gestionada una excepción. Tal vez induzca a los programadores a utilizar excepciones para controlar el flujo normal de ejecución, que es confuso y por lo general ineficaz.

Estos patrones de diseño en concreto son tan importantes que ya vienen incorporados en Java. Otros vienen incluidos en otros lenguajes, tal vez algunos nunca lleguen a estar incorporados a ningún lenguaje, pero continúan siendo útiles.

2.4.5.1 Cuando (no) utilizar patrones de diseño

La primera regla de los patrones de diseño coincide con la primera regla de la optimización: retrasar. Del mismo modo que no es aconsejable optimizar prematuramente, no se deben utilizar patrones de diseño antes de tiempo. Seguramente sea mejor implementar algo primero y asegurarse de que funciona, para luego utilizar el patrón de diseño para mejorar las flaquezas; esto es cierto, sobre todo, cuando aún no ha identificado todos los detalles del proyecto (si comprende totalmente el dominio y el problema, tal vez sea razonable utilizar patrones desde el principio, de igual modo que tiene sentido utilizar los algoritmos más eficientes desde el comienzo en algunas aplicaciones).

Los patrones de diseño pueden incrementar o disminuir la capacidad de comprensión de un diseño o de una implementación, disminuirla al añadir accesos indirectos o aumentar la cantidad de código, disminuirla al regular la modularidad, separar mejor los conceptos y simplificar la descripción. Una vez que se aprenda el vocabulario de los patrones de diseño será más fácil y más rápido comunicarse con otros individuos que también lo conozcan. Por ejemplo, es más fácil decir "ésta es una instancia del patrón Visitor" que "éste es

un código que atraviesa una estructura y realiza llamadas de retorno, en tanto que algunos métodos deben estar presentes y son llamados de este modo y en este orden".

La mayoría de las personas utiliza patrones de diseño cuando perciben un problema en su proyecto -algo que debería resultar sencillo no lo es- o su implementación como por ejemplo, el rendimiento. Examine un código o un proyecto de esa naturaleza. ¿Cuáles son sus problemas, cuáles son sus compromisos? ¿Qué le gustaría realizar que, en la actualidad, es muy difícil lograr? A continuación, compruebe una referencia de patrón de diseño y busque los patrones que abordan los temas que le preocupan.

Un patrón de diseño es el trabajo de una persona que ya se encontró con el problema anteriormente, intentó muchas soluciones posibles y escogió y describió una de las mejores. Esto es algo de lo que debería aprovecharse.

2.4.5.2 Patrones de creación

Fabricación Pura

Solución: Asignar un conjunto altamente cohesivo de responsabilidades a una clase artificial que no representa nada en el dominio del problema; una cosa inventada para dar soporte a una alta cohesión un bajo acoplamiento y reutilización.

Esa clase es una fabricación de la imaginación. En teoría, las responsabilidades que se asignan brindan soporte a una Alta Cohesión y Bajo Acoplamiento, de modo que el diseño de la fabricación sea muy limpio o puro. De ahí el nombre: Fabricación Pura.

Problema: ¿A quién asignar la responsabilidad cuando uno está desesperado y no quiere violar los patrones Alta Cohesión y Bajo Acoplamiento?

Los diseños orientados a objetos se caracterizan por implementar como clases de software las representaciones de conceptos en el dominio de un problema del mundo real; por ejemplo, una clase Venta y Cliente. Pese a ello, se dan muchas situaciones donde el asignar responsabilidades exclusivamente a las clases del dominio origina problemas por una mala cohesión o acoplamiento o bien por un escaso potencial de reutilización.

Ejemplo: Supongamos por ejemplo, que se necesita soporte para guardar las instancias Venta en una base de datos relacional. En virtud del patrón Experto, en cierto modo se justifica asignar esta responsabilidad a la clase Venta. Pero reflexionemos sobre las siguientes implicaciones:

- La tarea requiere un número relativamente amplio de operaciones de soporte orientadas a la base de datos, ninguna de las cuales se relaciona con el concepto de vender; por tanto, la clase Venta reduce su cohesión.
- La clase Venta ha de ser acoplada a la interfaz de la base de datos relacional (interfaz que suele proporcionar el proveedor de las herramientas de desarrollo); de ahí que mejore su acoplamiento. Y este ni siquiera se realiza con otro objeto del dominio, si no con una interfaz idiosincrásica de la base de datos.
- Guardar los objetos en una base de datos relacional es una tarea muy general en que debemos brindar soporte a muchas clases. Asignar estas responsabilidades a la clase Venta indica que habrá poca reutilización o mucha duplicación en otras clases que cumplen la misma función.

En conclusión, aunque en virtud del patrón Experto, Venta es un candidato lógico para guardarse a sí misma en una base de datos, da origen a un diseño de baja cohesión, alto acoplamiento y bajo potencial de reutilización, precisamente el tipo de situación desesperada que exige inventar algo.

Una solución razonable consiste en crear una clase nueva que se encargue tan sólo de guardar los objetos en algún tipo de almacenamiento persistente: una base de datos relacional; lo llamaremos AgentedeAlmacenamientoPersistente. Esta clase es una Fabricación Pura, una mera abstracción de la imaginación.

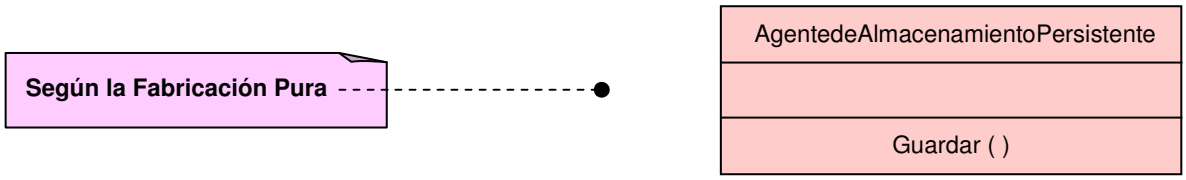


Fig. 51.- Clase generada atendiendo el patrón Fabricación Pura.

Sirve para resolver los siguientes problemas de diseño:

- La Venta conserva su buen diseño, con alta cohesión y bajo acoplamiento.
- La clase AgentedeAlmacenamientoPersistente es un objeto extremadamente genérico y reutilizable.

Crear una fabricación pura en este ejemplo es precisamente la situación donde se justifica su uso: hay que eliminar un diseño deficiente, con mala cohesión y acoplamiento y cambiar por un buen diseño que ofrezca un mayor potencial de reutilización.

Explicación para diseñar una fabricación pura debe buscarse ante todo un gran potencial de reutilización, asegurándose para ello de que sus responsabilidades sean pequeñas y cohesivas. Estas clases tienden a tener un conjunto de responsabilidades de granularidad fina.

Una fabricación pura suele partirse atendiendo a su funcionalidad y por lo mismo, es una especie de objeto de función central.

Generalmente se considera que la fabricación es parte de la capa de servicios orientada a objetos de alto nivel en una arquitectura.

Muchos patrones actuales del diseño orientado a objetos constituyen ejemplos de Fabricación Pura: Adaptador, Observador, Visitante y otros más.

Beneficios

- Se brinda soporte a una Alta Cohesión porque las responsabilidades se dividen en una clase de granularidad fina que se centra exclusivamente en un conjunto muy específico de tareas afines.

- Puede aumentar el potencial de reutilización debido a la presencia de las clases de Fabricación Pura de granularidad fina, cuyas responsabilidades pueden utilizarse en otras aplicaciones.

Problemas posibles: Puede perderse el espíritu de los buenos diseños orientados a objetos que se centran en los objetos y no en las funciones, pues las clases de Fabricación Pura casi siempre se dividen atendiendo a su funcionalidad; dicho con otras palabras, se confeccionan clases destinadas a conjuntos de funciones. Si se abusa de ello, la creación de clases de Fabricación Pura, originará un diseño centrado en procesos o funciones que se implementa en un lenguaje orientado a objetos.

El Patrón Prototipo

El patrón de diseño Prototype (Prototipo), tiene como finalidad crear nuevos objetos duplicándolos, clonando una instancia creada previamente.

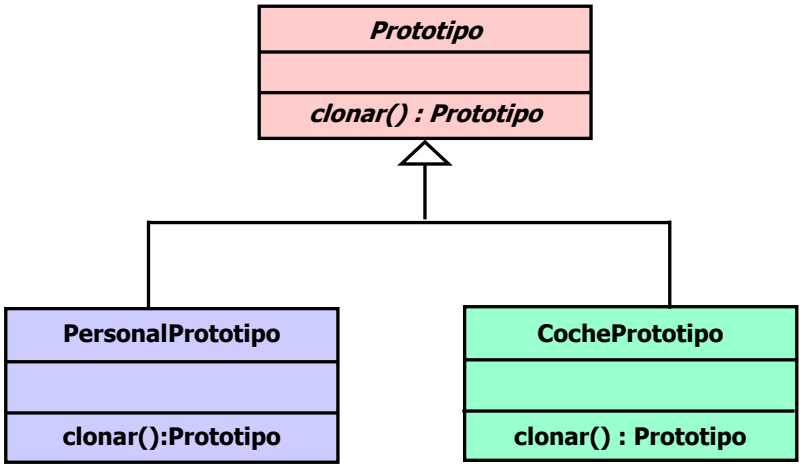


Fig. 52.- Ejemplo de patrón Prototipo.

Motivación: Este patrón es inspirado en ciertos escenarios donde es preciso abstraer la lógica que decide que tipos de objetos utilizará una aplicación, de la lógica que luego usarán esos objetos en su ejecución. Los motivos de esta separación pueden ser variados, por ejemplo, puede ser que la aplicación deba basarse en alguna configuración o parámetro en tiempo de ejecución para decidir el tipo de objetos que se debe crear.

Solución: Este patrón propone la creación de distintas variantes del objeto que nuestra aplicación necesite, en el momento y contexto adecuado. Toda la lógica necesaria para la decisión sobre el tipo de objetos que usará la aplicación en su ejecución debería localizarse aquí. Luego, el código que utilizan

estos objetos solicitará una copia del objeto que necesite. En este contexto, una copia significa otra instancia del objeto. El único requisito que debe cumplir este objeto es suministrar la prestación de clonarse. Cada uno de los objetos prototipo debe implementar el método Clone().

Implementación: Esta es una solución en el lenguaje C# de DotNet:

```
using System;
// "Prototype"
abstract class Prototype {
        private string _id;
        public Prototype( string id ) {
                _id = id;
        }
        public string ID {
                get{ return _id; }
        }
        abstract public Prototype Clone();
}
class ClsConcreta1 : Prototype {
        public ClsConcreta1 ( string id ) : base ( id ) {}
        override public Prototype Clone() {
                // Shallow copy
                return (Prototype)this.MemberwiseClone();
        }
}
class PrototypeClient {
        public static void Main(string[] argv) {
                // Instancio ClsConcreta1 y luego la clono
                ClsConcreta1 p1 = new ClsConcreta1 ("Clone-I");
                ClsConcreta1 c1 = (ClsConcreta1) p1.Clone();
                Console.WriteLine( "Clonación: {0}", c1.ID );
        }
}
```

El Patrón Builder (Constructor)

Propósito: Permite a un cliente construir un objeto complejo especificando sólo su tipo y contenido, ocultándole todos los detalles de la construcción del objeto.

Motivación: Supongamos un editor que quiere convertir un tipo de texto (p. ej. RTF) a varios formatos de presentación diferentes y que puede ser necesario en el futuro definir nuevos tipos de representación.

- El lector de RTF (RTFReader) puede configurarse con una clase de Conversor de texto (TextConverter) que convierta de RTF a otra representación.
- A medida que el RTFReader lee y analiza el documento, usa el conversor de texto para realizar la conversión: cada vez que reconoce un token RTF llama al conversor de texto para convertirlo.
- Hay subclases de TextConverter para cada tipo de representación.
- El conversor (Builder) está separado del lector (director): se separa el algoritmo para interpretar un formato textual de cómo se convierte y se representa.

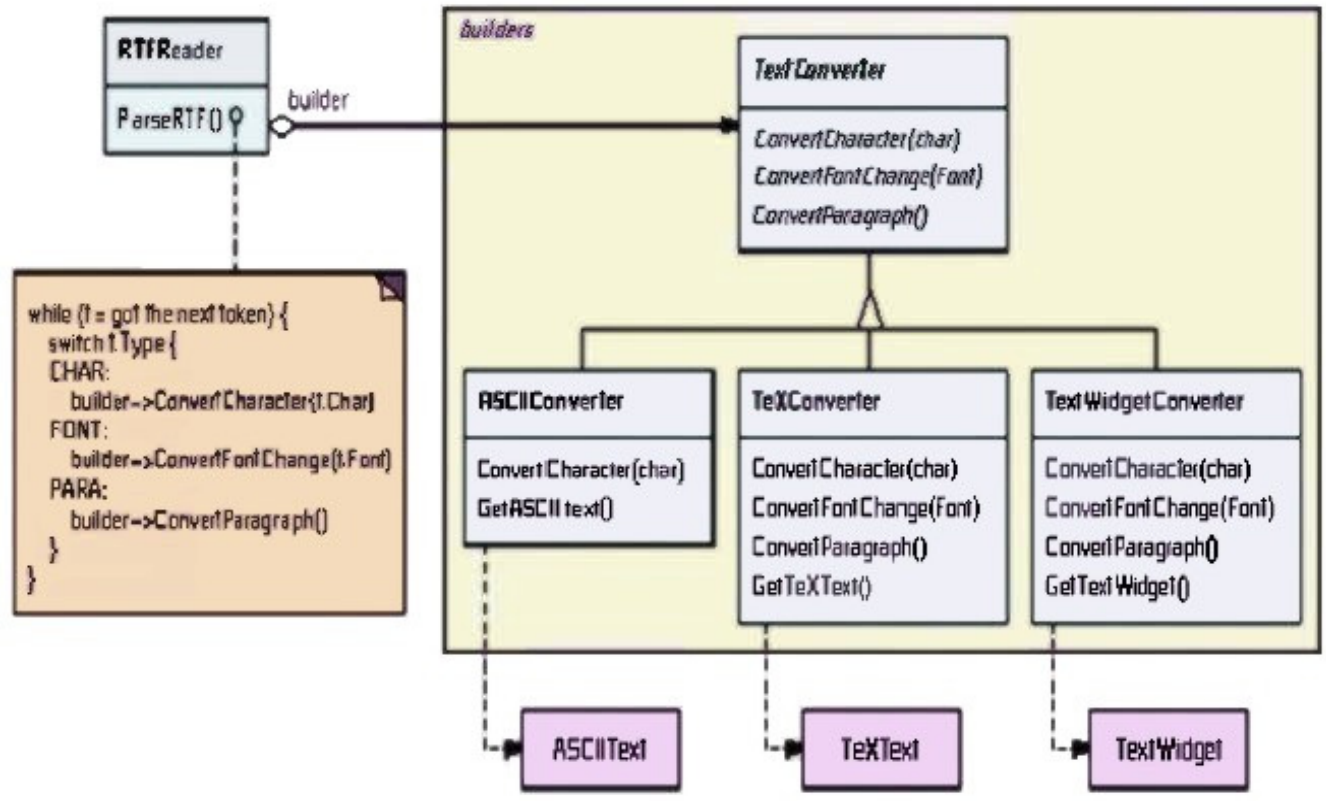


Fig. 53.- Patrones de creación.

Aplicación

- Cuando el algoritmo para crear un objeto complejo debe ser independiente de las partes que constituyen el objeto y cómo se juntan.
- Cuando el proceso de construcción debe permitir representaciones diferentes para el objeto que se está construyendo.

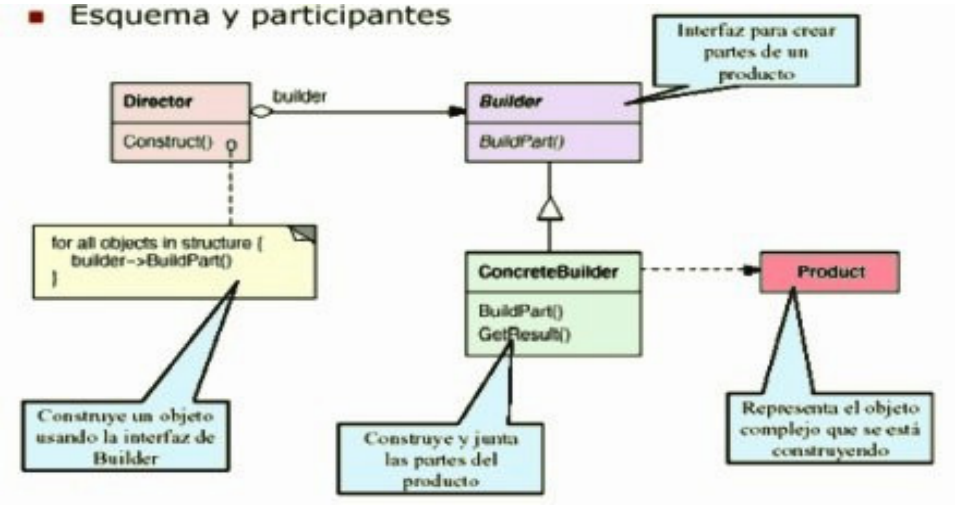


Fig. 54.- Patrones de creación.

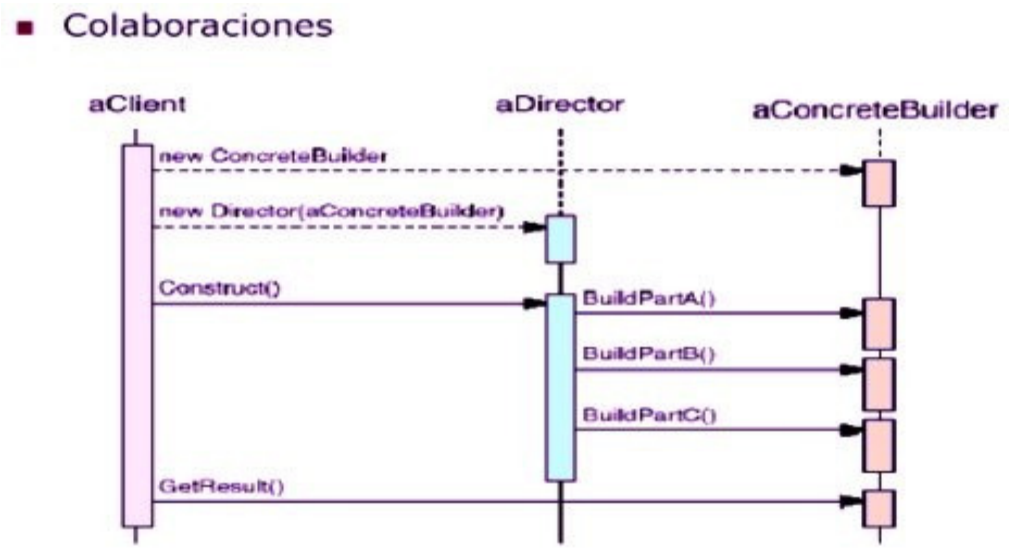


Fig. 55.- Patrones de creación.

Consecuencias

- Permite variar la representación interna de un producto.
    - El Builder ofrece una interfaz al Director para construir un producto y encapsula la representación interna del producto y cómo se juntan sus partes.
    - Si se cambia la representación interna basta con crear otro Builder que respete la interfaz.
- Separa el código de construcción del de representación.
    - Las clases que definen la representación interna del producto no aparecen en la interfaz del Builder.
    - Cada ConcreteBuilder contiene el código para crear y juntar una clase específica de producto.
    - Distintos Directors pueden usar un mismo ConcreteBuilder.
- Da mayor control en el proceso de construcción.
    - Permite que el Director controle la construcción de un producto paso a paso.
    - Sólo cuando el producto está acabado lo recupera el Director del Builder.

Implementación y patrones relacionados

- El Builder define las operaciones para construir cada parte.
    - El ConcreteBuilder implementa estas operaciones.
- Con la Factoría Abstracta también se pueden construir objetos complejos.
    - Pero el objetivo del patrón Builder es construir paso a paso.
    - El énfasis de la Factoría Abstracta es tratar familias de objetos.
- El objeto construido con el patrón Builder suele ser un Composite.
- El Método Factoría se puede utilizar por el Builder para decidir qué clase concreta "instanciar" para construir el tipo de objeto deseado.
- El patrón Visitor permite la creación de un objeto complejo, en vez de paso a paso, dando todo de golpe como objeto visitante.

El Patrón Singular

El Patrón de Diseño Singleton (instancia única) está diseñado para restringir la creación de objetos pertenecientes a una clase o el valor de un tipo a un único objeto.

Su intención consiste en garantizar que una clase sólo tenga una instancia y proporcionar un punto de acceso global a ella.

El patrón Singleton se implementa creando en nuestra clase un método que crea una instancia del objeto sólo si todavía no existe alguna. Para asegurar que la clase no puede ser instanciada nuevamente se regula el alcance del constructor (con atributos como protegido o privado).

La instrumentación del patrón puede ser delicada en programas con múltiples hilos de ejecución. Si dos hilos de ejecución intentan crear la instancia al mismo tiempo y esta no existe todavía, sólo uno de ellos debe lograr crear el objeto. La solución clásica para este problema es utilizar exclusión mutua en el método de creación de la clase que implementa el patrón.

Las situaciones más habituales de aplicación de este patrón son aquellas en las que dicha clase controla el acceso a un recurso físico único (como puede ser el ratón o un archivo abierto en modo exclusivo) o cuando cierto tipo de datos debe estar disponible para todos los demás objetos de la aplicación.

El patrón Singleton provee una única instancia global gracias a que:

- La propia clase es responsable de crear la única instancia.
- Permite el acceso global a dicha instancia mediante un método de clase.
- Declara el constructor de clase como privado para que no sea "instanciable" directamente.

Ejemplo de implementación: Una implementación **correcta** en el lenguaje de programación Java para programas multi-hilo es la solución conocida como "inicialización en demanda" sugerida por Bill Pugh:

```
public class Singleton {

    // El constructor privado no permite que se genere un constructor por defecto
(público)

    private Singleton() {}

    public static Singleton getInstance() {

        return SingletonHolder.instance;

    }

    private static class SingletonHolder {

            private static Singleton instance = new Singleton();

    }

}
```

Patrones relacionados:

- **Abstract Factory:** Muchas veces son implementados mediante Singleton, ya que normalmente deben ser accesibles públicamente y debe haber una única instancia que controle la creación de objetos.

- **Monostate:** Es similar al Singleton, pero en lugar de controlar el "instanciado" de una clase, asegura que todas las instancias tengan un estado común, haciendo que todos sus miembros sean de clase.

El Patrón Singular también es útil para objetos grandes y caros que no deben ser "instanciados" múltiples veces.

La razón por la que debe utilizarse un método de fábrica, en vez de un constructor, es la segunda debilidad de los constructores de Java: siempre devuelven un objeto nuevo, nunca un objeto ya existente.

2.4.5.3 Patrones Estructurales

Adapter

El Patrón Adapter convierte la interfaz de una clase en la interfaz que el cliente espera.

El Adapter permite que clases con interfaces incompatibles puedan trabajar juntas.

Este patrón se denomina también Wrapper.

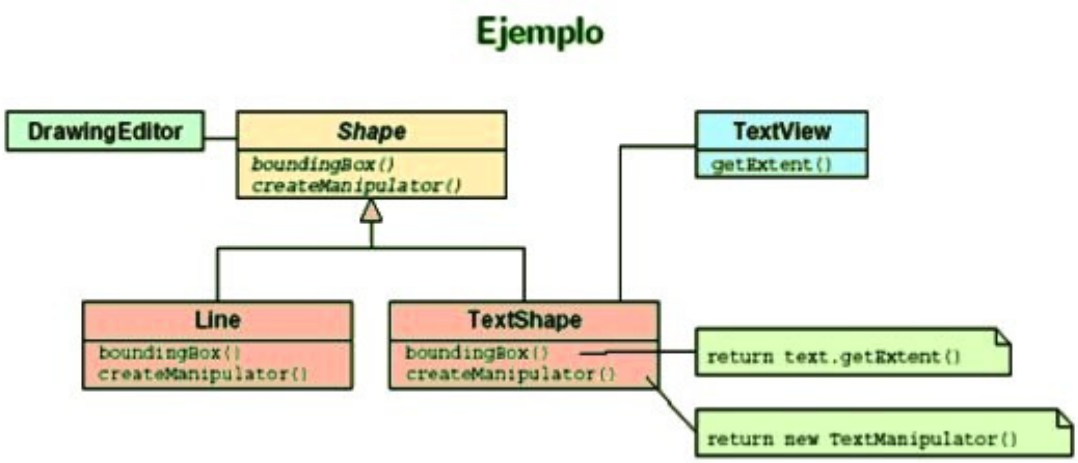


Fig. 56.- Ejemplo de Patrón Adapter.

Aplicación

- Se desea utilizar una clase ya existente pero cuya interfaz no coincide con la que se necesita.
- Se desea crear una clase que colabora con otras clases que no tienen interfaces compatibles.
- Se desea adaptar varias subclases ya existentes adaptando la interfaz de su clase padre como un (Object Adapter).

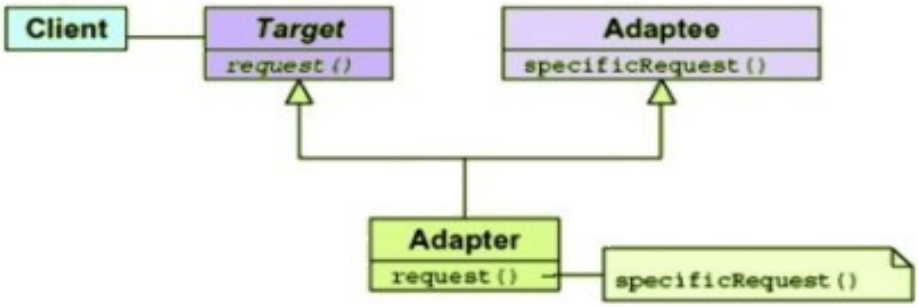


Fig. 57.- Estructura Class Adapter.

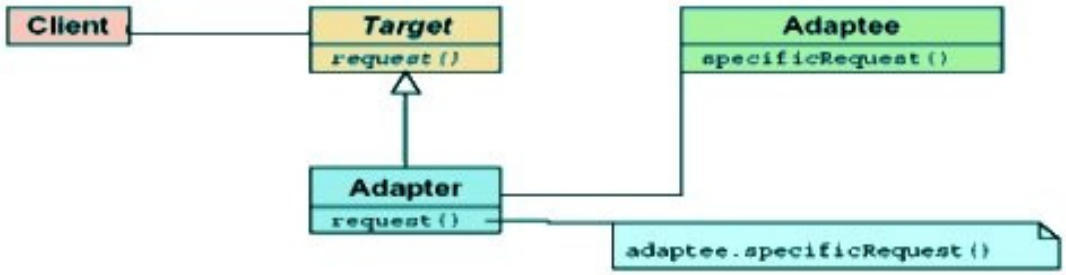


Fig. 58.- Estructura Object Adapter.

Participantes

- **Target:** Define la interfaz específica del dominio en el que se quiere hacer uso de la clase que se adapta.
- **Client:** Utiliza objetos que implementan la interfaz definida por el Target.
- **Adaptee:** Presenta su interfaz original, que es la que se tiene que adaptar.
- **Adapter:** Adapta la interfaz del objeto adaptado a la definida por el Target.

Consecuencias

- Class Adapter.

  - No sirve para adaptar una clase y todas sus subclases.
  - Hace que el Adapter herede el comportamiento del Adaptee.
  - Provoca la creación de un único objeto, no necesita indirecciones adicionales.
- Object adapter.
  - Permite que un Adapter funcione con varios Adaptee.
  - Hace más complicado heredar el comportamiento del Adaptee.
- ¿Cuánto trabajo tendrán los Adapter?

Detalles de implementación

- El Class Adapter es menos flexible que el Object Adapter.
- El Object Adapter permite que un adaptador trabaje con muchos Adaptee (el propio Adaptee y todas sus subclases)

El Patrón Puente

El Puente es uno de los patrones básicos y su definición formal es la siguiente:

"Desacoplar una abstracción de su implementación de tal forma que las dos puedan variar independientemente".

Probablemente nosotros aplicamos este patrón más de lo que nos damos cuenta. Vamos a tomar un ejemplo común.

Al escribir código en nuestros formularios y clases queremos naturalmente atrapar las cosas que pueden salir mal y usualmente (considerándonos desarrolladores) queremos decir a los usuarios cuándo van a ocurrir. La solución más obvia y sencilla, es incluir un par de líneas de código en el método apropiado. El resultado puede verse así:

```
IF NOT <Una función que devuelve True/False>
  lcText = "Chequeo fallido para devolver el valor correcto" + CHR(13)
  lcText = lcText + "Presione cualquier tecla para re-entrar el valor"
  WAIT lcText WINDOW
  RETURN .F.
ENDIF
```

En toda nuestra aplicación pueden existir docenas de situaciones donde mostramos una ventana Wait de este tipo, para mantener al usuario al corriente de lo que está ocurriendo y esto funciona perfectamente bien mientras ocurre una de las dos cosas. Cualquier usuario le dirá que realmente odia estas molestas ventanas de espera "Wait Windows" y preferirá una ventana estilo "message box", o peor, tendremos

que implantar el código en un entorno que no soporta una ventana tipo "Wait Windows". Es posible lograrlo con un componente COM, o en un formulario Web. Ahora podemos ir y buscar cada ocurrencia de este código por nuestra aplicación, para cambiarla para que garantice el nuevo requerimiento. Las posibilidades de hacer esta tarea correcta la primera vez (no olvide ninguna ocurrencia y re-codifique cada una perfectamente) están muy cerca de cero. Incluso, si confiamos en que nuestros resultados son correctos, tendremos que hacer un grupo de verificaciones para asegurarnos.

Entonces, ¿qué tiene que ver esto, con el patrón Puente? Bueno, la razón de que tenemos ese problema es porque hemos fallado al reconocer que estábamos acoplando una abstracción (mostrando un mensaje al usuario) y su implementación (la "Wait Window"). Si hubiéramos hecho un puente en su lugar, hubiéramos evitado el problema. Así es como se muestra el mismo código si lo hubiéramos implementado, utilizando un Patrón de Puente.

```
IF NOT <Una función que devuelve True/False>
  lcText = " Chequeo fallido para devolver el valor correcto" + CHR(13)
  loMsgHandler = This.oMsgHandler
  loMsgHandler.ShowMessage( lcText )
  RETURN .F.
ENDIF
```

La diferencia radica en que usando esta segunda forma no sabemos más, ni nos preocuparemos por cómo se mostrará el mensaje (por eso tampoco necesitamos la línea "presione cualquier tecla", que puede ser agregada en el manipulador de mensaje si es requerido). Todo lo que se necesita conocer es dónde tomar una referencia al objeto que va a manipular el mensaje por nosotros, (por supuesto, esto es posible, porque en este requerimiento todo posible manipulador implementará la interfaz apropiada, en este caso el método ShowMessage()).

Este es el origen de la referencia, que es un puente. En este ejemplo la propiedad "oMsgHandler" proporciona el puente entre el código que requiere un mensaje y el mecanismo para manipularlo.

Ahora todo lo que necesitamos es cambiar la forma en que nuestro mensaje es manipulado, para cambiar el objeto de referencia guardado en esta propiedad. Esto es algo que pudo haberse hecho incluso en tiempo de ejecución, independientemente del entorno en el que el objeto padre se haya "instanciado". Esta estrategia desacopla exitosamente la abstracción de la implementación y como resultado, nuestro código es mucho más reutilizable.

¿Cuáles son los componentes de un puente?

Un puente tiene dos componentes esenciales, la "abstracción", que es el objeto responsable de inicializar una operación, y la "implementación", que es el objeto que la lleva a cabo, siguiente figura. La abstracción conoce su implementación porque guarda una referencia a la misma, o porque lo posee (por ejemplo, lo contiene). Observe que, a pesar de esta estructura, el diagrama denota que la abstracción crea la implementación y que no es en absoluto un requerimiento del patrón. Un puente también puede hacer uso

de una referencia ya existente para una misma implementación de objeto. De hecho, diseñar puentes de esta forma, puede ser muy eficiente porque diferentes objetos abstracción pueden ser utilizados en un mismo objeto implementación.

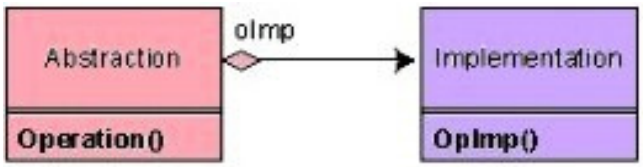


Fig. 59.- El Patrón de Puente Básico.

No hay que confundir el envío de mensajes con el puente. Si un método de un botón de comandos llama a un método en su formulario padre, esto implementa directamente la acción necesaria, que es enviar un mensaje, no un puente. Sin embargo, si el método del formulario va a llamar a otro objeto para que lleve a cabo la acción, entonces tenemos un puente.

Interesante, otra posibilidad ocurre cuando un formulario (u otro contenedor) tiene una propiedad que guarda una referencia a un objeto de implementación. Un objeto contenedor (por ejemplo un botón de comandos en un formulario) puede acceder a esta propiedad directamente y tomar una referencia local al objeto de implementación. Puede entonces, llamar a los métodos del objeto de implementación directamente. Ahora, ¿este es un puente, o un mensaje enviado? De hecho, probablemente podría argumentar ambos lados del caso con igual justificación. Sin embargo, la respuesta en realidad no importa. Este asunto surge sólo porque Visual FoxPro expone las propiedades como "Públicas" de forma predeterminada y también implementa puntos hacia atrás, lo que permite a los objetos dirigirse a sus padres directamente. En la mayoría de los entornos orientados a objetos, es sencillamente imposible para los objetos comportarse tan toscamente.

Entonces, para implementar un puente necesita identificar qué objeto va a realizar la función de abstracción y cuál la implementación. Una vez que lo haya definido, todo lo que queda por decidir es cómo el puente entre los dos será codificado y esto dependerá enteramente del escenario.

El Patrón Composite

Intención

- Componer objetos en jerarquías todo-parte y permitir a los clientes tratar objetos simples y compuestos de manera uniforme.

Aplicabilidad: utilizar cuando...

- Se quiera representar jerarquías de objetos todo-parte.
- Se quiera que los clientes puedan ignorar la diferencia entre objetos simples y compuestos y tratarlos uniformemente.

Por ejemplo: suponga que se desea representar un árbol de objetos (como el organigrama de una compañía) o una lista enlazada de objetos.

La parte central de una solución es usar la relación de herencia/agregado que ilustra la siguiente figura.

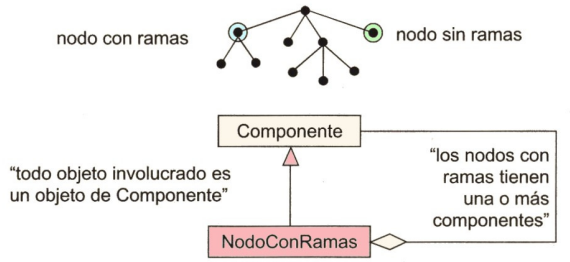


Fig. 60.- Base para las estructuras Compuesto y Decorador.

La idea es mostrar que algunas componentes agregan otras componentes, como se ve en la siguiente figura. El modelo de clases incluye componentes, objetos de Ramas, que no agregan otros componentes. El patrón funciona de la siguiente manera. Un objeto de Compuesto ejecuta hacer() de manera que depende de un objeto de Ramas, que es una ejecución sencilla, o un objeto de NodoConRamas. Un objeto de NodoConRamas llama cada uno de sus "descendientes" agregados para ejecutar hacer(). Estos objetos de Componente continúan el proceso hacia debajo de la estructura de árbol.

Suponga, por ejemplo, que el árbol es en realidad un organigrama administrativo de la organización y "hacer()" es "despliegueEmpleado()". En este caso, un mejor nombre para Componente sería Empleado y para NodoConRama sería Supervisor. Un término para Rama sería algo como ContribuciónIndividual. Llamar a despliegueEmpleado() en el objeto de Componente, que representa la organización completa, ocasiona que cada uno de los objetos ejecute su método despliegueEmpleado() y después llame a despliegueEmpleado() para todos sus subordinados. Para los vicepresidentes (por ejemplo, NodoConRamasTipoA que es SeniorVP) el método despliegueEmpleado() puede tener un efecto diferente, como la inclusión de una versión biográfica para impresión comparada con despliegueEmpleado() para los supervisores de primera línea (donde NodoConRamasTipoB es el PrimerSupervisorLinea), etcétera.

En general la estructura de este patrón es la siguiente:

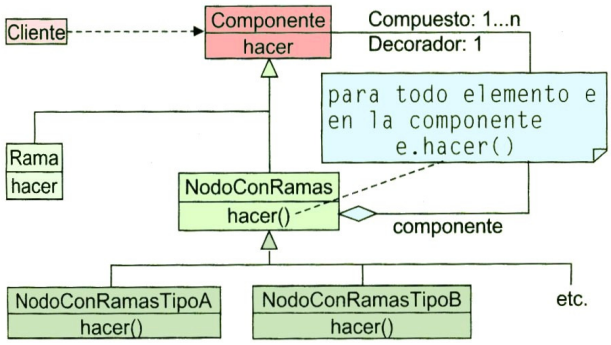


Fig. 61.- Patrones de diseño Compuesto y Decorador.

Participantes

- **Componente:** Declara la interfaz para la composición de objetos e implementa acciones por defecto cuando es oportuno.
- **Simple:** Representa los objetos de la composición que no tienen hijos e implementa sus operaciones.
- **Compuesto:** Implementa las operaciones para los componentes con hijos y almacena a los hijos.
- **Cliente:** Utiliza objetos de la composición mediante la interfaz de Componente.

Colaboraciones

- Los clientes usan la interfaz de Componente.
- Los objetos simples responden directamente, mientras que los compuestos suelen reenviar la operación a sus componentes.

Consecuencias (ventajas e inconvenientes)

- Permite tratamiento uniforme de objetos simples y complejos.
- Simplifica el código de los clientes, que sólo usan una interfaz.
- Facilita añadir nuevos componentes sin afectar a los clientes.
- Es difícil restringir los tipos de los hijos.
- Las operaciones de gestión de hijos en los objetos simples pueden presentar problemas: seguridad frente a flexibilidad.

El Patrón Decorador

El Decorador describe una solución al problema de agregar una funcionalidad a un objeto sin cambiar realmente el código en el objeto.

La definición formal del Decorador, es:

"Adjuntar responsabilidad adicional a un objeto dinámicamente".

La necesidad de cambiar dinámicamente la funcionalidad o comportamiento de un objeto surge típicamente en una de estas dos situaciones. Primero, cuando el código fuente del objeto está no disponible, puede ser que el objeto es un control o una biblioteca de clases de terceras partes está utilizada. Segundo, cuando la clase que es ampliamente utilizada para la responsabilidad específica, se necesita sólo en una o más situaciones particulares y puede ser inapropiado agregar simplemente el código de la clase.

Como ejemplo veremos el problema de determinar la cantidad de la tasa para aplicar el precio en dependencia de la localidad.

Supongamos que tenemos un objeto base de cálculo (CH15.vcx::cntTaxRoot) que trabaja exponiendo un método llamado CalcTax(), que toma dos parámetros, un valor y un índice y retorna la tasa calculada. Afrontamos el problema básico de tratar con la tasa en el formulario utilizando un patrón de puente para separar la implementación de la interfaz. Sin embargo, esto no es suficientemente flexible para acoplar con diferentes índices de tasa en diferentes localidades.

El Patrón Decorador nos permite solucionar este problema sin necesidad de crear más clases. En lugar de definir un nuevo objeto el que tiene exactamente la misma interfaz como el cálculo de la tasa, pero que incluye el código necesario para determinar la tasa apropiada para una localización dada. El objeto "mira" justo el cálculo de la tasa como su cliente; ya que guarda una referencia al objeto de cálculo de tasa y puede "pre-procesar" cualquier requerimiento de un índice y luego, simplemente puede generar la implementación real cuando esté lista.

¿Cuáles son los componentes del Decorador?

Un Decorador, tiene dos requerimientos esenciales. Primero, necesitamos la clase Implementación que define la interfaz y el núcleo de funcionalidad. Segundo, nosotros necesitamos un Decorador que reproduce la interfaz de implementación y guarda una referencia a él. El objeto cliente ahora dirige las llamadas que pueden ir directamente a la implementación, en su lugar del objeto Decorador. La estructura básica del patrón Decorador se muestra en la figura siguiente.

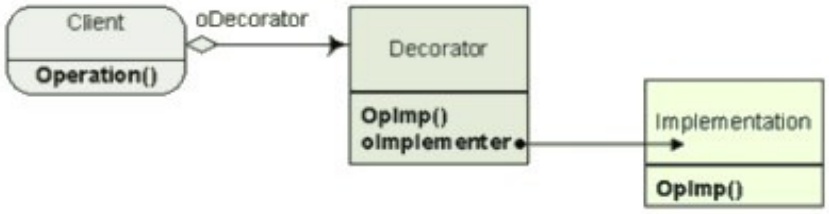


Fig. 62.- Patrón Decorador.

Este patrón es en realidad un puente extendido. Debido a que el cliente está afectado, puede dirigir el objeto decorador como si realmente fuera la implementación al final del puente estándar, porque la interfaz de los dos es la misma. Debido a que la implementación está afectada, la petición mira exactamente igual que si fuera directamente desde el cliente. No necesita nunca conocer que el decorador existe.

Resumen del Patrón Decorador

El Patrón Decorador es utilizado cuando deseamos modificar el comportamiento básico de una instancia específica sin la necesidad siquiera de crear una nueva subclase, o cambiar el código en el original. El Decorador, esencialmente actúa como un pre-procesador para su implementación interponiendo su cliente y la implementación.

El Patrón Fachada

Se le da el nombre de Fachada a la clase definida que ofrece una interfaz común con un conjunto heterogéneo de interfaces. Las interfaces heterogéneas pueden ser un conjunto de funciones, un esquema, un grupo de otras clases o un subsistema (local o remoto).

Contexto/Problema: Se requiere una interfaz común unificada con un conjunto heterogéneo de interfaces, como la de un subsistema. ¿Qué debe hacerse?

Intención: El patrón Fachada simplifica el acceso a un conjunto de clases proporcionando una única clase que todos utilizan para comunicarse con dicho conjunto de clases.

Ventajas

- Los clientes no necesitan conocer las clases que hay tras la clase Fachada.
- Se pueden cambiar las clases "ocultas" sin necesidad de cambiar los clientes. Sólo hay que realizar los cambios necesarios en Fachada.

El Problema

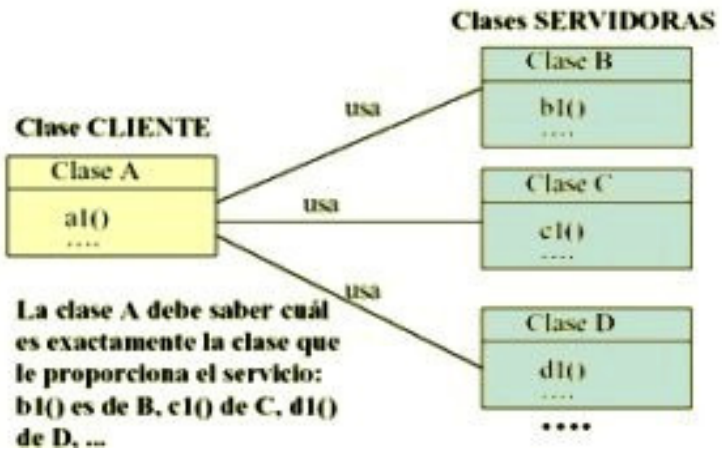


Fig. 63.- Problema a resolver con el Patrón Fachada.

Solución: Definir una sola clase que unifique las interfaces y asignarle la responsabilidad de colaborar con el subsistema.

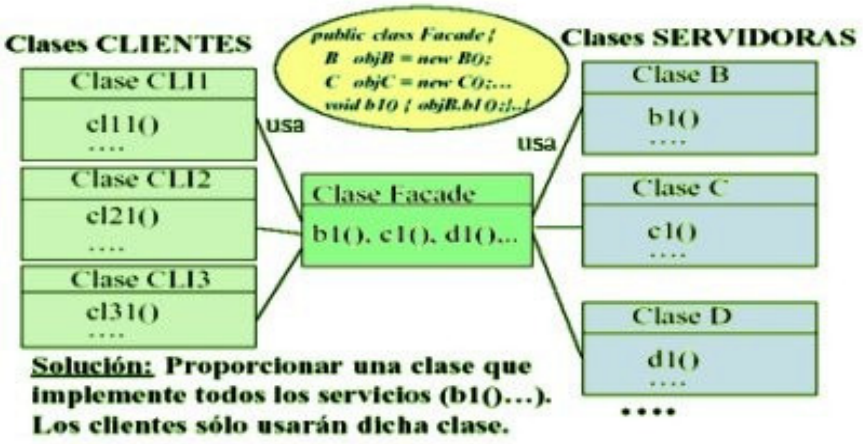


Fig. 64.- Clase Fachada.

Ejemplo

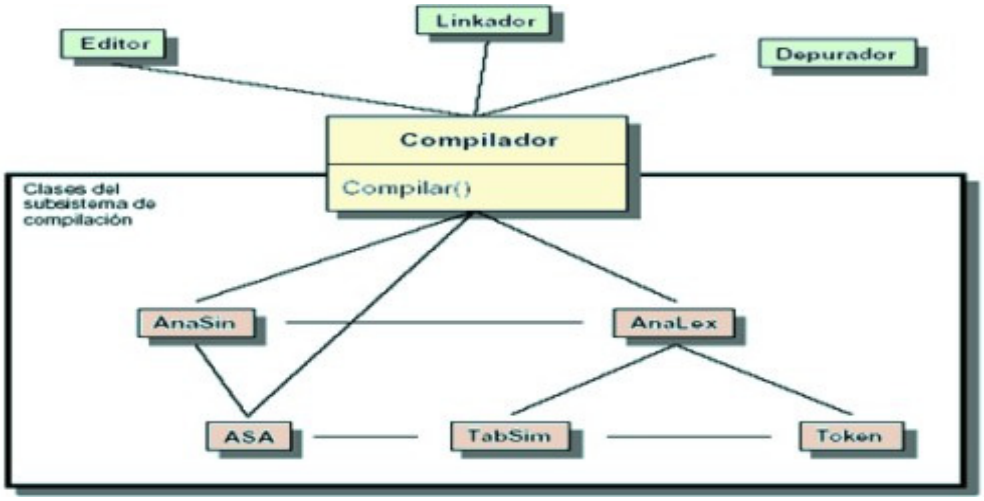


Fig. 65.- Ejemplo práctico del Patrón Fachada.

El patrón Peso Ligero

Propósito

Uso de objetos compartidos para soportar eficientemente un gran número de objetos de poco tamaño.

Motivación

En una aplicación editor de documentos, ¿modelamos los caracteres (letras y símbolos) mediante una clase?

Un "flyweight" es un objeto compartido que puede ser utilizado en diferentes contextos simultáneamente.

- No hace asunciones sobre el contexto.
- Estado intrínseco vs. Estado extrínseco.
- Estado intrínseco se almacena en el "flyweight" y consiste de información que es independiente del contexto y se puede compartir.
- Estado extrínseco depende del contexto y por tanto no puede ser compartido.
- Objetos clientes son responsables de pasar el estado extrínseco al "flyweight" cuando lo necesita.
- Objetos "flyweight" se usan para modelar conceptos o entidades de los que se necesita una gran cantidad en una aplicación, por ej. caracteres de un texto.
- "Flyweight" letra de un texto:
    - Estado intrínseco: código de la letra o símbolo.
    - Estado extrínseco: información sobre posición y estilo.

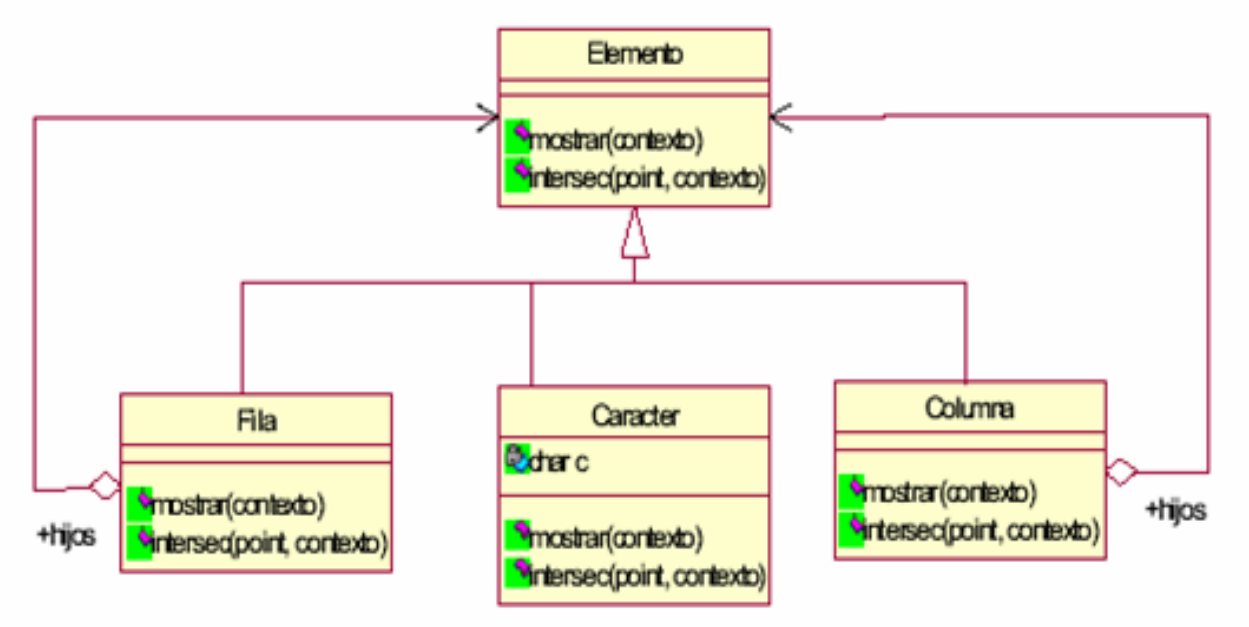


Fig. 66.- Patrón Peso Ligero.

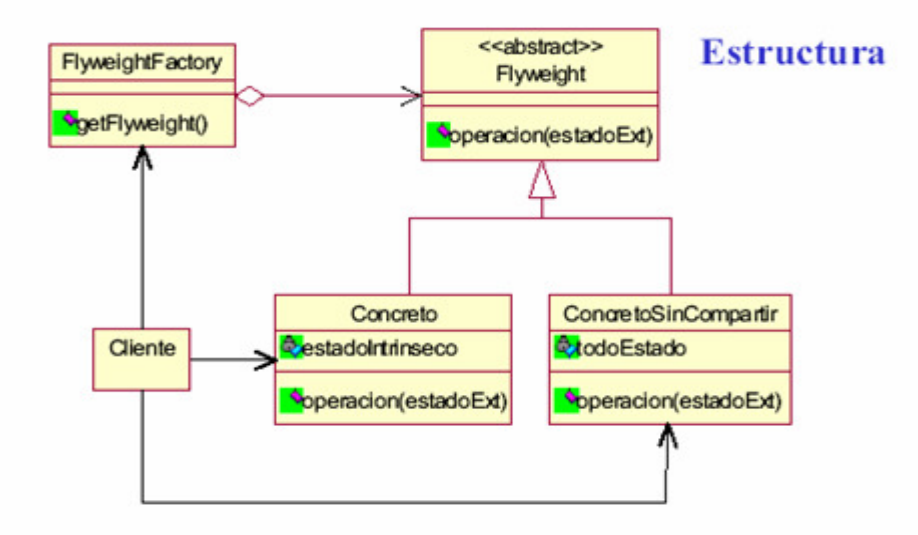


Fig. 67.- Estructura del Patrón Peso Ligero.

Aplicabilidad

- Aplicarlo siempre que se cumplan las siguientes condiciones:
  - Una aplicación utiliza un gran número de cierto tipo de objetos.
  - El costo de almacenamiento es alto debido al excesivo número de objetos.
  - La mayor parte del estado de esos objetos puede hacerse extrínseco.
  - Al separar el estado extrínseco, muchos grupos de objetos pueden reemplazarse por unos pocos objetos compartidos.
  - La aplicación no depende de la identidad de los objetos**.**

Consecuencias

- Puede introducir costos run-time debido a la necesidad de calcular y transferir el estado extrínseco.
- La ganancia en espacio depende de varios factores:
  - La reducción en el número de instancias.
  - El tamaño del estado intrínseco por objeto.
  - Si el estado extrínseco es calculado o almacenado.
- Interesa un estado extrínseco calculado.
- Se suele combinar con el Composite.

Implementación

- La aplicabilidad depende de la facilidad de obtener el estado extrínseco. Idealmente debe poder calcularse a partir de una estructura de objetos que necesite poco espacio de memoria.
- Debido a que los flyweight son compartidos, no deberían ser instanciados directamente por los clientes: clase FlyweightFactory.

El Patrón Proxy

Propósito

- Proporcionar un sustituto (surrogate) de un objeto para controlar el acceso a dicho objeto.

Motivación

- Diferir el costo de crear un objeto hasta que sea necesario usarlo: creación bajo demanda.
- Un editor de documentos que incluyen objetos gráficos.
- ¿Cómo ocultamos que una imagen se creará cuando se necesite? Manejar el documento requiere conocer información sobre la imagen.
- Hay situaciones en las que un cliente no referencia o no puede referenciar a un objeto directamente, pero necesita interactuar con él.
- Un objeto Proxy puede actuar como intermediario entre el objeto cliente y el objeto destino.
- El objeto Proxy tiene la misma interfaz como el objeto destino.
- El objeto Proxy mantiene una referencia al objeto destino y puede pasarle a él los mensajes recibidos (delegación).

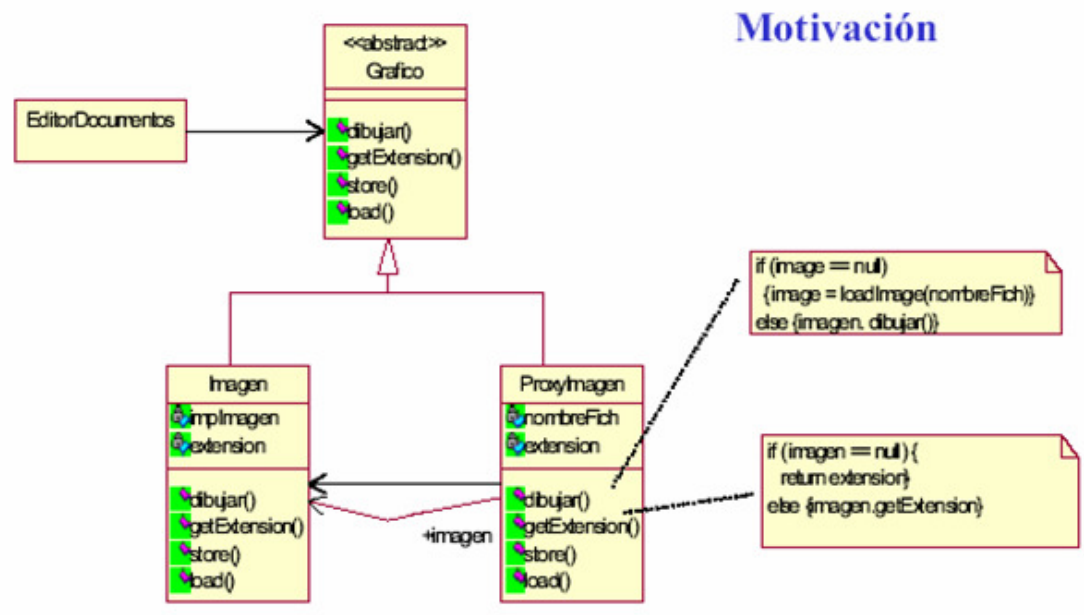


Fig. 68.- Motivación del Patrón Proxy.

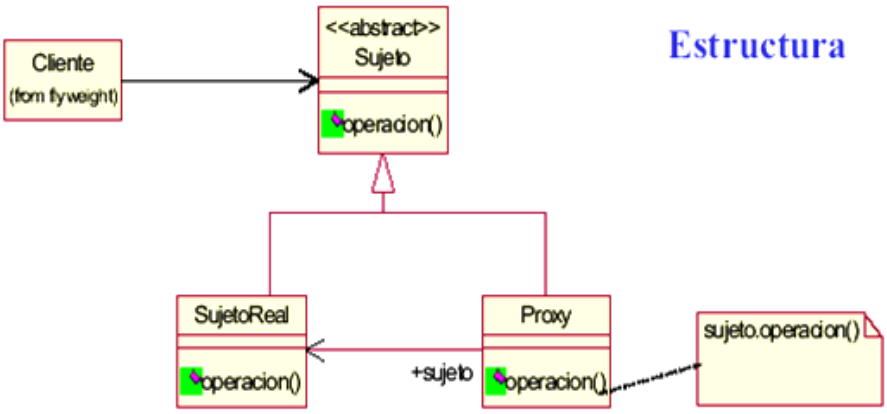


Fig. 69.- Estructura del Patrón Proxy.

Aplicabilidad

- Siempre que hay necesidad de referenciar a un objeto mediante una referencia más rica que un puntero o una referencia normal.

- Situaciones comunes:
  - Proxy acceso remoto (acceso a un objeto en otro espacio de direcciones).
  - Proxy virtual (crea objetos grandes bajo demanda).
  - Proxy para protección (controlar acceso a un objeto).
  - Referencia inteligente (smart reference, proporciona operaciones adicionales).

Consecuencias

- Introduce un nivel de indirección para:
  - Un Proxy remoto oculta el hecho que objetos residen en diferentes espacios de direcciones.
  - Un Proxy virtual tales como crear o copiar un objeto bajo demanda.
  - Un Proxy para protección o las referencias inteligentes permiten realizar tareas de control sobre los objetos accedidos.

El Patrón Cadena de Responsabilidad

Una definición formal de Cadena de Responsabilidad es:

"Evitar el acoplamiento del emisor del requerimiento y el receptor, al obtener más de un objeto con posibilidad de tratar el requerimiento. Es la cadena de objetos que reciben y pasan la petición a lo largo de la cadena hasta que un objeto los manipule".

En una Cadena de Responsabilidades cada objeto sabe cómo evaluar la petición para una acción y si no la puede controlar por sí mismo, sabe solamente cómo pasarla a otro objeto, por ello la palabra "cadena". La consecuencia de esto es que el cliente, (que inicia la petición de la acción) ahora sólo necesita conocer sobre el primer objeto en la cadena. Por otra parte, cada objeto en la cadena, solo necesita conocer también sobre un objeto, el siguiente en la cadena. La Cadena de Responsabilidades puede ser implementada utilizando una cadena predefinida o estática, o las cadenas se pueden construir en tiempo de ejecución teniendo cada objeto su propio sucesor cuando sea necesario.

¿Cuáles son los componentes de una cadena de responsabilidad?

Una Cadena de Responsabilidad puede ser implementada al crear una clase "manipuladora" (handler) abstracta (para especificar la interfaz y funcionalidad genérica) y crear subclases concretas para definir varias posibles implementaciones (figura siguiente). Sin embargo, no existen requerimientos absolutos para todos los miembros de la cadena de responsabilidades, para descender a partir de la misma clase, proporcionando que todos ellos soporten la interfaz necesaria para integrar con otros miembros de la cadena. Los objetos clientes necesitan una referencia a la subclase específica, la cual es su punto de entrada individual a la cadena. (Observe que no todos los clientes necesitan utilizar el mismo punto de entrada).

Observe que hasta ahora, hemos visto el Patrón de Puente básico, porque cada enlace en la cadena es en realidad un puente entre una abstracción y una implementación. Lo diferente es que un único objeto puede desempeñar varios roles en dependencia con su situación. De esta manera, el primer enlace tiene al cliente como abstracción y el primer handler concreto, como la implementación. Sin embargo, el segundo enlace tiene ahora el primer handler desempeñando el rol de abstracción y el segundo handler como implementación. Este patrón puede, en teoría al menos, repetirse hasta infinito.

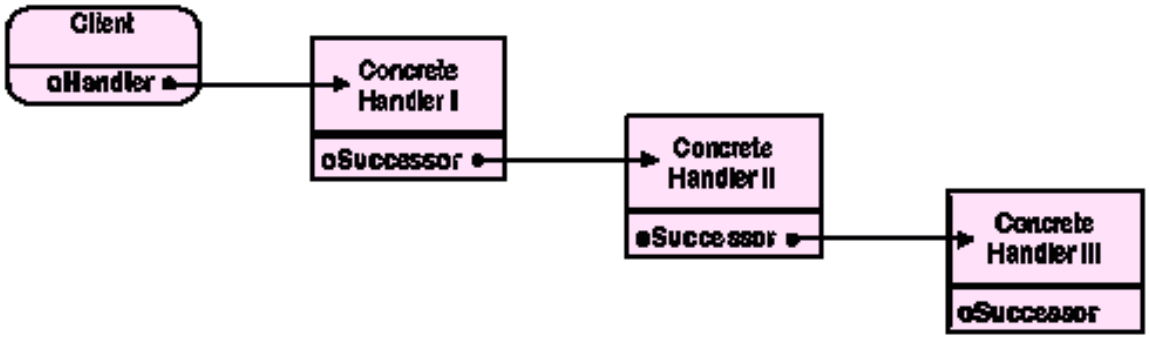


Fig. 70.- Patrón Cadena de Responsabilidad.

Patrón Cadena de Responsabilidades Básico

Con el objeto de implementar un Patrón de Cadena de Responsabilidades, necesita definir una clase handler abstracta y tantas subclases diferentes como necesite. La diferencia con el Patrón de Estrategia es que la clase abstracta debe definir el mecanismo, por el cual un objeto puede determinar si puede manipular el requerimiento para todos los manipuladores o pasarlos a una propiedad para guardar una referencia al objeto siguiente en la cadena. De hecho, no existen requerimientos absolutos para todos los manipuladores que hereden del mismo manipulador abstracto, proporcionando que ellos adhieran la mínima interfaz definida.

Propósito

- Evita acoplar el emisor de un mensaje a su receptor dándole a más de un objeto la posibilidad de manejar la solicitud. Se define una cadena de objetos, de modo que un objeto pasa la solicitud al siguiente en la cadena hasta que uno la maneja.

Motivación

- Facilidad de ayuda sensible al contexto.
- El objeto que proporciona la ayuda no es conocido al objeto que inicia la solicitud de ayuda (por ej. un Button).

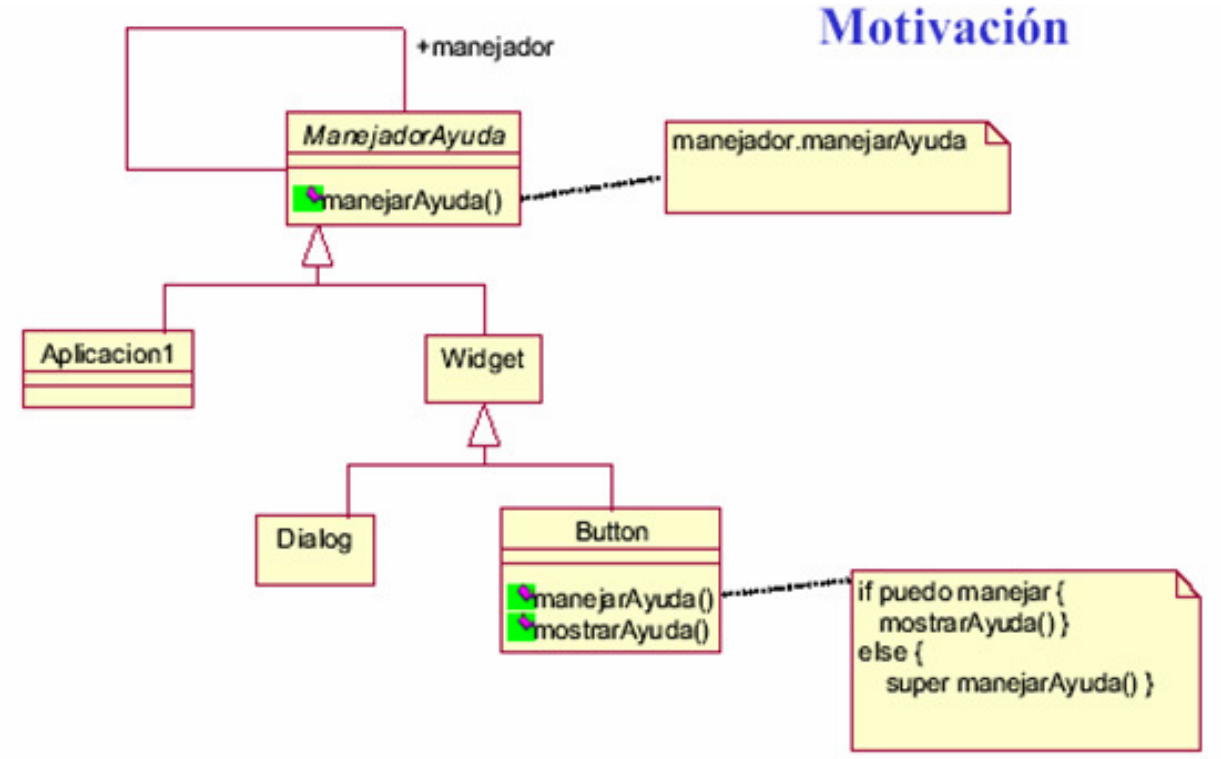


Fig. 71.- Patrón Cadena de Responsabilidades Básico 1.

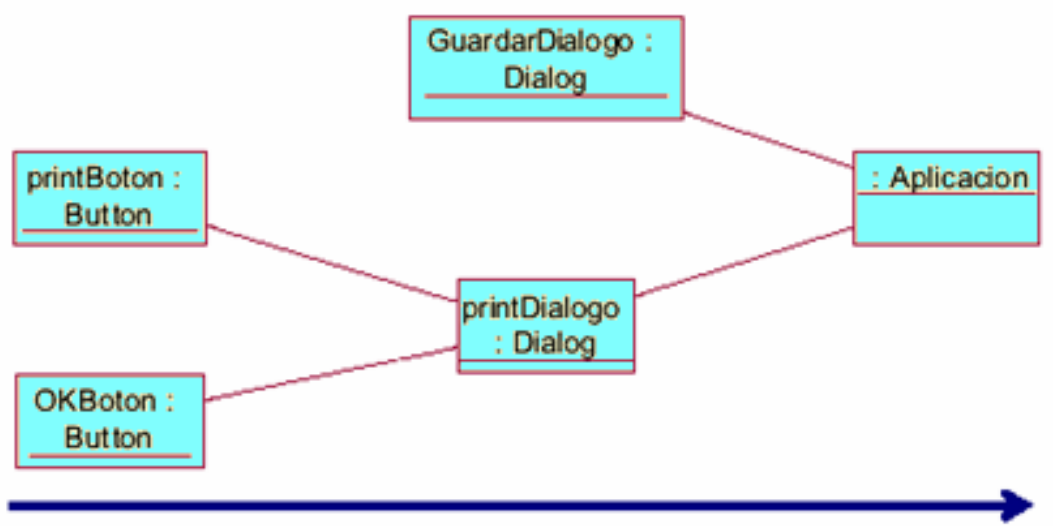


Fig. 72.- Patrón Cadena de Responsabilidades Básico 2.

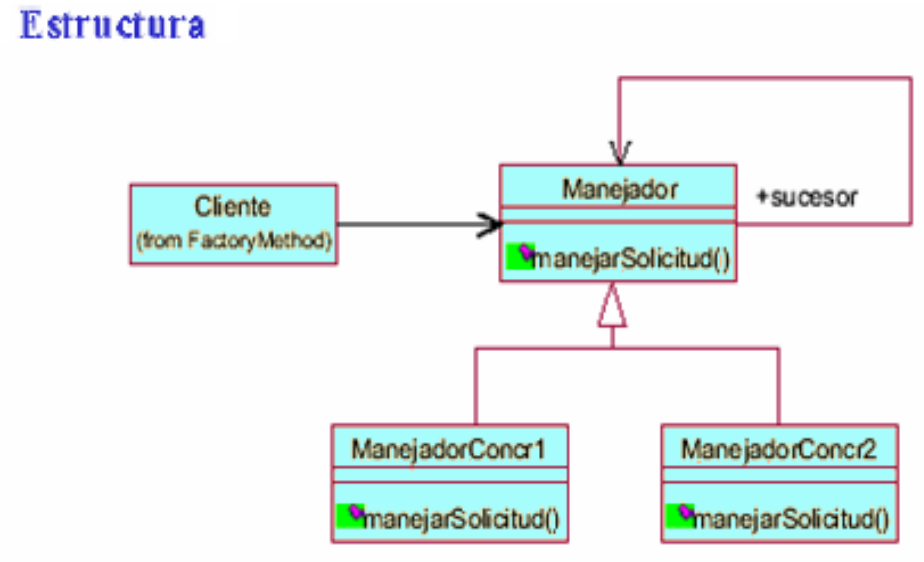


Fig. 73.- Patrón Cadena de Responsabilidades Básico 3.

Aplicabilidad

- Más de un objeto puede manejar una solicitud y el manejador no se conoce a priori.
- Se desea enviar una solicitud a uno entre varios objetos sin especificar explícitamente el receptor.
- El conjunto de objetos que puede manejar una solicitud puede ser especificado dinámicamente.

Consecuencias

- Reduce acoplamiento.
- Proporciona flexibilidad al asignar responsabilidades.
- No se garantiza el manejo de la solicitud.

Patrón Comando

Propósito

- Encapsula un mensaje como un objeto, permitiendo parametrizar (adecuar) a los clientes con diferentes solicitudes, añadir a una cola las solicitudes y soportar funcionalidad deshacer/rehacer (undo/redo).

Motivación

- Algunas veces es necesario enviar mensajes a un objeto sin conocer el selector del mensaje o el objeto receptor.
- Por ejemplo widgets (botones, menús, etc.) realizan una acción como respuesta a la interacción del usuario, pero no se puede hacer explícito en su implementación.

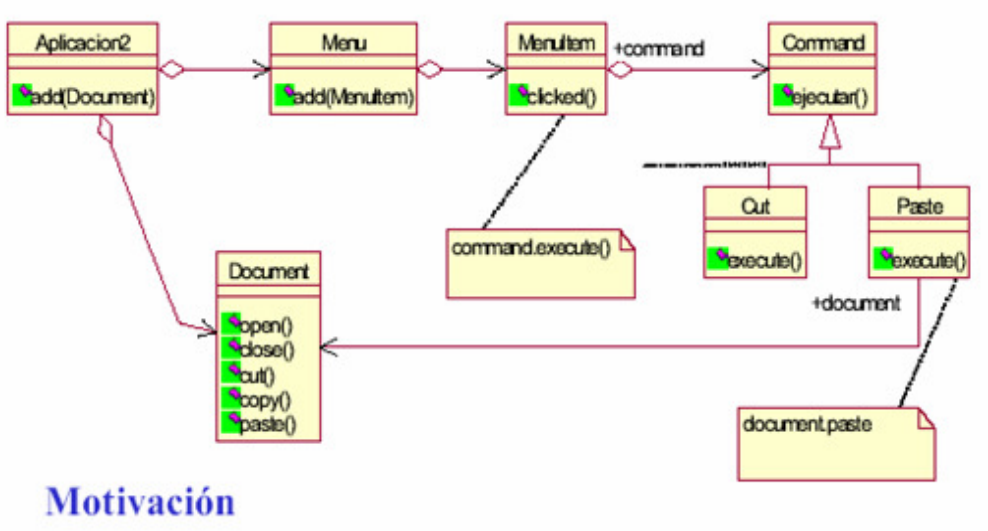


Fig. 74.- Motivación del Patrón Comando.

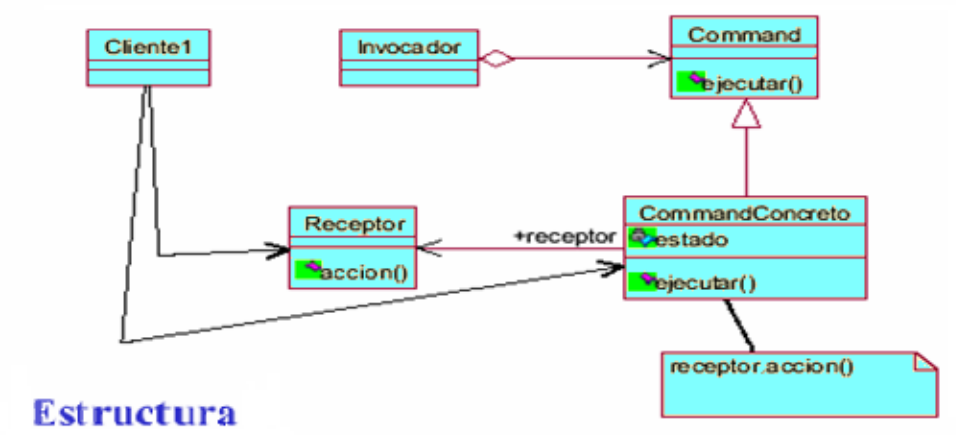


Fig. 75,- Estructura del Patrón Comando.

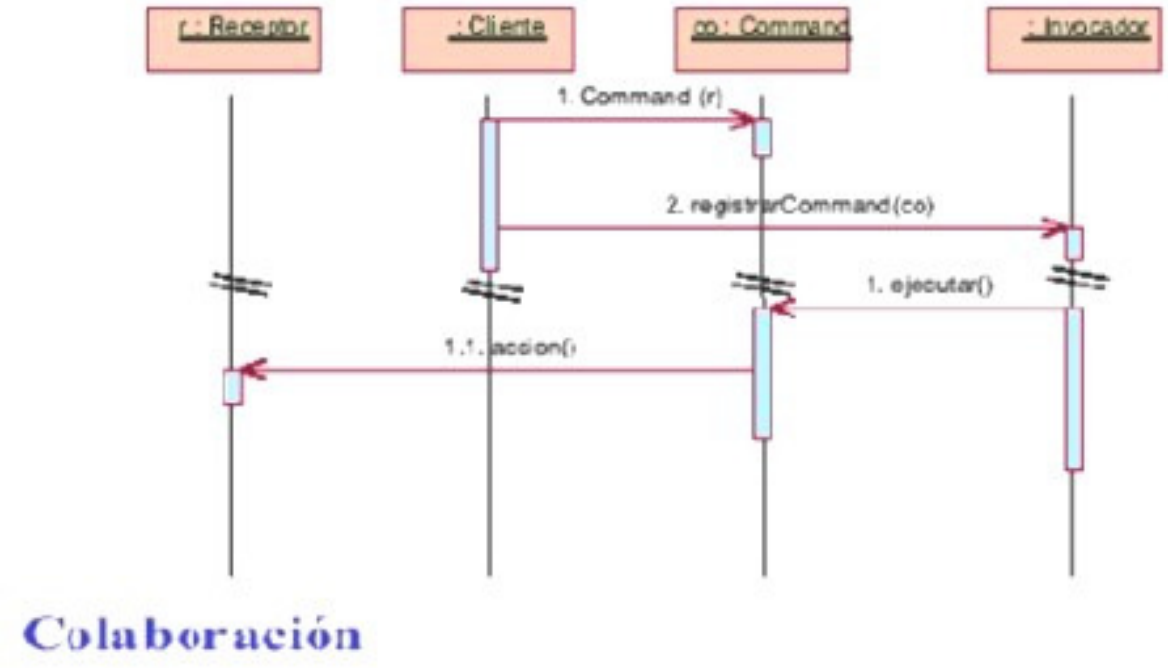


Fig. 76.- Colaboración del Patrón Comando.

Aplicabilidad

- Parametrizar objetos por la acción a realizar (alternativa a funciones Callback: función que es registrada en el sistema para ser llamada más tarde; en C++ se puede usar punteros a funciones)
- Especificar, añadir a una cola y ejecutar mensajes en diferentes instantes: un objeto command tiene un tiempo de vida independiente de la solicitud original.
- Soportar facilidad undo/redo.
- Recuperación de fallos.

Consecuencias

- Desacopla el objeto que invoca la operación del objeto que sabe cómo realizarla.
- Cada subclase CommandConcreto especifica un par receptor/acción, almacenando el receptor como un atributo e implementando el método ejecutar.
- Objetos command pueden ser manipulados como cualquier otro objeto.
- Se pueden crear command compuestos (aplicando el patrón Composite).
- Es fácil añadir nuevos commands.

Implementación

- ¿Cuál debe ser la "inteligencia" de un command?
  - No delegar en nadie.
  - Encontrar dinámicamente el objeto receptor y delegar en él.
- Soportar undo/redo.
  - Atributos para almacenar estado y argumentos de la operación.
  - Lista de objetos commands.
  - En la lista se colocan copias.

El Patrón Intérprete

Propósito

- Dado un lenguaje, definir una representación para su gramática junto con un intérprete que utiliza la representación para interpretar sentencias en dicho lenguaje.

Motivación

- Interpretar una expresión regular.
- Usa una clase para representar cada regla, los símbolos en la parte derecha son atributos.
- Usar si la gramática es simple y la eficiencia no es importante.

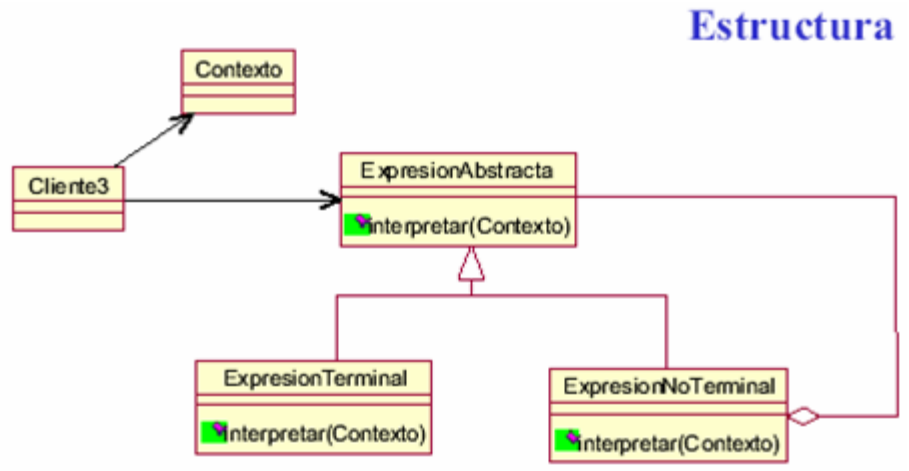


Fig. 77.- Estructura del Patrón Intérprete.

El Patrón Iterador

Propósito

- Proporciona una forma para acceder a los elementos de una estructura de datos sin exponer los detalles de la representación.

Motivación

- Un objeto contenedor (agregado o colección) tal como una lista debe permitir una forma de recorrer sus elementos sin exponer su estructura interna.
- Debería permitir diferentes métodos de recorrido.
- Debería permitir recorridos concurrentes.
- No queremos añadir esa funcionalidad a la interfaz de la colección.
- Una clase Iterator define la interfaz para acceder a una estructura de datos (p. ej. una lista).
- Iteradores Externos vs. Iteradores Internos.
- Iteradores Externos: recorrido controlado por el cliente.
- Iteradores Internos: recorrido controlado por el iterador.

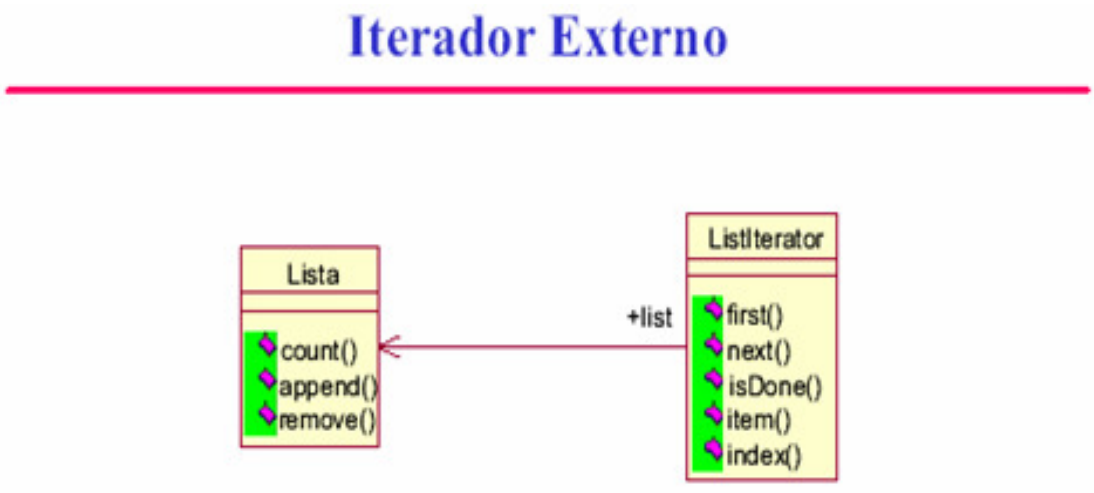


Fig. 78.- Patrón Iterador Externo.

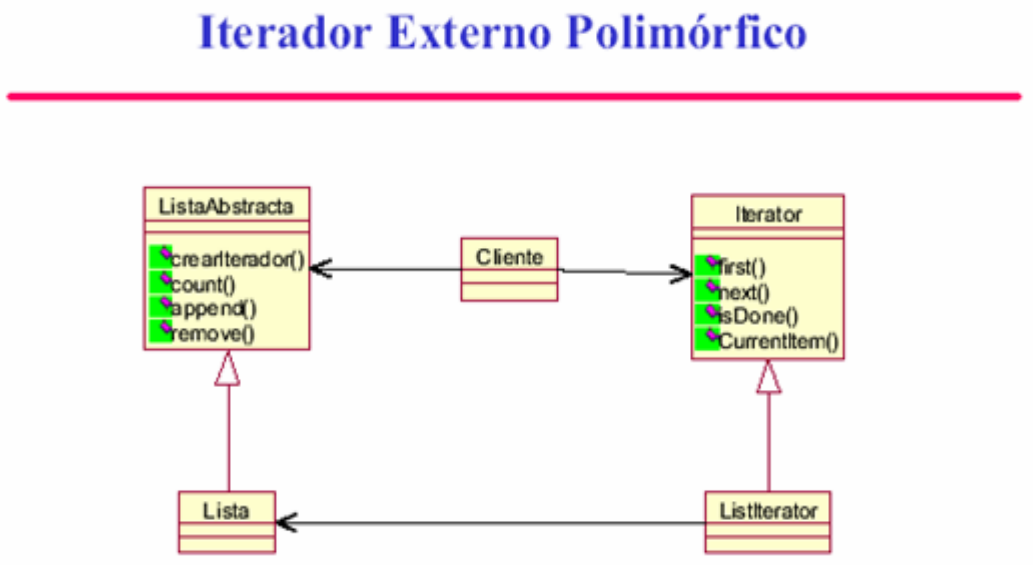


Fig. 79.- Patrón Iterador Externo Polimórfico.

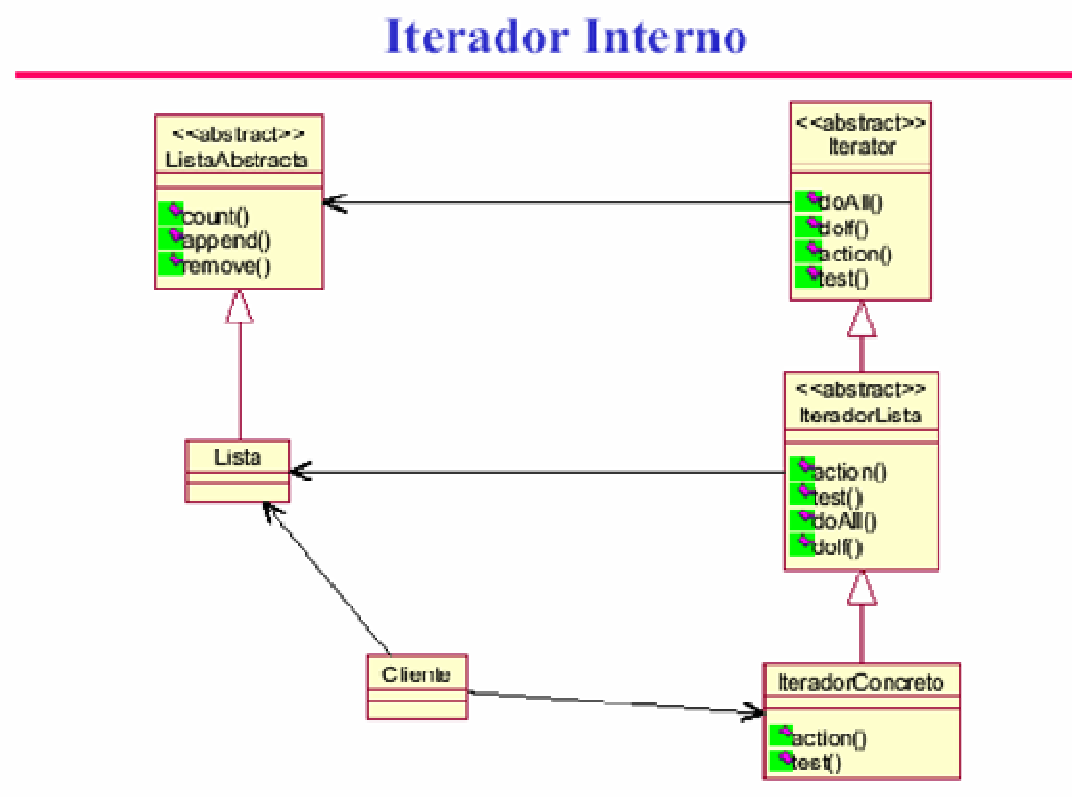


Fig. 80.- Patrón Iterador Interno.

Consecuencias

- Simplifica la interfaz de un contenedor al extraer los métodos de recorrido.
- Permite varios recorridos concurrentes.
- Soporta variantes en las técnicas de recorrido.

Implementación

- ¿Quién controla la iteración?
  - Externos vs. Internos.
- ¿Quién define el algoritmo de recorrido?
  - Agregado: Iterador sólo almacena el estado de la iteración (Cursor). Ejemplo de uso del Patrón Memento.
  - Iterador: es posible reutilizar el mismo algoritmo sobre diferentes colecciones o aplicar diferentes algoritmos sobre una misma colección.
  - ¿Es posible modificar la colección durante la iteración?
  - Colección e Iterador son clases íntimamente relacionadas.
  - Suele ser usado junto con el patrón Composite.

Componentes

En la última mitad de la década de los 90 hubo un creciente interés en el concepto de "componentes". Estas son entidades reutilizables que no requieren conocimiento del software que las usa. Los objetos COM y los Javabeans son ejemplos de tecnologías de componentes. Los componentes pueden ser objetos en el sentido usual de orientación a objetos, excepto que satisfacen guías adicionales dirigidas a hacerlas autocontenidas. Usan otras componentes mediante agregación y por lo general interactúan con otras componentes a través de los eventos.

2.4.6 Arquitectura de Software

La Arquitectura del Software alude a la "estructura global del Software y a las formas en que la estructura proporciona la integridad conceptual de un sistema" [SHA 95a]. En su forma más simple, la arquitectura es la estructura jerárquica de los componentes del programa (módulos), la manera en que los componentes interactúan y la estructura de datos que van a utilizar los componentes. Sin embargo, en un sentido más amplio, los "componentes" se pueden generalizar para representar los elementos principales del sistema y sus interacciones.

La Arquitectura de Software desarrolla un modelo mental de cómo debe funcionar la aplicación con cinco a siete componentes (muy burdo). Por supuesto, el resultado depende de la aplicación pero puede

beneficiarse con arquitecturas que otros han desarrollado en el pasado, igual que el diseño de un puente de suspensión se beneficia con el estudio de puentes construidos antes.

A continuación se presentan las arquitecturas más comúnmente desarrolladas:

Arquitectura de Flujo de Datos

Algunas aplicaciones se ven mejor como datos que fluyen entre las unidades de proceso. Los diagramas de flujo de datos (DFD) ilustran este panorama. Cada unidad de procesamiento del DFD se diseña independiente de los otros. Los datos emanan de fuentes, como el usuario y en algún momento fluyen de regreso al usuario o a sumideros como bases de datos.

Como ejemplo mostraremos una aplicación bancaria como se muestra en la siguiente figura.

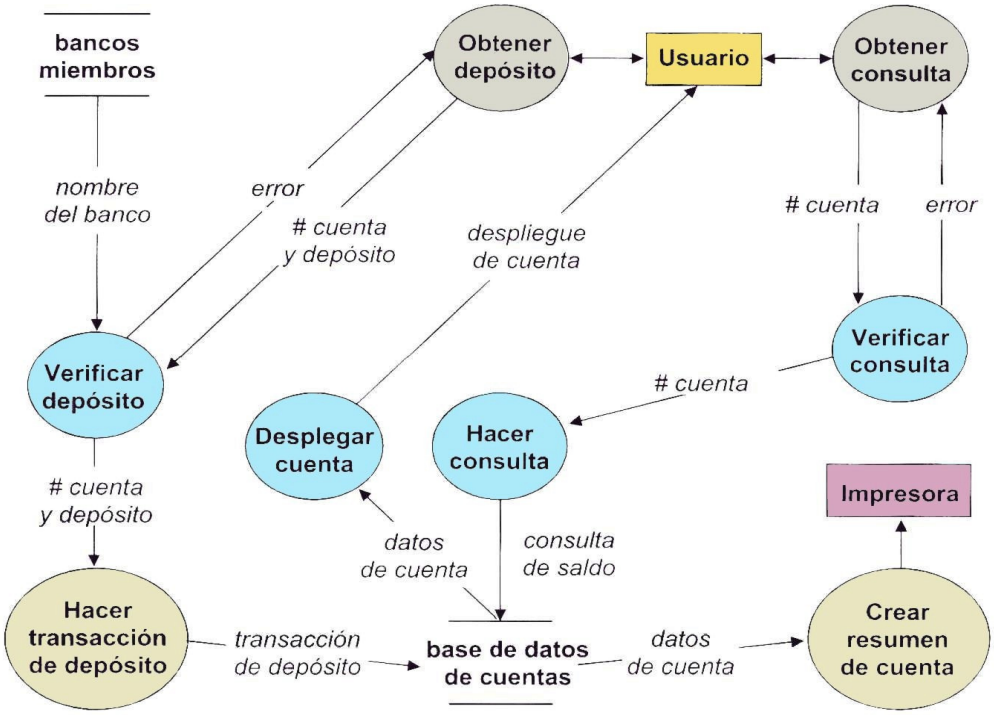


Fig. 81.- Diagrama parcial de Flujo de Datos para una aplicación de cajero automático.

Los datos fluyen del usuario a un proceso obtener depósito, que envía el número de cuenta y la cantidad de depósito a un proceso que verifica la consistencia de los datos. Si los datos son consistentes, se pueden

enviar a un proceso que crea una transacción y los pasos que siguen. Los DFD pueden estar anidados. Por ejemplo, "crear consulta de transacción" puede descomponerse en un Diagrama de Flujo de Datos más detallado. Un tipo de Arquitectura de Flujo de Datos, como la de la figura siguiente, recibe el nombre de "tubería y filtro". Estas son las Arquitecturas de Flujo de Datos donde los elementos de procesamiento (filtros) aceptan flujos como entrada (secuencias de un elemento de datos uniformes) en cualquier momento y producen flujos de salida. Cada filtro debe diseñarse para que sea independiente de los otros filtros. La característica de la arquitectura de la siguiente figura se implementa, por ejemplo, con tuberías (pipes) UNIX.

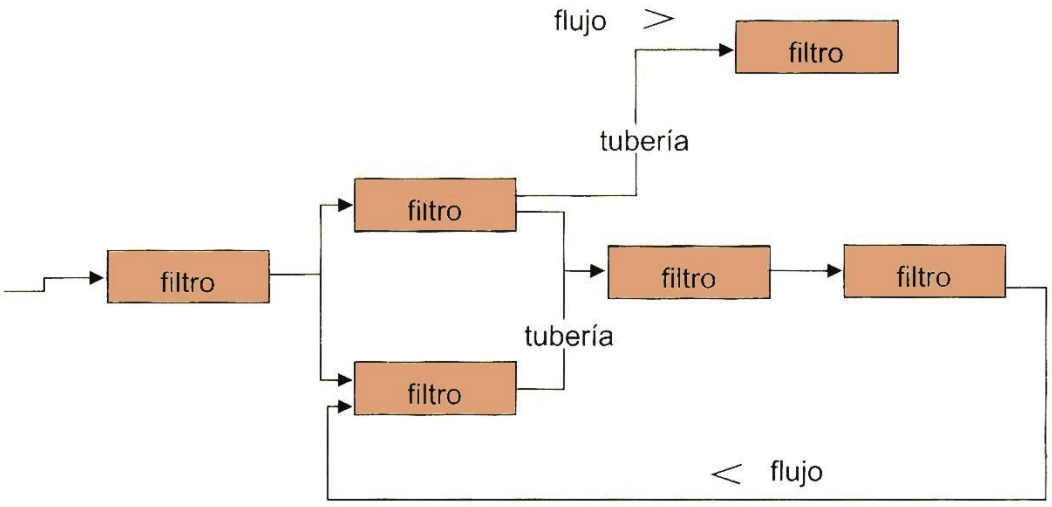


Fig. 82.- Arquitectura de Tubería y Filtros.

Las Arquitecturas de Tubería y Filtro tienen la ventaja de la modularidad. Un ejemplo se muestra en la figura anterior. Aquí la aplicación mantiene las cuentas conforme llegan las transacciones en tiempos aleatorios desde las líneas de comunicación. La arquitectura incluye un paso para registrar las transacciones en caso de que falle el sistema. La función retirar tendrá un retiro por entrada, como VictorVelazquezCuentaNum12345Cantidad$3500.00, o sólo VictorVelazquez12345$3500.00, es decir, una cadena de caracteres y la dirección de entrada del banco como BancoNum9876. Los elementos de procesamiento, las elipses, esperan hasta que llegan todos los datos requeridos para "activarse" y realizar sus operaciones.

En general, no existe una manera uniforme de "mapear" los diagramas de flujo de datos (DFD) en modelos de clases; sin embargo, en ocasiones, las unidades funcionales del DFD pueden macerarse directamente sobre los métodos de clases, como se ve en la siguiente figura.

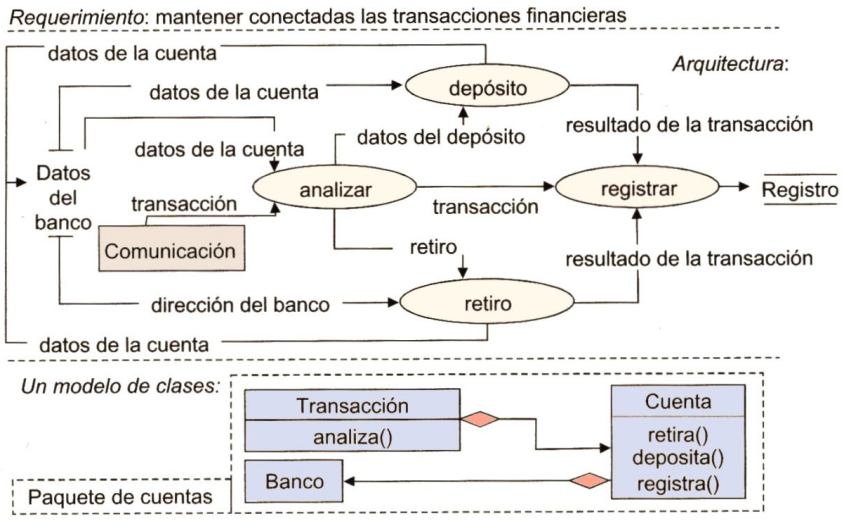


Fig. 83.- Ejemplo de las opciones en la Arquitectura de Flujo de Datos en Tubería y Filtros.

El creciente uso de la computación distribuida ha acelerado la aplicación de computación orientada a flujos porque la llamada remota a las funciones con frecuencia se implementa mediante la conversión de la llamada en un flujo de caracteres. Esto se hace, por ejemplo, con la solicitud del método remoto (RMI Remote Method Invocation) de Java. El RMI usa seriación que convierte objetos en cadenas de caracteres. Además, la entrada/salida (I/O) muchas veces se implementa con flujos o cadenas y realizar la I/O en un lenguaje como Java no suele ser más que un proceso de filtrado.

En el caso especial en que los filtros son sólo lotes dados de datos, el resultado es un lote secuencial de flujo de datos. Como ejemplo considere una aplicación bancaria que calcula la cantidad de dinero disponible para préstamos hipotecarios (avalados por propiedades) y la cantidad de dinero disponible para préstamos no asegurados. En la figura siguiente se sugiere un Diagrama de Flujo de Datos. Este DFD es secuencial por lotes porque las funciones se ejecutan usando casi todos los datos de entrada juntos. Por ejemplo, se recolectan los fondos disponibles para préstamos hipotecarios usando todos los datos de la cuenta. Esto es opuesto al ejemplo de transacciones de la figura anterior, donde hay muchas transacciones, casi continuas y cada una usa los datos seleccionados de sus fuentes.

La figura siguiente también muestra el mapeo en un modelo de clase donde las funciones del flujo de datos se analizan como métodos de la clase Banco. Los "lotes" de procesamiento se ejecutan corriendo los métodos relevantes de esta clase.

Durante décadas, el flujo de datos ha sido la forma común más importante para expresar las arquitecturas y parece que seguirá siendo útil por más tiempo. Los ingenieros piensan de manera natural en el flujo de una "estación" de procesamiento a otra y en el procesamiento que se lleva a cabo en cada estación. Las desventajas de los Diagramas de Flujo de Datos incluyen que el mapeo al código no es claro, sea orientado a objetos o no.

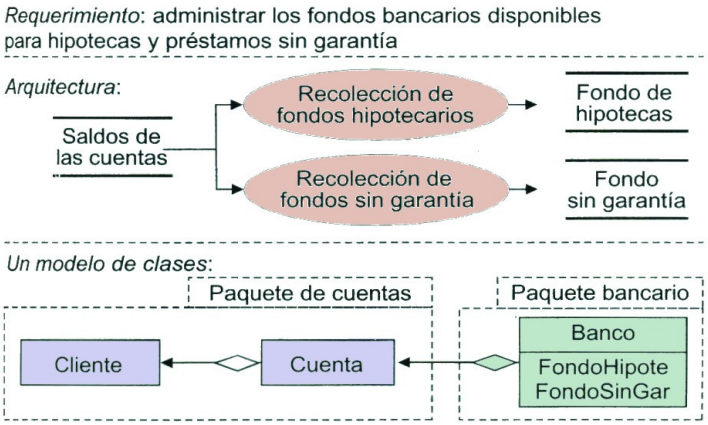


Fig. 84.- Ejemplo de una Arquitectura de Flujo de Datos secuencial por lote.

Componentes Independientes

La arquitectura de "Componentes Independientes" consiste en componentes que operan en paralelo (al menos en principio) y se comunican entre si de vez en cuando. Tal vez el ejemplo más obvio se pueda encontrar en Internet, donde miles de servidores y millones de navegadores en paralelo todo el día y de manera periódica se comunican entre sí.

Arquitectura Cliente-Servidor

En una relación Cliente-Servidor, la componente del Servidor satisface las necesidades del Cliente cuando lo solicita. Las relaciones Cliente-Servidor tienen la ventaja de un bajo acoplamiento entre las dos componentes que participan. Estas relaciones se aplican a la Ingeniería de Software en general, cuando más de una persona realiza la implementación. Es natural enviar un paquete de clases a cada desarrollador o grupo de desarrolladores que requieren los servicios de las clases que son responsabilidad de otros. En otras palabras, los paquetes de los desarrolladores con frecuencia se relacionan como Cliente y Servidor. El problema es que estos servicios suelen tener diferentes estados de terminación conforme el proyecto avanza.

Un componente actúa de manera más efectiva como servidor cuando su interfaz es "estrecha", es decir, que la interfaz (en esencia una colección de funciones) no contiene partes innecesarias, se colecta en un solo lugar y está definida con claridad.

Las arquitecturas Cliente-Servidor fueron una característica firme en las décadas de 1980 y 1990. Muchas de ellas sustituyeron las arquitecturas de Mainframe/Terminal. Más tarde, las arquitecturas Cliente-Servidor se volvieron más elaboradas y variadas. Algunas se diseñan ahora como arquitecturas de tres grados (tiers) en lugar de las originales de dos (Cliente-Servidor). El tercer grado está entre el Cliente y el Servidor y proporciona un nivel útil de no dirección. Una asignación común es diseñar la GUI para el cliente, el sistema de administración de la base de datos o administración de procedimientos para el grado intermedio y diversos programas de aplicación y/o bases de datos para el tercer grado. El grado intermedio puede ser un transportador (bus) de datos común como el Common Object Request Broker (CORBA). De

otra manera, el grado intermedio puede operar vía estándar binario como COM (Common Object Model de Microsoft). Por último, el Internet pude considerarse una especie de arquitectura Cliente-Servidor donde "un servidor/decenas de clientes" se sustituye por un "un servidor/millones de clientes".

Arquitectura de Procesadores de Comunicación Paralelos

Otro tipo de arquitectura de "Componentes Independientes" se llama Proceso de Comunicación Paralela. Esta arquitectura se caracteriza por varios procesos que se ejecutan al mismo tiempo (en términos de Java, hilos o threads). La siguiente figura muestra una arquitectura para una aplicación bancaria diseñada para manejar transacciones múltiples simultáneas en cajeros automáticos.

Cuando el cliente "n" usa un cajero automático, se crea el objeto cliente "n" (1 en la figura siguiente). Este objeto crea la sesión "m", un hilo o proceso paralelo (2), denotado por media flecha. Después la sesión "m" recupera un objeto de Cuentas como cliente "n" cheques (3). Luego el cliente realiza una transacción de depósito en un objeto de cheques (4). En paralelo se crean objetos de Clientes como cliente "n+1" y operan en otros hilos, como sesión "k".

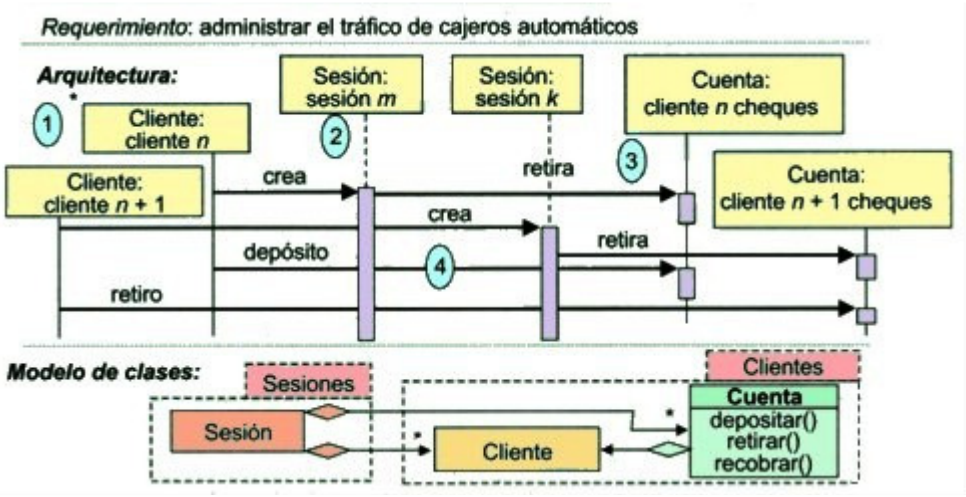


Fig. 85.- Ejemplo de Arquitectura de Procesos de Comunicación Paralelos.

Cuando una aplicación requiere procesamiento paralelo, una opción común es la Arquitectura de Procesadores de Comunicación Paralelos. Esta arquitectura se puede usar para esquemas que coordinan áreas independientes de manera conceptual. En su libro clásico [Di], Dijkstra mostró que concebir un proceso como la combinación de partes paralelas, con frecuencia puede simplificar los diseños. Un ejemplo es una simulación de clientes en un banco. Por tradición, muchas de estas simulaciones se diseñaban sin paralelismo, almacenando y manejando los eventos involucrados. Sin embargo, estos diseños pueden simplificarse si el movimiento de cada cliente es un proceso separado (como un objeto thread en Java).

Este diseño de Procesos de Comunicación Paralelos tiene la ventaja de que se ajusta bien a las actividades que simula.

Arquitecturas de Sistemas por Eventos

Esta arquitectura ve la aplicación como un conjunto de componentes, cada una de las cuales espera hasta que ocurre un evento que la afecta. Muchas aplicaciones contemporáneas son sistemas por eventos. Un procesador de palabras, por ejemplo, espera a que el usuario oprima un icono o una opción del menú. Después reacciona en consecuencia para guardad un archivo, aumentar el tamaño de la letra, etcétera. Los sistemas por eventos con frecuencia funcionan como los sistemas de transición de estados.

Arquitecturas de Almacenamiento

Una arquitectura construida en esencia alrededor de datos se llama Arquitectura de Almacenamiento. Los más comunes son sistemas diseñados para realizar transacciones en una base de datos. Por ejemplo, una compañía de electricidad mantiene una base de datos de los clientes que incluye detalles de los mismos, como uso mensual de energía, saldo, historia de pagos, reparaciones, etcétera. Un diseño típico para este tipo de Arquitectura de Almacenamiento se presenta en al figura siguiente.

Fig. 86.- Arquitectura Típica de Almacenamiento.

Otros ejemplos de aplicaciones con Arquitecturas de Almacenamiento incluyen entornos de desarrollo interactivos (IDE, Interactive Development Environments). Los IDE se aplican a varios procesos como edición y compilación de una base de datos de archivos fuente y archivos objeto.

Muchas aplicaciones, como los IDE, aunque no son aplicaciones puras de bases de datos, las involucran. El lenguaje SQL (Structured Query Language) es una forma común para expresar búsquedas.

Las Arquitecturas de Pizarrón, desarrolladas para las aplicaciones de Inteligencia Artificial, son de almacenamiento que se comportan de acuerdo con las reglas publicadas.

El último tipo de Arquitecturas de Almacenamiento que se mencionarán son las Arquitecturas de Hipertexto. El uso más común del hipertexto es en Internet.

Con frecuencia la industria usa la palabra "almacenamiento" para denotar una aplicación que proporciona

una visión unificada de una colección de bases de datos. Los almacenamientos no cambian la estructura de estas bases de datos, pero permitirán un acceso uniforme a ellas. Este es un caso especial de las Arquitecturas de Almacenamiento definidas por Garlan y Shaw [Ga].

Las Arquitecturas de Almacenamiento ocupan una gran porción de las aplicaciones, ya que muchas tienen bases de datos como parte central. Cuando el procesamiento es despreciable comparado con el formato de los datos en la base de datos, las Arquitecturas de Almacenamiento son apropiadas. Por otro lado, la presencia de una base de datos grande, en ocasiones puede ocultar el hecho de que una gran cantidad de procesamiento puede determinar la arquitectura. Es fácil que la programación adecuada (por ejemplo, los "procedimientos de almacenamiento") se convierta en una aplicación desordenada que debió concebirse de manera diferente en el modelo de almacenamiento.

Arquitectura de Capas

Una Capa de una Arquitectura es una colección coherente de artefactos de software, casi siempre un paquete de clases. En su forma más común, una Capa usa cuando mucho otra Capa y es usada por otra Capa cuando mucho. Construir aplicaciones Capa por Capa puede simplificar mucho el proceso. Algunas Capas, como marcos de trabajo, pueden servir para varias aplicaciones.

La figura siguiente contiene un ejemplo de Arquitectura de Capas para una aplicación de impresión en el banco Ajax. En esta arquitectura existen cuatro Capas y en la figura siguiente se muestra la dependencia en la dirección opuesta respecto a la figura anterior. La capa de aplicación, Impresión Banco Ajax, tiene que ver con impresión y formato. Estas últimas están construidas sobre una capa de proveedores-vendedores, no mostrada, que contiene herramientas necesarias como ordenar y buscar. Por lo común, una capa se realiza como un paquete de clases. Por ejemplo, la biblioteca común de Ajax comprende clases que se usan en todas las aplicaciones de Ajax y maneja aspectos como el logo del banco y sus reglas. La relación de "uso" puede ser la herencia, el agregado o la referencia de objetos. En el ejemplo sólo se aplica el agregado entre las capas.

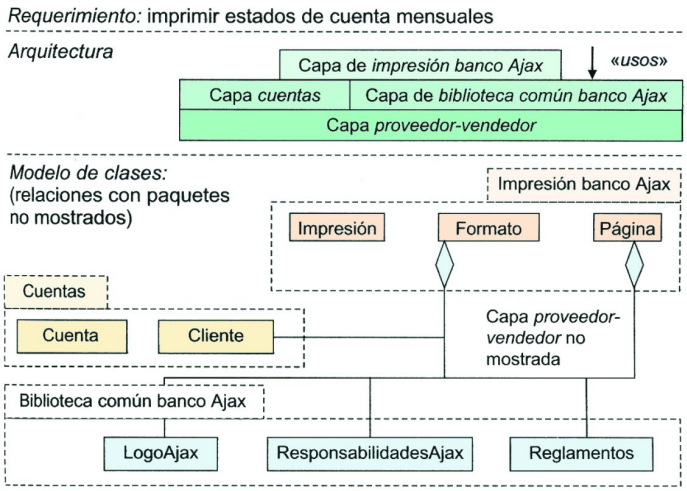


Fig. 87.- Ejemplo de Arquitectura de Capas que usa agregados.

Una forma común de Capas es la Arquitectura Cliente-Servidor. En esta forma, la capa del cliente se apoya en la capa del servidor para obtener los servicios que requiere. En general, el cliente reside en la computadora del usuario y el servidor en una computadora más grande centralizada. Con frecuencia el servidor se refiere a una base de datos.

Las Arquitecturas clásicas Cliente-Servidor tienen un alto grado de dependencia en el sentido de que los clientes y servidores deben tener código duro con conocimiento uno del otro. Este problema se puede resolver empleando la Arquitectura de Tres Grados. En ella se introduce una capa intermedia que sirve de aislante entre el cliente y el servidor. Esta capa intermedia se puede usar para aumentar la flexibilidad de la arquitectura en una variedad de maneras. Por ejemplo, si varios servidores pueden servir una petición del cliente, la capa intermedia se puede diseñar para encontrar un servidor dinámico apropiado. La capa intermedia con frecuencia se implementa mediante un tipo de software conocido como middleware. CORBA, desarrollado por el consorcio Object Management Group, es un middleware estándar.

Las arquitecturas por capas tienen grandes beneficios por el reuso. La biblioteca de clases java es en efecto un sistema de capas muy exitoso (por ejemplo, el paquete (capa) Applet apoyado en el paquete awt, que a su vez se apoya en el paquete lang, etcétera).

Uso de Arquitecturas Múltiples dentro de una aplicación

Las aplicaciones suelen usar varias arquitecturas. Tal vez tenga sentido, por ejemplo, un paquete como una base de datos, otro paquete como procesos de comunicación paralelos e incluso otro como un sistema activado por eventos.

Notación de la Arquitectura, Estándares y Herramientas.

Notación

El lenguaje de modelado unificado (UML) es una notación gráfica aceptada para describir los diseños orientados a objetos. También están los Diagramas de Flujo de Datos que se aplican de manera independiente de la orientación a objetos. Lo mismo es cierto para los Diagramas de Entidad-Relación, que describen las relaciones entre los datos y los almacenes de datos.

Herramientas

Se usan varias herramientas de Ingeniería de Software asistida por computadora (CASE) para facilitar el proceso de Ingeniería de Software. Algunas herramientas representan clases y sus relaciones, como Rational Rose de IBM y Together de Borland. Estas herramientas facilitan los bosquejos de los modelos de objetos, ligándolos con el código fuente y los Diagramas de Secuencia correspondientes.

Al seleccionar una herramienta de modelado, se obtiene una lista de requerimientos para las herramientas usando procedimientos similares al proceso de análisis de requerimientos para el desarrollo de las aplicaciones de software. El siguiente es un ejemplo de algunos requerimientos para herramientas de modelado.

- [Esencial] Facilitar el dibujo de Modelos de Objetos y Diagramas de Secuencia.

- Crear clases con rapidez.
- Editar las clases de manera sencilla.
- Acercamiento a partes del modelo.

- [Deseable] Posible saltar directo de un modelo de objetos al código fuente.
- [Esencial] No debe costar más de $X por usuario.
- [Opcional] Ingeniería Inversa (crear modelos de objetos a partir del código fuente).

2.4.7 Herramientas a nivel Arquitectura contra herramientas para Diseño Detallado e Implementación

Los paquetes de herramientas intentan abarcar la Arquitectura, el Diseño Detallado y la Implementación. Varios proveedores están desarrollando la capacidad de crear hipervínculos entre el código fuente con la documentación y viceversa. Las herramientas orientadas a la implementación como Javadoc pueden ser útiles para complementar el proceso de diseño. Las herramientas Cliente-Servidor como Powerbuilder se pueden aplicar a arquitecturas específicas, aun cuando también especifican implementaciones. Javadoc es útil para paquetes de navegación porque proporciona una lista alfabética de las clases y la jerarquía precursora de cada una.

Los entornos de desarrollo interactivo (IDE) se entregan con compiladores y se usan como herramientas de modelado parcial. Los IDE OO en general muestran la herencia de manera gráfica y con frecuencia, son atractivos para los desarrolladores debido a su cercanía con el proceso de compilación y depuración. Desde 1999, los IDE no cuentan con el desarrollo suficiente para facilitar un trabajo de arquitectura y diseño verdadero.

Las herramientas de ensamble de componentes crean aplicaciones arrastrando y soltando iconos que representan elementos de procesamiento. Los entornos de Java Beans son de este tipo. En estos entornos, a partir de estas bibliotecas, se pueden obtener beans (objetos de Java cuyas clases se ajustan al estándar de Java Beans) a la medida y relacionados entre sí mediante eventos. El estándar de Java Beans se creó con el propósito expreso de facilitar los ensambles sencillos por medio de herramientas gráficas.

Una desventaja de usar las herramientas de modelado es la dependencia del proyecto en una tercera parte o proveedor. Además de las complicaciones de la aplicación y el proyecto mismo, los Ingenieros deben preocuparse por la factibilidad de las herramientas del proveedor. Suponga que el proveedor quiebra o que las actualizaciones de la herramienta son demasiado costosas, ¿Cómo afectará esto al proyecto?

A pesar de estos aspectos, el uso de herramientas de diseño y desarrollo ha aumentado de manera firme. Las herramientas adecuadas nivelan la productividad y lo más probable es que a la larga los factores económicos favorezcan su uso.

2.4.8 Estándares IEEE/ANSI para expresar diseños

El documento de diseño de software (DDS) estándar de IEEE 1016-1987 (confirmado en 1993) proporciona guías para la documentación del diseño. La tabla de contenido se muestra a continuación:

Arquitectura

1. Introducción
    - 1.1 Propósito
    - 1.2 Alcance
    - 1.3 Definiciones, acrónimos y abreviaturas
2. Referencias
3. Descripción de descomposición
    - 3.1 Descomposición en módulos
        - 3.1.1 Módulo 1, descripción
        - 3.1.2 Módulo 2, descripción
    - 3.2 Descomposición de procesos concurrentes
        - 3.2.1 Proceso1, descripción
        - 3.2.2 Proceso 2, descripción
    - 3.3 Descomposición de datos
        - 3.3.1 Entrada de datos 1, descripción
        - 3.3.2 Entrada de datos 2, descripción
4. Descripción de dependencia
    - 4.1 Dependencias entre módulos
    - 4.2 Dependencias entre procesos
    - 4.3 Dependencias entre datos
5. Descripción de interfaz
    - 5.1 Interfaz de módulo
        - 5.1.1 Módulo 1, descripción
        - 5.1.2 Módulo 2, descripción
    - 5.2 Interfaz de proceso
        - 5.2.1 Proceso 1, descripción
        - 5.2.2 Proceso 2, descripción

6 Diseño detallado
- 6.1 Diseño detallado de módulos
    - 6.1.1 Módulo 1, detallado
    - 6.1.2 Módulo 2, detallado

6.2 Diseño detallado de datos
- 6.2.1 Entidad de datos 1, detallada
- 6.2.2 Entidad de datos 2, detallada

Tabla 11.- Tabla de contenido del documento de diseño de software IEEE 1016 -1987 (confirmado en 1993).

Las guías de IEEE que acompañan el estándar (1016.1-1993) explican cómo puede escribirse el DDS para varios estilos de arquitectura.

Aseguramiento de la Calidad (QA) de la Arquitectura elegida

El personal de QA debe participar en la revisión de la arquitectura. Además QA desarrolla los planes de pruebas para cada componente de la arquitectura tan pronto como se define.

Las arquitecturas se inspeccionan contra los requerimientos. Recuerde que el beneficio por detección de defectos es más alto en las etapas iniciales del proyecto como la selección de Arquitectura. Las métricas mencionadas proporcionan una base para la inspección.

2.5 Implementación

Implementación de Unidades

La implementación se refiere a programación. Unidad se refiere a la parte más pequeña de la implementación a la que se dará mantenimiento por separado y puede ser un método individual o una clase.

Metas de la implementación

El propósito de la implementación es satisfacer los requerimientos de la manera que específica el diseño detallado. Aunque el diseño detallado debe ser suficiente como documento contra el que se programa, es común que el programador examine todos los documentos anteriores al mismo tiempo (arquitectura, requerimientos D y requerimientos C), para ayudar a disminuir las inconsistencias entre documentos.

2.5.1 Mapa conceptual típico del Proceso de Implementación Unidades

La siguiente figura muestra un proceso típico mediante el cuál se produce el código.

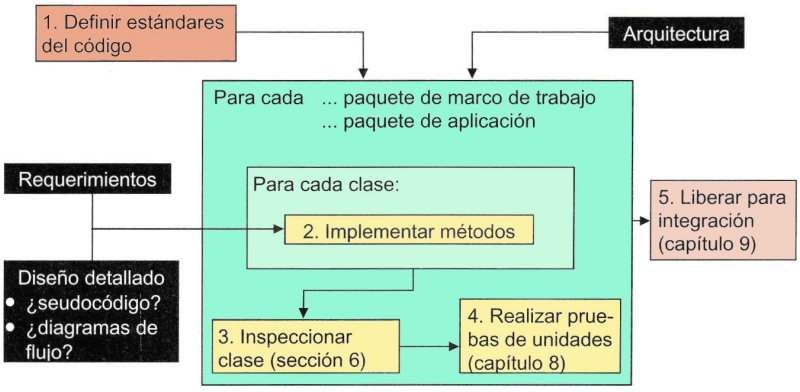


Fig. 88.- Mapa conceptual para la Implementación.

- En la figura, los estándares de codificación se identifican de manera que el código fuente tenga una apariencia común.
- La arquitectura determina cuáles son el marco de trabajo y los paquetes de aplicación. Cada clase de cada paquete se implementa codificando los métodos determinados por los requerimientos y el diseño detallado. Los paquetes del marco de trabajo se requieren antes de poder construir los paquetes de aplicación.
- Cada clase se inspecciona tan pronto está lista.
- Cada clase se prueba.
- Los paquetes o clases se liberan para la integración en la aplicación que surge.

2.5.2 Una manera de preparar la Implementación

La siguiente lista enumera una secuencia de acciones que preparan al programador para la implementación.

- Confirmar los diseños detallados que deben implementarse.
  - Sólo código a partir de un diseño escrito (parte del DDS).
- Preparar la medición del tiempo dedicado, clasificado por:
  - Diseño detallado residual; revisión del diseño detallado; codificación; revisión del código; compilación y reparación de defectos de sintaxis; pruebas de unidades y reparación de defectos encontrados en las pruebas.
- Preparar para registrar los defectos usando una forma.
  - **Severidad:** Importante (requerimiento no satisfecho), trivial o ninguno.
  - **Tipo:** Error, nombre, entorno, sistema, datos, otro.
- Comprender los estándares requeridos.
  - Para codificación.
  - Para la documentación personal que debe guardar.
- Estimar el tamaño y el tiempo con base en sus datos anteriores.
- Planear el trabajo en segmentos de ± 100 líneas de código.

Algunos de los métodos más complejos se proporcionarán en seudo código o diagramas de flujo. El seudo código del documento de diseño detallado se puede convertir en código con comentarios. Se desarrolla un plan de pruebas para cada unidad. Se inspecciona cada una de las unidades del diagrama. Una vez que se implementa una clase, se puede usar una herramienta de Ingeniería Inversa (como Javadoc) para generar los aspectos del diseño detallado, según se ilustra en la figura siguiente.

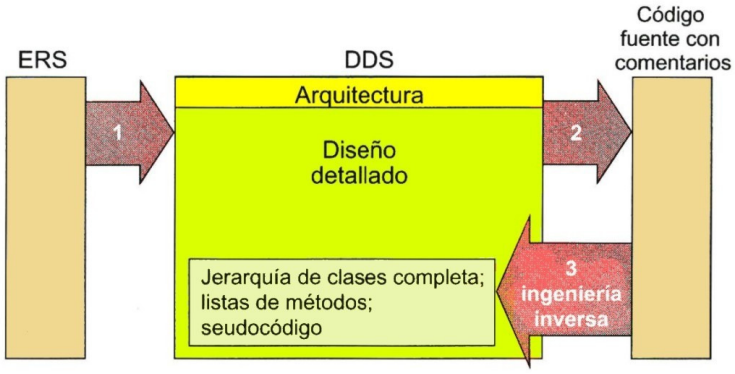


Fig. 89.- Uso de Ingeniería Inversa para diseño muy detallado.

La Ingeniería Inversa sería aconsejable después de la implementación inicial porque el código fuente tiende a ser más actualizado que el nivel más bajo del diseño detallado. También puede ser útil para el código heredado cuando el diseño está mal documentado. En términos generales, la Ingeniería Inversa debe ser un recurso sólo si en verdad es necesaria. En algunas ocasiones se utiliza para cubrir la falta de diseño. En otras palabras, los Ingenieros se enfrascan en el código antes de tiempo y después se justifican con un "diseño" obtenido con Ingeniería Inversa a partir del código. Lo cual no es lo adecuado.

2.5.3 Programación y estilo

La imagen popular de la programación como el acto de someter material tecleado a un compilador es sólo una pequeña parte de la imagen. La meta real de la Ingeniería de Software es crear el código correcto (es decir, justo el apropiado para los requerimientos), pero los compiladores sólo pueden verificar la sintaxis y generar un código objeto. Lo correcto es responsabilidad humana. Por lo tanto, es esencial que el profesional esté convencido por completo de la exactitud del código antes de someterlo a compilación. Aunque en principio es posible compilar primero y verificar que sea exacto o correcto después, esto es tan poco efectivo como corregir la sintaxis de una carta antes de estar seguro que expresa el pensamiento adecuado. Más aún, los programadores tienden a omitir la verificación exhaustiva del código de programa una vez que compila sin errores (de sintaxis).

En la siguiente tabla se indican los pasos que pueden seguir los programadores.

Una manera de Implementar el Código

- Planear la estructura y el diseño residual para el código. (Complete los diseños detallados que faltan, si los hay).
  - Note las precondiciones y poscondiciones.
  - Registre el tiempo dedicado.

- Auto inspeccione su diseño y/o estructura.
  - Observe tiempo dedicado, tipos de defectos, fuente (etapa) y severidad.
- Teclee su código.
  - No compile todavía.
  - Intente los métodos enumerados a continuación.
  - Aplique los estándares requeridos.
  - Codifique de manera que la verificación sea sencilla.
    - Use métodos formales si es apropiado.
- Auto inspeccione el código: no compile todavía.
  - Asegure que su código realiza el trabajo requerido.
    - El compilador nunca hará esto por usted, ¡sólo verifica la sintaxis!
  - Registre el tiempo dedicado, defectos encontrados, tipo, fuente y severidad.
- Compile su código.
  - Repare los defectos de sintaxis.
  - Registre tiempo dedicado, tipo de defectos, severidad y líneas de código.
- Pruebe su código.
  - Aplique los métodos de prueba de unidades.

2.5.4 Principios Generales de una Implementación Acertada

La siguiente tabla recomienda dos principios generales para programar:

- Intente el reuso primero.
- Cumpla los propósitos.

Si su código debe usarse sólo de manera particular, escríbalo de modo que no se pueda usar de ninguna otra forma.

Se ha resaltado la necesidad de diseñar las propias aplicaciones de manera que permitan el reuso de las componentes que se construyen. Con el mismo ánimo, es muy recomendable considerar el reuso de código existente confiable, antes de escribir el propio. Por ejemplo, considere usar componentes GUI de Java Swing o un Java Bean antes de desarrollar su propia interfaz gráfica. Una búsqueda rápida en Internet de las componentes de reuso casi siempre es una inversión que vale la pena.

Si tiene en mente un propósito de cómo otras partes de la aplicación deben usar el código que está desarrollando, entonces trate de cumplir con esa intención. Con frecuencia esto es evidente en las interfaces de usuario, donde no se permite al usuario introducir datos ilegales. Sin embargo, se hace hincapié en ello para el procesamiento interno.

El principio de "cumplir con las intenciones" es análogo a construir curvas e islas en las carreteras para dirigir el tránsito justo por las trayectorias que deseaban los Ingenieros de Tránsito y no por otras. Este cumplimiento de intenciones hace a las carreteras más seguras y se extiende a cada rama de la Ingeniería. A continuación se presentan ejemplos del principio de "cumplir las intenciones" en Ingeniería de Software.

- Haga calificadores como "abstract" para cumplir las intenciones correspondientes. Si esto ocasiona errores de compilación, significa que todavía no se comprende bien su programa y no hay daños. Lo que se quiere evitar en especial son los errores durante la ejecución.
- Haga que las constantes, variables y clases sean lo más locales posible. Por ejemplo, defina contadores de ciclos dentro de los ciclos, no les dé un alcance mayor.
- Use le patrón de diseño Solitario si debe haber una sola instancia de una clase.
- En términos generales, haga que los miembros sean inaccesibles si no hay una intención específica de tener acceso directo a ellos.
    - Los atributos deben ser privados. Se llega a ellos a través de funciones de acceso más públicas, si se requiere.
    - Los métodos deben ser privados si son sólo para uso por métodos de la misma clase.
- Incluya ejemplos en la documentación. Los programadores suelen dudar al hacer esto, pero los ejemplos pueden ayudar mucho al lector.
- Enumere los métodos en orden alfabético en lugar de intentar encontrar una orden de llamada entre ellos. Algunos programadores prefieren agrupar en métodos privados, protegidos y públicos y después hacer subgrupos de métodos estáticos y no estáticos.

2.5.5 Manejo de Errores

Los desarrolladores se enfrentan de manera constante con qué hacer con datos potencialmente ilegales. Un ejemplo de datos ilegales es un número de cuenta que no corresponde a una cuenta de banco real. Aunque se intente hacer la implementación lo más sencilla posible, el mundo real no es sencillo. Una gran parte de la programación se dirige al manejo de errores. Un enfoque disciplinado es esencial: seleccione un enfoque, establézcalo y asegúrese de que todo el equipo los entiende y respeta. Una manera de manejar errores se explica a continuación:

- Siga el proceso de desarrollo acordado; inspeccione.
- Considere introducir clases para encapsular los valores legales de los parámetros.
    - Constructor privado, funciones integradas para crear instancias.
    - Detecta muchos errores durante la compilación.
- Cuando los requerimientos especifican el manejo de errores, debe implementarse según se requiere.
    - Use excepciones si se pasa la responsabilidad del manejo de errores.
- Para las aplicaciones que nunca deben detenerse, prevea todos los defectos de implementación posibles (como uso de predeterminados o default).

  - Sólo si la operación desconocida es mejor que nada (poco usual).
- De otra manera, siga una política consistente para verificar los parámetros.
  - Debe apoyarse más que nada en buen diseño y proceso de desarrollo.

Nuestra meta real es la prevención de errores y no su corrección. Utilizar un proceso bien definido, inspeccionar las etapas, etcétera, es la primera línea de defensa esencial. Ciertos patrones de diseño también pueden ayudar a prevenir errores. Por ejemplo, si un método evaluar() acepta sólo "auto", "camioneta" o "camión" como parámetros, entonces quizá valga la pena no usar cadena(string) como parámetro, porque produce la posibilidad de parámetros ilegales. Sería mejor definir una clase como VehiculoEspecializado con un constructor privado y funciones integradas.

Cuando los valores posibles de los parámetros están restringidos, pero son infinitos, todavía puede ser valiosa una clase separada. Por ejemplo, la edad de una persona es un entero digamos entre 0 y 105, en este caso un método obtenerAñoNacimiento (int edadP) tal vez deba manejar errores. De hecho, el mismo procesamiento de error tendría que repetirse para todos los métodos que piden edad como parámetro.

Una segunda línea de defensa para manejar datos potencialmente ilegales es interactuar con la fuente de datos hasta que la entrada cambie a una legal antes de continuar con el proceso. Esto es posible para una gran parte de la programación de interfaces de usuario, donde con frecuencia se puede asegurar que sólo se permiten entradas legales. Si las únicas cadenas permitidas que pueden introducirse en un campo de texto son "auto", "camioneta" y "camión" es sencillo evitar que el usuario continúe hasta que dé un dato legal. Un cuadro con una lista es una forma común de hacerlo. Sin embargo, aún aquí pueden introducirse algunos errores sutiles. Por ejemplo, el usuario puede poner su fecha de nacimiento como 1/1/80 y edad (en el 2006) de 36. Es posible verificar inconsistencias, pero el diseñador tendrá la responsabilidad de pensar en todas las verificaciones de límites y consistencia posibles (a veces llamadas "reglas del negocio"). Pero, puede ser difícil perfeccionar esta verificación cuando están involucrados muchos campos. Si la entidad externa que proporciona los datos es una aplicación separada., existe aún más posibilidad de error.

Dado que debe manejarse la posibilidad de errores, ¿cómo se programa un método para manejar una entrada ilegal? (Por ejemplo, un método que da el saldo para un número de cuenta cuando las precondiciones requieren con claridad que el parámetro de la cuenta sea legal). Si se siguieron todos los aspectos del proceso de desarrollo, entonces los parámetros del método siempre serán legales al llamar al método; pero, ¿debe programarse una verificación del valor del parámetro para el caso de que el diseño o la implementación tengan fallas? Esto depende de los requerimientos.

Por ejemplo, suponga que existe un requerimiento del sistema de que la operación continua de la aplicación es lo más importante, aun cuando su ejecución se degrade o tenga fallas. En este caso, el programador debe manejar todas las entradas, incluso las que no tengan sentido. Como ejemplo, considere una aplicación que supervisa las funciones del corazón y controla el oxígeno que se suministra al paciente. Suponga que se está codificando un método "process" que tiene un parámetro entero que contiene el tipo de medición donde el tipo de medición debe ser un número positivo. Suponga que esta aplicación no puede darse el lujo de parar cuando se da al método interno un entero ilegal debido a un defecto de desarrollo. Entonces, el código verifica la entrada: 1) establece los valores predeterminados seguros, si es posible; 2) coloca toda la aplicación en un modo de operación predeterminado o 3) invoca una excepción y pasa la responsabilidad al que llama. Por lo común la aplicación también envía una señal de alerta.

En ocasiones, la recepción de entradas ilegales se puede manejar de manera consistente con los

requerimientos explícitos. Esto ocurre, por ejemplo, cuando los datos se transmiten por una línea de comunicación defectuosa. El método de recepción puede diseñarse para esperar ciertos datos, pero con frecuencia la aplicación requiere de manera explícita que continúe la ejecución, aun cuando los datos no sean legales. Aquí, deben verificarse los datos y procesarse los errores de acuerdo con los requerimientos (por ejemplo, "si la señal no está entre 3.6 y 10.9, descartarla y escuchar la siguiente señal…").

¿Qué se hace con los métodos cuyo comportamiento excepcional no está determinado por los requerimientos? Primero, sus precondiciones deben especificarse de manera exhaustiva para aclarar la condición bajo la que se llama. Pero aun así ¿deben verificarse los valores de sus parámetros para asegurar que se cumplen las precondiciones? Se hará la distinción entre ejecución durante el desarrollo y ejecución durante el despliegue.

La ejecución durante el desarrollo permite incluir código de pruebas y verificación en muchas partes de una aplicación, tal vez se desee insertar un código que verifique las precondiciones.

La ejecución del producto entregado requiere una perspectiva diferente. Si se llama al método con parámetros negativos, esto indica un defecto en la aplicación. Sería deseable una protección contra los propios errores, pero la cura debe ser preferible a la enfermedad.

Sin políticas claras, una verificación como ésta para los productos entregados puede ser como una pendiente resbalosa: no es factible verificar cada parte de una aplicación (por ejemplo, ¿debe verificarse el código de verificación?). Dado un tiempo fijo, existe un potencial significativo para desperdiciarlo en insertar código de auto protección a costa de inspeccionar el material para que sea correcto.

Los desarrolladores pierden el control de su aplicación cuando se usa un valor predeterminado arbitrario cuyas consecuencias no se conocen. Existen varias razones para ello. No es ético distribuir, sin advertencia, una aplicación que maneja un desarrollo defectuoso con una continuación incorrecta deliberada (es decir, una continuación no establecida en los requerimientos). Un defecto es un error, pero un valor predeterminado arbitrario no especificado de manera explícita en los requerimientos es un encubrimiento. En términos prácticos, con frecuencia es preferible restablecer una aplicación abortada que ejecutarla de modo incorrecto (piense en una aplicación que traza el curso de un avión). Segundo, un error de procesamiento indisciplinado oculta un defecto y se vuelve costoso encontrarlo si se compara con permitir que la aplicación se detenga (deseable: durante las pruebas). Tercero, debe haber consistencia en las expectativas del diseño. El código que en realidad dice "creo que el diseño de esto es incorrecto, pero es lo que esperamos de que la aplicación ejecute si estuviera bien hecha", no resuelve el problema de diseño y es mala Ingeniería.

2.5.6 Estándares de Programación

El uso de estándares mejora la disciplina, lo legible y lo portátil de un programa. Se presentará un conjunto de normas que se pueden usar como ejemplo y para reflexionar. Algunos de los siguientes estándares se adaptaron de Scout Amber [Am].

Convenciones de Nombres

Debe usarse una convención de nombres para las variables. Los Ingenieros tienden a ser emocionales en

cuanto a sus convenciones favoritas y muchas veces el consenso es imposible. De todos modos, las convenciones son necesarias. Debe asignarse un tiempo limitado para decidir las convenciones y un método para cumplirlas. Por ejemplo, puede designarse un miembro del equipo delinear las convenciones, enviarlas por correo electrónico a los otros miembros para sus comentarios y concretar las opciones con la aprobación del líder del equipo. Debe hacer una guía de la aceptación de excepciones.

El siguiente es un ejemplo de las convenciones de nombres:

- Las entidades se nombran con palabras concatenadas como en longitudCilindro. Es fácil entenderlas y ahorran espacio. Deben permitirse excepciones de espacios a juicio del programador. Por ejemplo, auto se puede expresar mejor como a_AAA_AA_A_Auto en lugar de seguir la convención a toda costa.
- Inicie los nombres de clases con mayúscula. Esto las diferencia de las variables. Algunas herramientas preceden el nombre de las entidades con letras o combinaciones de letras estándar, por ejemplo, C para las clases como en CCliente. Esto es útil cuando es más importante saber el tipo de nombre que ese inconveniente.
- Inicie los nombres de las variables con minúsculas. Las constantes puede ser una excepción.
- Nombre las constantes con mayúsculas como en SOY_UNA_CONSTANTE. Es difícil leer SOYUNACONSTANTE; SoyUnaConstante puede confundirse con una clase; soyUnNombre no indica que sea una constante.
- Inicie (o termine) el nombre de variables de instancias de clase con un guión bajo como en _horaDelDia para distinguirlo de otras variables, ya que son globales para su objeto. Esta convención se ha usado.

  Otra convención podría ser agregar un sufijo I para indicar variables de instancia, como en horaDelDiaI. Cada variable de instancia es global para la instancia de la clase y cuando se encuentra en un bloque de código es útil saberlo.
- Considere usar una notación para distinguir las variables estáticas de una clase. Una forma puede ser utilizar el sufijo S como en numAutosConstrS. Recuerde que una variable estática es global para una clase y es útil saber que una variable encontrada en un bloque de código es de éstas.
- Use "obtener" "establecer" y "es", para los métodos de acceso como obtenerNombre(), establecerNombre(), esCaja() (que regresa un valor booleano).
- Aumente éstos con otros "llamadores" y "establecedores" estandarizados de colecciones, por ejemplo insertarEnNombre(), removerDeNombre(), nuevoNombre().
- Considere una convención de parámetros. Una convención es usar el prefijo "un" como en sum(int unNum1P, int unNum2P).

Métodos para documentar con una descripción de lo siguiente

- Precondiciones y poscondiciones.
- Qué hace el método.
- Porqué lo hace.

- Qué parámetros debe pasar.
- Excepciones que lanza.
- Razón para la elección de visibilidad.
- Maneras en que cambian las variables instanciadas.
- Fallas conocidas.
- Descripción de pruebas, establezca si se ha probado el método y la localización de su texto de prueba.
- Historia de cambios si no se usa un sistema de administración de configuración.
- Ejemplo de cómo trabaja el método.
- Documentación especial para métodos de hilos y sincronizados.

Use un estándar consistente para la separación. Como las líneas en blanco son útiles para separar las secciones de código dentro de los métodos, un estándar consistente es usar líneas dobles en blanco entre los métodos.

Dentro de los métodos, considere estándares como los siguientes.

- Realice una operación por línea.
- Intente mantener los métodos para una sola pantalla.
- Use paréntesis dentro de las expresiones para aclarar el significado, aún cuando no lo requiera la sintaxis del lenguaje. Esto es una aplicación de la máxima "si lo sabes, muéstralo".

Al dar nombre a las clases, use nombres en singular como Cliente, a menos que el propósito expreso sea coleccionar objetos (en cuyo caso, Clientes puede ser adecuado). Para evitar la proliferación de clases, en ocasiones es deseable tener una clase que recolecte sus propias instancias. Esto se haría con un miembro de datos estático de la clase.

Documentación de Atributos

- Proporcione toda las invariantes aplicables (factores cuantitativos de los atributos, como "36< _longitud*_ancho <193").

Para cada atributo:

- Establezca su propósito.
- Proporcione todos los invariantes aplicables (hechos cuantitativos acerca del atributo, como "1< _edad <130" o "36< _longitud*_ancho <193".

Inicialización de Atributos

Los atributos siempre deben estar inicializados para que el programador tenga control de su programa.

2.5.7 Programas con Demostración Formal que son correctos

Decir que un programa "tiene demostración formal de que es correcto" significa que se proporciona una demostración matemática para mostrar que el programa satisface sus requerimientos. La demostración se basa en los requerimientos para el código y el texto del código. La demostración es independiente de la compilación en un compilador y es independiente de las pruebas. Esta es una capacidad ideal.

Una buena demostración matemática es un argumento bien construido que convence al autor y a otros de que la declaración es cierta. El lenguaje de matemáticas es la clave de las demostraciones, pero éstas se pueden aplicar sólo para demostrar declaraciones expresadas en términos matemáticos.

Primero, el requerimiento mismo debe expresarse con precisión. Para ello se usa un lenguaje especial como el lenguaje "Z" para establecer requerimientos formales. Se puede también usar una forma más sencilla para los requerimientos como precondiciones y poscondiciones. Las precondiciones especifican todas las suposiciones hechas al invocar la función. Las poscondiciones especifican el estado requerido a la conclusión de la ejecución de la función.

Como ejemplo, considere el siguiente requerimiento para una función f().

Precondición: g es un arreglo de enteros de longitud n (g esta "dado").

Poscondición: r=max{g[0], g[1], …, g[n-1]} (r es el "resultado").

De manera informal, este requerimiento llama a f() para determinar el valor máximo del arreglo g. Por supuesto es sencillo programar esto, pero se hará en una forma para la que es más fácil la demostración de que es correcta.

Una manera de familiarizarse con programar fragmentos de código que se demuestra que son correctos es aplicar la forma (pseudocódigo) mostrada en la figura siguiente. Para demostrar que las variables terminan con los valores requeridos, el programador sólo tiene que verificar que el ciclo termina. Si el programador puede demostrar esto, entonces la afirmación las variables no tienen los valores requeridos ya no es válida cuando termina el ciclo (doble negativa) y por lo tanto, se alcanza el estado deseado.

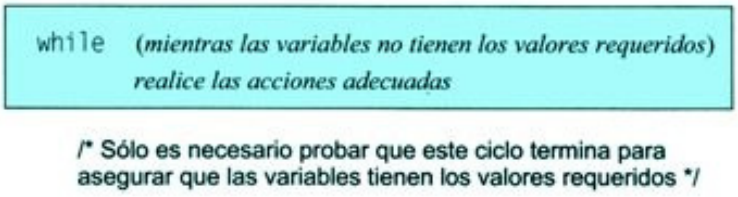


Fig. 90.- Un esquema sencillo para probar que es correcto.

A primera vista, los programas con demostración formal de que son correctos parecen un camino muy diferente para programar, pero en realidad no es así. El proceso intuitivo que se usa no es diferente; la

formalidad sólo ayuda a controlar el proceso con mucha mayor precisión. Por ejemplo, la mayoría de las personas programaría una función max() estableciendo un ciclo y acumulando el máximo "hasta ahora". En otras palabras, durante los cálculos se mantendría cierta la siguiente afirmación:

$$[j <= n-1] \text{ AND } [\, r = \text{máx}\,\{g[0], g[1], \ldots, g[j]\}\,] \text{ // afirmación 1}$$

Una afirmación que debe mantenerse cierta como ésta se conoce como invariante. También se dice que la afirmación se "mantiene invariante" (sin cambio). Aunque los valores de las variables individuales dentro de ella (j y r) cambien, la afirmación completa permanece cierta. Se puede pensar que una invariante es un tipo de sube y baja en el que las variables se coordinan de manera continua para mantenerlo nivelado. Cuando el cálculo de max() avanza y j aumenta, los valores de r y j se coordinan para que la afirmación se mantenga cierta. Observe que max() es el nombre de la función que se está desarrollando, mientras que la palabra "máximo", usada en la afirmación de la invariante, está dirigida a las personas y se refiere a un concepto matemático.

Una invariante se usa en un programa como un tipo de armazón sobre la que se construye el resto del programa. Es análoga a la infraestructura para Ingeniería Estructural: algo "dado" sobre lo que se puede construir para satisfacer los requerimientos. Esto se sugiere en la siguiente figura, donde se desarrolla una infraestructura que proporciona una "invariante" sobre la cual se puede construir una carretera elevada.

Un programa con Demostración Formal de que es correcto para max() se muestra en la siguiente figura. Observe que se usa un ciclo while() en lugar de for() porque es más sencillo usar los ciclos while con demostraciones de que son correctos. Se coloca un límite n porque las computadoras reales no pueden garantizar la efectividad del programa para cualquier n sin límite.

```
// Definir I: 0 ≤ j ≤ n−1 < 100 y r = máx{g[0], g[1], …, g[j]}
// Después de los comandos siguientes, I es cierta:
int r=g[0];
int j=0;
// Este bloque mantiene a I cierta
while( j < n-1 )
{     if( g[ j+1 ]>r )
                r=g[ j+1 ];
      ++j;
}
```

Fig. 91.- Programa con Demostración Formal de que es correcto para sumar un arreglo de longitud n, 1 de 2.

En la siguiente figura, se establece una demostración de que este programa es correcto.

/* Suponiendo que el ciclo termina (demostración adelante), en este punto se sabe que $j < n - 1$ ya no es cierta. También se ha mantenido I invariante (cierta). Al unir esto,

- $j < n - 1$ es falso, AND
- $j <= n - 1$ (de la afirmación 1), AND
- r = máx $\{g[0], g[1], \ldots, g[j]\}$ (de la afirmación 1)

de manera que $j = n - 1$ AND r = máx $\{g[0], g[1], \ldots, g[n - 1]\}$
— que era la meta
Sólo queda demostrar que el ciclo de *while* termina. Como I se mantiene invariante, la cantidad $n - j$ es siempre positiva; además, $n - j$ disminuye en 1 cada iteración. La única manera en que esto puede ocurrir es si el ciclo termina. */

Fig. 92.- Programa con Demostración Formal de que es correcto para sumar un arreglo de longitud n, 2 de 2.

Este ejemplo sencillo ilustra el argumento de los científicos e ingenieros que piensan que, con el tiempo, la Ingeniería de Software usará cada vez más los métodos formales. Señalan que estos métodos formales sólo hacen más profesional y precisa la manera en que ya se implementan las aplicaciones.

2.5.8 Herramientas y Entornos para Programación

Con frecuencia se dice que el hombre es un hacedor de herramientas y esto no es menos cierto para los desarrolladores de software.

Los entornos de desarrollo interactivos (IDE, Interactive Development Environment) tienen un uso amplio para permitir que los programadores produzcan más código en menos tiempo. Incluyen características de "arrastrar y dejar" (drag-and-drop) para formar las componentes de la interfaz gráfica, representaciones gráficas de los directorios, depuradores (debuggers), ayudas automáticas (wizards) y otros.

Las aplicaciones deben tener alguna forma de código fuente con la finalidad de que se puedan entregar. Muchas organizaciones del nivel 1 de CMM llevan esto al extremo y producen sólo código fuente. Los proveedores de herramientas se han dado cuenta de que pueden contar con que sus clientes producirán código fuente y con esperanzas de que esté documentado, los ha motivado para producir herramientas de Ingeniería Inversa que tienen el código fuente como entrada y que desarrollan documentación limitada.

Varias herramientas orientadas a objetos (como Rational Rose, Together/J/C++) generan el código fuente a partir de los modelos de objetos. No se puede esperar que estas herramientas de Ingeniería Directa generen más que los esqueletos del código con los que el programador debe trabajar para producir la implementación. Las mismas herramientas realizan Ingeniería Inversa mediante la producción mecánica de modelos de objetos a partir del código fuente (de ahí el término "Ingeniería de Viaje Redondo" (round trip)).

La historia de las herramientas en otras ramas de la ingeniería (como CAD/CAM) sugiere que las herramientas de programación tendrán mejoras significativas continuas que seguirán apoyando las habilidades de programación y reducirán las tareas penosas y mecánicas.

2.5.9 Métricas Estándar para el Código Fuente

Cuenta de Líneas

Las "Líneas de Código" constituyen una medida útil, aunque no perfecta, para programación. Debe establecerse una manera estándar de contar. Por ejemplo:

- Cómo contar las declaraciones que ocupan varias líneas (¿1 o "n"?).
- Cómo contar los comentarios (¿0?).
- Cómo contar las líneas que consisten en while, for, do, etcétera (¿1?).

La elección depende del precedente, el estándar usado para los datos históricos y las herramientas automáticas para contar que se adopten. Mantener los datos de las líneas de código consistentes con las definiciones seleccionadas es mucho más importante que la selección exacta de la definición.

Métricas de IEEE

La siguiente es una muestra pequeña de medidas del IEEE 982 que indica como cuantificar la calidad del código fuente.

IEEE Métrica 14. Mediciones de la Ciencia de Software.

Cantidades medidas usadas:

Sea n_1= número de operadores distintos (+,*, etcétera) en el programa. Por ejemplo, en el programa {x=x+y; z=x*w;}, n_1= 2.

Sea n_2= número de operandos distintos en el programa. Por ejemplo, en {x=x+y; z=x*w;} n_2=4, ya que + y * implica cada uno dos operandos.

Sea N_1= número total de ocurrencias de los operadores en el programa.

Sea N_2= número total de ocurrencias de los operandos en el programa.

Muestra de estimaciones de Halstead:

Longitud estimada del programa: $n_1 (\log n_1) + n_2 (\log n_2)$.

Dificultad del programa: $(n_1 N_2)/(2n_2)$.

Se encuentran más detalles de mediciones de la Ciencia de Software en [Ha5].

IEEE Métrica 16. Complejidad ciclomática. Esta medida determina la complejidad estructural de un bloque de código, en esencia, con la cuenta de ciclos, que son factores importantes para la complejidad. Se puede usar para identificar módulos cuya complejidad deba intentarse reducir.

Un ejemplo de una política basada en esta medida es requerir que todos los módulos con complejidad ciclomática que excedan la medida por 20% pasen una revisión especial.

Cantidades medidas usadas

N: número de nodos (declaraciones de programa).

E: número de aristas (una "arista" une el nodo m con el nodo n, si la sentencia n puede seguir de inmediato a la sentencia m).

Un método para calcular la complejidad ciclomática es determinar E-N+1. Otra manera de obtener este número es contar el número de regiones cerradas formadas por aristas que no se pueden dividir en regiones cerradas más pequeñas. Esto es evidente en la figura siguiente.

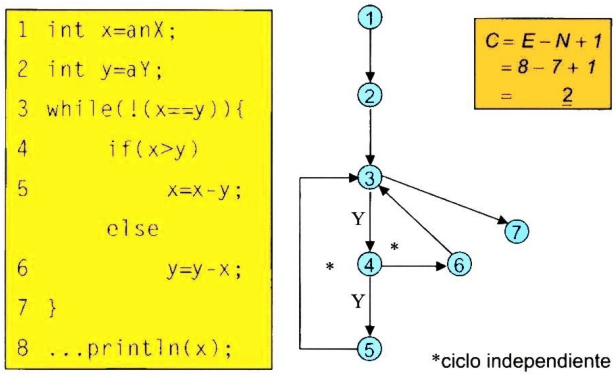


Fig. 93.- Complejidad Ciclomática.

Métricas Especiales para el Código Fuente

La complejidad ciclomática expresa el número de ciclos en un programa, pero no diferencia entre los ciclos anidados y no anidados. En general, los ciclos dentro de ciclos son más propensos a error que las secuencias de ciclos independientes.

No es difícil crear métricas que determinen aspectos significativos identificados. Por ejemplo, para medir la complejidad de ciclos, se puede estimar que cada ciclo anidado aumenta la complejidad en un orden de magnitud (de manera burda, un factor de 10). El siguiente ejemplo anidado puede tener un factor de complejidad anidada como sigue.

Ciclo 1 - cuenta 1
Ciclo 2 - cuenta 1
    Ciclo 2.1 - cuenta 10 (ciclo dentro de ciclo 2)
    Ciclo 2.2 - cuenta 10
        Ciclo 2.2.1 - cuenta 100
Ciclo 3 - cuenta 1
Ciclo 4 - cuenta 1
Ciclo 5 - cuenta 1
    Ciclo 5.1 - cuenta 10

Al usar esta métrica, el valor del resultado de la complejidad es 135. Esta métrica se puede normalizar si se divide entre las líneas de código y se compara con un promedio de la compañía o división.

2.5.10 Inspección de Código

Recuerde que la severidad es una parte esencial de los datos de la inspección, pues permite asignar prioridades a los defectos y proporciona un programa de tiempos de trabajo racional. (Vea la tabla siguiente). No vale la pena dedicar tiempo de una reunión de inspección a los defectos triviales. Una buena manera de manejarlos es que los inspectores den un listado del código fuente marcado directamente al autor.

Severidad	Descripción
Importante	Requerimiento(s) no satisfecho(s)
Media	No importante ni trivial
Trivial	Un defecto que no afectará la operación o el mantenimiento.

Tabla 12.- Clasificación de severidad de defectos usando prioridades.

Problema de lógica: (Casos o pasos olvidados; lógica duplicada; descuido de condiciones extremas; funciones innecesarias; mala interpretación; prueba de condiciones faltantes; verificación de variables equivocadas; ciclo de iteración incorrecto, etcétera).

Problema de cálculo: (Ecuaciones insuficientes o incorrectas; pérdida de precisión; convención de los signos mal).

Problema de interfaz/tiempo: (Interrupciones mal manejadas; tiempo de E/S incorrecto; mala asociación de subrutina/módulo).

Problema de manejo de datos: (Datos mal inicializados; acceso o almacenamiento de datos incorrecto; escala o unidades de datos incorrectos; dimensión de datos incorrecta).

Alcance de los Datos: (Datos de sensor incorrectos o faltantes; datos del operador incorrectos o faltantes; datos externos incorrectos o faltantes; datos de salida incorrectos o faltantes; datos de entrada incorrectos o faltantes; etcétera).

Documentación: (Descripción ambigua, etcétera).

Calidad del documento: (Se aplica a estándares no cumplidos, etcétera).

Mejoramiento: (Cambio en requerimientos del programa, etcétera).

Falla causada por el arreglo anterior.

Interoperabilidad: (No compatible con otro software o componentes).

Problema de cumplimiento de estándares.

Otros: (ninguno de los anteriores).

2.5.11 Documentación Personal de Software

Es necesario que cada Ingeniero conserve documentación de su trabajo actual; esta documentación tiene un nombre, como "Archivo de Documentación de Software" o "Documentación Personal de Software" (DPS). Este documento permite al Ingeniero reportar el estado en todo momento y parte de él se convierte en parte del archivo del proyecto. El DPS puede incluir los elementos de la siguiente figura.

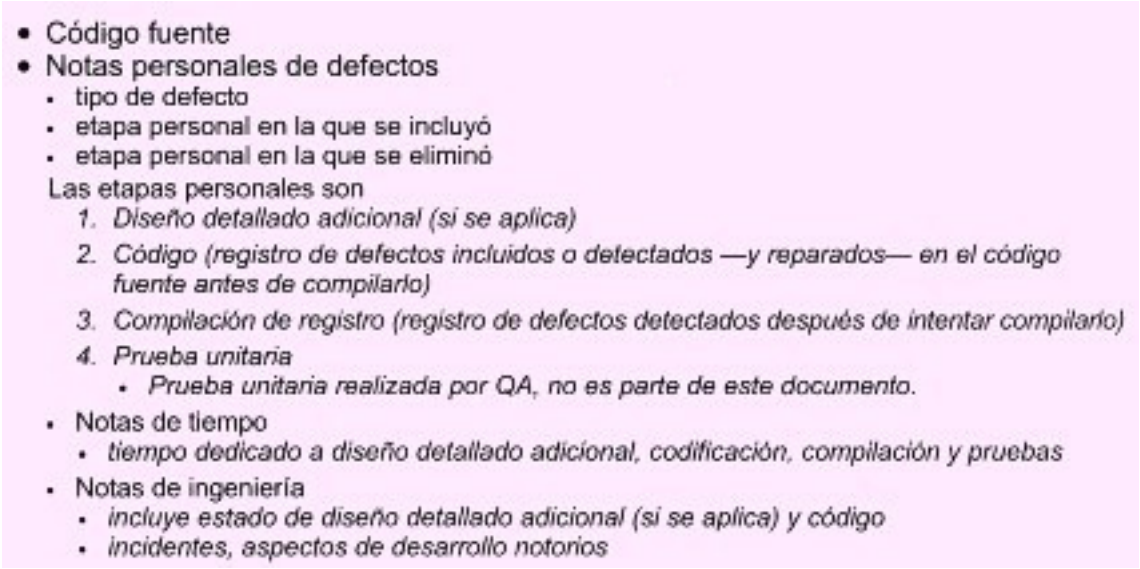


Fig. 94.- Documentación Personal de Software.

El proceso de Software Personal (Personal Software Process) del Software Engineering Institute exige la información de tiempo y defectos. El equipo o el líder del proyecto determinan cómo usar y archivar el DPS de todos los miembros. La palabra "personal" no quiere decir que el DPS es propiedad del Ingeniero; el trabajo que una organización paga es propiedad de la organización. En esencia, se evalúa a los Ingenieros según los productos terminados de su proceso personal y de acuerdo con lo adecuado de la documentación del mismo. En general, no se evalúan en todos los pasos de ese proceso.

2.6 Proceso de Pruebas

2.6.1 Pruebas de Unidades

No se puede probar una aplicación para cada posibilidad, porque el número de maneras en que puede ejecutarse un programa de computadora no trivial es ilimitado. Así, al probar no se puede demostrar que la aplicación no tiene defectos, como pueden hacer las pruebas de que es correcto. Las pruebas sólo pueden mostrar la presencia de defectos.

A menudo, hacer pruebas se malinterpreta en esencia como un proceso de establecer la confianza, como en "probar para asegurar que es correcto". En ocasiones, ésta es la meta de las pruebas, en especial justo antes de la entrega, o en las pruebas de regresión. Sin embargo, un propósito importante de las pruebas es casi opuesto al de establecer la confianza. El propósito no es demostrar que una aplicación es satisfactoria, sino determinar con firmeza en qué parte no lo es.

El tiempo dedicado a las pruebas implica un gasto considerable y se intenta obtener el máximo beneficio de este gasto. Para una aplicación en proceso de pruebas, cuantos más defectos se encuentren por dólar de mano de obra, más alto será el pago de la inversión en las pruebas. Entonces, el propósito de probar es encontrar el mayor número de defectos, con el más alto nivel de severidad posible. Esto se resume a continuación.

Meta de las pruebas: Maximizar el número y severidad de los defectos encontrados por dólar gastado.

- Así, haga pruebas pronto.

Límites de las pruebas: Probar sólo puede determinar la presencia de los defectos, nunca su ausencia.

- Use demostraciones formales de que es correcto para establecer la "ausencia".

Las pruebas son responsables de más de la mitad del tiempo dedicado a los proyectos. La recompensa por encontrar un defecto pronto en el proceso es al menos un ahorro de diez veces comparado con detectarlo en la etapa de integración o –peor aun- después de la entrega. En consecuencia, las pruebas se realizan pronto y con frecuencia.

Igual que con el aseguramiento de la calidad en general, las pruebas de código deben realizarlas personas diferentes a las que lo desarrollaron. Cuando un Ingeniero desarrolla un código, se forma una visión de lo que debe hacer ese código y al mismo tiempo, desarrolla circunstancias típicas en las que debe ejecutarse el código. Sin duda puede suponerse que el código mostrará algunos problemas en esas circunstancias particulares. De manera consciente o no, éstas constituyen los casos de prueba del desarrollador. Así, cuando un individuo prueba su propio código tiende a ocultar justo eso que debe descubrir.

Las "pruebas de unidades" son el primer tipo de prueba que se aplica. El siguiente nivel consiste en las pruebas de integración. Esto valida la funcionalidad global de cada etapa de la aplicación parcial. Por último, las pruebas del sistema y de aceptación validan el producto final. La siguiente figura, ilustra este tipo de pruebas y sus relaciones

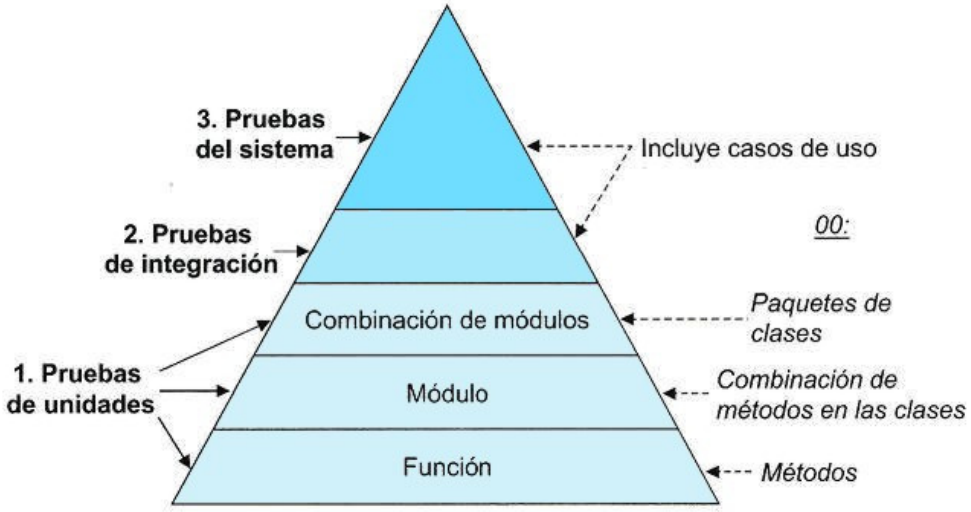


Fig. 95.- Pruebas de Visión Global.

Significado de "Pruebas de Unidades"

La meta de las Pruebas de Unidades es estructural mientras que el otro tipo de pruebas es funcional típico. Como una analogía, probar cada cable de un puente de módulo en la fábrica es un tipo de pruebas de unidades ya que involucra unidades estructurales. Como se muestra en la figura anterior; en general las funciones son las partes más pequeñas a las que se aplican las pruebas de unidades. La siguiente unidad en tamaño es el módulo (la clase, en el caso de orientación a objetos). Algunas veces, las combinaciones de módulos se consideran "unidades" para los propósitos de las pruebas.

Las unidades a las que se aplican las "pruebas de unidades" son los bloques de construcción de la aplicación, un poco como los ladrillos individuales sobre los se apoya una casa. Mientras algunos defectos en unos cuantos ladrillos no tienen un efecto serio en la casa, las aplicaciones de software sí pueden ser muy sensibles a los defectos de cada bloque. Una vez integradas las partes defectuosas en una aplicación, puede tomar una cantidad enorme de tiempo identificarlas y repararlas. Así, los bloques de construcción de software deben ser completamente confiables y esta es la meta de unidades.

Una prueba de unidad es un complemento de la inspección y del uso de la formalidad correcta.

En términos del proceso de desarrollo de software unificado, las pruebas unitarias se llevan a cabo durante las iteraciones de Elaboración y también en las primeras iteraciones de la Construcción como lo sugiere la siguiente figura.

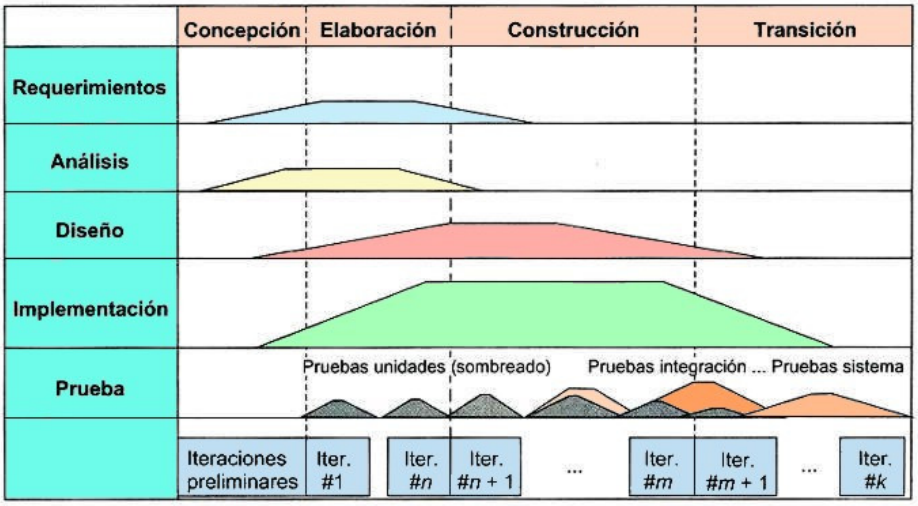


Fig. 96.- Pruebas de Unidades en el Proceso Unificado [Ja].

2.6.2 Mapa Conceptual Típico de las Pruebas de Unidades

La siguiente figura, basada en el estándar IEEE 1008-1987, muestra un mapa conceptual típico para las Pruebas de Unidades. Se dan los pasos para el proceso de Pruebas de Unidades.

- Los insumos para el proceso de planeación de pruebas consiste en los requerimientos y el diseño detallado. Recuerde que cada requerimiento corresponde a un código bien definido (una función, cuando es posible). El diseño detallado suele contener clases y métodos adicionales. Éstos también son importantes para la calidad de las aplicaciones y deben probarse en el mismo grado que los requerimientos individuales. La salida del proceso de planeación de pruebas es un plan de pruebas de unidades (por ejemplo "1) probar método 84; 2) probar método 14,…,m) probar clase 26,…").
- El siguiente paso es la adquisición de los datos de entrada y salida asociados con cada prueba. Se contará con algunos de estos datos para pruebas anteriores (por ejemplo, iteraciones anteriores o versiones anteriores del producto). El producto de este paso se llama conjunto de pruebas.
- Por último, se ejecutan las pruebas.

Se presentan los pasos que requiere el estándar de IEEE (1008-1987) para las pruebas de unidades, los cuales amplían el mapa conceptual anterior.

- Planear el enfoque general, recursos y programa de tiempos.
- Determinar las características basadas en los requerimientos que deben probarse.
- Refinar el plan general.

- Diseñar un conjunto de pruebas.
- Implementar el plan y diseño refinados.
- Ejecutar los procedimientos de pruebas.
- Verificar que terminen.
- Evaluar el esfuerzo y la unidad de las pruebas.

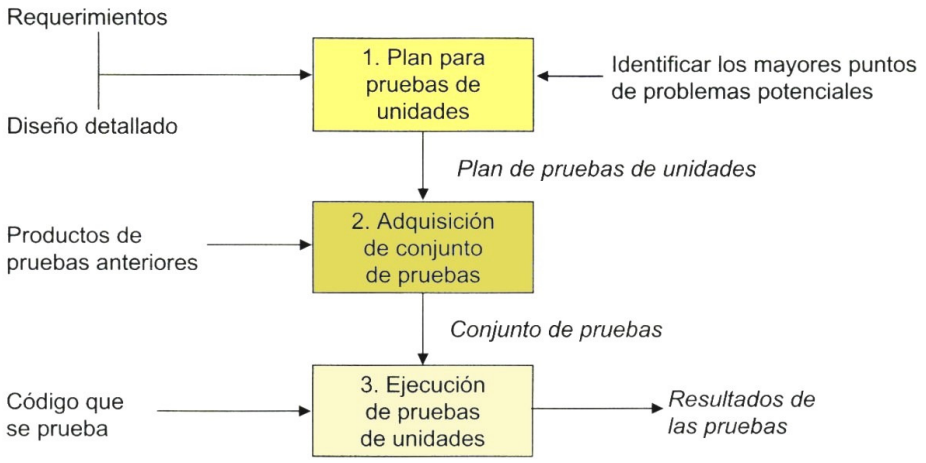


Fig. 97.- Mapa conceptual para Pruebas de Unidades Copyright © 1986 IEEE.

2.6.3 Tipos de Pruebas

Pruebas de caja negra, caja blanca y caja gris

Cuando el único interés es si una aplicación o parte de ella proporciona la salida adecuada, se prueba que cumpla cada requerimiento al usar una entrada apropiada. Esto se llama prueba de caja negra porque no se pone atención al interior de la "caja" (la aplicación): sería lo mismo si la caja fuera "negra". Las pruebas de caja negra pueden ser suficientes si se puede asegurar que agotan todas las combinaciones de entrada. Esto probaría al cliente que todos los requerimientos se satisfacen. Sin embargo, ninguna prueba cubre todas las posibilidades.

Una prueba de caja negra es como probar un puente con el cruce de varias combinaciones de vehículos. Esto no es suficiente porque también deben verificarse los elementos del puente y la manera en que se ensambló. Esto último es la idea de las pruebas de caja blanca. La siguiente figura ilustra estas pruebas.

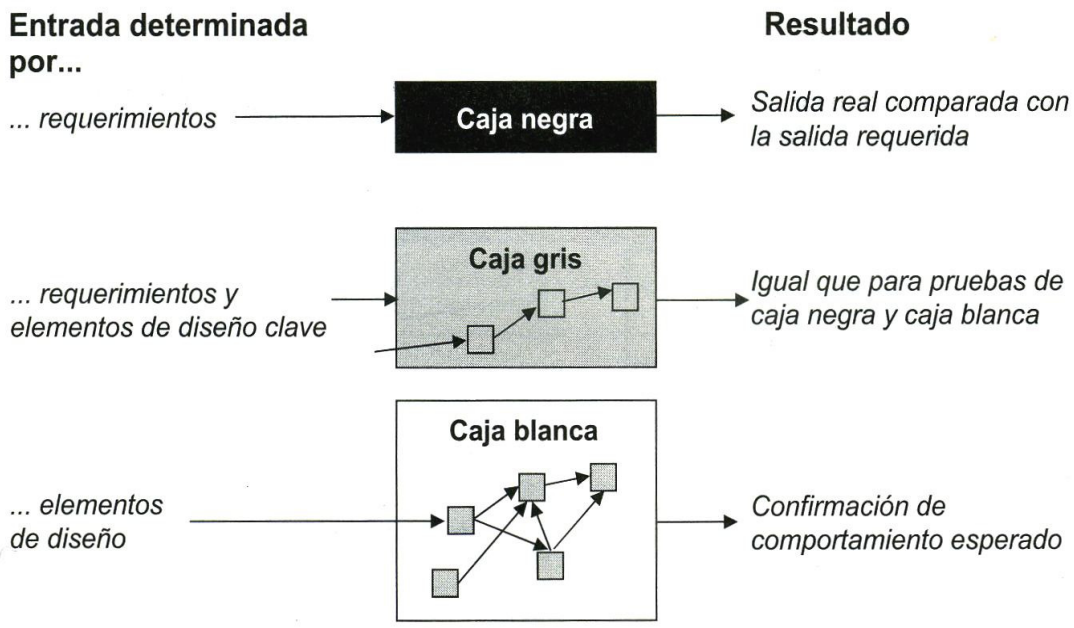


Fig. 98.- Pruebas de caja negra, caja gris y caja blanca.

La meta de las pruebas de caja blanca es probar las líneas de falla más probables de la aplicación. Para realizar pruebas de caja blanca, primero se desglosa el diseño de la aplicación en busca de trayectorias y otras particiones de control y datos. Después se diseñan pruebas que recorran todas o algunas de estas trayectorias y se ejecutan todas las partes o elementos. Un nombre que describe mejor esta actividad es "pruebas de caja de vidrio".

Las pruebas de "caja gris" consideran el trabajo interior de la aplicación o unidad bajo prueba, pero sólo en un grado limitado, también pueden incluir aspectos de caja negra.

Peticiones equivalentes para pruebas de caja negra

Como no se pueden probar todas las combinaciones, se buscan casos de prueba representativos, la siguiente figura, ilustra el conjunto de casos de prueba posibles para tres variables (principal, tasa de interés y estimación de la inflación) en una aplicación de finanzas. El problema es representar de la mejor manera el conjunto infinito de posibilidades con un conjunto finito tan representativo como sea posible. La partición de equivalencia es la división de los datos de entrada de las pruebas en subconjuntos tales que si cualquier entrada única tiene éxito, entonces es probable que todas las demás entradas tengan éxito.

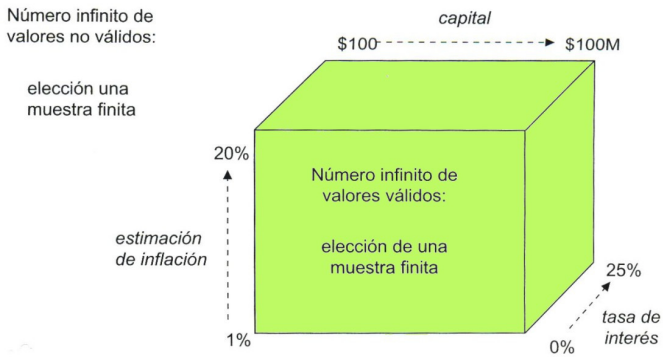


Fig. 99.- Prueba de intervalos de entrada.

La figura siguiente ilustra la partición de equivalencia para un método que calcula una cantidad basada en un capital dado, una tasa de interés dada y una estimación de inflación dada.

La partición de equivalencia sombreada se describe en esta figura, por ejemplo, es:

"estimación de inflación entre 15% y 20%", "capital entre $65 M y $100M" y "tasa de interés entre 0% y 5%".

El pago máximo de realizar las pruebas casi siempre se obtiene a partir de los valores de las fronteras.

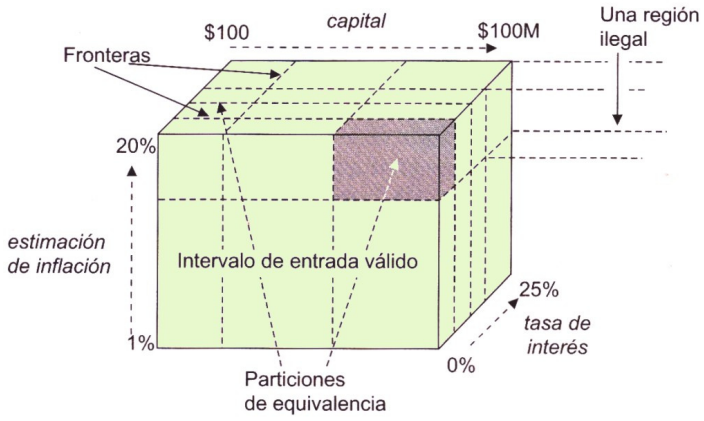


Fig. 100.- Prueba de particiones y fronteras de entrada.

Análisis del valor de la frontera para pruebas de caja negra.

En general, se llega a las particiones de equivalencia mediante la investigación de los valores que limitan las variables dentro de la aplicación. Por ejemplo, si la estimación de la inflación está entre 1% y 20%,

esto proporciona dos fronteras. Suponga que la aplicación maneja las estimaciones mayores que 15% de manera diferente a las menores que este valor. Esto introduciría una frontera adicional, como se muestra en la figura.

Las fronteras también pueden tomar una forma como x+y=7. Esto daría como resultado una condición como la de while(x+y≥7).

Al diseñar las pruebas, también se usan los valores fuera de estas fronteras (es decir, la entradas no válidas) como datos de prueba. Una vez establecidas las fronteras de las clases de equivalencia, los datos de prueba se generan del modo sugerido en la figura siguiente

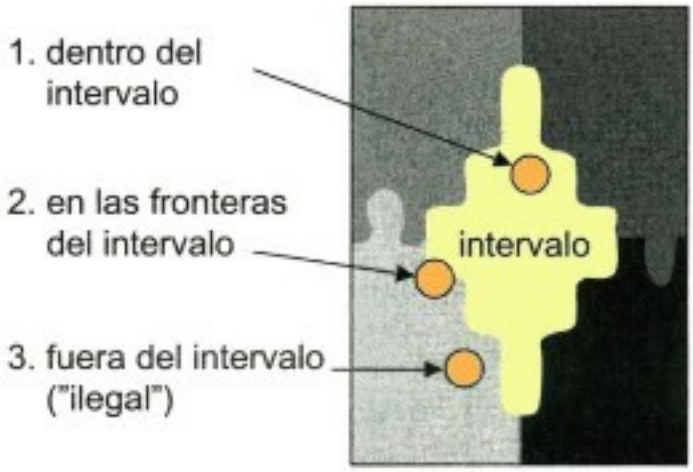


Fig. 101.- Intervalos de Prueba: casos elementales.

Cobertura de sentencia de pruebas de caja blanca

Cada sentencia en un programa debe ejecutarse en al menos una prueba. Esta cobertura es necesaria. Sin embargo, igual que en el ejemplo de la figura siguiente, la cobertura de sentencias de ninguna manera es suficiente para asegurar que un programa sea correcto. Los requerimientos para el programa de la figura siguiente, se especifican en el diagrama de flujo. El caso de prueba "u==2, v==0 y x==3" ejecuta cada comando en la implementación y genera la salida correcta (x==2.5). No obstante, el programa tiene defectos ya que no implementa el diagrama del flujo. Por ejemplo, genera "x==1" a partir del caso de prueba "u==3, v==0 y x==3", mientras que debía generar "x==2" según lo especifica el diagrama de flujo.

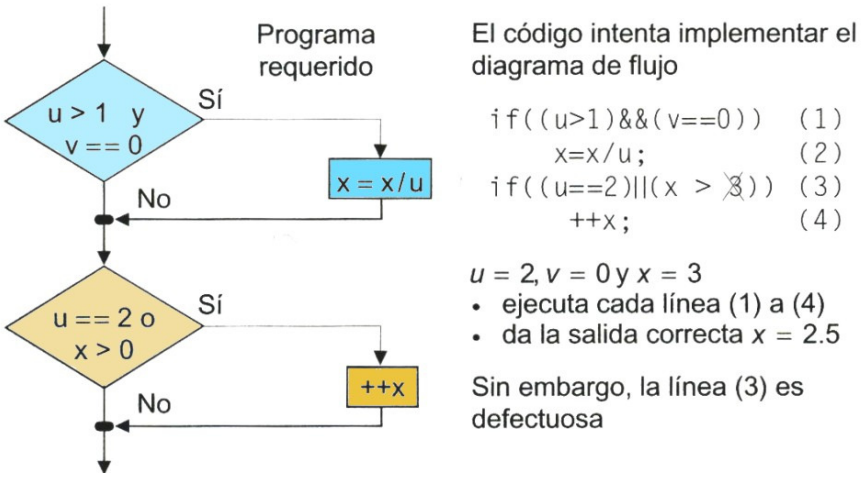


Fig. 102.- La cobertura de cada sentencia no es suficiente [My].

Cobertura de decisiones para pruebas de caja blanca

La cobertura de decisiones asegura que el programa tome cada rama de cada decisión. Por ejemplo, si se considera el diagrama de flujo de la figura siguiente, se verifica que dada "si" y cada "no" se ha seguido por lo menos una vez en las pruebas de conjunto.

De hecho, los ciclos implementan una secuencia de declaraciones condicionales. Por ejemplo, el ciclo:

```
For {i=0; i<3; ++i}
        v[i] = w[i+1] + w[i];
```

Se puede "extender" como la secuencia siguiente de declaraciones condicionales:

```
// Para i==0;
i=0;
v[i] = w[i+1] + w [i];
++i;
// Para i==1;
if ( i<3 )  {
        v[i] = w[i+1] + w[i];
        ++i;
}
// Para i==2;
if ( i<3 )  {
        v[i] = w[i+1] + w[i];
        ++i;
}
```

La cobertura de decisiones asegura que se ejecute cada iteración de cada ciclo. Esto se puede hacer de manera que cada iteración de cada ciclo se ejecute al menos una vez. Este enfoque no es difícil para ciclos, pero introduce complicaciones en los ciclos de while. Por ejemplo, ¿cómo se probaría cada iteración de while (u<v) { ….. } // ?

En ocasiones se pueden enumerar todas las posibilidades: otras se puede hacer una partición en grupos. Sin embargo, algunas veces es casi imposible una cobertura completa de las decisiones para ciclos while. Recuerde que los ciclos while suelen ser fáciles de manejar en los métodos formales y la inspección. Esto ilustra la naturaleza complementaria de los métodos formales, las inspecciones y las pruebas.

En general, la cobertura de decisiones incluye una declaración de cobertura, ya que el seguir todos los puntos de ramificación en todas las ramificaciones suele hacer que se encuentren todas las declaraciones en el código. Las excepciones son los programas con puntos de entrada múltiples. La cobertura de decisiones puede ser deficiente porque algunas de ellas pueden ocultar a otras. Por ejemplo, en la declaración:

```
If (A && B) . . . .
```

La condición B nunca se prueba si la condición A es falsa. Lo mismo en la declaración

```
If (A | | B) . . . .
```

La condición B nunca se prueba si la condición A es cierta.

Para manejar este problema, se puede aplicar la prueba de condiciones múltiples. Esta es una forma exhaustiva de probar las condiciones que examina cada combinación al menos una vez. El diseño de casos de prueba puede ser una tarea tediosa porque es necesario rastrear hacia atrás todo el programa a partir de cada condición para determinar la entrada adecuada. La generación de software con pruebas automáticas es esencial para delinear estos conjuntos de pruebas.

Hasta ahora, se ha centrado la atención en asegurar que todas las declaraciones se ejecuten y que los resultados sean los esperados. Esto constituye una técnica de caja gris en cuanto a que se prueba la entrada/salida (caja negra) y también todas las declaraciones (caja blanca). Además debe asegurarse que, conforme se ejecuta, el programa hace las transiciones por los estados planeados. Las pruebas basadas en afirmaciones lo logran.

Pruebas basadas en afirmaciones

Recuerde que las afirmaciones son declaraciones que se relacionan con las variables y expresan el estado. Por ejemplo, para expresar el estado de una aplicación de un cajero automático después que el usuario insertó su tarjeta e introdujo el número de identificación personal, se aplica la siguiente afirmación típica:

```
(tarjetaInsertada == cierto) && (nip == VALIDO)
```

Se usan afirmaciones para mantener el control intelectual de los cálculos y probar que el código sea correcto. En muchos casos, las afirmaciones son invariantes (no cambian) en todo un bloque estratégico de código. A menudo es útil insertar comandos en el código fuente que informan si una afirmación que se espera que sea cierta, en realidad lo es. Esta técnica de caja blanca se llama prueba basada en afirmaciones. Por ejemplo, si se supone que la suma de las variables "x" y "y" es 10 para todos los cálculos, entonces "assert(x+y==10);" se puede verificar durante la ejecución. La función assert(<argumento>) puede informar si el argumento es cierto o falso. Si se desea, los cálculos se pueden

suspender si la afirmación (assertion) es falsa, etcétera. Algunas veces las funciones assert() se dejan en el código fuente para la ejecución. Como en:

```
assert(dosisRayosXFatalPotencial<137);
```

Aunque por lo general se usan sólo para las pruebas de validación.

2.6.4 Aleatoriedad en las pruebas

En términos generales, se usan entradas de pruebas aleatorias para evitar muestras sesgadas en las pruebas. Por ejemplo, si se desea evaluar la opinión de la población respecto a un político, se toma una muestra aleatoria de la población. Esto aplica la aleatoriedad con un enfoque definido: primero se identifica el tipo de prueba, después se aplica la aleatoriedad para obtener datos de una muestra no sesgada para esta prueba. Lo mismo se cumple para las pruebas de software. Una vez seleccionado el tipo de prueba (como cobertura de decisiones) y definidos los límites de los datos, todavía queda una libertad sustancial para las entradas posibles. En el caso ideal, esta entrada debe elegirse al azar.

Como ejemplo, considere realizar una prueba de frontera para una aplicación que verifica si una secuencia de cuatro números reales x1, x2, x3 y x4 es un dato válido como muestra en un experimento. Suponga que "datos válidos" significa que:

$-5 < x1 \leq 10$

$x1 + x2 \leq x3$

$x3 \geq x4$

Las pruebas de frontera requieren la selección de los datos de prueba con las siguientes restricciones.

$x1 = -5$(ilegal); $x1=10$

$x1 + x2 = x3$

$x3 = x4$

Observe que x1 debe tener sólo dos valores, x3 se determina una vez seleccionados x1 y x2, y x4 se determina por x3. Así, debe elegirse a partir de un número infinito de valores de x2 y estos valores se eligen al azar para evitar el sesgo.

2.6.5 Planeación de pruebas de unidades

Se requiere un enfoque sistemático para las pruebas porque el número de unidades potenciales que se deben probar suele ser grande. Es sencillo establecer que "cada parte del trabajo debe probarse"; sin embargo, esto tiene muy poco significado porque se asigna sólo una cantidad finita de recursos (duración y personas-hora) a la etapa de pruebas. Así, la meta es detectar cuantos errores sea posible al nivel más serio posible con los recursos disponibles. Los pasos para el plan de pruebas de unidades se enumeran en los pasos siguientes:

- Decida la filosofía de las pruebas de unidades.
  - ¿Es responsable un ingeniero individual (común)?
  - ¿Revisadas por otros?
  - ¿Diseñadas y realizadas por otros?

El primer aspecto es identificar qué "unidades" deben probarse y quién lo hará.

Para proyectos de desarrollo orientado a objetos, una organización común para las pruebas de unidades es probar los métodos de cada clase, después las clases de cada paquete y luego el paquete como un todo.

El plan ideal y la ejecución de las pruebas de unidades debe realizarlo alguien diferente al desarrollador y de hecho, en ocasiones se hace en la parte de aseguramiento de la calidad de la organización.

Aunque esto tiene beneficios de independencia, requiere que los Ingenieros de AQ comprendan el diseño con gran detalle. Algunas organizaciones no tienen esta capacidad y asignan a AQ sólo a las pruebas de nivel alto. Las pruebas de unidades con frecuencia se dejan al grupo de desarrollo y se realizan de la manera en que ellos eligen. De cualquier forma, las pruebas quedan disponibles para inspección y para la posible incorporación a las pruebas de alto nivel. Parte de la independencia de AQ se puede capturar haciendo que los Ingenieros de Desarrollo realicen las pruebas de unidades en el código de otros.

- Decida qué/donde/cómo documentar.
  - ¿Conjunto de documentos personales individuales (común)?
  - ¿Cómo/cuando incorporar otros tipos de pruebas?
  - ¿Se incorporan documentos formales?
  - ¿Se usan utilidades de herramientas/pruebas?

La documentación de las pruebas de unidades consiste en los procedimientos de prueba, los datos de entrada, el código que ejecuta las pruebas y los datos de salida. Las pruebas de unidades pueden estar en el paquete de código o en documentos separados. La ventaja de unirlas con el código que prueban es la convergencia. La desventaja es que aumenta el tamaño del código fuente. Se pueden usar los precompiladores para eliminar el código de pruebas antes de compilar el producto que se entrega.

Las unidades y controladores se usan para ejecutar las pruebas de unidades que se documentan para uso futuro.

- Determine el grado de las pruebas de unidades (de antemano).
  - No sólo "realice pruebas hasta que el tiempo expire".
  - Asigne prioridades para que las pruebas importantes se hagan.

Dado que es importante probar "todo", debe definirse el alcance de las pruebas con cuidado. Por ejemplo, si una aplicación bancaria consiste en retiros, depósitos y consultas, las pruebas de unidades podrían especificar que cada método debe probarse con una cantidad igual de datos ilegales, de frontera y legales; o tal vez los métodos de retiro y depósito se prueban tres veces más que los de consulta, etcétera. En general, los métodos que cambian el estado (valores de las variables) suelen probarse en forma más extensa que los otros. Deben definirse los límites de lo que constituye la prueba "de unidad". Por ejemplo, ¿incluyen la prueba de paquetes?, o ¿se considera otro tipo de prueba?

Los Ingenieros también especifican el alcance de las pruebas de antemano, es decir, cuándo debe terminar el proceso de pruebas. Por ejemplo, ¿debe probarse cada unidad durante un periodo fijo?, ¿hasta obtener las tres primeras fallas?

Recuerde que la idea de las pruebas es realizar las que tienen mayores posibilidades de exponer errores. Al asignar prioridades a las pruebas, según su posibilidad de producir defectos, se tiende a optimizar el tiempo dedicado a ellas. El enfoque de prioridades depende de la unidad que se prueba. Los ciclos son una fuente importante de errores, lo mismo que las fronteras e interfaces. En la última categoría, las pruebas que incluyen subunidades, como las funciones, tienen abundancia de defectos potenciales, ya que cada subunidad espera que las otras sean de cierta clase y estas expectativas con frecuencia son erróneas.

- Decida cómo y dónde obtener los datos para las pruebas.

Se han mencionado entradas legales, en frontera e ilegales para estos datos. También se requiere cierta generación aleatoria. Cuando es posible, se usan herramientas que generan datos de entrada para las pruebas analizando el código fuente y detectando las fronteras y ramificaciones de los datos. Además, puede disponerse de una buena cantidad de versiones anteriores de la aplicación, fuentes estándar, puntos de referencia industriales, etcétera. Todo esto se documenta para referencia futura y reuso.

- Estime los recursos requeridos.
    - Use datos históricos si están disponibles.

Igual que con toda la planeación, se identifican las personas-mes y la duración requeridas para realizar las pruebas de unidades. La fuente más importante de esta estimación consiste en los datos históricos. Aunque las pruebas de unidades pueden unirse al código fuente, la ejecución por separado da datos invaluables.

- Registre tiempo, cuenta de defectos, tipo y fuente.

Los Ingenieros que participan determinan la forma exacta en la que registrarán el tiempo dedicado a pruebas de unidades, cuenta de defectos y los tipos de esos defectos. Los datos obtenidos se usan para evaluar el estado de la aplicación y pronosticar la calidad que tendrá el producto y su fecha de terminación. Además, los datos se convierten en parte del registro histórico de la organización.

A continuación se muestra una lista de verificación para realizar las pruebas de métodos:

- Verificar la operación con valores normales de los parámetros. (Prueba de caja negra basada en los requerimientos de la unidad).
- Verificar la operación en los valores límite de los parámetros (caja negra).
- Verificar la operación para valores de parámetros fuera de los límites (caja negra).
- Asegurar que ejecutan todas las instrucciones. (Cubrimiento de declaraciones).
- Verificar todas las trayectorias, incluidos ambos lados de todas las ramas (cubrimiento de decisiones).
- Verificar el uso de todos los objetos llamados.
- Verificar el manejo de todas las estructuras de datos.
- Verificar el manejo de todos los archivos.
- Verificar la terminación normal de todos los ciclos (parte de la demostración de que son correctos).

- Verificar la terminación anormal de todos los ciclos.
- Verificar la terminación normal de todas las recursiones.
- Verificar la terminación anormal de todas las recursiones.
- Verificar el manejo de todas las condiciones de error.
- Verificar el tiempo y la sincronización.
- Verificar todas las dependencias de hardware.

Para los métodos que surgen del diseño, a menudo no se cuenta con requerimientos explícitos contra los cuales realizar las pruebas. En el ideal, deben escribirse requerimientos para todas las clases de diseño una vez que éste se crea. Cuando no se escriben requerimientos separados, como suele ocurrir, deben diseñarse (quien hace las pruebas) los casos de prueba contra la funcionalidad que debe tener esa clase, una situación lejana al ideal.

Listas de verificación

Una vez probados los métodos individuales de una clase, se puede pasar la prueba de la clase como un todo. Esto significa ejecutar sus métodos en combinación o someter los objetos de la clase a eventos como una acción del ratón. Una vez más, es probable que un conjunto puramente aleatorio de combinaciones de métodos desperdicie tiempo y de todos modos deje huecos en la cobertura.

Pruebas orientadas a atributos

Las pruebas orientadas a atributos están diseñadas para enfocarse en un solo atributo y predecir los efectos de ejecutar los diferentes métodos en secuencia (como establecerSaldo(100); sumaSaldo(70); obtenerSaldo()). La secuencia se ejecuta y se verifica que el valor resultante del atributo se convierta en lo que se predijo. De hecho, las pruebas orientadas a atributos se centran en las pruebas de secuencia de métodos.

Pruebas de invariantes de clase

Las invariantes de clase son restricciones entre los atributos de la clase que deben permanecer ciertas en los lugares indicados de la ejecución. Las pruebas de invariantes de clase consisten en observar la veracidad de cada invariante, ejecutar las secuencias de métodos y verificar que las invariantes sean ciertas.

Algunos lenguajes (como el Eiffel) y algunos API (como SunTest) están equipados con funciones que prueban la validez de las aseveraciones, casi siempre llamadas assert(…). Una aseveración es una declaración que puede ser cierta o falsa; por ejemplo, "x==y". Con frecuencia las aseveraciones son invariantes.

Pruebas basadas en los Estados

Los objetos de las clases muchas veces se pueden interpretar como transiciones entre estados en respuesta a los eventos. Por lo tanto, deben probarse esas clases en términos de su estado. Una prueba

consistiría en la estimulación del sistema de manera que transite por una secuencia. También se introducirían eventos del sistema de manera que transite por esta secuencia. Igualmente se insertarían eventos que no se aplican a estados particulares, para asegurar que no afectan a la aplicación.

Para diseñar y ejecutar pruebas orientadas a objetos se requiere mucho tiempo, en especial debido al amplio uso de datos de entrada necesario. Se dispone de herramientas que registran las interacciones del usuario con el sistema y reproducen estas acciones. Además la verificación de aseveraciones puede incluirse en el código para verificar que el sistema está en el estado que se supone.

2.7 Integración

Debido a que las aplicaciones son complejas, deben construirse con partes que primero se desarrollan por separado. La "integración" se refiere a este proceso de ensamble. Se realizan varios tipos de pruebas en los ensambles parciales de la aplicación y en toda ella.

La etapa de integración del proceso en cascada suele producir sorpresas desagradables por incompatibilidad de las partes que se integran. Por esto, el Proceso de Desarrollo de Software Unificado, en particular, intenta evitar la integración "explosiva" mediante la integración continua con múltiples iteraciones. Las partes sombreadas de la siguiente figura muestran que en realidad la integración se lleva a cabo durante las iteraciones de construcción y transición.

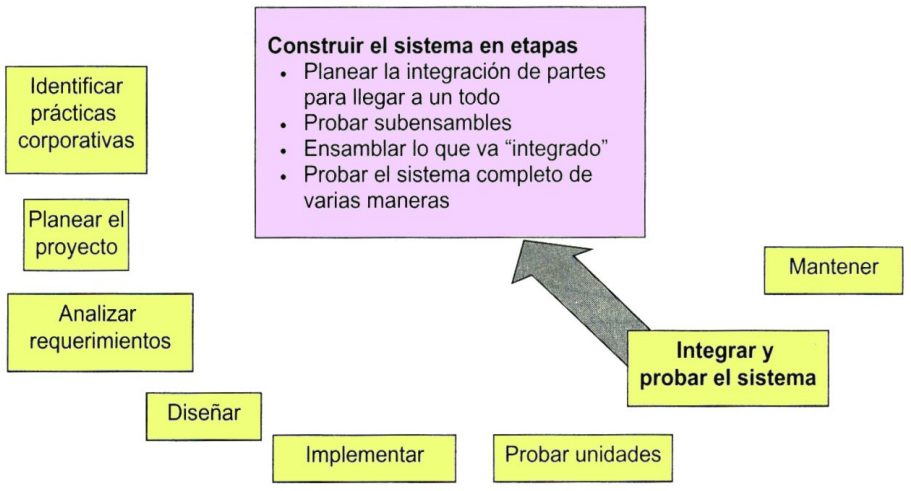


Fig. 103.- Mapa conceptual de Ingeniería de Software.

Es probable que ocurran pérdidas de información al ir de una etapa del proceso de desarrollo a la siguiente. Como Myers [My] señala, la figura siguiente ilustra los lugares en el proceso en cascada donde puede ocurrir una mala interpretación.

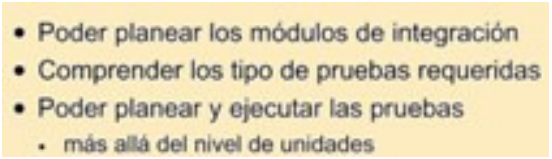


Fig. 104.- Metas de aprendizaje.

Por esta pérdida potencial de información se usan las pruebas y la integración continuas. Mientras que, sin duda, las pruebas de unidades tienen un gran valor dentro de su entorno final como parte de la aplicación, esto no sustituye las pruebas de unidades exhaustivas anteriores a la inclusión de cada parte.

2.7.1 Verificación, Validación y Pruebas del Sistema

La "verificación" pregunta si se está "construyendo bien". En otras palabras, ¿se construyen en la etapa presente justo aquellos artefactos que se especificaron en la etapa anterior? Cuando se aplica a la integración, la verificación se reduce a confirmar que se están uniendo justo las componentes que se planeó ensamblar, justo en la forma que se planeó ensamblarlas. Esa verificación se puede realizar mediante la inspección de los productos de la integración.

La "validación" pregunta si se "construye lo correcto". En otras palabras, ¿se satisfacen los requerimientos según se establecen en el ERS? En la etapa de integración, esto se realiza mediante las pruebas del sistema.

Al completar la construcción, una iteración, o la aplicación completa, las pruebas exhaustivas requieren que primero se revisen las pruebas de unidades de las funciones (métodos) y módulos (clases o paquetes). Sin embargo, ahora se prueban en el contexto, no asiladas. Se requieren menos controladores, que llevan a menos complicaciones y defectos. Si ésta es la construcción final, de hecho no se necesitan controladores. La diferencia entre las pruebas de unidades aisladas y las realizadas en el contexto se ilustra en la figura siguiente, donde las unidades son funciones.

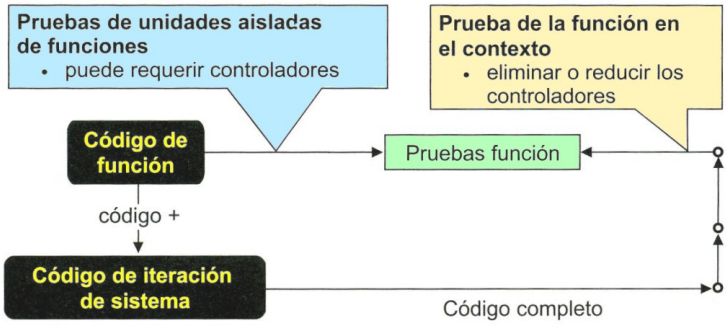


Fig. 105.- Pruebas de Unidades en el contexto.

La siguiente figura muestra el flujo de artefactos (más que nada documentos y código) entre las etapas del proyecto y entre los diferentes tipos de pruebas. Las pruebas de funciones y módulos se ejecutan de dos maneras, la primera vez se aíslan como pruebas de unidades. La segunda, en el contexto de toda la aplicación. Por eso se numeran dos veces.

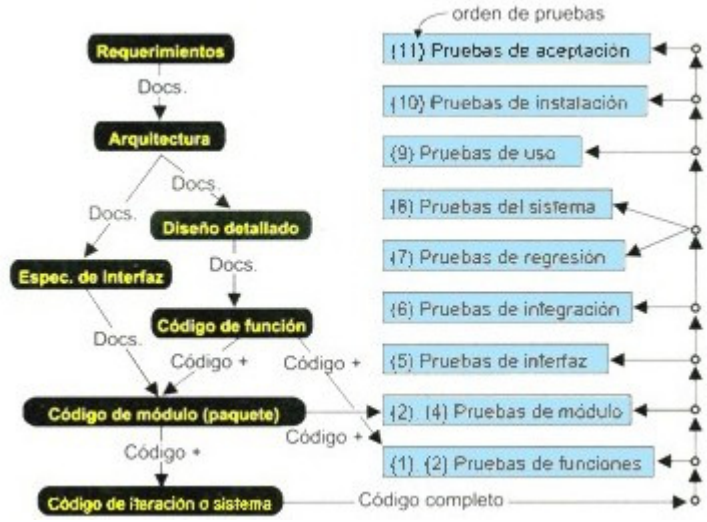


Fig. 106.- Visión general de pruebas: flujo de Artefactos (según Myers [My]).

La figura anterior muestra qué documento(s) de las diferentes pruebas se prueban de nuevo. Hay que recordar que la validación es el proceso de asegurar que se construye lo adecuado y por lo tanto, las pruebas contra los requerimientos originales afectan. Las otras pruebas, el proceso de verificación, aseguran que la aplicación se construye de manera deseada. Por ejemplo, las pruebas de interfaz verifican que la implementación es un reflejo fiel de las interfaces proyectadas.

Una vez hecha la integración total o parcial del código para el sistema, es posible probar las partes en el contexto de todo el sistema en lugar de aisladas. Para enfocar las pruebas en las partes designadas se debe idear una entrada apropiada.

- Deben crearse controladores para realizar las pruebas de unidades de funciones y clases, con la posibilidad de errores innecesarios y cobertura incompleta. Si no se puede dejar el código, por espacio o por razones de organización, este software se puede dejar a un lado disponible para uso futuro. Otra alternativa es incluirlos o excluirlos mediante un precompilador para conmutarlos (incluir/excluir código de pruebas de unidades).
- De manera similar, es posible probar de nuevo otros módulos (por ejemplo, paquetes) en el contexto del sistema.
- Las pruebas de interfaz confirman la validez de las interfaces entre los módulos.
- El propósito de las pruebas de regresión es verificar que las adiciones al sistema no hayan

degradado las capacidades preexistentes. En otras palabras, se realiza una prueba de regresión contra los requerimientos que ya se satisfacían, antes de la adición de nuevas capacidades. Sólo cuando un artefacto pasa la prueba de regresión, está listo para que se pruebe la operación del código agregado.

- La prueba de integración se realiza sobre un sistema parcialmente construido para verificar que el resultado de integrar software adicional (como clases) opera como se planeó.
- La prueba del sistema se realiza en toda la aplicación o en las versiones designadas.

  Las pruebas del sistema y de integración se realizan contra la arquitectura. En otras palabras, verifican que la arquitectura se haya seguido y que funcione como se planeó.

  Las pruebas del sistema también validan los requerimientos, tanto funcionales como no funcionales. Los requerimientos funcionales incluyen los requerimientos de desempeño como velocidad de ejecución y uso de almacenamiento.
- Las pruebas de uso validan la aceptación de la aplicación por los usuarios finales.
- La prueba de instalación se hace con la aplicación instalada en la plataforma dada.
- El cliente hace las pruebas de aceptación para validar la aceptación de la aplicación.

2.7.2 Proceso de Integración

La siguiente figura ilustra un proceso de Integración posible para la primera "iteración" de la construcción de un puente suspendido (versión de un solo nivel), así como otro para la segunda "iteración" (versión de nivel doble). Cada iteración es una etapa coherente del desarrollo. Se planea una secuencia cuidadosa de actividades llamada "construcción", que completa la iteración.

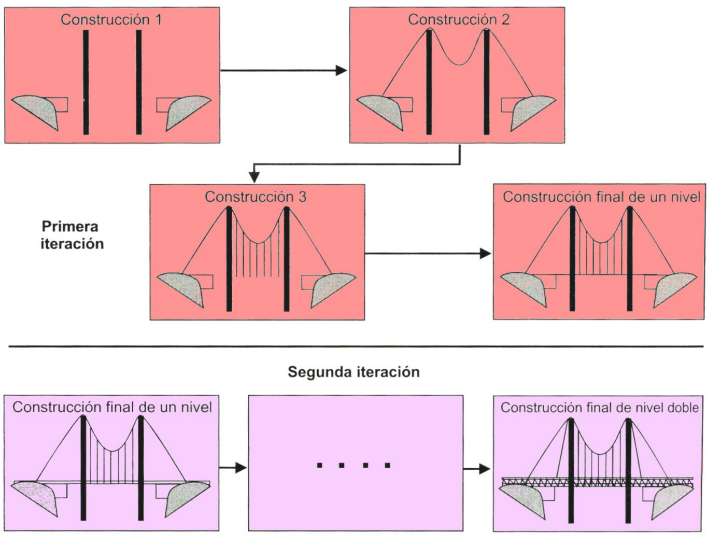


Fig. 107.- Proceso de Construcción dentro de las Iteraciones.

El tipo más sencillo de integración consiste en agregar nuevos elementos a la base (el código existente) en cada iteración alrededor de una espiral como se ilustra en la figura siguiente. La etapa de "implementación" consiste en la codificación de partes nuevas, seguida de su integración en la base.

Fig. 108.- Integración con Desarrollo en Espiral.

El proceso de integración para el Software no es menos un arte y una ciencia que el proceso de integración de los proyectos físicos y puede ser muy complejo. Igual que con el ejemplo del puente, cada iteración de software se construye en etapas. Esto se ilustra para el proceso de Software Unificado en la siguiente figura.

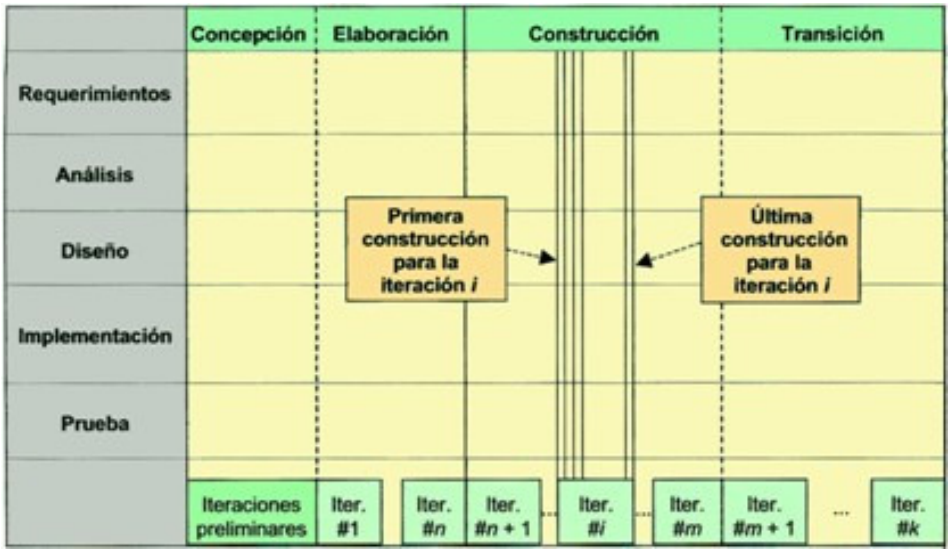


Fig. 109.- Relación de construcciones e iteraciones en el Proceso Unificado [Ja1].

En dicha figura se muestra los grupos de iteraciones (por ejemplo, las iteraciones de desarrollo) con la iteración (por ejemplo, la iteración i) dividida en varias construcciones. Esta organización extensa es relevante para los grandes proyectos.

Cuando se desarrolla la arquitectura, una consideración importante es la facilidad con que se pueden integrar las partes. Sin embargo, a diferencia de las aplicaciones físicas, rara vez es factible completar los módulos individuales de software antes de la integración. Una razón es que los módulos de Software típicos sirven a varios clientes, mientras que los módulos físicos sirven a un número muy limitado de "clientes", con frecuencia sólo uno. Por otro lado, conforme los requerimientos de software se comprenden mejor, los nuevos clientes de cada módulo se vuelven palpables. Así, como se ilustra en la siguiente figura, las construcciones de software a menudo deben integrar unidades parcialmente construidas, como en la secuencia "típica" en lugar de la secuencia "orientada a objetos".

Aunque el proceso de construcción típico tiene la desventaja de trabajar con unidades incompletas, posee la ventaja de ejercer la integración antes en el proceso de desarrollo. Esto ayuda a eliminar los riesgos al evitar la integración "explosiva".

Las dificultades de integrar las aplicaciones resaltan la importancia de diseñar unidades (como clases y paquetes) para centrarse en el propósito lo más posible y disminuir sus interfaces mutuas todo lo que se pueda. Esto con la finalidad de cumplir con las metas de "alta cohesión" y "bajo acoplamiento".

La siguiente figura proporciona una manera de establecer una integración y construir un plan:

- Comprender la descomposición de la arquitectura.
  - Intentar hacer una arquitectura sencilla de integrar.
- Identificar las partes de la arquitectura que implementará cada iteración.
  - Construir clases de marcos de trabajo primero, o en paralelo.
  - Si es posible, integrar "continuamente".
  - Construir suficientes GUI para anclar las pruebas.
  - Documentar los requerimientos para cada iteración.
  - Intentar construir de abajo arriba al menos parte del tiempo.
    - Para que las partes estén disponibles cuando se requieran.
  - Intentar planear las iteraciones para eliminar los riesgos.
    - Los riesgos más altos primero.
  - Especificar las iteraciones y construir de manera que cada Caso de Uso se maneje por completo por una.
- Descomponer cada iteración en construcciones si es necesario.
- Planear las pruebas, revisar e inspeccionar el proceso.
- Refinar el programa para reflejar los resultados.

Fig. 110.- Una forma de planear la integración y las construcciones.

Realizar pruebas se simplifica con la incorporación de implementaciones de Casos de Uso completos en cada construcción en lugar de sólo partes de los Casos de Uso. Crear Casos de Uso relativamente pequeños desde el principio facilita su ajuste en las construcciones. Como las interfaces de usuario tendrán que construirse y probarse en algún momento, es preferible si se puede hacer pronto –completas o partes importantes- para que las pruebas de la aplicación que evoluciona se puedan realizar a través de ellas. La alternativa es construir interfaces temporales para usarlas durante las pruebas de integración.

Jacobson et al. Señalan que en general es más sencillo usar el desarrollo de construcciones de abajo arriba para planearlas. Este enfoque crea partes antes de que se usen para construir unidades más grandes. El proceso de abajo arriba se puede combinar de modo útil con la implementación de clases de marcos de trabajo que, sin embargo, es un proceso de arriba abajo.

2.7.3 Mapa conceptual típico del Proceso de Integración y Pruebas del Sistema

La figura siguiente muestra una secuencia típica de las acciones para integrar un sistema de software.

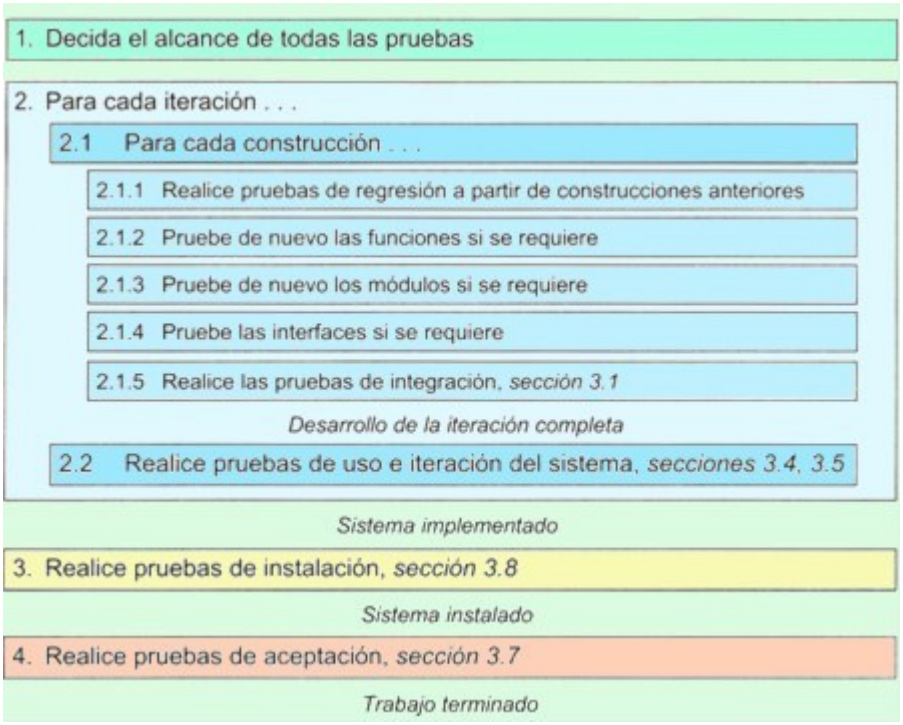


Fig. 111.- Mapa conceptual para la Integración y Pruebas del Sistema.

La figura anterior enumera los factores significativos para determinar el orden de integración. Este orden depende de las demandas del proyecto. Para proyectos riesgosos, la atención se centra en la integración de las partes como riesgo tan pronto como sea posible para calibrar la efectividad del diseño. Mostrar las partes específicas de la aplicación al cliente también dicta un orden de integración. De otra manera, se integrarían los módulos usados antes de los módulos que los usan, minimizando así el uso de código de control temporal.

2.7.4 Pruebas de Integración

La prueba de Integración verifica cada construcción e iteración de Integración. La siguiente figura, muestra una manera en que las pruebas de integración se pueden planear y llevar a cabo en conjunto con las pruebas unitarias y de regresión:

- Decida cuándo y dónde almacenar, reutilizar y codificar las pruebas de Integración.
  - Muestre esto en la programación de tiempos del proyecto.
- Ejecute tantas pruebas unitarias (de nuevo) como el tiempo permita.
  - Esta vez en el contexto de la construcción.
    - No se requieren controladores esta vez.
  - De prioridades según la mayor probabilidad de encontrar defectos.
- Realice pruebas de regresión.
  - Para asegurar que las capacidades existentes siguen funcionando.
- Asegure que los requerimientos de las construcciones están bien especificados.
- Ejecute los Casos de Uso que debe implementar la construcción.
  - Pruebe contra ERS.
- Ejecute las pruebas del sistema soportadas por esta construcción.

Fig. 112 .- Una manera de planear y ejecutar las pruebas de Integración.

Los Casos de Uso son una fuente ideal de casos de prueba de Integración. Como se mencionó, Jacobson et al [Ja] recomiendan acomodar cada Caso de Uso dentro de una construcción. La idea es que los Casos de Uso se construyan sobre los que ya están integrados para formar pruebas cada vez más representativas del uso de la aplicación. Esto se ilustra en la figura siguiente.

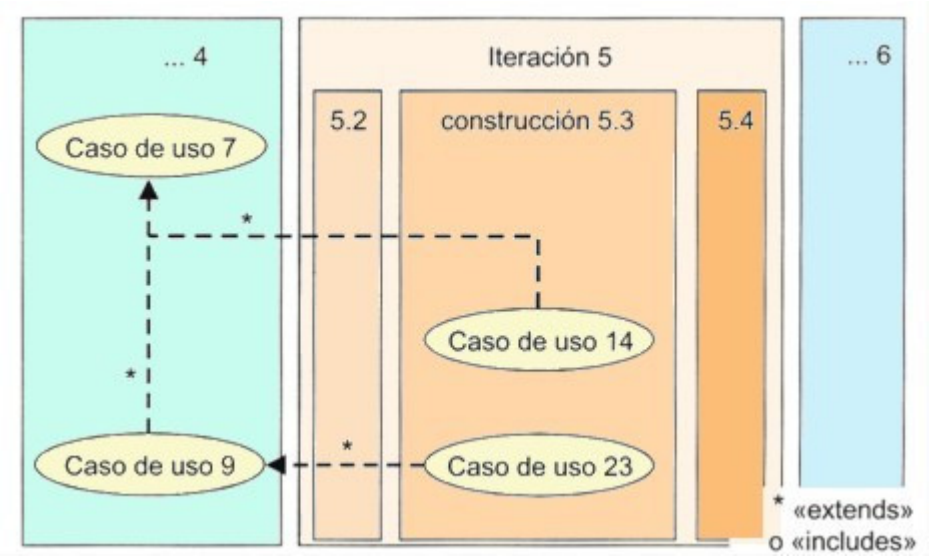


Fig. 113 .- Relación entre Casos de Uso, iteraciones y construcciones.

Se requiere un número grande de pruebas para validar una aplicación de manera adecuada y necesitan una organización metódica. Un estilo de organizar los casos de prueba es ponerlos en un paquete en clases especialmente creadas para las pruebas. Una clase, o quizá un paquete entero, se puede dedicar a probar toda la aplicación.

Por lo general, las construcciones consisten en el código de varios desarrolladores y es común encontrar muchos problemas cuando se integra el código para crear la construcción. Por ello, se intenta comenzar la integración y las pruebas de integración pronto en el proceso, para ejecutar el código en su contexto final.

Las pruebas de Integración se realizan mientras las construcciones avanzan. Estas pruebas suelen consistir en pruebas de regresión, con pruebas adicionales agregadas para validar las nuevas adiciones. No es práctico hacer pruebas de Integración formales completas, continuas. En consecuencia, las pruebas de Integración a una menor escala se aplican en periodos regulares y frecuentes; validan sólo que parezca que el sistema opera como debe. Estas pruebas se conocen como "pruebas de humo" y sólo dan seguridad a los programadores para seguir trabajando en la misma forma (si no aparecen problemas), o indican problemas que podrían ocasionar grandes demoras en la integración.

Un calendario de integración con frecuencia tiene la forma que describe la siguiente figura, que toma una aplicación bancaria por ejemplo. Conforme se acerca el fin de la tarea de construir los módulos, se integran en la base (es decir, se unen en el producto oficial en evolución) uno a la vez. En este caso, el proceso de integración tiene lugar entre las semanas 23 y 31.

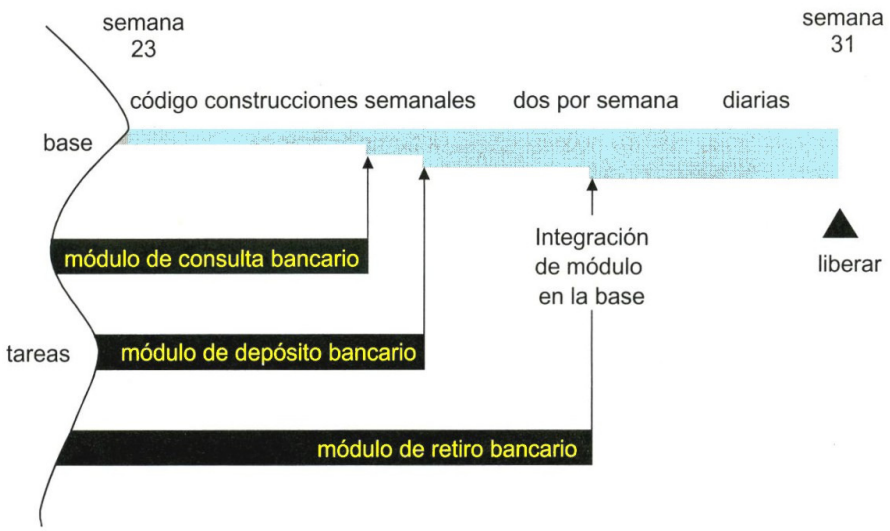


Fig. 114.- Programa de construcción de codificación final e Integración: ejemplo bancario.

El proceso de compilar y probar construcciones parciales a menudo se realiza durante la noche, cuando el desarrollo está congelado mientras se realizan las pruebas. Esto se muestra en la figura siguiente

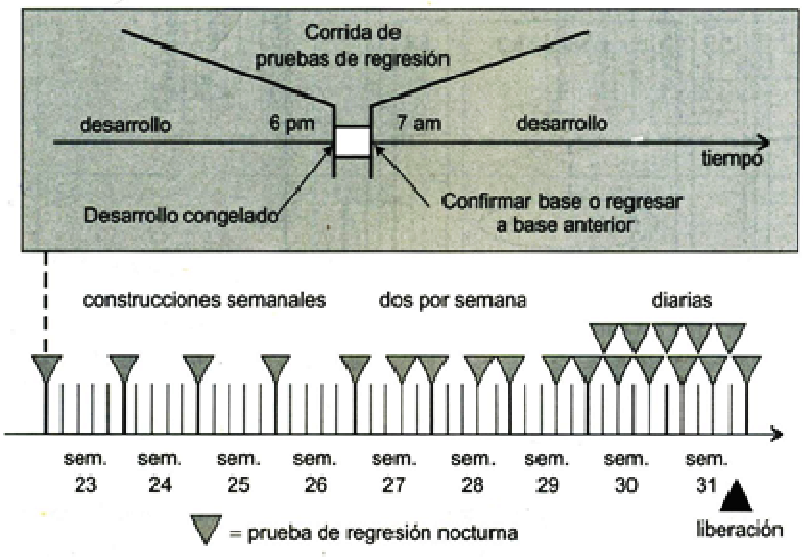


Fig. 115.- Proceso de Integración de código diario típico.

Conforme se acerca la fecha de liberación de la construcción o aplicación, la frecuencia de las pruebas de regresión aumenta hasta que se hace todos los días; casi siempre por la noche como se muestra en la figura anterior. Si la prueba de regresión indica que la funcionalidad existente todavía está presente, entonces el código integrado se convierte en parte de la base. Por otro lado, si la prueba de regresión muestra que el código agregado crea fallas en la funcionalidad preexistente, la decisión será regresar al código base que se tenía antes de integrar el nuevo material, es decir, "des"-integrarlo. Este tipo de programa de pruebas de integración y regresión diarias se ha usado, por ejemplo, en Microsoft, según lo informan Tucumano y Selby [Cu].

Pruebas de desarrolladores y artefactos

En esta sección se revisarán los artefactos involucrados en el proceso de pruebas de Integración, como lo sugiere el Proceso de Desarrollo de Software Unificado (USDP):

- **Modelo de Casos de Uso:** Conjunto de Casos de Uso que describen el uso típico de la aplicación y los Diagramas de Secuencia que los describen con detalle.
- **Casos de prueba:** Los datos de entrada para cada prueba.
- **Procedimientos de prueba:** La manera en que se deben establecer y ejecutar las pruebas y evaluar los resultados. Estos pueden ser los procedimientos del manual o los que usan herramientas de automatización de pruebas.
- **Evaluación de pruebas:** Resumen, detalles y efectos de los defectos encontrados.
- **Plan de pruebas:** Plan global para realizar las pruebas, incluye el orden.
- **Componentes de las pruebas:** Código fuente para las pruebas en sí y para el código de la aplicación que se debe probar.
- **Defectos:** Informes de los defectos descubiertos como resultado de este proceso, clasificado por la severidad y tipo.

El proceso de pruebas de Integración USDP implica los papeles del Ingeniero de Pruebas, el Ingeniero de Componentes y el que prueba el sistema. Sus responsabilidades se muestran en la figura siguiente.

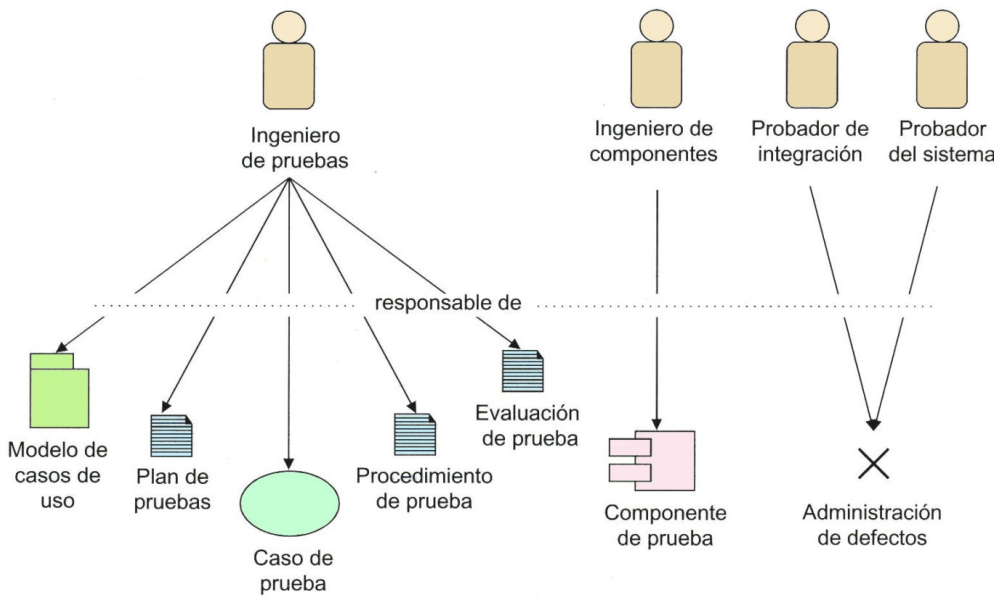


Fig. 116.- Artefactos y papeles para las pruebas de Integración [Ja1]

Prueba de interfaz

Muchas fallas de aplicaciones se deben a problemas con las interfaces entre las componentes. En el caso de la ejecución de un proyecto, algunos grupos encuentran más sencillo comunicarse entre ellos que con otros grupos, de manera que es fácil interpretar mal las interfaces de diseño y programación. La "sección interfaces" del documento de diseño de software (DDS) es la "Biblia" para las interfaces de la aplicación. Una vez desarrollados los módulos, se pueden probar las interfaces. Esto se hace generando tráfico entre las interfaces, casi siempre en la forma de llamadas de las funciones.

Prueba del Sistema

La Prueba del Sistema es la culminación de las pruebas de Integración. Consiste en pruebas de caja negra que validan la aplicación completa contra sus requerimientos. Siempre que es posible, las Pruebas del Sistema se realizan mientras la aplicación se ejecuta en su entrono requerido. Sin embargo, en ocasiones habrá que conformarse con pruebas de ejecuciones del sistema en un entorno o configuración que no es equivalente a la del cliente. Por ejemplo, no se consideraría necesario probar partes pequeñas de programas (applets) en todo tipo de computadoras personales. Por otro lado, los applets deben probarse en todas las versiones importantes de todos los exploradores principales (de Internet).

Dado que las pruebas del sistema aseguran que los requerimientos se cumplen, deben validar de modo sistemático cada requerimiento. Se requiere una escritura de pruebas considerable para forzar la demostración de cada requerimiento. En este punto también se validan los Casos de Uso.

El Proceso de Desarrollo de Software Unificado intenta organizar la mayoría de los requerimientos mediante Casos de Uso, en cuyo caso las pruebas son más sencillas si se compara con la prueba de los requerimientos atómicos individuales.

La tabla siguiente enumera casi todos los tipos de pruebas del sistema:

- Volumen.
  - Somete al producto a la entrada de grandes cantidades de datos.
- Utilidad.
  - Mide la reacción del usuario (por ejemplo, calificación de 0 a 10).
- Desempeño.
  - Mide la velocidad para varias circunstancias.
- Configurabilidad.
  - Configura los distintos elementos de Hardware/Software.
    - Por ejemplo, mide el tiempo de preparación.
- Compatibilidad.
  - Con otras aplicaciones designadas.
    - Por ejemplo mide el tiempo de adaptación.
- Confiabilidad/disponibilidad.
  - Mide el tiempo de operación en tiempos largos.
- Seguridad.
  - Sujeta a intentos comprometedores.
    - Por ejemplo, mide el tiempo promedio para entrar (romper) al sistema.
- Uso de recursos.
  - Mide el uso de RAM, espacio en disco, etcétera.
- Aptitud de instalación.
  - Mide el tiempo de instalación.
- Recuperabilidad.
  - Fuerza actividades que desactivan la aplicación.
    - Mide el tiempo de recuperación.
- Funcionalidad.
  - Da servicio a las aplicaciones en diferentes circunstancias.
    - Mide el tiempo de servicio.
- Carga/tensión.
  - Sujeta a datos extremos y tráfico de eventos.

Fig. 117 .- Tipos de Pruebas de Sistemas.

La confiabilidad/disponibilidad se determina mediante medidas como el tiempo medio entre fallas (MTBF,

Mean Time Between Failures). Para obtener el MTBF primero se establece una definición de "falla"; por ejemplo, la deshabilitación total de la aplicación.

En realidad se pueden definir varios niveles de falla. Para calcular el MTBF, un probador inicia la aplicación, registra la hora y ejecuta la aplicación con un escenario aleatorio (idealmente), hasta que el sistema falla. Registra la hora y calcula el tiempo transcurrido. Este proceso se realiza muchas veces. El MTBF es el promedio de los tiempos obtenidos.

El termino funcionalidad mencionado anteriormente en la figura, se refiere a la facilidad o dificultad con que la aplicación se mantiene operativa. Por ejemplo, una aplicación de Sistema Experto se apoya en su base de conocimientos, que debe ser capaz de modificarse con facilidad.

Prueba de Utilidad

Una buena interfaz puede mejorar mucho el valor de la aplicación. La Prueba de Utilidad valida la aceptación de la aplicación por los usuarios.

Prueba para los requerimientos de interfaz de usuario

La tarea principal de las Pruebas de Utilidad es asegurar que la aplicación satisface los requerimientos establecidos. Además puede requerirse un tiempo específico.

Una manera de hacer esto es cuantificar el nivel de satisfacción que informan los usuarios al utilizar la aplicación.

Medidas de Utilidad

Los criterios de uso se especifican de antemano. Por ejemplo, es posible que se requiera que una muestra aleatoria de 30 usuarios de la aplicación de finanzas del hogar califiquen la aplicación como muestra la siguiente tabla.

Característica	Promedio (de 10)
Facilidad de visión	8.5
Facilidad de uso	9.5

Tabla 13.- Ejemplo de valores métricos para la utilidad.

La estadística determina el tamaño de muestra adecuado que depende del tamaño de la base de clientes pronosticada y de la probabilidad deseada de las conclusiones erróneas.

En la práctica, los datos de utilidad serían más detallados que los de la tabla anterior. Por ejemplo, Kit [Ki] enumera los criterios de la figura siguiente como esenciales para estas pruebas.

- Accesibilidad.
  - Facilidad con la que entran, navegan y salen los usuarios.
    - Por ejemplo, medida por el tiempo promedio que toma…
- Rapidez de respuesta.
  - Qué tan rápido permite la aplicación al usuario lograr sus metas especificadas.
    - Por ejemplo, medida del tiempo promedio que toma.
- Eficiencia.
  - Que tan pequeños son los pasos requeridos para la funcionalidad elegida.
    - Por ejemplo, también medida por el tiempo mínimo de una muestra de usuarios.
- Comprensión.
  - La facilidad con que se entiende y usa el producto mediante la documentación y la ayuda.
    - Por ejemplo, medida de tiempo que toman las investigaciones estándar.

Fig. 118 .- Atributos clave para las Pruebas de Utilidad.

Además de estas cualidades, se necesitarán medidas específicas de la aplicación como:

¿Qué tan fácil diría que es entrar a una forma de informe de accidentes estándar (en una escala de 1 a 10)?

Al diseñar los cuestionarios, el reto es obtener los datos que permitan a los ingenieros enfocar el desarrollo a remediar las deficiencias más serias, sin exceder los límites de tiempo y paciencia del usuario. Puede ser costoso reunir los datos de utilidad porque con frecuencia los usuarios esperan una compensación por el tiempo y el problema de proporcionar retroalimentación.

Prueba de Regresión

Cuando la aplicación crece, las Pruebas del Sistema adquieren un significado especial. Esto es en especial notorio cuando se hace un cambio en una aplicación grande y los desarrolladores necesitan validar el hecho de que el cambio no haya afectado la funcionalidad existente. La primera pregunta después de integrar el cambio casi siempre es la siguiente: "¿es este producto el mismo que el anterior, con la funcionalidad agregada?", en general, la reinspección no es práctica, por lo que una manera práctica importante de ayudar a contestar esta pregunta es verificar que el sistema continúe pasando el mismo conjunto de pruebas diseñado antes de hacer los cambios. Este proceso de verificación se llama Pruebas de Regresión.

Las Pruebas de Regresión se llevan a cabo con frecuencia. Si el tiempo no permite realizar las pruebas completas, entonces se seleccionan aquéllas con mayor probabilidad de fallar debido a los cambios.

Pruebas de Aceptación

La organización desarrolladora y la del cliente son las partes de un contrato. Cuando termina un trabajo, un desarrollador inteligente obtiene una declaración definitiva del cliente que establece que de hecho la aplicación está entregada. Las Pruebas de Aceptación están diseñadas para asegurar al cliente que se construyó la aplicación estipulada. Las Pruebas de Aceptación tal vez no sean muy diferentes de las Pruebas del Sistema que diseña el desarrollador, pero la organización del cliente es el testigo y se ejecutan en la plataforma que van a operar.

A menudo los clientes requieren hacer pagos sobre el avance basados en entregas intermedias, o implementaciones y diseño parciales y requieren la Prueba de Aceptación.

Pruebas de Instalación

El hecho de que se haya probado una aplicación en el entorno propio no asegura que trabajará de manera apropiada en el entorno del cliente porque existen muchas posibilidades de error al cambiar de entorno. La Prueba de Instalación consiste en probar la aplicación en su configuración de hardware final. Esto implica instalar la aplicación en su entorno meta, después ejecutar el conjunto de Pruebas del Sistema. Para las aflicciones comprimidas, las Pruebas de Instalación consisten en ejecutar la aplicación en plataformas que tipificarían los entornos de los clientes.

2.7.5 Documentación de Integración y Pruebas

Estándares para la documentación de pruebas.

El estándar de ANSI/IEEE para la documentación de pruebas se muestra en la siguiente figura .

1. **Introducción**
2. **Plan de pruebas**
   elementos a probar, alcance, enfoque, recursos, programa de tiempos, personal
3. **Diseño de prueba**
   elementos a probar, enfoque, plan detallado
4. **Casos de prueba**
   establece los datos de entrada y los eventos
5. **Procedimientos de prueba**
   pasos para preparar y ejecutar los casos de prueba
6. **Informe de transmisión de elementos de prueba**
   elemento a probar, localización física de resultados, persona responsable para transmitir
7. **Archivo de prueba**
   registro cronológico, localización física de prueba, nombre del probador
8. **Informe de incidentes de prueba**
   documentación de cualquier evento que ocurre durante la prueba que requiere mayor investigación
9. **Informe de resumen de la prueba**
   resume lo anterior

Fig. 119.- Documentación de Prueba de Software ANSI/IEEE 829-2983 (confirmado en 1991) Copyright © 1983 IEEE.

- La sección de introducción explica el contexto de las pruebas y su filosofía global. Por ejemplo, si la aplicación controla el equipo de una sala de urgencias, entonces ahí es donde se explicaría el enfoque global de las pruebas de los modelos, que llevaría a realizar esas pruebas en esas condiciones.
- El plan de pruebas explica cómo deben organizarse personas, software y equipo para realizar el trabajo de prueba. Por ejemplo, "Juan probará el módulo de tiempos entre las semanas 30 y 33; Pedro probará el módulo del monitor cardiaco entre las semanas 26 y 30. Eduardo probará la integración de estas pruebas ente las semanas 31 y 33".
- La prueba de diseño proporciona el siguiente nivel de detalle más allá del plan de pruebas. Desglosa los elementos de software involucrados, describe el orden en el que deben probarse y nombra los casos de prueba que deben aplicarse. Por ejemplo, "Juan probará el módulo de tiempos aislado entre las semanas 30 y 33 usando el procedimiento 892 y el controlador 8910; Pedro probará el módulo del monitor cardiaco entre las semanas 26 y 30, usando el procedimiento de prueba 555 y el controlador 3024; Eduardo probará la construcción que integra los dos módulos (construcción 7) usando…".
- Los casos de prueba consisten en conjuntos de datos y el estímulo preciso que debe aplicarse para llevar a cabo el diseño de la prueba. Por ejemplo, el módulo del monitor cardiaco debe operar con el archivo 892, que proporciona datos específicos de pacientes en tiempos específicos. Debe describirse justo en dónde se localiza este archivo.
- Los procedimientos de prueba son los pasos detallados completos para ejecutar este plan de prueba. Incluyen todos los procedimientos de preparación, nombres de los archivos de casos de prueba e informes. Por ejemplo, el procedimiento de prueba 892 puede ser el siguiente:
  - Compilar el módulo de tiempos junto con el controlador 9810.
  - Establecer la trayectoria del directorio para….
  - Cargar el contenido del archivo 672 en un archivo con nombre de entrada en el mismo directorio que el código objeto.
  - Ejecutar el código con las siguientes opciones…
  - En la ventana que aparece, introducir "Jones" en el cuadro de texto del nombre…

  La razón para este nivel de detalle es que si una prueba indica un defecto, es importante conocer las circunstancias exactas en las que ocurre el defecto. Sin la documentación de los pasos detallados, las pruebas no se pueden reproducir de manera confiable y el defecto puede no ser reproducible. En ese caso, es difícil o imposible repararlo.
- El informe de transmisión de elementos de prueba resume qué pruebas se realizaron, quién las hizo, qué versiones se usaron, etcétera.
- El registro de la prueba es un recuento detallado de lo que transpiró durante la prueba. Esto puede ser importante al tratar de reconstruir las situaciones cuando la prueba falla.
- El informe de incidentes de la prueba detalla las ocurrencias notorias que tuvieron lugar durante la prueba. Los ejemplos son desviaciones de la manera normal de operar la aplicación y los errores cometidos en el proceso de prueba.

Organización de la documentación de Integración y Pruebas

Un buen documento para describir cómo deben ensamblarse las partes de una aplicación es el documento de administración de la configuración (PACS en términos del IEEE). Esto se ilustra en la figura siguiente. Esta organización de documento indica que el PAPS describe el PACS y quién es el responsable. El PACS describe los procedimientos específicos para mantener (almacenar, coordinar, etcétera) las diferentes versiones de los distintos documentos, incluido el PAPS. También especifica exactamente dónde se localizan estos documentos. La última especificación crece y se puede describir mejor en un apéndice del PACS. En particular el PACS y sus apéndices deben hacer referencia a la documentación de las pruebas (el DPS-Documentación de Pruebas de Software- en términos del IEEE), para tener un rastreo cuidadoso de pruebas ejecutadas, casos de prueba correspondientes, procedimientos, planes, etcétera y las versiones reales del código que prueban.

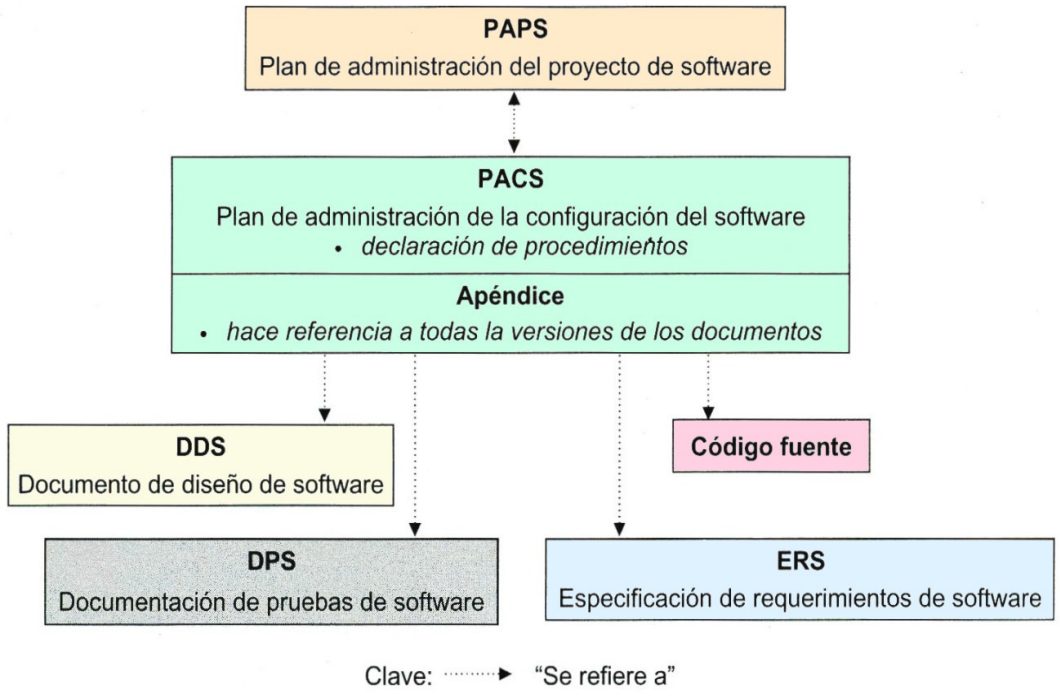


Fig. 120.- Documentación de Pruebas de Software en el contexto.

La anterior figura ilustra las relaciones entre los diferentes documentos de prueba y sus relaciones con la documentación existente. La documentación de pruebas para la aplicación, el DPS incluye todos los tipos de pruebas descritos en la aplicación. También puede incluir una unidad de pruebas, que depende del grado en el que la prueba unitaria se documentó.

Los artefactos de las diferentes pruebas se reutilizan, como indican las líneas punteadas de la figura anterior. Por ejemplo, las pruebas de construcción usan planes de pruebas, diseño, casos y procedimientos desarrollados para probar las construcciones anteriores. Las pruebas del sistema usan artefactos (casos de prueba, etcétera) que se desarrollaron para probar la construcción final. Las pruebas de aceptación usan artefactos de las Pruebas del Sistema y las Pruebas de Instalación usan artefactos de las Pruebas de Aceptación.

2.7.6 Iteraciones de Transición

Una vez integrada la aplicación, se requieren varias actividades antes de poder liberarla. Los pasos se resumen en Jacobson et al. En las iteraciones de transición de su Proceso de Desarrollo de Software Unificado. Las metas de estas iteraciones se describen en la figura siguiente. Esta figura también resume las cantidades relativas de requerimientos, análisis, etcétera, necesarios para estas iteraciones durante la etapa de transición.

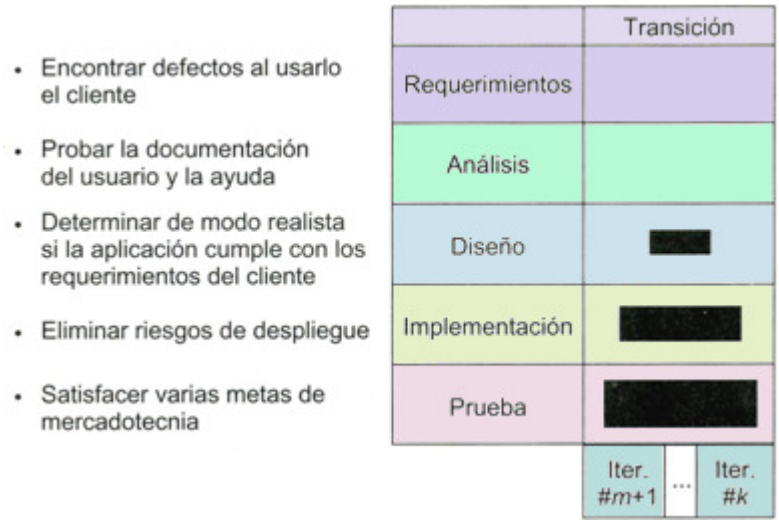


Fig. 121.- Metas de las iteraciones de Transición.

Versiones Alfa y Beta

En muchos casos, los usuarios prospectivos internos, igual que los clientes, están dispuestos a participar en el proceso de Pruebas del Sistema. Este proceso está controlado por las versiones Alfa y las versiones Beta liberadas, como se ilustra en la figura siguiente.

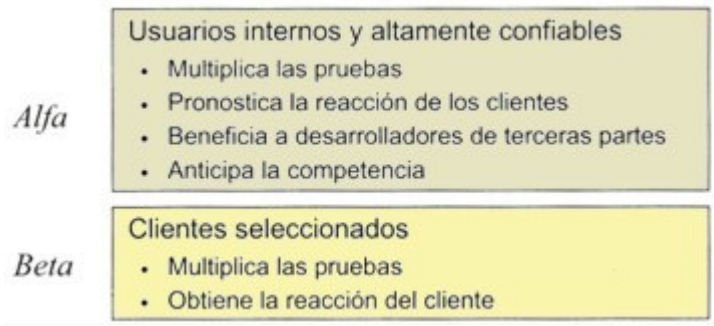


Fig. 122.- Versiones Alfa y Beta.

Las versiones Alfa se dan a los usuarios internos o a un grupo selecto y confiable de usuarios externos para los primeros usos antes de la liberación. El propósito de las versiones Alfa es proporcionar retroalimentación a la organización de desarrollo e información de defectos de un grupo más grande que el de los probadores, sin afectar la reputación del producto no liberado. Después de repartir las versiones Alfa, se liberan las versiones Beta.

Las versiones Beta se dan a parte de la comunidad de clientes con el entendimiento de que informarán acerca de los errores encontrados. Además, las versiones Alfa y Beta se usan para convencer a los clientes potenciales de que en realidad se trata de un producto que respalda las promesas de los proveedores. En ocasiones, la distribución de las aplicaciones preliberadas es una técnica de negocios estratégica usada para desanimar a los clientes en la compra de productos que compiten e inducirlos a esperar nuevas versiones de la aplicación que se prueba.

Una motivación importante para ser probador de Alfas y Betas es obtener conocimientos avanzados del producto. Los desarrolladores obtienen información de la aplicación (casi siempre de las API) para poder empezar a desarrollar las aplicaciones que las usan. Los usuarios comienzan a tomar decisiones para comprar la aplicación.

Mapa conceptual de las iteraciones de Transición

La figura siguiente muestra los pasos comunes para llevar a cabo las iteraciones de transición (finales). Los "criterios de detención" son las condiciones en las cuales el producto debe liberarse para las pruebas de aceptación. Si estas condiciones no se determinan de antemano, las pruebas suelen seguir hasta que se acaba el tiempo. Este no es el uso más efectivo del tiempo de pruebas.

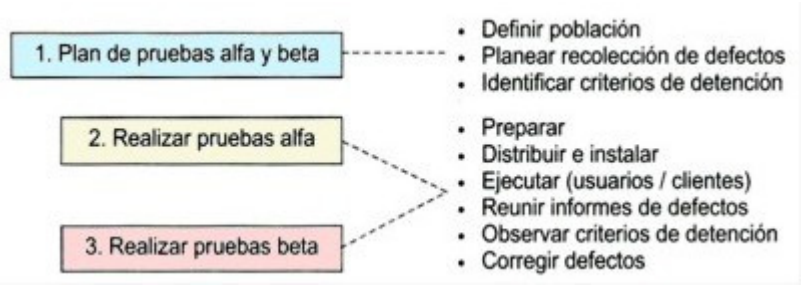


Fig. 123.- Mapa conceptual para las iteraciones de Transición.

Un ejemplo de un criterio de detención es "un máximo de dos defectos de nivel medio o bajo encontrados por semana en las pruebas beta". Kit [Kit] clasificó los criterios de detención como se sugiere en la figura siguiente.

La figura siguiente, menciona los "defectos restantes"; pero ¿cómo se puede estimar el número de defectos restantes? Un método es "plantar". Consiste en insertar una variedad de defectos en la aplicación, después determinar el porcentaje de ellos que detectan los probadores independientes dentro de un periodo dado. Este porcentaje se usa después para estimar el número de defectos que quedan.

Por ejemplo si se encuentran 3 de 50 fallas plantadas durante una prueba dada, se pueden estimar 47/3= 15.67 fallas no detectadas por cada falla que sí se detecta. Así, si se encuentra un total de 100 fallas no plantadas durante el mismo tipo de prueba, entonces existen alrededor de 100X15.67=1567 fallas restantes no detectadas en el sistema.

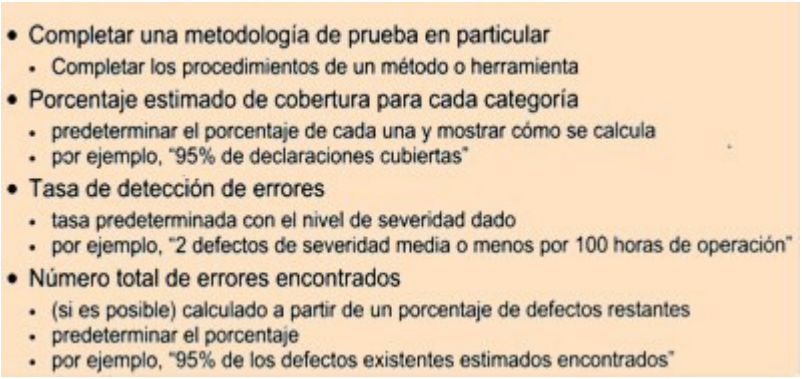


Fig. 124.- Criterios de Detención (Kit [Ki]).

Al aplicar algunos criterios, los proyectos pueden usar gráficas como las de la figura siguiente, a fin de determinar cuándo liberar el producto. En este ejemplo, se aplican tres criterios de detención. El criterio de

detención para la tasa de detección de errores es "cuando mucho siete defectos encontrados por cada mil horas de pruebas por semana durante al menos cuatro semanas consecutivas".

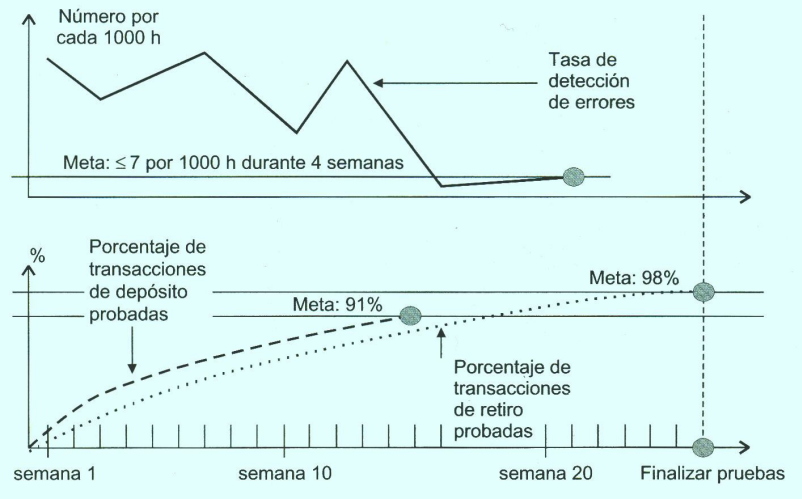


Fig. 125.- Criterios de detención: representación gráfica.

(Para lograr 1000 horas de pruebas en una semana se ejecutan en varias copias de la aplicación en paralelo). Como ejemplo se toma una aplicación bancaria y prueba transacciones de "depósito" y "retiro" que se cuentan por separado. En el caso que se ilustra en la figura, el último criterio que debe cumplirse es el "porcentaje de transacciones de retiro probadas", por lo que el producto no se libera antes de esto (semana 26).

2.7.7 Calidad en Integración, Verificación y Validación

Metas de calidad

Los buenos planes de Integración se piensan con cuidado y las pruebas efectivas del sistema y la Integración deben ser exhaustivas. Las métricas como la siguiente promueven estas cualidades.

Métricas para la Integración y Pruebas del Sistema

Las siguientes métricas se tomaron del diccionario estándar de medidas del IEEE 982.1-1988 [IEEE 982].

- **IEEE 1.** Densidad de falla = [número de fallas distintas encontradas en pruebas]/[número de líneas de código].
- **IEEE 2.** Densidad de defectos = [número de fallas distintas encontradas por inspección]/[número de líneas de código].

- **IEEE 5.** Cobertura de pruebas de defectos = [número de requerimientos funcionales probados]/[número total de requerimientos].
- **IEEE 10.** Indice de madurez del software= $[M-F_a - F_c- F_d] /M$, donde

  M = número de partes en la base actual.

  F_a = número de partes en la base actual agregadas respecto a la base anterior.

  F_c = número de partes en la base actual cambiadas respecto a la base anterior.

  F_d = número de partes eliminadas respecto a la base anterior.

  Las "partes" pueden ser funcionales, clases, paquetes, módulos u otros seleccionados de manera consistente de principio a fin. Las aplicaciones maduras dan un índice de madurez cercano a 1, lo cual significa que el número de partes afectadas es pequeño comparado con el total.
- **IEEE 18.** La confiabilidad de la prueba es la probabilidad de que k corridas aleatorias del sistema producirán resultados correctos. Esto se estima ejecutando N corridas y contando el número S de éxitos. La probabilidad de éxito es entonces S/N y la probabilidad de ejecutar k corridas con éxito es el producto de la probabilidad de cada éxito: [S/N][S/N]…[S/N] o $[S/N]^k$. Los datos para cada corrida se eligen de manera aleatoria e independiente de la selección de la prueba anterior.
- **IEEE 20.** Tiempo medio para descubrir las siguientes k fallas. Esta cantidad se estima de manera análoga a la confiabilidad de la corrida (punto 18).
- **IEEE 21.** Nivel de pureza del software. Esta medida es una estimación de la falta de fallas en el programa durante las etapas operativas.
- **IEEE 22.** Estima el número de fallas restantes (plantadas). Esta estimación se obtiene "plantando" fallas en la aplicación de la manera más aleatoria posible, digamos N de ellas. Si s es el número de fallas plantadas y f es el número de fallas no plantadas encontradas en el mismo periodo, entonces la estimación es f*N/s
- **IEEE 24.** Cobertura de las pruebas. Esto mide si las pruebas realizadas están completas (es decir, la fracción del trabajo implementado multiplicada por la fracción de pruebas implementadas). La fórmula es:

  TC (porcentaje) = [[núm. requerimientos implementados] / [núm. requerimientos]] * [[núm. programas primitivos probados] / [núm. total de programas primitivos]] * 100 (convierte a porcentaje).

  Los programas primitivos son unidades del programa que se pueden probar e incluyen métodos, clases y paquetes.
- **IEEE 30.** Tiempo medio hasta la falla (MTTF). Esto se mide con el registro de los tiempos entre todos los pares de fallas sucesivos observados y el promedio de estos tiempos. El "tiempo" a menudo es tiempo transcurrido y no de CPU.
- **IEEE 36.** Prueba de exactitud. Esta prueba estima la confiabilidad del proceso de pruebas y es un producto derivado de la prueba 22 descrita.

  Prueba de exactitud = N_f / N, donde N es el número de fallas plantadas y N_f es el número de fallas plantadas encontradas durante el periodo de pruebas designado.

Inspección de Integración y Pruebas del Sistema

Varios aspectos de la integración son sensibles al proceso de inspección. Incluyen las partes del PACS que se relacionan con la secuencia de integración y los diferentes planes de pruebas (como el Plan de Integración y el Plan de Pruebas del Sistema). Un ejemplo de un defecto en el Plan de Integración es la ausencia de los elementos necesarios (módulos, clases, etcétera) de una etapa de integración dada; esto es, un elemento que debe estar presente para formar una construcción o Caso de Uso que se pueda probar. Un ejemplo de un defecto en la integración es la ausencia de un paso de prueba que es parte del Caso de Uso correspondiente.

La secuencia de construcción y sus pruebas puede ser compleja; de ahí el beneficio de las inspecciones donde varias mentes analizan los planes.

Participación de Aseguramiento de la Calidad (QA) en la Integración y Pruebas del Sistema.

El final de un proyecto es un buen momento para evaluar el proceso usado y preparar las mejoras. Una organización típica aspira a pasar al siguiente nivel de CMM. La tabla siguiente recuerda los cinco niveles de CMM.

1 Inicial	No definido, ad hoc
2 Repetible	Costo de rastreo, programa de tiempos, funcionalidad después del hecho
3 Definido	Documentado, estandarizado, se puede adaptar
4 Administrado	Mediciones detalladas; control
5 Optimizado	Mejora continua cuantificable del proceso

Tabla 14.- Modelo de Madurez de Capacidades.

Como ejemplo, suponga que nuestra organización está en el nivel 3 e intenta lograr el nivel 4. Entonces, el equipo habrá intentado hacer mediciones detalladas y controlar el proyecto. Un posmortem podría tomar la forma que se muestra en la siguiente tabla.

	Colección de métricas	Controlabilidad	Aspectos de acción
Requerimientos	Sólo se mantienen 2 de 4 métricas de requerimientos	Buena; problema: descuido, no falta de tiempo	Designar un ingeniero de AQ para mantener el ERS y las cuatro métricas. LD por 3/1.
Diseño	No define las métricas	Mala; terminado en 140% de la duración programada	HR para seleccionar las tres métricas mejores para este tipo de aplicación,; estimar costos y programar consecuencias; por 3/5 LD para describir prioridades de las actividades de diseño; por 4/1. ST para decidir si el tiempo asignado a diseño fue insuficiente, o si el proceso fue ineficiente e identificar las razones; por 3/5
Implementación	Buena	Terminada en 130% de la duración programada	Revisar métodos de estimación de líneas de código; identificarlas tres mejores razones de sobrestimación; JA por 3/5
Integración y liberación	Aplica demasiadas métricas inútiles	Adecuada	Eliminar métrica de “tasa de subscriptores al sitio beta”. Reasegurar efectividad de otras métricas usadas; BV por 3/10.

Tabla 15.- Ejemplo postmortem: análisis de requerimientos mediante la Integración del Sistema.

2.7.8 Herramientas para Integración y Pruebas del Sistema

El gran volumen de pruebas suelen requerir herramientas de prueba automatizadas. Jacobson et al. [Ja1] sugieren que al menos 75% de las pruebas sean automatizadas y el resto manual. Las capacidades de las herramientas de prueba seleccionadas se enumeran en la figura siguiente.

Capacidades de las herramientas automatizadas de prueba de sistemas

- Registrar acciones de ratón y teclado para permitir reproducción repetida.
  - Sin la posibilidad de registrar y reproducir las acciones de ratón y teclado, los probadores están limitados a realizar estas pruebas a mano una y otra vez. Esto es tedioso y costoso. Además, quizá los resultados no sean comparables directamente porque las personas no pueden duplicar sus acciones con precisión. Kit [Kit] clasificó las herramientas de captura / reproducción.
  - Las herramientas más comunes de captura / reproducción son pruebas intrusivas de software nativo. Un ejemplo de pruebas de sistema no intrusivas es la prueba de una aplicación de comando y control militar en tiempo real donde la interacción del usuario se simula con hardware separado conectado a la aplicación que se prueba. El hardware externo se programa para proporcionar estímulos al comando y control de manera que la aplicación no pueda distinguir estos estímulos de los datos reales del usuario.
    Las herramientas de registro / reproducción pueden ser muy útiles, pero son sensibles a cambios en la interfaz de usuario. Un pequeño cambio en la IU puede invalidar un conjunto completo de pruebas ejecutadas de manera mecánica.
- Correr varias veces los textos de prueba.
  - La posibilidad de ejecutar pruebas automáticas de la aplicación ahorra a los probadores repetirlas a mano con diferentes parámetros.
- Activar registro de resultados de pruebas.
  - Esto evita que los probadores tengan que implementar esta función.
- Registrar tiempo de ejecución.
  - Las herramientas automatizadas de pruebas pueden medir y registrar el tiempo transcurrido y el uso de CPU.
- Registrar errores de la corrida.
  - Algunas herramientas automáticas de prueba pueden registrar errores encontrados mientras se ejecuta la aplicación.
- Crear y administrar pruebas de regresión.
  - Hay que recordar que las pruebas de regresión se requieren para validar el hecho de que las modificaciones a la versión anterior no introducen nuevos errores. Las pruebas de regresión cambian con el tiempo, al implementar cada vez más capacidades. Algunas herramientas automáticas de prueba pueden mantener un registro de estas pruebas y aplicarlas cuando se pida.
- Generar informes de pruebas.
  - Las herramientas automáticas de prueba incluyen generadores de informes que eliminan la necesidad de escribir los numerosos informes a mano o escribir el propio generador de informes.
- Generar datos de pruebas.
  - Entre las herramientas de prueba más útiles están las que prueban la generación de datos para las pruebas. Estas herramientas generan datos de entrada para satisfacer muchas de las disciplinas de caja blanca y caja negra presentadas anteriormente. Un ejemplo es la generación de combinaciones aleatorias de entradas. Algunas herramientas de prueba también facilitan la integración y las pruebas del sistema de caja gris para la interacción de los módulos. Sin embargo, debe observarse que no se espera que estas herramientas generen la salida correcta para casos de pruebas, ya que esta capacidad es la razón por la que se construye la aplicación.
- Registrar uso de memoria.
  - Estas herramientas son útiles al probar el desempeño de la corrida de las aplicaciones.
- Administrar casos de pruebas.
- Analizar la cobertura.
  - Quienes analizan la cobertura toman como entrada el producto, junto con la combinación de pruebas. Proporcionan un análisis de lo que cubre esa combinación. Estos analistas pueden verificar varios tipos de cobertura de las pruebas, que incluyen la cobertura de declaraciones.

Fig. 126.- Capacidades de las herramientas automatizadas de Prueba de Sistemas.

Probar requiere el uso repetido de formas. Las plantillas de documentos son las "herramientas" de prueba más elementales, pero las que más se usan. Las plantillas se pueden basar en los estándares de documentación de pruebas como la "Documentación de Pruebas de Software" de ANSI/IEEE 829-1983 (confirmado en 1991).

Aunque los programas de pruebas automatizadas sustituyen muchas de las tareas de programación de pruebas, su uso a menudo requiere habilidades de programación significativas. Por ejemplo, los registradores de las acciones de ratón y teclado deben atrapar esos eventos y esto requiere que los programadores tengan buenos conocimientos de cómo se generan estos eventos, para que las herramientas de prueba puedan interceptar la aplicación de manera adecuada.

2.8 Bases de Datos

El tema de Base de Datos es muy importante ya que es muy común que como parte de un sistema exista al menos un repositorio de datos en donde se registre la información importante para el negocio, así como también algunos otros datos importantes para que funcione el sistema, dentro del diseño y algunas arquitecturas las bases de datos son conocidas como la capa de persistencia, queriendo decir con ello que esta capa arquitectónica será la que se encargue de guardar toda la información que se requiera para que el sistema funcione adecuadamente.

Se pueden escribir libros enteros de este tema ya que es muy extenso y existen varias clases de bases de datos, cada una con una metodología diferente de diseño.

Lo primero que haremos es especificar que es una base de datos y se puede definir como un conjunto de datos que pertenecen al mismo contexto almacenados sistemáticamente para su posterior uso.

Las bases de datos se pueden clasificar en diferentes formas y una de las clasificaciones es el tipo de información que almacenan:

Bases de datos Transaccionales (OLTP): Es un acrónimo de On Line Transactional Processing que nos sirven para almacenar transacciones como pueden ser ventas, compras, órdenes, transacciones bancarias, clientes, etcétera. Estas bases de datos se caracterizan por la forma dinámica en que sus datos se actualizan.

Bases de datos para análisis (OLAP): Este es un acrónimo de On Line Analysis Processing. Estos tipos de base de datos nos sirven para analizar las tendencias de ciertos elementos con respecto al tiempo principalmente, por ejemplo cómo han sido las ventas en los diferentes lugares en los diferentes meses o años, qué productos son los que más se venden y en qué lugares, etcétera. Se caracterizan por tener ciertos indicadores que se comparan en el tiempo y también suelen tener estas bases de datos la información histórica de varios años, por lo que son generalmente muy grandes y su actualización no es tan dinámica (pueden actualizarse una vez por mes) y en general se toma la información de varias bases de datos (OLTP) y se realizan procesos para adaptar los datos para obtener los indicadores por periodos.

Otra de las clasificaciones que se pueden hacer es de acuerdo a su modelo de datos y de acuerdo con ellos se tiene lo siguiente:

Bases de datos Jerárquicas: Son bases de datos que como su nombre lo indica almacenan la información en una jerarquía, su estructura es similar a la estructura de datos de árbol en donde hay un nodo padre del cual se deriva la información y de allí se van derivando los nodos hijos hasta llegar a nodos que no tienen más hijos llamados hojas.

Las bases de datos jerárquicas son especialmente útiles en el caso de aplicaciones que manejan un gran volumen de información y datos muy compartidos permitiendo crear estructuras estables y de gran rendimiento.

Una de las principales desventajas de este modelo es su incapacidad de representar eficientemente la redundancia de datos.

Bases de datos de red: Este es un modelo ligeramente distinto del jerárquico; su diferencia fundamental es la modificación del concepto de nodo; se permite que un mismo nodo tenga varios padres (posibilidad no permitida en el modelo jerárquico).

Fue una gran mejora con respecto al modelo jerárquico, ya que ofrecía una solución eficiente al problema de redundancia de datos; pero, aun así, la dificultad que significa administrar la información en una base de datos de red ha significado que sea un modelo utilizado en su mayoría por programadores más que por usuarios finales.

Base de datos Relacional: Es el modelo más usado en la actualidad, para modelar y sus fundamentos fueron postulados en 1970 por Edgard Frank Codd en los laboratorios IBM y se consolidó como un nuevo paradigma en los modelos de bases de datos. Su idea fundamental es el uso de "relaciones" y está basado en la teoría de conjuntos.

Estas relaciones podrían considerarse en forma lógica como conjuntos de datos llamados "tuplas". Pese a que ésta es la teoría de las bases de datos relacionales creadas por Edgar Frank Codd, la mayoría de las veces se conceptualiza de una manera más fácil de imaginar. Esto es pensando en cada relación como si fuese una tabla que está compuesta por registros (las filas de una tabla), que representarían las tuplas y campos (las columnas de una tabla).

En este modelo, el lugar y la forma en que se almacenen los datos no tienen relevancia (a diferencia de otros modelos como el jerárquico y el de red). Esto tiene la considerable ventaja de que es más fácil de entender y de utilizar para un usuario esporádico de la base de datos. La información puede ser recuperada o almacenada mediante "consultas" que ofrecen una amplia flexibilidad y poder para administrar la información.

El lenguaje más habitual para construir las consultas a bases de datos relacionales es SQL, Structured Query Language o Lenguaje Estructurado de Consultas, un estándar implementado por los principales motores o sistemas de gestión de bases de datos relacionales.

Durante su diseño, una base de datos relacional pasa por un proceso al que se le conoce como normalización de una base de datos, en el capítulo 3 se tratará con más detalle este tema.

Durante los años '80 (1980-1989) la aparición de dBASE produjo una revolución en los lenguajes de programación y sistemas de administración de datos. Aunque nunca debe olvidarse que dBASE no utilizaba SQL como lenguaje base para su gestión.

Bases de datos Orientadas a Objetos: Este modelo, bastante reciente y propio de los modelos informáticos orientados a objetos, trata de almacenar en la base de datos los objetos completos (estado y comportamiento).

Una base de datos orientada a objetos es una base de datos que incorpora todos los conceptos importantes del paradigma de objetos:

- **Encapsulación:** Propiedad que permite ocultar la información al resto de los objetos, impidiendo así accesos incorrectos o conflictos.
- **Herencia:** Propiedad a través de la cual los objetos heredan comportamiento dentro de una jerarquía de clases.
- **Polimorfismo:** Propiedad de una operación mediante la cual puede ser aplicada a distintos tipos de objetos.

En bases de datos orientadas a objetos, los usuarios pueden definir operaciones sobre los datos como parte de la definición de la base de datos. Una operación (llamada función) se especifica en dos partes. La interfaz (o signatura) de una operación incluye el nombre de la operación y los tipos de datos de sus argumentos (o parámetros). La implementación (o método) de la operación se especifica separadamente y puede modificarse sin afectar la interfaz. Los programas de aplicación de los usuarios pueden operar sobre los datos invocando a dichas operaciones a través de sus nombres y argumentos, sea cual sea la forma en la que se han implementado. Esto podría denominarse independencia entre programas y operaciones.

Se está trabajando en **SQL3,** que es el estándar de SQL92 ampliado, que soportará los nuevos conceptos orientados a objetos y mantendría compatibilidad con SQL92.

Capítulo 3. Generación del Sistema de Producción.

3.1 Análisis

Parte del análisis se realizó en el Capítulo 1, donde se enlistaron las funciones del sistema, las necesidades del negocio, etcétera. En seguida se hará la parte del análisis plasmando estas necesidades, funciones y características en Casos de Uso.

Como se recordará, los Casos de Uso son de suma utilidad para el análisis de un sistema, pues nos favorecen de una forma muy adecuada la comunicación con el usuario y la comprensión de detalles ya que nos plantean la utilización paso a paso de un proceso del sistema, con lo cual se conoce este proceso desde el punto de vista del usuario y además se divide el sistema en procesos que nos hacen trabajar con un problema a un nivel adecuado de complejidad ya que en general los sistemas completos son herramientas que tienen un nivel de complejidad bastante alto y si no se analiza ordenadamente, se puede uno perder fácilmente en tantos detalles y procesos.

Conforme se vayan haciendo los Casos de Uso completos de toda la aplicación se podrá hacer un seguimiento de la lista de funciones del sistema y de esta forma se verá como todas las características de negocio se van transformando en diagramas que nos van indicando como interactúa el usuario con el sistema que se convertirá en nuestra solución.

El siguiente diagrama muestra los Casos de Uso de la generación de órdenes.

Fig. 127.- Casos de Uso para la generación de órdenes.

El diagrama anterior es un esquema de todos los Casos de Uso relacionados con las órdenes de producción y como se puede visualizar se divide todo este proceso en partes más pequeñas que nos permiten tener un mejor control de lo que se está analizando y al no incluir en esta etapa aspectos técnicos el usuario se siente mucho más cómodo e involucrado para esta fase del desarrollo. A continuación se muestra el Caso de Uso de la generación de órdenes.

3.1.1 Caso de Uso de la Generación de Ordenes

Caso de Uso: Generar Orden.

Sección: Principal.

Actores: Encargado de producción.

Propósito: Generar una orden de producción.

Resumen: El encargado de producción es notificado que habrá de generar unas órdenes de producción a partir de una explosión de materiales que puede tener uno o más productos a fabricar, este captura el número de explosión y una vez mostrada la lista de productos dentro de la misma, selecciona la prioridad de fabricación de los productos de la explosión, posteriormente selecciona el o los número(s) de órdenes que desea generar y el sistema genera la(s) orden(es) de producción y la(s) orden(es) de embarque.

Tipo: Primario y esencial.

Referencias cruzadas: Funciones (R3.1, R3.2, R3.3, R3.4, R3.5, R3.6, R3.7, R3.8).

Casos de Uso: El encargado de planeación y control de la producción debe haber terminado el Caso de Uso Generar Explosión.

Curso normal de los eventos

Acción de los actores	Respuesta del sistema
1. Este Caso de Uso comienza cuando el encargado de producción selecciona una explosión de materiales para generar órdenes de producción.	2. El sistema muestra todos los productos los cuales se les hizo una explosión de materiales y que habrá que fabricar.
3. El encargado selecciona una prioridad de fabricación en caso de haber más de un producto y ejecuta la generación de órdenes.	4. El sistema valida que haya los materiales suficientes y genera los números de orden.
5. El encargado Selecciona los números de orden y selecciona la opción de generar la orden.	6. El sistema genera la orden con las cantidades y lotes adecuados, muestra las órdenes generadas así como los registros de embarque.

Cursos alternos

- **Línea 4:** El sistema encuentra que no hay materiales suficientes para generar la orden e indica esta situación mostrando la lista de materiales, las cantidades necesarias para generar la orden y la cantidad de material a comprar.
- **Línea 6:** El sistema encuentra que un material cuyos lotes no pueden combinarse no tiene suficiente cantidad para producir la orden, el sistema no generará la orden y avisará de esta situación al encargado de producción.

A partir de este Caso de Uso se pueden comenzar a visualizar las entidades y conceptos que intervienen en el mismo, así como las relaciones entre las entidades. Una forma de plasmar estas entidades y la forma en que se relacionan es mediante los modelos conceptuales, dichos modelos permiten ir visualizando como se relacionan las entidades, aún son conceptos y no se puede decir cuales se pueden llegar a convertir en clases en un modelo orientado a objetos pero si permite en conjunto ver como se relacionan las cosas para llevar a cabo los procesos que el negocio necesita, estos modelos cabe resaltar que se van depurando a lo largo del tiempo ya que conforme se van conociendo más aspectos de cómo son los procesos y su interacción, se podrán hacer mejores modelos conceptuales, de hecho también es importante señalar que estos deben mostrar los aspectos más relevantes de esta interacción y que un modelo más que ser correcto o incorrecto se busca sea útil a las siguientes fases de desarrollo, es decir, que muestre las cosas necesarias para el análisis y que de allí se parta hacia el diseño pero que no se recargue tanto este modelo que llegue a ser incomprensible para los analistas y diseñadores que vayan a hacer uso de él.

El modelo puede ir con o sin atributos, los atributos son el nombre de la relación y la multiplicidad principalmente, en este caso el diagrama contiene atributos ya que se consideró que es mejor debido a que es más claro aunque el modelo en si se ve un poco mas recargado por lo cual debe de establecerse un balance, no cargarlo de conceptos y propiedades si estas no son relevantes pero si mostrar todo lo que se considere importante y que brindará un mejor conocimiento de los requerimientos, conforme se van detallando y viendo Casos de Uso se podrá ir completando el modelo conceptual y se puede también hacer modelos de diferentes bloques o grupos de funcionalidad del sistema para que de esta manera no resulten modelos extremadamente complejos y que sirvan estos modelos sólo como un requisito pero que al ser poco útiles realmente no se aproveche todo su potencial y además se invierta tiempo en algo que no va a servir de base para facilitar el diseño y el conocimiento del negocio.

El modelo que se presenta a continuación sólo muestra el modelo conceptual para las órdenes. Es un modelo parcial a modo de ilustrar como se generan estos diagramas y como se fue haciendo el análisis del sistema, ya que sería muy extenso mostrar y explicar a detalle todos estos diagramas.

El modelo conceptual con los atributos para los Casos de Uso relacionados con las órdenes es el siguiente:

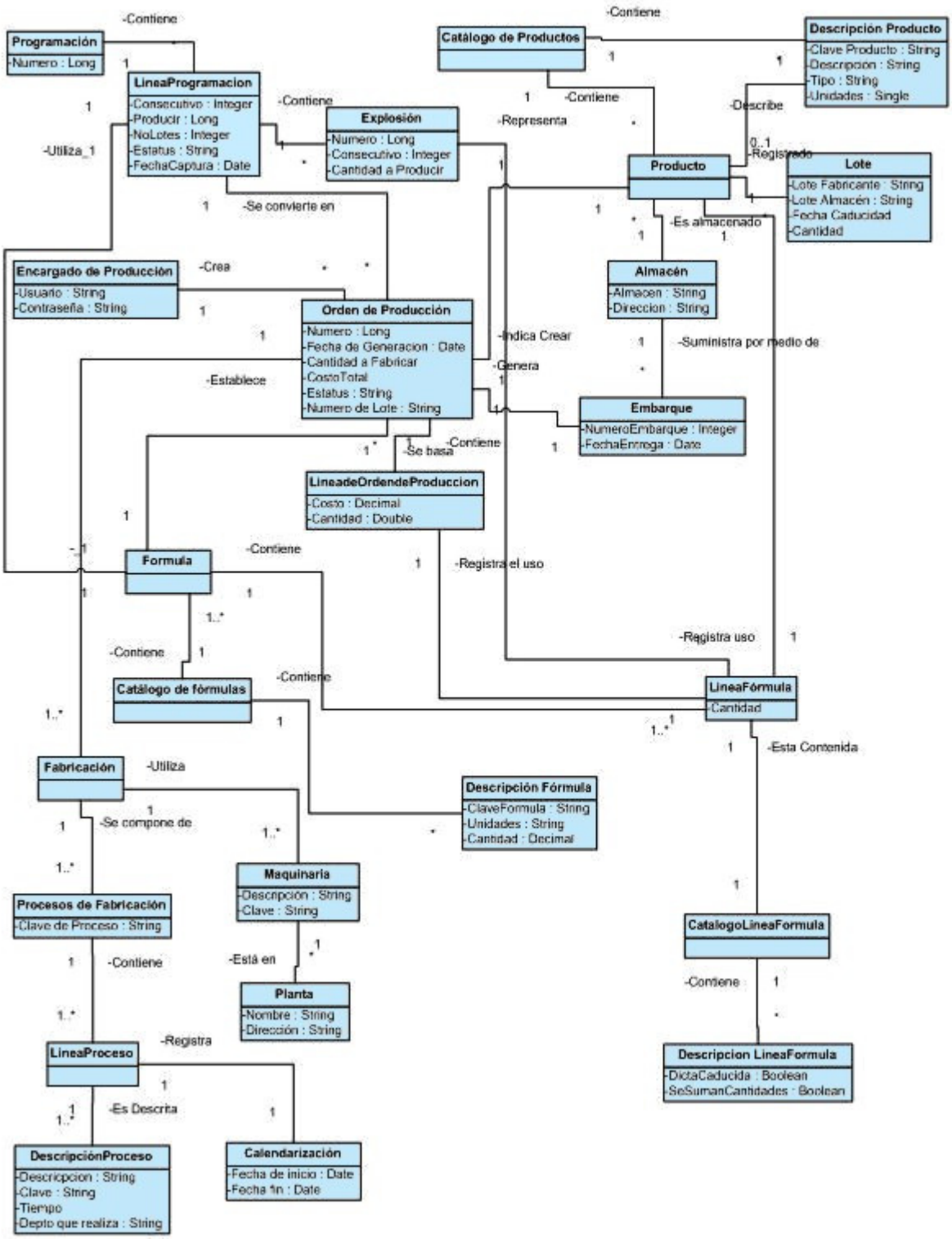


Fig. 128.- Modelo Conceptual para la Generación de Ordenes.

Como ya se dijo en la parte teórica, el glosario de un sistema nos permitirá describir las definiciones principales que existen dentro del entorno del mismo, esto nos será sumamente útil para disminuir las ambigüedades que pueda haber ya que al unificar criterios se trata de asegurar que cuando se habla de algún termino se sabe exactamente a lo que se refiere y además cuando un nuevo miembro se integra, su curva de aprendizaje disminuye considerablemente al leer el glosario, ya que se familiariza con los términos usados más rápidamente.

No se trata de definir todos los conceptos, únicamente los que se consideran más adecuados, representativos o importantes para poder entender bien las entidades que están involucradas en el sistema a producir. Y al igual que en los modelos se debe establecer un balance entre la simplicidad y la utilidad, es decir, no recargar de tantos conceptos el glosario de tal forma que resulte demasiado complicado y abrumador consultarlo, pero tampoco tan escueto que no aclare los conceptos y termine archivado y olvidado mientras existen discrepancias en los conceptos entre los integrantes del equipo o entre el cliente y los integrantes del equipo de desarrollo.

El glosario con las definiciones de los Casos de Uso relacionados a las órdenes es el siguiente:

Término	Categoría	Comentario
Programación	Tipo	Una programación es una entidad que agrupa varias explosiones de materiales, por ejemplo, para suministrar un pedido se engloban todos los productos a producir para surtir el mismo.
Explosión	Tipo	Es un proceso en el cual se verifica que una programación de producción de un producto tenga los materiales necesarios para fabricarlo. Una explosión es un conjunto de líneas de productos que son necesarios para generar un elemento de una programación.
Orden de Producción	Tipo	Es un listado de materiales que contienen lotes y cantidades para producir un producto dado.
Fórmula	Tipo	Es una lista de materiales y cantidades para producir un producto.
Proceso de fabricación	Tipo	Es el conjunto de pasos a seguir para fabricar un producto.
Embarque	Tipo	Una orden con los materiales a ser llevados del almacén a la planta de producción.
Lote	Tipo	Es un identificador que se le da a los productos para tenerlos mejor ubicados y tener información muy clara y específica de ciertos aspectos relevantes como podría ser la fecha de fabricación, así mismo es una forma de agrupar productos por proceso de fabricación, es decir, todos los productos con un lote fueron fabricados en un mismo proceso de fabricación.
Fase de producción	Tipo	En el caso de este sistema existen 3 fases de producción: Granel, Semiterminado y Terminado. Y son las etapas en las que se divide la fabricación de un producto.
Granel		Es la primera etapa de producción y propiamente se llama así porque en dicha fase se fabrican los productos en su forma más básica, es decir en cantidades especificadas por la orden de producción pero sin ningún tipo de presentación o agrupamiento. Tal cual se especifica en la orden de producción por ejemplo de esta etapa salen millares de pastillas o litros de jarabe.
Semiterminado		Es la segunda fase de producción que se caracteriza por ser donde se acondiciona el producto fabricado en la primera etapa y se agrupa, por ejemplo ,los millares de pastillas producidos en la fase de granel en esta etapa se ponen en tiras de celofán o en frascos.
Terminado		En esta última fase se busca darle la presentación final al producto, por ejemplo, las tiras de 12 pastillas se ponen en cajas de 3 tiras y con presentación para venta al público o con presentación para venta al ISSSTE, etcétera, se etiquetan y se terminan de poner en las envolturas adecuadas.

Tabla 16.- Glosario usado en la Generación de Ordenes.

El diagrama de secuencia como se trató en la parte teórica, nos muestra los eventos que se generan normalmente en un Caso de Uso y cómo el usuario y el sistema interactúan y los eventos que se generan en el sistema para atender las peticiones del mismo. Este tipo de diagramas son muy útiles ya que nos permite visualizar los eventos de entrada al sistema y de una forma muy esquemática cómo el sistema reacciona, esto nos permite irnos adentrando en la solución que vamos a diseñar y a darnos cuenta de cómo el usuario irá interactuando y cuáles deben ser los eventos que disparen la diferente funcionalidad que tendrá el sistema, es una forma bastante adecuada de ir conociendo el problema ya que al ser de una forma esquemática y reducida a un Caso de Uso en la mayoría de los casos, la porción de complejidad tomada es bastante manejable y nos permite ir conociendo el problema de manera gradual y en todos sus aspectos, desde la problemática, las entradas que debe tener el sistema, los procesos que debe ejecutar, etcétera, todo esto nos permite tener un conocimiento amplio del problema cuya solución queremos construir y entre más amplio sea el panorama y más aspectos conozcamos del mismo podremos brindar una solución más adecuada, que finalmente es el objetivo de crear un sistema, brindar una herramienta útil a sus usuarios.

A continuación veremos el diagrama de secuencia para el Caso de Uso generar órdenes:

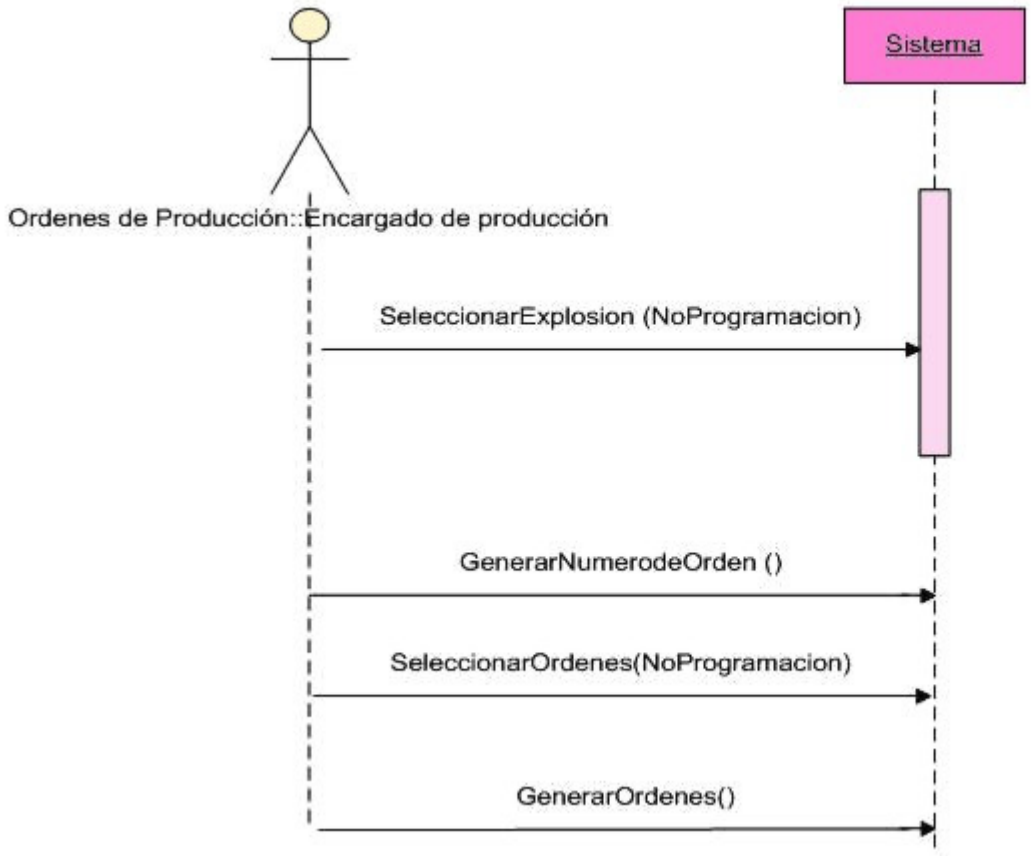


Fig. 129.- Diagrama de Secuencia para el Caso de Uso Generar Ordenes.

El análisis de un sistema como se recordará nos sirve para conocer el problema y nos dará la respuesta a las diferentes preguntas de qué o cuáles son los diferentes aspectos tanto del problema como de la solución del mismo mientras que el diseño se cuestiona el "como" se harán las cosas. En este sentido los contratos son la parte del análisis que nos ayudan a definir el comportamiento de un sistema; ya que describen el efecto que sobre él tienen las operaciones y nos darán en detalle los pasos que deben seguir los eventos una vez que se disparan, se enuncian que condiciones previas habrán de existir como precondiciones y en caso de posibles excepciones también se documentan.

Normalmente se debería poder hacer un seguimiento de las funciones que debe tener el sistema, para ello los contratos cuentan con la sección de referencias cruzadas, ya que con esto nos podemos dar cuenta cuáles de las funciones abarca un contrato y por supuesto al finalizar todos los contratos y Casos de Uso, se deben abarcar todas las funciones.

Los siguientes contratos que se muestran a continuación son los contratos que cubren el Caso de Uso Generar Ordenes:

Contrato

Nombre:	SeleccionarProgramacion (NoProgramacion: Entero)
Responsabilidades:	Seleccionar un número de Programación sobre la cual se generarán uno o más números de orden.
Tipo:	Sistema.
Referencias cruzadas:	R3.1
Notas:	
Excepciones	1. El número de explosión no existe, el sistema indicará que se cometió un error.
Salida:	
Precondiciones:	El usuario ya esta reconocido por el sistema.
Poscondiciones:	Se creó una instancia de Programación. Se crearon instancias de línea de Programación. Se asoció línea de programación con Programación. Se creo una instancia de fórmula para cada línea de Programación. Se asoció una instancia de fórmula para cada línea de Programación. Se creó una instancia de producto para cada línea de Programación. Se asoció una instancia de producto a cada línea de programación.

Tabla 17.- Contratos (1 de 3) que cubren el Caso de Uso Generar Ordenes.

Nombre:	GenerarNumeroOrden ()
Responsabilidades:	Generar un número de orden para cada línea de una programación.
Tipo:	Sistema.
Referencias cruzadas:	R3.2,R3.3
Notas:	
Excepciones:	1. El número de explosión no existe, el sistema indicará que se cometió un error. 2. Si no existe suficiente material para generar una o más órdenes, se indicará que no se pueden realizar la generación de algunas órdenes y se mostrará la lista de materiales que son necesarios, las existencias y las cantidades a comprar para satisfacer las necesidades.
Salida:	
Precondiciones:	Se debe indicar al sistema una prioridad en cuanto a cuales líneas de programación tendrán los primeros números de orden y cuál línea tendrá el último número de orden.
Poscondiciones:	Se creó una instancia de OrdenProduccion. Se asoció OrdenProduccion con LineaProgramación. Se estableció OrdenProduccion.Estatus a “Validación”. Se creó una instancia de Fórmula. Se asoció la instancia de Fórmula a OrdenProducto. Se creó una instancia de LineaFormula. Se asoció LineaFormula a Formula. Se creo Producto. Se asoció Producto a LineaFormula. Se creó una instancia de Lote. Se asoció Lote a Producto. Se estableció el atributo OrdenProduccion.Numero. Si es una orden de Granel se estableció el atributo OrdenProduccion.Cantidad a la Cantidad Contenida en la fórmula como lote estándar.

Tabla 18.- Contratos (2 de 3) que cubren el Caso de Uso Generar Ordenes.

Nombre:	GenerarOrdenes ()
Responsabilidades:	Generar las órdenes de producción para cada línea de una explosión de materiales.
Tipo:	Sistema.
Referencias cruzadas:	R3.4, R3.5, R3.6, R3.7, R3.8
Notas:	
Excepciones:	1. El número de explosión no existe, el sistema indicará que se cometió un error. 2. Si no existe suficiente material para generar una o más órdenes, se indicará que no se pueden realizar la generación de algunas órdenes y se mostrará la lista de materiales que son necesarios, las existencias y las cantidades a comprar para satisfacer las necesidades.
Salida:	
Precondiciones:	Se debe indicar al sistema cuales órdenes se generarán de todas las que pertenecen a la explosión de materiales.
Poscondiciones:	Se modificó la propiedad Lote.Cantidad. Se generó una instancia de embarque. Se asoció la instancia del Embarque. Se modificó el atributo Embarque.NumeroEmbarque. Se modificó el atributo Embarque.FechaEntrega. Se modifico el atributo OrdenProduccion.Costo. Se modifico el atributo OrdenProduccion.NumerodeLote. Se modificó la propiedad OrdenProduccion.Estatus a “Generada”.

Tabla 19.- Contratos (3 de 3) que cubren el Caso de Uso Generar Ordenes.

3.1.2 Entorno

Como se dijo anteriormente el sistema será uno de un conjunto de varios sistemas que interactuarán entre ellos para poder llevar un adecuado manejo de la empresa, así como el generar la información adecuada para por un lado tomar decisiones adecuadas a corto plazo y por otro sentar las bases para una explotación de la información por herramientas del tipo OLAP (Online Analytical Processing). Existe ahora una infraestructura que ha ido mejorando con el tiempo y en el presente hay una red LAN, un par de servidores de archivos y otro de datos, lo que permitirá que la información se centralice en estas

máquinas. La idea es que una sola base de datos se encargue de manejar la información de toda la empresa y que varios sistemas concurran e interactúen para proveer un adecuado manejo y actualización de la información que se genera y modifica continuamente de acuerdo a las diversas operaciones que se efectúan en la organización.

Existen también como parte de la infraestructura los productos de desarrollo Microsoft Visual Studio Versión 6 del que se utiliza el Visual Basic y Microsoft SQL Server versión 7 con los cuales se están desarrollando el resto de las aplicaciones que conforman esta nueva solución. El sistema se calcula que generará alrededor de unas 1,100 órdenes de producción al año y esto implicará aproximadamente más de 10,000 transacciones de materiales ya que son en promedio unos tres materiales usados en cada fase de producción.

Se calcula que la base de datos atenderá aproximadamente a unos 7 usuarios al mismo tiempo por parte de este sistema más los usuarios de los otros sistemas, que podrán llegar a ser unos 20 o 30 al mismo tiempo, con un crecimiento anual de un 10 % en los próximos años.

El tiempo de respuesta de los procesos o transacciones deben ser lo más cortos posibles ya que el proceso de generar una orden aunque implique realizar varios movimientos de materiales, así como cálculos de costos tendrá que hacerse tanto de una manera segura como rápida.

3.1.3 Beneficios

Es importante mencionar que ya había un sistema previo de producción pero que dicho sistema en primer lugar no interactuaba mucho con los demás sistemas y por otro lado no manejaba costos en línea de las órdenes, esto es, se obtenían dichos costos por medio de procesos conocidos como "por lotes" (o batch), ni tampoco manejaba el tiempo que se requiere en producir cada orden con el propósito de tener la información suficiente para medir la productividad de los procesos de fabricación de la empresa.

El sistema también en esta interacción generará movimientos en el inventario de forma automática a la hora de generar una orden de acuerdo con los materiales que indique la fórmula y en las cantidades que se calculen para la orden, así como también en el momento del cierre de la misma al ingresar la cantidad producida del producto indicado en la orden. Si hay una devolución de materiales o una cancelación también se generarán automáticamente ingresos de materiales en el inventario.

Otro beneficio importante es la flexibilidad de haber dividido todo el proceso de fabricación en tres etapas y que cada una tiene su fórmula, esto es muy importante porque con el mismo sistema se pueden producir medicamentos, suturas, inyectables, jarabes, lociones, etcétera, o incluso cualquier otro producto del que se tengan ingresados su fórmula, sus procesos productivos y el producto a fabricar en si, es decir, si se quisiera fabricar cualquier producto simplemente habría que ingresarlo, introducir sus componentes y darlo de alta dentro de las fórmulas. Esto por ejemplo, como dije anteriormente nos permite fabricar múltiples productos en diferentes presentaciones, como por ejemplo nos permite generar un medicamento en grageas, inyectable o solución, así como presentarlo en frasco, sobres, tiras de cápsulas, etc. Y por último presentarlo para el sector salud, para venta al público o alguna institución como ISSSTE o Seguro Social. De allí vienen las tres etapas, la primera nos genera propiamente el medicamento o producto, la segunda nos da la presentación de dicho producto, ya sea frasco, sobre, ampolleta, tira de celofán

etcétera y por último la tercera nos indica la presentación final, por ejemplo, caja con 2 tiras de 10 para venta al público.

Un beneficio que nos brindará el dividir en tres etapas la fabricación es que se puede registrar para cada una de ellas las mermas. Y al ir registrando cuales son éstas en cada etapa, así como los tiempos que en cada fase y proceso de fabricación se dan, podemos en algún momento optimizar los mismos, así como dar incentivos a las personas más productivas del área.

3.1.4 Selección de metodologías

Como se vio en el capítulo anterior, existen varias formas de atacar un problema. La decisión de escoger alguna metodología sobre otra puede depender de varios factores, entre ellos pueden ser las características del sistema, la facilidad de uso, relación costo/beneficio e inclusive no se debe minimizar la importancia de que los desarrolladores y personal en general estén familiarizados con alguna metodología lo cual disminuye el tiempo de tener que familiarizarse con otra nueva, siempre y cuando las desventajas que presente la metodología usada respecto a la nueva no sean muy importantes, claro está. Son muchos los factores que intervienen en la selección de las metodologías y de los productos a utilizar en un proyecto, aunque muchas veces los factores de más peso que se toman en cuenta son la familiaridad que se tengan con las metodologías y productos a usar, ya que el utilizar nuevos métodos implica capacitación, curva de aprendizaje y un proceso de tiempo y labores en el cual las personas asimilan las nuevas formas de trabajar y las desarrollan plenamente, naturalmente en ocasiones las ventajas de hacer esta inversión de tiempo y esfuerzo valen la pena pero no siempre es el caso, a veces solo es necesario refinar el conjunto de métodos que se utilizan para mejorar el rendimiento y calidad del proyecto.

Una de las partes importantes a escoger es el proceso de desarrollo y para este sistema en particular se escogió el RUP (Rational Unified Process), esto debido a que es una metodología bastante aceptada en la industria de desarrollo de software y además es una forma práctica de dividir la solución en etapas manejables, es similar al desarrollo en espiral en el sentido que hay iteraciones y que está dividido en varias fases el desarrollo, sólo que se escogió este proceso de desarrollo por ser más robusto y bastante documentado, así mismo algunos integrantes del personal ya tenían un primer acercamiento con este proceso de desarrollo.

Hay bastante literatura de apoyo para esta metodología a diferencia de los métodos ágiles en general e integra innovaciones con respecto a metodologías como el desarrollo en espiral ya que es una variante un poco más moderna y ordenada en el sentido que las fases están mejor definidas.

En la parte de administración de proyectos, existen más que metodologías, prácticas que son comunes y exitosas y que están mencionadas en la sección de Administración de Proyectos de Software, entre las cuales se menciona la gestión tomando en cuenta las 4 p (que son personal, producto, proceso y proyecto), así como tratar en todo momento de identificar y retirar posibles riesgos que pueden tener influencia negativa sobre nuestro proyecto. En relación con los riesgos y el proceso cabe decir que el hacerlo en iteraciones tiene la ventaja de poder ir tomando la experiencia de las primeras iteraciones para evitar riesgos y problemas que se vayan suscitando e ir perfeccionando de manera proactiva la administración del proyecto, de esta forma para nuestro caso se tomaron en cuenta ambos aspectos, el de identificar y retirar los riesgos y la gestión tomando en cuenta las 4 p.

Para analizar los requerimientos del sistema se eligieron los Casos de Uso por considerarlos que son una forma adecuada para interactuar con los usuarios, una forma descriptiva y sencilla que nos permitirá mediante diagramas y texto expresar el funcionamiento y la interacción entre el usuario y el sistema, así como definir diferentes roles o tipos de usuarios. Se tiene con esto una buena comunicación con el cliente y no se satura el conocimiento inicial del problema con cuestiones demasiado formales, repetitivas y redundantes. En general cuando se utilizan Casos de Uso, en primer lugar se comienzan a plasmar en diagramas que nos dan una visión general del sistema dividido en procesos que tienen un inicio y un fin de interacción con el usuario y en segundo lugar cuando ya se comienzan a hacer uno a uno los Casos de Uso en texto, más bien se comienza narrando como el usuario y el sistema interactúan para alcanzar un objetivo y en base a esto se empieza a conocer los requerimientos, cómo se espera que el sistema responda, cuál es la información de entrada que el sistema necesita, qué información de salida brindará el sistema e inclusive se está con estos diagramas dividiendo la complejidad del sistema en partes mucho más manejables que nos permiten ir poco a poco conociendo el problema. Al tener los diagramas de Casos de Uso se están viendo los diferentes procesos con los que el sistema contará y es mucho más fácil verlo en un diagrama e irlo viendo en texto caso por caso que en un documento ver todas las características y toda la complejidad sin ninguna división ni orden específico, además de que los Casos de Uso no están ligados a ningún tipo de diseño, ni tampoco nos liga o ata con otras metodologías que se tuvieran que seguir una vez seleccionada esta forma de análisis, por estas características es que utilizamos los Casos de Uso y también porque es un método muy generalizado y documentado en la industria del software y porque también se tiene cierta experiencia en la utilización de los Casos de Uso.

Para la parte de diseño se podía escoger una de dos posibilidades: por un lado el diseño Orientado a Objetos y por otro el diseño Procedimental, se escogió el primero porque entre otras cosas es más fácil de reutilizar, existe una afinidad entre entidades de negocio y las clases producidas, por lo que diseñar un sistema de esta forma representa mayor facilidad de transición que va de los modelos conceptuales al diseño en clases, en un sistema complejo es más sencillo diseñar clases que se parezcan a las entidades reales que interactúan en los procesos que dividir por funciones una aplicación, por ejemplo es más fácil conceptualizar en un sistema de ventas una clase que se llame "factura" y que contenga varios métodos que me permitan interactuar con dicha clase que generar una serie de procedimientos que hagan lo que se puede hacer en una factura y que sean llamados en muchos lugares de mi programa. De esta manera la primera es la que guarda más similitud con el modelado de entidades reales que seccionar el diseño por funcionalidad. Además, muchos de los lenguajes y tecnologías nuevas de software no sólo dan soporte a este tipo de programación, su forma de manejo interno de muchos programas y paquetes por no decir que casi todos, es a base de objetos y cada día se amplía más el uso de la programación orientada a objetos en la industria del software, de hecho ahora lo raro o poco común es ver sistemas hechos de forma procedimental. Ahora bien, entre las metodologías de este tipo de programación para diseño de sistemas, se encuentra el UML; este es un lenguaje de modelado que nos permitirá diseñar el sistema ya que contiene muchas vistas diferentes de modelado, nos permite ver el sistema y diseñarlo desde varios puntos de vista o perspectivas y es un lenguaje que se ha convertido prácticamente en un estándar para modelar y diseñar sistemas. Este lenguaje de diseño es ampliamente usado y existe bastante bibliografía y software que nos servirán para modelar el sistema, razón por la cual fue escogido para diseñar el mismo. También hubo un gran apoyo en los patrones de diseño que son realmente prácticas y soluciones que en general han dado resultado para diseñar adecuadamente los sistemas y que tomando dichos patrones en cuenta se busca tener diseños óptimos y sobre todo soluciones ya probadas a ciertas situaciones que nos evitarán reinventar formas de atacar problemas que son muy generales.

Para el diseño de la base de datos se utilizo básicamente el modelado entidad-relación en el cual se modelan los datos que será necesario guardar en la base de datos agrupados de acuerdo al establecimiento de entidades, a las cuales se les dan atributos y se comienza a generar relaciones entre las mismas, estas entidades a su vez se convierten en las tablas de la base de datos, los atributos serán las columnas de las tablas y las relaciones en constraints, llaves primarias y foráneas, una vez teniendo este modelo de entidad-relación se optimiza de acuerdo a ciertas reglas llamadas formas normales las cuales son principalmente tres aunque algunos autores manejan 5 formas normales. Este método de normalización es muy usado en bases de datos relacionales para verificar que el diseño sea óptimo ya que reduce principalmente su tamaño al minimizar la duplicación de información y por supuesto el rendimiento al hacer consultas más específicas y directas. Gran parte de un desempeño adecuado de una base de datos proviene de que esté correctamente diseñada y otra parte es la creación de índices y otras acciones que nos lleven a tenerla en un estado óptimo, pero si esta no cuenta con el requerimiento mínimo de estar bien diseñada, no sólo se arriesga el rendimiento de la base, seguramente será muy complicado hacerle cambios, darle mantenimiento y poder hacer extracciones de datos útiles para visualizar ya que servirá como repositorio de información pero no tendrá una estructura sólida.

La implementación se realizó tratando de establecer estándares de programación, como por ejemplo el estandarizar los nombres de las variables de tal forma que se les puso los a tres primeros caracteres la abreviación del tipo de datos que tiene la variable y un nombre adecuado de acuerdo a su función, es importante por ejemplo, el quitar nombres de variables como i o x que no dicen absolutamente nada y que si se encuentran a lo largo de un programa vuelve complicado el darle mantenimiento y hace que sea muy probable que un desarrollador se equivoque fácilmente ya que al carecer de sentido puede darle un uso inadecuado o guardar un valor que altere el flujo normal del programa, otro de los estándares es que todas las variables se les asigne un tipo de datos, este estándar no sólo mejora la legibilidad y la facilidad de mantenimiento, además mejora mucho el rendimiento del sistema. Otro estándar de programación que se adoptó fue el de poner un encabezado a cada función, subrutina o método que se hizo, poniendo en dicho encabezado el nombre del autor, la fecha de la última modificación o creación, la versión y una descripción muy breve de lo que hace este grupo de líneas de código. Esto que en realidad nos quita unos cuantos minutos, nos ayuda tremendamente en el momento que se le da mantenimiento ya que nos indica a quien podemos consultar en caso de un cambio, qué debe hacer la rutina y la última vez que se le agregó código lo cual nos facilita el mantenimiento de la misma y recuperamos por mucho el tiempo gastado en poner un encabezado.

Otra práctica que llevamos a cabo y que en general nos dio muy buenos resultados es el no programar cosas que no fueran diseñadas ya que el tomar decisiones importantes de diseño en el momento que se programa no es una buena práctica debido a que normalmente aunque se puede llegar a un programa que funcione difícilmente tendrá la calidad suficiente en términos de poderse reutilizar y de estar adecuadamente acoplada a su entorno, con esto me refiero a que si no se diseñó adecuadamente es poco probable que tomemos ese código y lo podamos utilizar transparentemente en otro sistema.

Otro estándar que seguimos es el manejo de errores que es muy importante llevarlo a cabo ya que ninguna aplicación está 100 por ciento exenta de fallar, incluso por causas externas a la propia programación del sistema como puede ser que en un momento dado se caiga el enlace a la base de datos y en una situación como esta el sistema no podrá ejecutarse correctamente, por ello es de suma importancia que se tenga un manejo de errores para que el programa no termine abruptamente y nos permita en caso dado tomar algunas acciones para poder manejar una posible contingencia con el sistema.

Existen numerosos tipos de pruebas y para este proyecto se contemplaron las de caja negra principalmente y pruebas de unidades, aunque también se consideraron y efectuaron varias pruebas con los usuarios reales del sistema efectuando ciclos completos de trabajo tomando en cuenta todos los requerimientos y todo el flujo de trabajo que el sistema ofrecerá para encontrar el máximo número de errores antes de liberar el sistema a producción. Normalmente con las pruebas de unidades se van encontrando los errores en funciones y en clases que naturalmente al detectarlos a tiempo facilitan la corrección y nos ahorran mucho tiempo ya que las revisiones son de partes más pequeñas de funcionalidad, las pruebas de caja negra nos ayudan a revisar si un proceso se realiza adecuadamente, es decir, si a una entrada específica corresponde la salida esperada no importando que flujo siga el programa, este tipo de pruebas nos ayuda a visualizar si con ciertas entradas específicas la parte del sistema que se está probando va a generar la salida esperada, en dado caso se van integrando los componentes o módulos a la aplicación, en caso contrario se corrigen dichos módulos, finalmente como se mencionó en las líneas anteriores, se realizaron varios ciclos completos y reales con datos tomados del sistema anterior y complementados con datos que si bien no eran del todo específicos para esas órdenes si eran reales ya que se pidió la ayuda del personal del área para complementarlos y se comprobó que el sistema si arrojaba los datos esperados.

Ahora bien, los productos utilizados fueron todos productos Microsoft ya que se tienen licencias "Small Business" de esta compañía y los sistemas operativos tanto del cliente como de los servidores desde antes de desarrollar este conjunto de sistemas son bajo plataforma MS Windows. Razón por la cual se decidió desarrollar sobre esta plataforma. Como manejador de base de datos se utilizó MS SQL Server 2000 que es un manejador de base de datos que ha mejorado mucho desde su primera versión y que en esta penúltima versión es un manejador robusto, no es quizá el mejor manejador de bases de datos pero las necesidades de la empresa las soporta bastante bien pudiendo brindar un adecuado rendimiento de los sistemas a crear y su costo es razonable para la cantidad de usuarios que se tendrán. También cabe mencionar que sobre todo este producto está muy ligado a la plataforma Windows por ser de la misma compañía y naturalmente cuenta con todo el soporte y apoyo para este manejador de base de datos por parte del sistema operativo; además de que a pesar de que no es como tal el manejador más rápido ni más poderoso, ni más configurable ni el que más datos soporta, pero tiene tres cualidades muy importantes por las cuales se tomó la decisión de utilizarlo. La primera es que a pesar de que no es tan configurable como podría ser por ejemplo Oracle, precisamente el hecho de tener muchas configuraciones y manejos automáticos nos aligera la administración de la base de datos y esto nos ayuda bastante ya que al ser una empresa mediana no se tienen los recursos ni las necesidades tan apremiantes de tener un DBA de tiempo completo, entonces lo que aparentemente podría ser una desventaja se puede tener como una ventaja dado también el tamaño y recursos destinados a este conjunto de sistemas en particular. La segunda ventaja es el convenio de licencias que ya se tiene con la compañía Microsoft que puede ser ampliado y que de alguna manera nos hace permanecer en una línea de productos bastante soportada y comercial, con amplio soporte y abundancia de consultoría en el mercado. Como tercera ventaja seria que para la cantidad de datos que se van a manejar a mediano plazo el costo-beneficio de utilizar este producto es bastante bueno.

Para mantener este esquema se decidió utilizar también productos Microsoft para el desarrollo y en este caso se utilizó la versión 6 del MS Visual Studio, la cual en estos momentos no es una plataforma que contenga tecnología de punta pero que es bastante soportada por todos los sistemas operativos de la plataforma MS Windows y la razón de usar estas herramientas es que ya se había comprado desde antes de que este desarrollo iniciara y también este es un producto con el cual los desarrolladores tienen mucha

experiencia, en este caso considero que se llevó casi al extremo el hecho de utilizar las herramientas con las que se cuenta y no invertir más ni en capacitación ni en nuevas licencias, pero aunque a largo plazo se deban quizás de hacer migraciones a nuevos productos de desarrollo el sistema para ese entonces por una lado ya habrá devengado su costo y por otro es casi seguro que para entonces sea necesario generar una versión que tenga muchas cosas nuevas ya que las reglas de negocio y la tecnología cambian de manera muy dinámica, además de que el producto usado soporta todas las necesidades actuales del sistema de una manera adecuada, si bien no es lo más nuevo en tecnologías.

Para el diseño del sistema (diagramas sobre todo), nos apoyamos en MS Visio 2003 que es parte de la suite de MS Office 2003 que nos permitió generar los diagramas y esquemas principalmente. Se inició el uso de diagramas con una versión de Rational Rose pero posteriormente se cambió a MS Visio. Visual Studio tiene un "plug-in" para ambos productos y a partir de los diagramas se pueden generar las clases con sus métodos y propiedades por supuesto esto solo nos facilita poder generar solamente la estructura de las clases, pero ayuda grandemente a que haya una sincronía entre nuestros diagramas y el código que se va a escribir para implementar el sistema.

3.1.5 Conexiones e interacción con otros sistemas

El sistema interactuará con otros sistemas, principalmente con el de almacenes ya que de estos tomará los materiales y se ingresarán los productos ya fabricados también hacia el mismo. Indirectamente también tiene que ver con la contabilidad al generarse unos costos de producción que hasta el momento serán únicamente contemplando materiales y en posteriores versiones se le agregarán los de mano de obra y otros gastos indirectos.

Básicamente la interacción que este sistema tendrá en un inicio, será la de compartir con los demás sistemas información, pero es probable que más adelante se puedan llegar a generar interfaces para que interactúen a nivel programático entre sistemas, con esto me refiero a que por ejemplo tal vez en un futuro desde el módulo de producción se pueda generar una compra en el sistema de compras automáticamente cada vez que hagan falta materiales por citar un ejemplo y de esta forma tener interacciones a nivel programas entre todo este conjunto de sistemas para la empresa. Esta funcionalidad en este momento está fuera de los alcances de la presente versión pero es probable que para futuras versiones se pueda llegar a solicitar este tipo de requerimientos.

3.2 Infraestructura con la que se trabajará

Se cuenta con una red Lan de 1Gbps, que brinda soporte a las comunicaciones internas de la empresa, existe también un acceso a Internet y además hay 3 servidores los cuales se usan de la siguiente manera; se tiene un servidor de base de datos, el cual consta de 2 procesadores Pentium IV a 2.4 GHz y contiene un arreglo de discos de 120 GB. Existe otro servidor de aplicaciones de características similares, en el cual radicarán las aplicaciones y otro servidor de archivos en el cual se guardan y respaldan todos los documentos que se consideran importantes y se tiene básicamente un servidor de archivos. Estos servidores tienen licencias de Microsoft Windows 2000 Server y el de base de datos cuenta con la

licencia de MS SQL Server 2000. Se tiene en el lado del cliente varias máquinas PC con diferentes versiones de MS Windows principalmente con Windows 2000 y con Windows XP. Esta es la estructura principal con la que se trabajará.

3.3 Diseño de Base de Datos

La base de datos que se utilizará es una base de datos relacional, por lo cual se usarán tablas y relaciones para organizar la información, se utilizó el modelado de entidad–relación para modelar la base de datos. Posteriormente para verificar que nuestras entidades y relaciones fueran correctas y sobre todo óptimas, se utilizaron los tres primeros postulados de un método de verificación llamado también postulados de normalización.

La normalización convierte una relación en varias sub-relaciones, cada una de las cuales obedece a reglas. Estas reglas se describen en términos de dependencia.

La normalización es un proceso que clasifica relaciones, objetos, formas de relación y demás elementos en grupos, en base a las características que cada uno posee. Si se identifican ciertas reglas, se aplica una categoría, si se definen otras reglas, se aplicará otra categoría.

Estamos interesados en particular en la clasificación de las relaciones de las bases de datos relacionales. La forma de efectuar esto es a través de los tipos de dependencias que podemos determinar dentro de la relación. Cuando las reglas de clasificación sean más y más restrictivas, diremos que la relación está en una forma normal más elevada. La relación que está en la forma normal más elevada posible es la que mejor se adapta a nuestras necesidades debido a que optimiza las condiciones que son de importancia para nosotros:

- La cantidad de espacio requerido para almacenar los datos es la menor posible.
- La facilidad para actualizar la relación es la mayor posible.
- La explicación de la base de datos es la más sencilla posible.

Existen básicamente tres niveles de normalización: Primera Forma Normal (1NF), Segunda Forma Normal (2NF) y Tercera Forma Normal (3NF). Cada una de estas formas tiene sus propias reglas. Cuando una base de datos se conforma a un nivel, se considera normalizada a esa forma de normalización. Por ejemplo, supongamos que su base de datos cumple con todas las reglas del segundo nivel de normalización. Se considera que está en la Segunda Forma Normal. No siempre es una buena idea tener una base de datos conformada en el nivel más alto de normalización. Puede llevar a un nivel de complejidad que pudiera ser evitado si estuviera en un nivel más bajo de normalización.

Primera Forma Normal: La regla de la Primera Forma Normal establece que las columnas repetidas deben eliminarse y colocarse en tablas separadas.

Por ejemplo esta forma normal se siguió cuando establecimos la tabla DFORM para almacenar los diferentes productos de que está formada una fórmula en lugar de poner varias columnas para colocar los contenidos de estos componentes.

Segunda Forma Normal: La regla de la Segunda Forma Normal establece que todas las dependencias parciales se deben eliminar y separar dentro de sus propias tablas. Una dependencia parcial es un término que describe a aquellos datos que no dependen de la clave de la tabla para identificarlos. Por ejemplo, en la base de datos, la descripción del producto sólo se encuentra en la tabla de productos, si estuviera en las tablas de fórmulas, órdenes o cualquier otra tabla entonces no se cumpliría con la segunda forma normal.

Tercera Forma Normal: La regla de la Tercera Forma Normal señala que hay que eliminar y separar cualquier dato que no sea dependiente de la clave. El valor de esta columna debe depender de la clave. Todos los valores deben identificarse únicamente por la clave. Para este caso cada dato en las columnas depende de la clave de cada una de las tablas por lo que se puede decir que nuestra base de datos está en la tercera forma normal, lo cual nos garantiza que esta optimizada.

La siguiente figura muestra el diagrama Entidad-Relación de la base de datos y nos muestra como se diseñó la estructura de base de datos, esto es, nos indica que tablas, campos y relaciones existen entre ellas. Ahora bien, esto es para su diseño, pero una vez puesta en producción dependiendo de mediciones de rendimiento se podrá decidir crear índices, el manejo de valores nulos y otro tipo de restricciones que ya son meramente parte del mantenimiento del sistema y que normalmente las realizan los administradores de bases de datos.

El adecuado funcionamiento de las bases de datos es muy importante para el rendimiento del sistema pues cuando una base de datos no está optimizada el rendimiento del sistema puede volverse muy bajo ya que aunque la aplicación funcione adecuadamente en el momento de manipular la información la capa de datos puede convertirse en un cuello de botella y en casos extremos donde se configura el tiempo de espera de una consulta puede llegar incluso a provocar fallos en el sistema debido a los altos tiempo de respuesta de la base de datos, así que aunque normalmente cuando se tienen pocos datos funcionen bien las bases de datos es necesario diseñarlas y mantenerlas en óptimas condiciones para de esta forma asegurar que nuestro sistema funcionará de manera adecuada a lo largo de su vida útil.

A continuación se muestra el diseño de base de datos para el Caso de Uso generar ordenes que es el ejemplo visto a lo largo del presente trabajo.

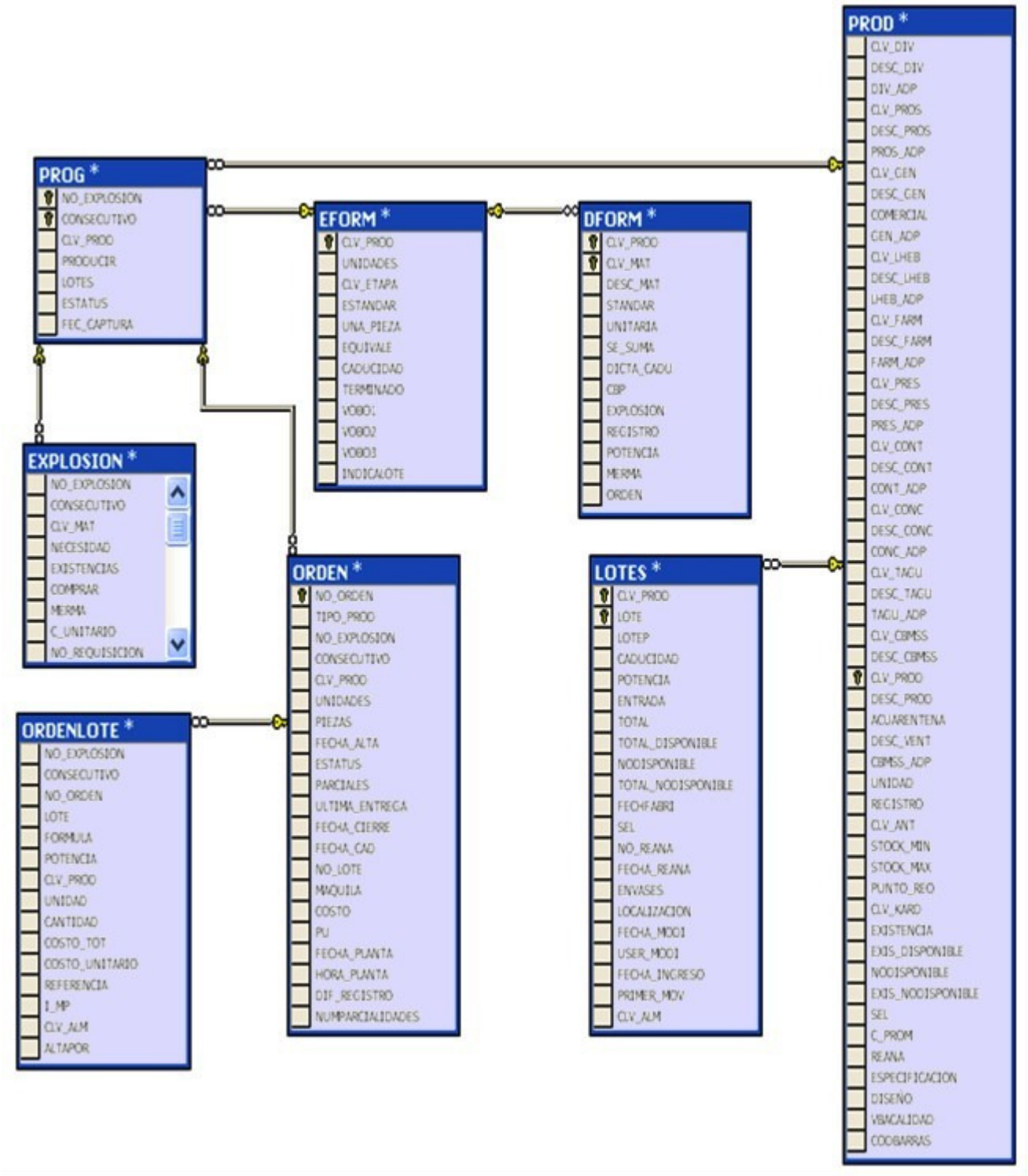


Fig. 130.- Diseño de Base de Datos para Caso de Uso Generar Ordenes.

3.4 Diseño del sistema

Lo primero que se hará por ciclo de desarrollo es escoger los Casos de Uso o versiones de Casos de Uso (en caso de que el caso o casos sean muy complejos) que se diseñarán para dicho ciclo.

Una de las cosas a diseñar dentro del sistema son las interfases, las cuales pueden ser diseñadas de maneras diversas, en nuestro caso el diseño lo hicimos en MS Visio pero también se pudo haber hecho uso de un prototipo no funcional en el producto elegido para desarrollar, lo cual nos puede acercar mucho a la vista final que tendrá el sistema e inclusive se puede directamente utilizar dicho prototipo para conectarlo con la capa del negocio.

En este caso se hizo un diseño de pantallas las cuales se mostraron al cliente para ver si eran funcionales, si contenían toda la información que se necesita y si esta se presenta de manera adecuada para poder generar la interfaz de usuario, cabe mencionar que es muy importante generar interfases adecuadas ya que para el usuario es la parte del sistema con la que el va a tener que interactuar, así que si no está presentada de manera adecuada, de forma que el usuario la sienta funcional y amigable es muy probable que aunque el sistema este correctamente construido y funcionalmente adecuado, la resistencia del usuario para utilizar el sistema puede llegar a ser muy grande. En esta búsqueda de una interfaz adecuada es importante buscar un equilibrio en el sentido de facilitarle las cosas al usuario, pero sin que ello implique hacer una interfaz tan compleja en el sentido de la programación que lleve demasiado tiempo y recursos, una interfaz adecuada es una interfaz que al usuario se le haga simple trabajar con ella y que cumpla con los objetivos de una adecuada interacción con el sistema.

Independientemente del producto en que se elaboren y de la forma que se diseñen hay que darle importancia a la comunicación con los usuarios ya que son estos los que diariamente interactuarán con nuestro sistema.

La estética en las pantallas o formas de un sistema varían de acuerdo al tipo de usuarios y tipo de sistema que se desarrollará, ya que no es lo mismo por ejemplo hacer unas formas de un sistema cliente-servidor para una empresa que tiene un perfil y usuarios muy específico y donde se le da más peso a la funcionalidad que a la estética que diseñar los formularios para un sitio Web de ventas, en donde además de la funcionalidad y eficiencia de nuestro sistema se tendrá que otorgar un peso mayor a la parte estética ya que dicha página será tomada en cierta medida como la imagen que la empresa quiere proyectar hacia el exterior y lógicamente en esos casos el equipo de desarrollo tendrá que contar con uno o más diseñadores gráficos que colaboren a darle este toque y apariencia profesional y a la vez vistosa al sistema que estamos generando.

Precisamente una ventaja de desarrollar sistemas en capas, es que son menos dependientes a los cambios en la capa de presentación y por lo tanto más flexibles a que en determinado momento se puedan cambiar ciertas partes de los formularios, inclusive formularios completos sin tener que hacer prácticamente nada en la capa de negocios, naturalmente algunas clases que brindan soporte a las formas si deben de ser modificadas o extendidas.

A manera de ejemplo y para establecer una continuidad dentro de este trabajo, se mostrarán en la parte de diseño, las pantallas para el Caso de Uso Generar Orden.

3.4.1 Diseño de Interfaz

En el primer formulario, se escribirá el número de explosión (o mejor dicho de programación), este mostrará en una segunda pantalla, todas las líneas de productos pertenecientes a la programación seleccionada cuyos materiales existan en el almacén en las cantidades suficientes.

En la segunda pantalla, se muestran estas líneas y se podrá seleccionar una prioridad de procesamiento para generar los números de orden de producción, previa validación de que hay las existencias suficientes de los productos que componen la orden.

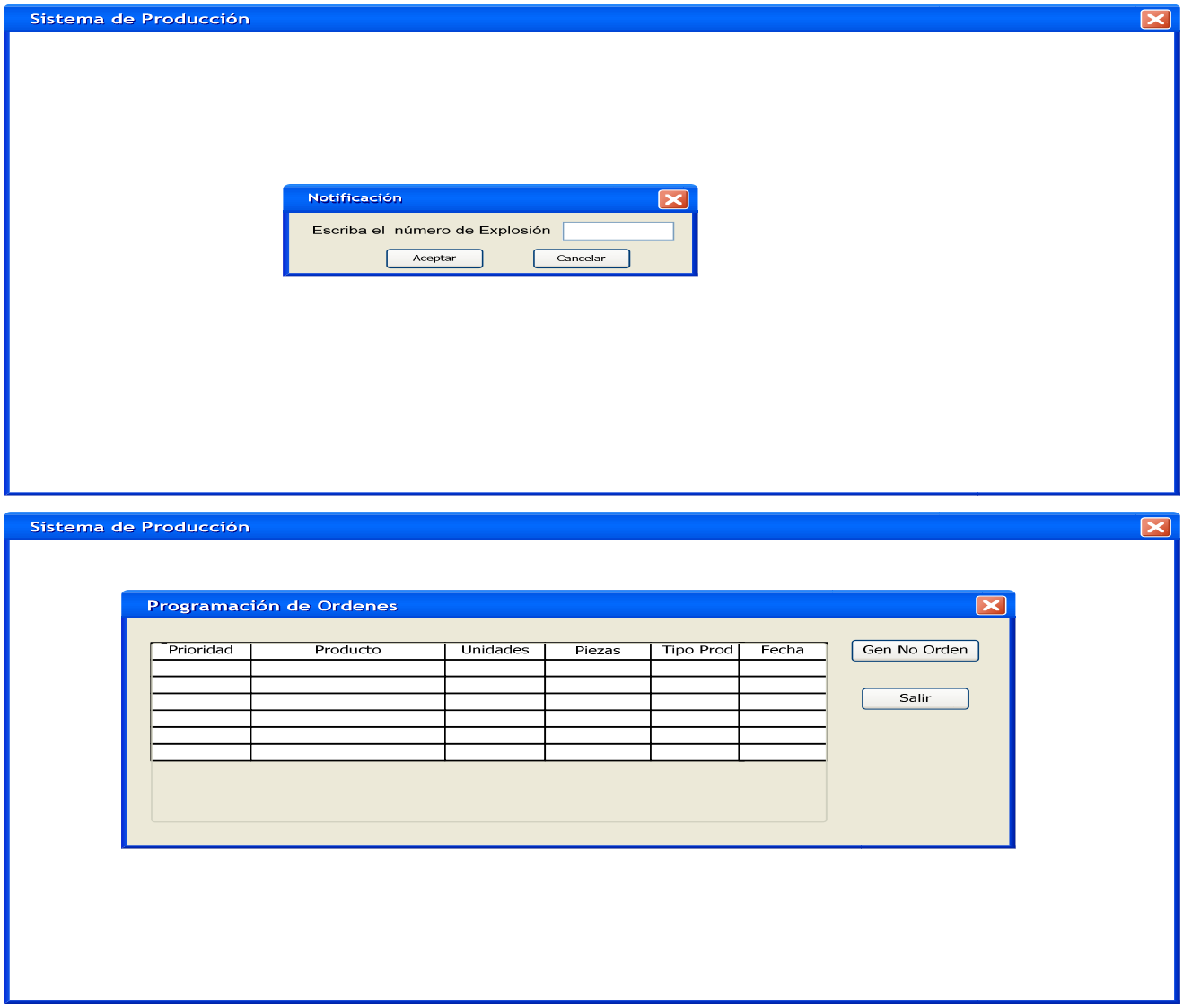
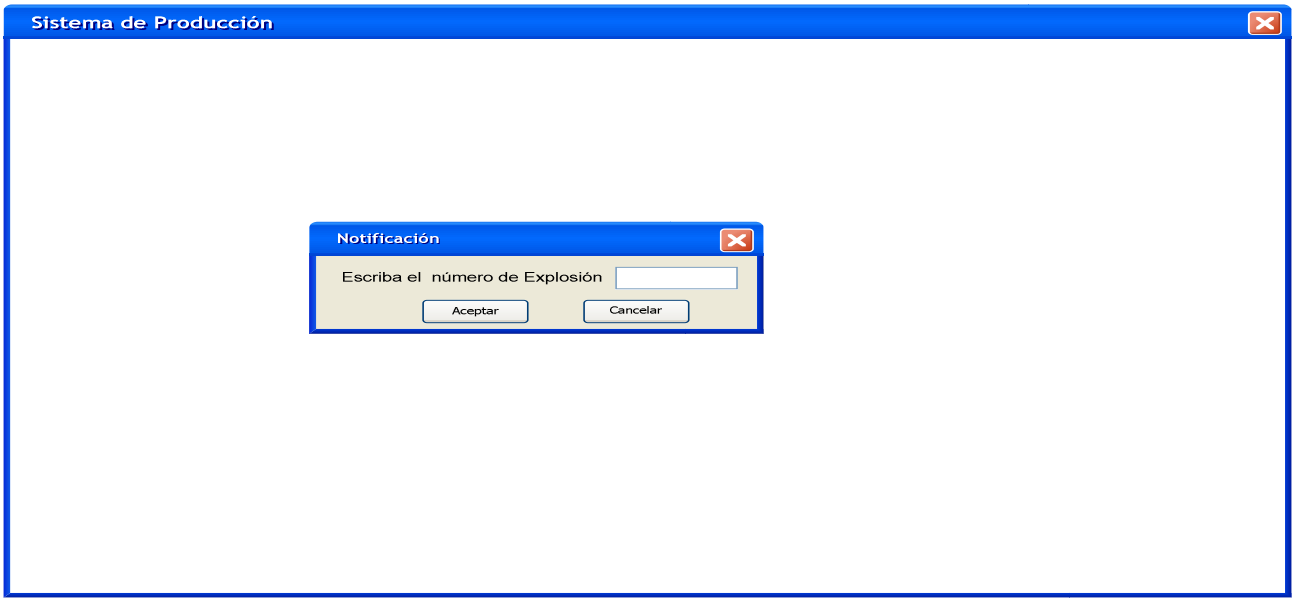


Fig. 131.- Interfaz de usuario para el Caso de Uso Generar Orden (1 de 2).

Una vez que se generaron los números de orden, el usuario deberá volver a ingresar el número de programación para que el sistema nos muestre ahora en esta ocasión todos los números de orden generados para cada una de las líneas de programación que cumplieron con tener las cantidades suficientes de insumos en almacén, es en esta segunda pantalla donde se seleccionarán los números de orden sobre los cuales se quiere producir ya como tal las órdenes de producción, ponerles una fecha de inicio y establecer los calendarios de generación de la orden o incluso devolver los materiales de una orden de producción que aún no haya sido comenzada.



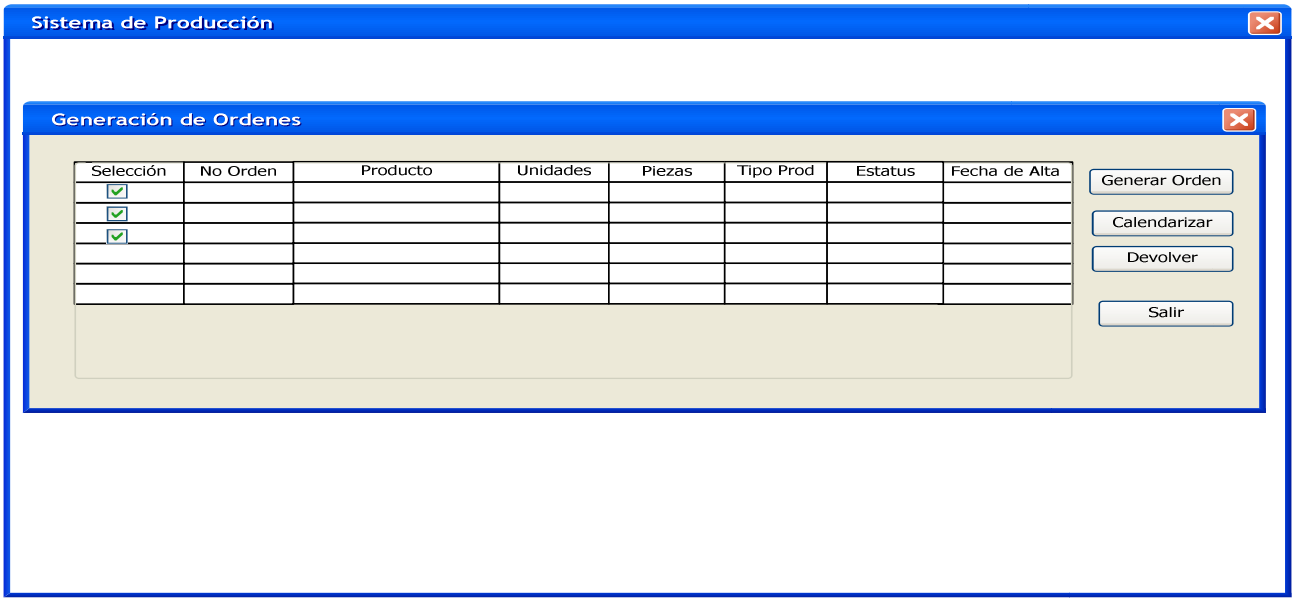
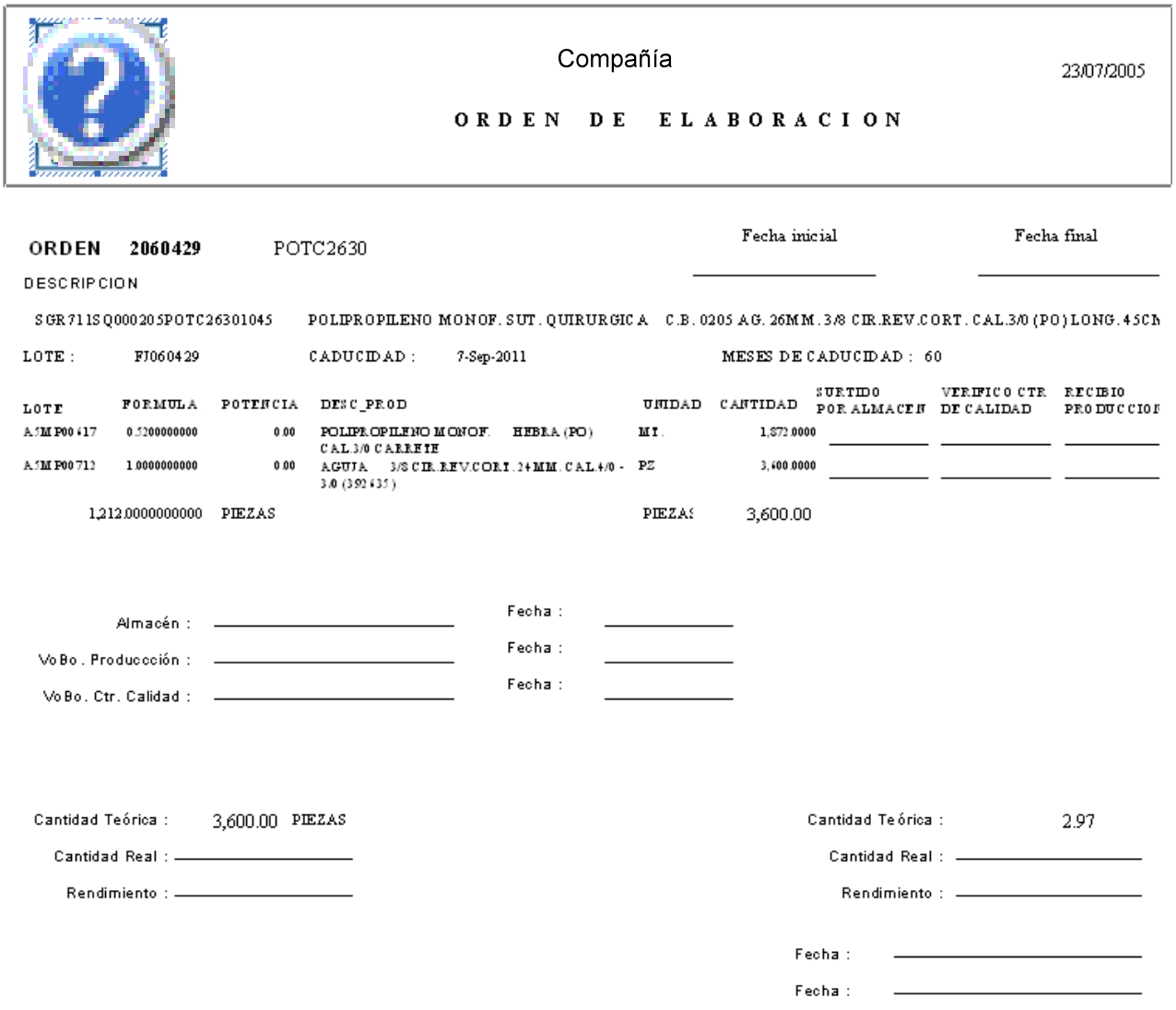


Fig. 132.- interfaz de usuario para el Caso de Uso Generar Orden (2 de 2).

Este es el formato que se propuso y se aceptó para el reporte de generación de una orden de producción, como se puede ver contiene toda la información necesaria para la generación de dicha orden. En este caso es un reporte que nos indica toda la información necesaria para la generación de una orden, pero realmente no se necesita la impresión de dichos reportes ya que el sistema de almacén tendrá dentro de su sistema la facilidad de poder ver las órdenes de embarque que es necesario atender. Este reporte nos ayudará en el caso que se necesite una orden impresa. Pero parte de la finalidad de este sistema también será que al existir una interacción entre los sistemas no sea necesario estar imprimiendo documentos, ni llevar archivos físicos de información, más bien la idea es que se sustituya toda esta documentación por archivos electrónicos en la manera de lo posible y de esta forma utilizar menos papel.



Compañía

23/07/2005

ORDEN DE ELABORACION

ORDEN 2060429 POTC2630 Fecha inicial Fecha final

DESCRIPCION

SGR711SQ000205POTC26301045 POLIPROPILENO MONOF. SUT. QUIRURGICA C.B. 0205 AG. 26MM. 3/8 CIR.REV.CORT. CAL.3/0 (PO) LONG. 45CM

LOTE : FJ060429 CADUCIDAD : 7-Sep-2011 MESES DE CADUCIDAD : 60

LOTE	FORMULA	POTENCIA	DESC_PROD	UNIDAD	CANTIDAD	SURTIDO POR ALMACEN	VERIFICO CTR. DE CALIDAD	RECIBIO PRODUCCION
A5MP00417	0.5200000000	0.00	POLIPROPILENO MONOF. HEBRA (PO) CAL.3/0 CARRETE	MI.	1,872.0000			
A5MP00712	1.0000000000	0.00	AGUJA 3/8 CIR.REV.CORT.24MM. CAL.4/0 - 3/0 (392435)	PZ	3,600.0000			
	1,212.0000000000	PIEZAS		PIEZAS	3,600.00			

Almacén : _____ Fecha : _____

VoBo. Producción : _____ Fecha : _____

VoBo. Ctr. Calidad : _____ Fecha : _____

Cantidad Teórica : 3,600.00 PIEZAS

Cantidad Real : _____

Rendimiento : _____

Cantidad Teórica : 2.97

Cantidad Real : _____

Rendimiento : _____

Fecha : _____

Fecha : _____

Fig. 133.- Reporte de Generación de una Orden de Producción.

3.4.2 Diagramas de Colaboración

Los Diagramas de Colaboración, como se vio anteriormente, nos permiten mostrar cómo las clases interactúan para lograr cumplir con las funciones del sistema.

Se tendrá un diagrama por evento del sistema o en el peor de los casos por evento que se considere importante del sistema y en caso de querer mostrar alguna función u operación relevante también se generará otro diagrama que muestre las interacciones entre objetos.

Estos diagramas son de suma utilidad ya que nos permiten establecer cómo van a interactuar nuestros objetos, los principales métodos que tendrán nuestras clases, propiedades relevantes y la secuencia de métodos y clases involucradas para atender un evento. Conforme esto se va haciendo para los diferentes eventos, nos podremos dar cuenta antes de codificar de la forma en que serán atendidas nuestras peticiones y encontrar la manera de reutilizar nuestros métodos y nuestras clases ya que estos diagramas son, por así decirlo, una vista de alto nivel de nuestro código, es como los planos de una casa en la cual a pesar de no tener la construcción físicamente, nos vamos dando cuenta de cómo se van dando los espacios, de cómo estas habitaciones por construir irán satisfaciendo nuestras necesidades. De igual forma todos los diagramas de diseño irán mostrando diversos aspectos a cubrir de nuestro sistema, de la misma manera que los diferentes planos (de instalaciones eléctricas, hidráulicos, etcétera) nos van mostrando los diferentes aspectos de nuestra construcción, desgraciadamente la Ingeniería de Software no tiene tanta antigüedad como la Ingeniería Civil y estamos apenas generando las primeras reglas, sentando las bases de lo que es esta rama de la Ingeniería. Aunque considero estamos avanzando a un ritmo bastante grande, ya que apenas hace unos 60 años el software era totalmente artesanal y servia para muy pocas aplicaciones (prácticamente solo aplicaciones militares) y hoy está casi en cualquier parte de nuestra vida cotidiana.

Al igual que el modelo conceptual, los Diagramas de Colaboración deben balancear en beneficio de la utilidad la cantidad de detalles que se manejarán ya que el uso excesivo de los mismos volverá muy complejo el diagrama y posiblemente éstos no sean de gran relevancia, si fuera necesario como se dijo anteriormente, se plasmará en otro diagrama el detalle de una operación o la forma de manejar algúna característica específica.

El primer diagrama corresponde al evento del sistema de Seleccionar Explosión, es importante hacer notar que en dichos diagramas no se muestra como el sistema va a interactuar con la base de datos ya que esta interacción se hizo mediante otra capa y se consideró que el involucrar esta interacción con la capa de datos lo único que haría es agregarle más complejidad innecesaria a los diagramas ya que lo que es seguridad y persistencia (acceso a base de datos) son considerados en una capa de servicios debido a que son de uso extensivo a lo largo de todo el sistema y no son parte de las reglas de negocio como tal.

Otro aspecto a señalar es el hecho de contar con la clase ManejadorGenerarOrden la cual nos ayuda a desacoplar la capa de presentación con la de la lógica de la aplicación y esto nos sirve para no ser tan dependientes de la interfaz y poder reutilizar más fácilmente las clases de la lógica de la aplicación.

Los Diagramas de Colaboración para el Caso de Uso generación de Ordenes se muestran a continuación y en el primer diagrama nos trae las líneas de programación de un número de programación que seleccionemos, es decir, con este diagrama podremos satisfacer la necesidad del usuario de que seleccione un número de programación y el sistema le muestre todos los elementos, por ejemplo, todos los productos ligados a dicho número de programación.

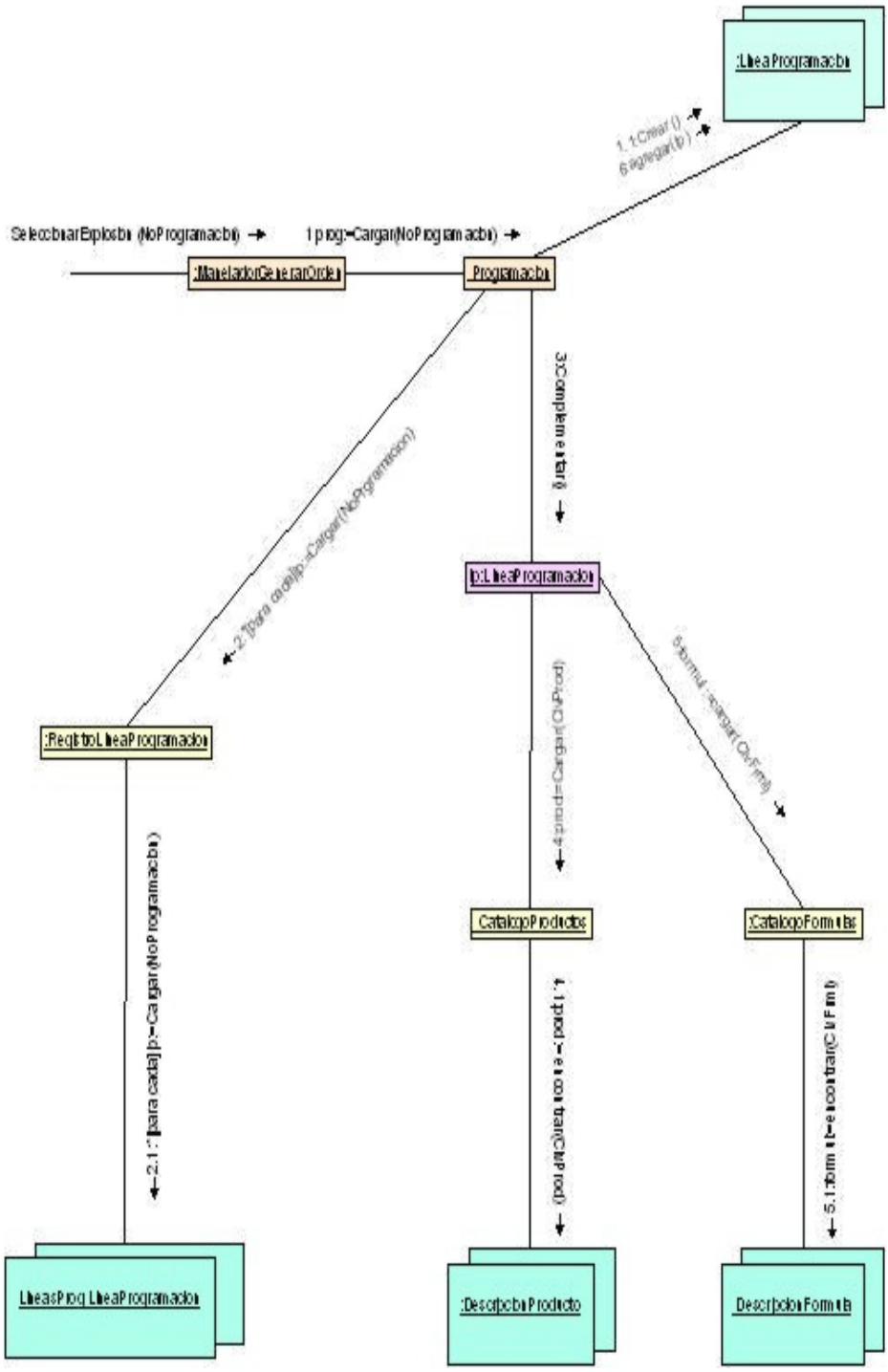


Fig. 134.- Diagrama de Colaboración para el evento SeleccionarExplosion.

El siguiente diagrama es el diagrama de colaboración de la operación de sistema GenerarNumeroOrden(), es decir, en este diagrama se muestra como las clases interactúan para generar los números de orden para los elementos de una programación que son válidos, que cumplen con tener los materiales en almacén.

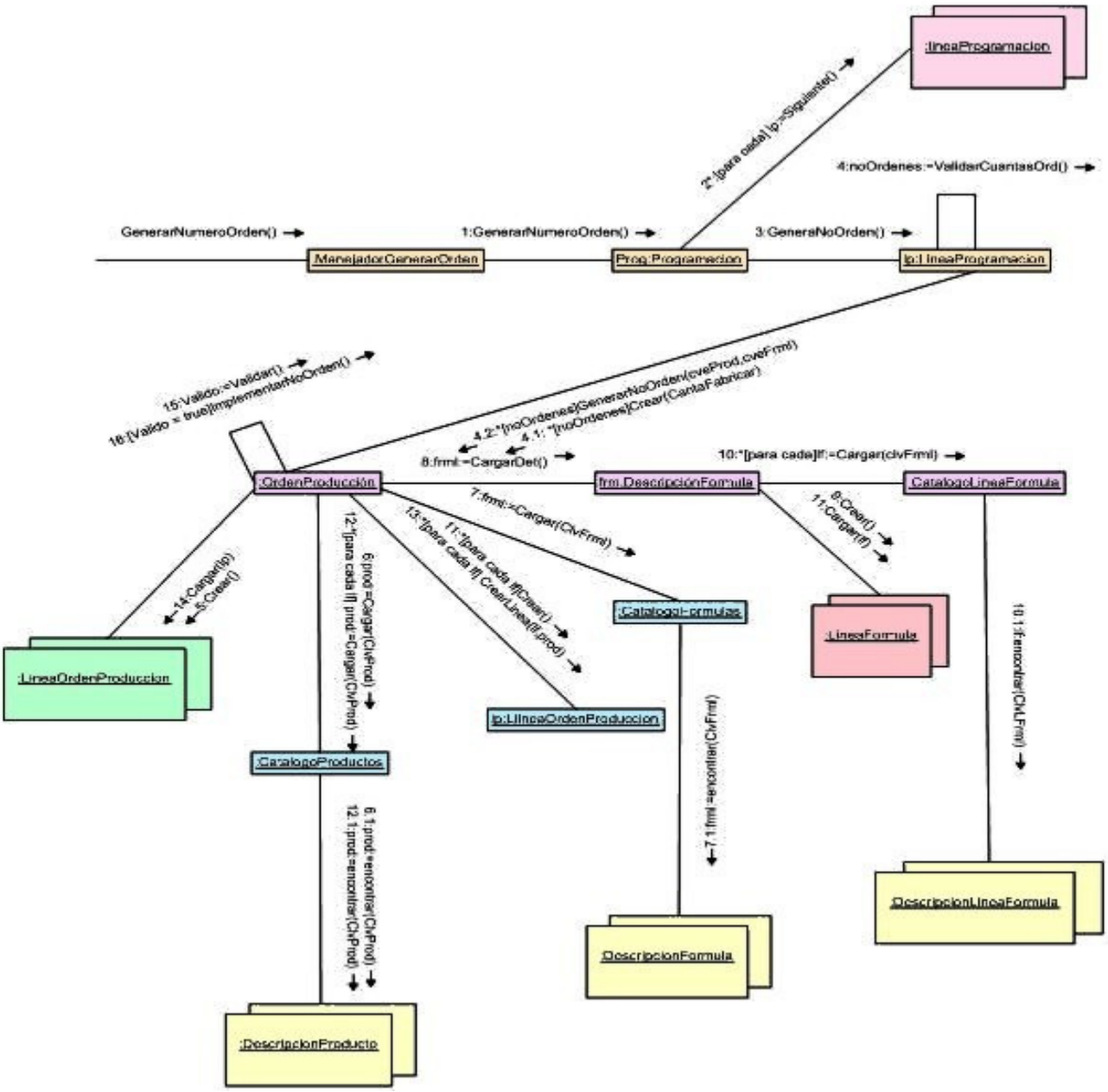


Fig. 135.- Diagrama de Colaboración para el evento GenerarNumeroOrden (1 de 2).

Debido a que la operación validar es compleja, se elaboró un diagrama de colaboración para ésta ya que este procedimiento se considera relevante pero si se incluye en el diagrama anterior le agregará mucha complejidad de tal forma que se consideró una mejor opción hacer un diagrama aparte que se muestra a continuación:

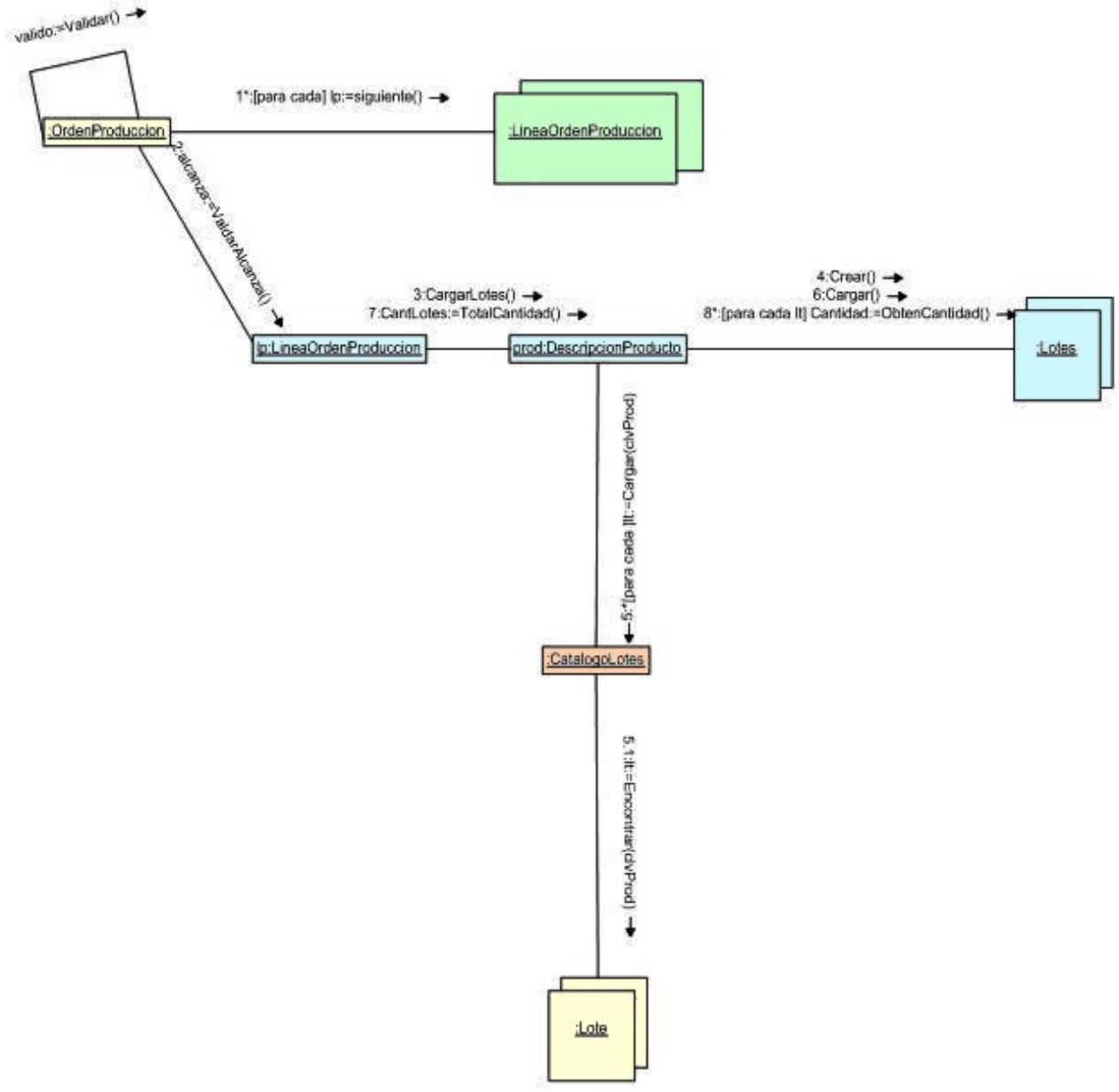


Fig. 136.- Diagrama de Colaboración para el evento GenerarNumeroOrden (2 de 2).

El siguiente diagrama muestra la colaboración de los objetos para el evento SeleccionarOrdenes(), este evento es disparado cuando el usuario selecciona las órdenes que cumplen con las restricciones y que se van a convertir en órdenes de producción ya con productos y lotes seleccionados:

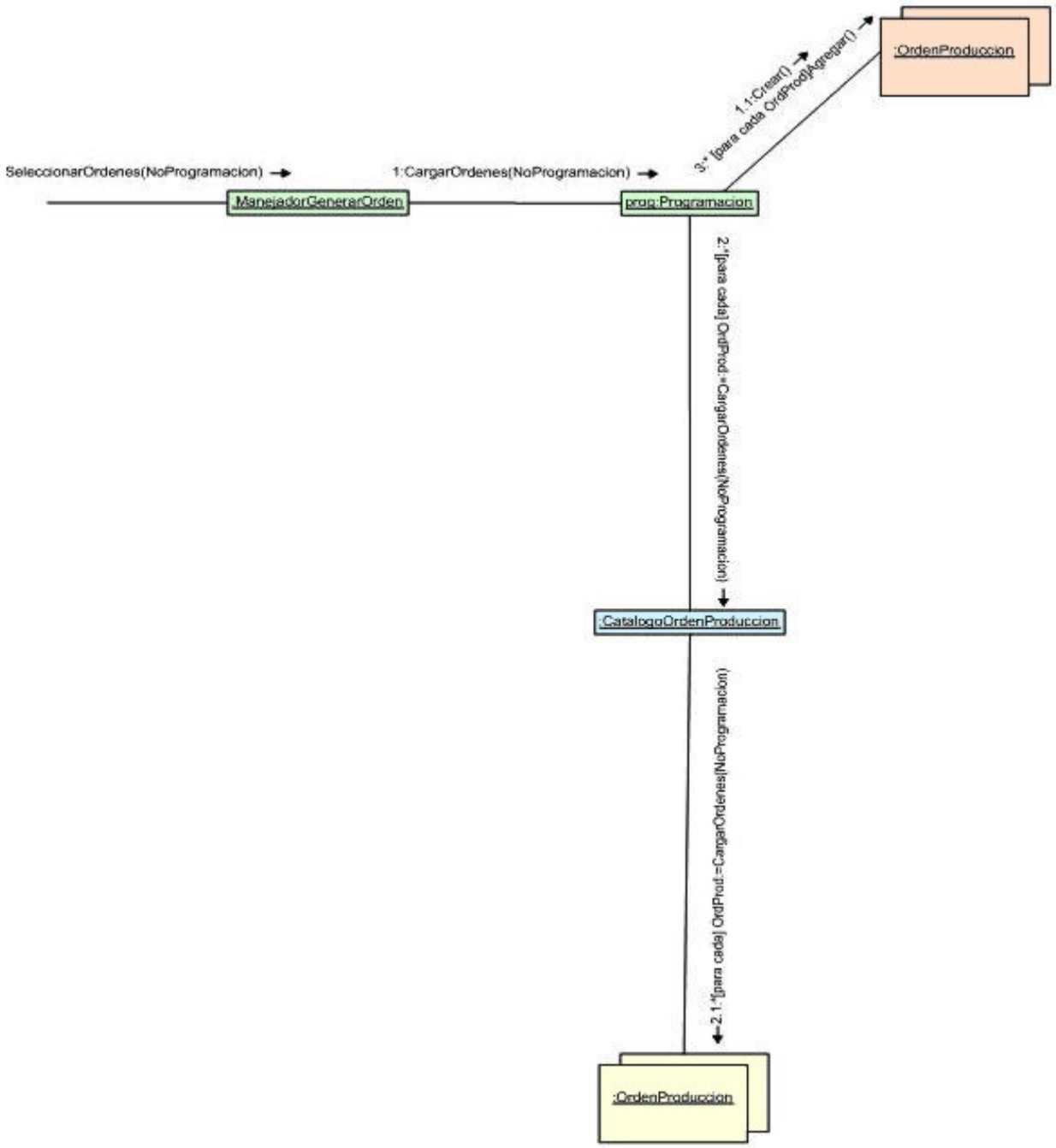


Fig. 137.- Diagrama de Colaboración para el evento SeleccionarOrdenes.

La figura siguiente muestra el diagrama de colaboración que corresponde al evento del sistema GenerarOrden(), aquí propiamente es el evento que genera la orden como tal en la cual se pueden ver todas las clases que intervienen en este evento:

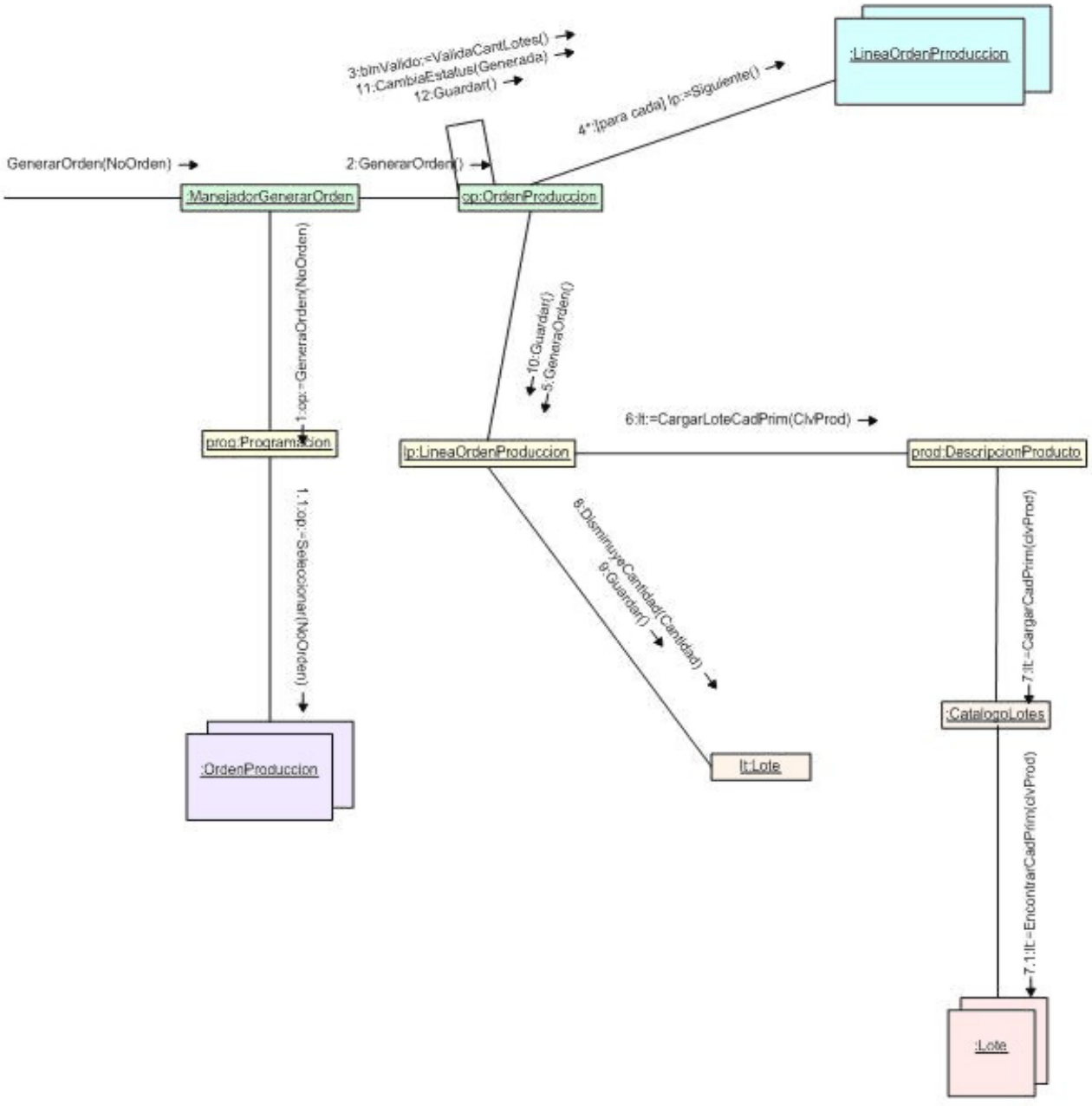


Fig. 138.- Diagrama de Colaboración para el evento GenerarOrden (1 de 2).

Finalmente como la operación de validar las cantidades que tienen los lotes es una operación relativamente compleja decidimos crear un diagrama de colaboración para esta última, este es el diagrama que representa esta operación:

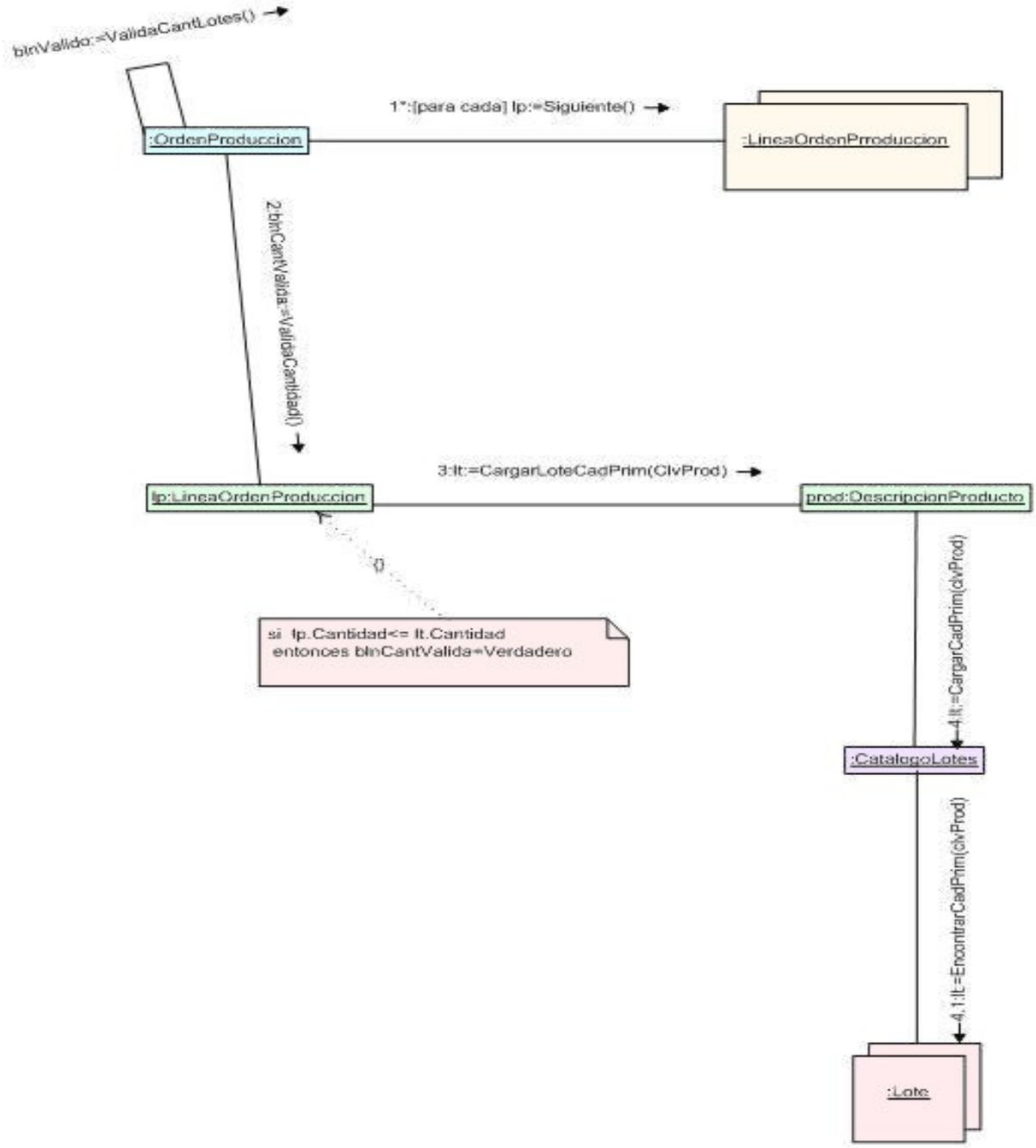


Fig. 139.- Diagrama de Colaboración para el evento GenerarOrden (2 de 2).

3.4.3 Diagramas de Clases

Basándose en los Diagramas de Colaboración se crearon los siguientes Diagramas de Clases, como se muestra a continuación, estos diagramas son muy parecidos a los diagramas del modelo conceptual, pero la diferencia radica en que éstas ya son entidades de software y no conceptos meramente. Aquí se dividió el diagrama para no hacerlo tan grande y que se pudieran apreciar de manera adecuada pero realmente es un solo diagrama que nos muestra las clases, sus propiedades y la manera en que están relacionadas:

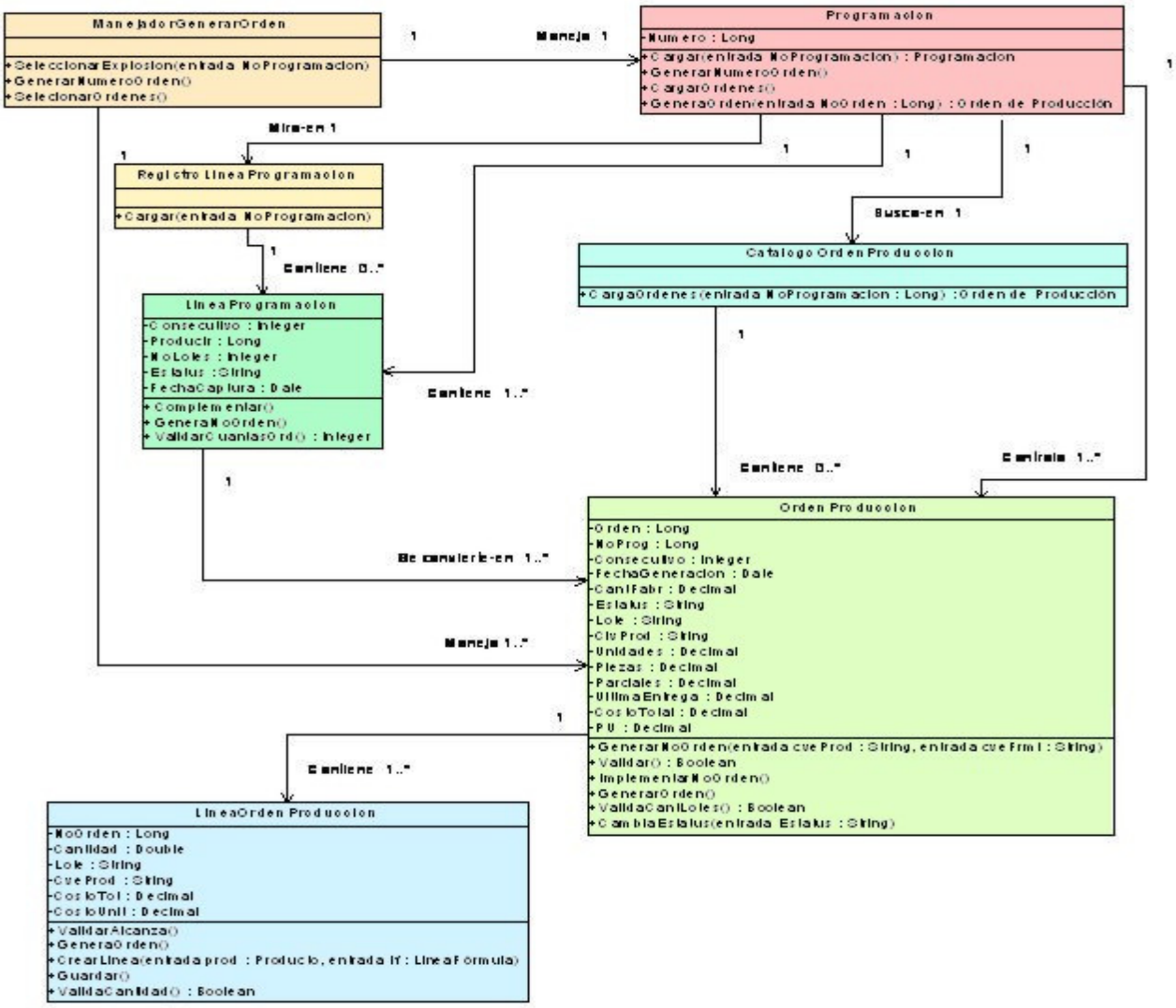


Fig. 140.- Diagrama de Clases para el Caso de Uso Generación de Ordenes (1 de 3).

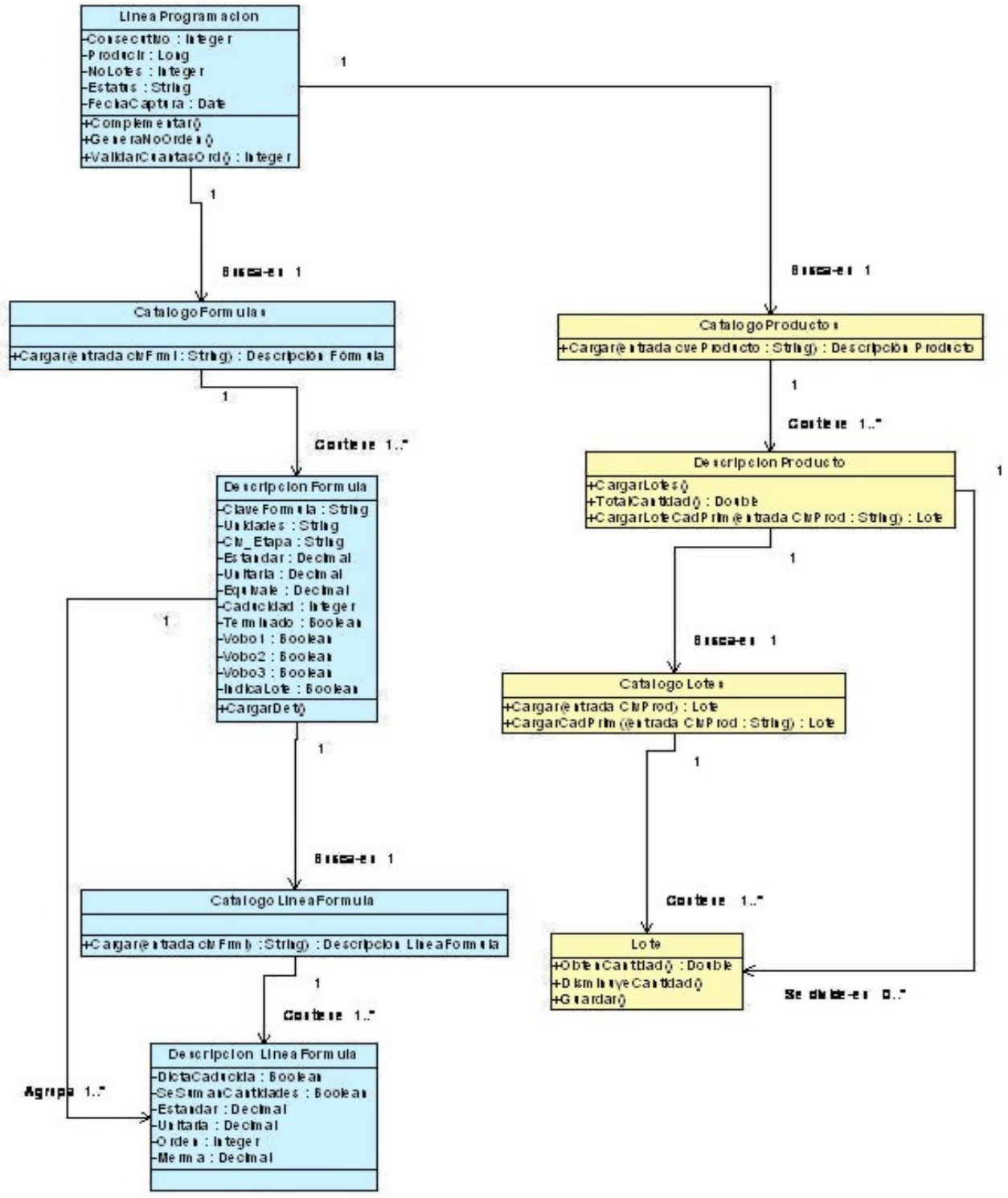


Fig. 141 .- Diagrama de Clases para el Caso de Uso Generación de Ordenes (2 de 3).

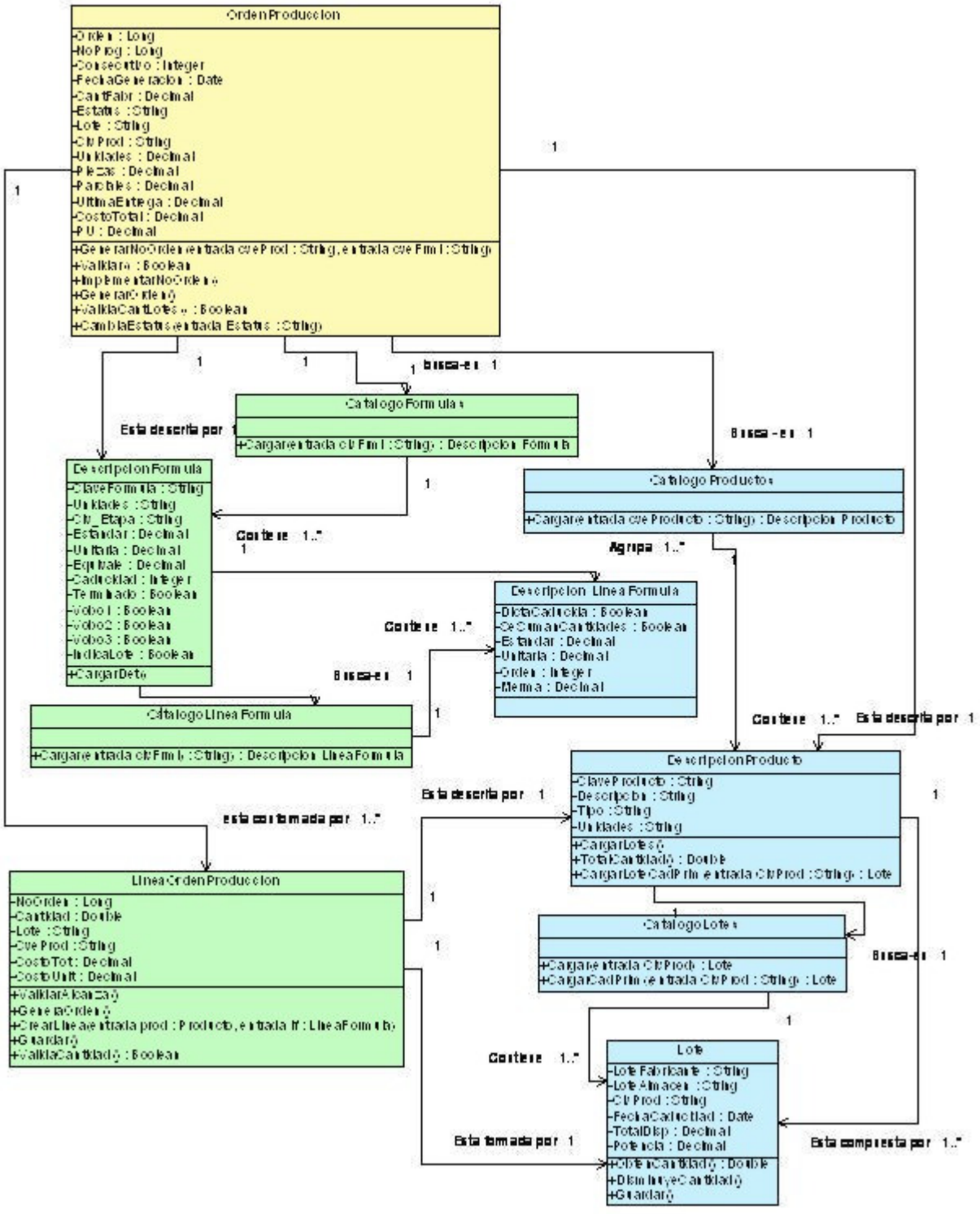


Fig. 142.- Diagrama de Clases para el Caso de Uso Generación de Ordenes (3 de 3).

3.4.4 Arquitectura del sistema

El diagrama siguiente, muestra la arquitectura del sistema que es una arquitectura en 3 capas como se muestra a continuación:

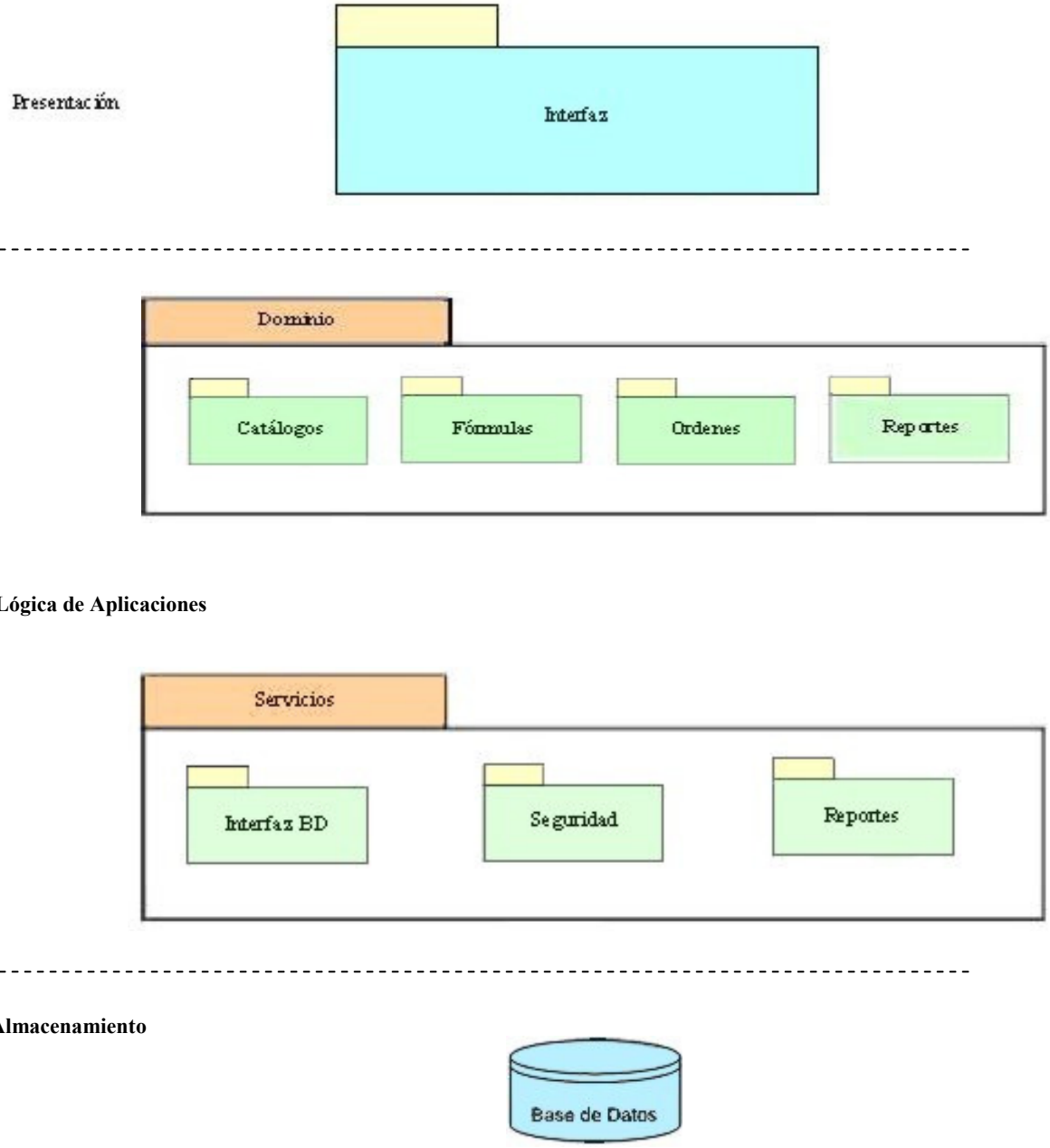


Fig. 143.- Arquitectura del Sistema.

Esta es la forma en que la aplicación se organizó, primero la capa de presentación, la cual incluye a todos los formularios, ventanas y todos los objetos con los que interactúa el usuario. La segunda capa que está dividida en dos, la capa de dominio o de reglas de negocio y la capa de servicios la cual maneja servicios tales como seguridad, acceso a base de datos, reportes, etcétera. Esto es importante porque de esta forma por una parte se aísla la lógica del negocio como tal y los servicios que aunque queden en la misma capa existe una diferencia y esto permite a la larga la reutilización de algunas partes del programa para otras aplicaciones y por otro lado al ser independientes estos servicios además de reutilizables podemos generarlos en paralelo y de tal forma que sean generales. Finalmente tenemos la capa de almacenamiento que es la base de datos de cuyo diseño ya se habló en los párrafos anteriores.

3.5 Pruebas

Normalmente las pruebas no las deben hacer las mismas personas que desarrollaron el sistema, esto debido a que el desarrollador normalmente ya está predispuesto a hacer que el sistema funcione. Dichas pruebas las hizo un equipo de personas diferentes y como el costo de pruebas a pesar de ser una muy buena inversión es una inversión grande y también el tamaño del proyecto no exigía pruebas tan exhaustivas como lo es por ejemplo un producto que va dirigido a un público totalmente abierto como lo es un procesador de palabras, entonces lo que se hizo fue realizar pruebas de caja negra, sin tomar en cuenta las instrucciones y sus trayectorias posibles de falla, más bien se verificó en este tipo de prueba que cada requerimiento fuera cumplido al usar una entrada apropiada, así como las entradas que generaban excepciones. Se tomó como unidad cada función implicada en los Casos de Uso involucrados en cada ciclo de desarrollo conforme fueron implementándose, también se hicieron pruebas de regresión, que son pruebas en las cuales se verifica que la inclusión de un elemento nuevo no afecte los preexistentes y finalmente se probó aproximadamente durante un periodo de 2 meses en el cual el usuario generó en paralelo las mismas órdenes de producción que estaban generando en ese mismo lapso y naturalmente se hicieron los ajustes necesarios para dejar el sistema en óptimas condiciones, además se documentaron y clasificaron los errores de acuerdo a su severidad, es decir, de acuerdo al impacto que ese error tenía en la aplicación.

Baja: Cuando ese error permite continuar la operación y simplemente causa una molestia leve como por ejemplo un error ortográfico en un titulo o una ventana.

Media: Cuando el error si impacta el flujo del proceso pero no es algo que no permita continuar el resto de las operaciones, como por ejemplo podría ser el mal manejo de una excepción que a pesar de que bajo esas condiciones no funciona, el resto de las entradas en ese proceso si realizan una respuesta adecuada y nos permiten continuar con la aplicación.

Alta: Finalmente la severidad alta de un error es cuando el sistema en algún punto no sigue el flujo esperado o presenta un error que no permite que continúe el flujo del programa.

La segunda clasificación que tomamos en cuenta es el tipo de error, si es de interfaz, de lógica o de manejo de errores y excepciones. Cabe mencionar que el hecho de estar probando desde los primeros ciclos nos ayudó muchísimo a que los índices de errores fuesen considerablemente menores en los siguientes, ya que ciertos problemas los aprendimos a evitar desde una fase temprana dentro del sistema, en cambio si se prueba hasta que el sistema está terminado se corre el riesgo de que se aprendan

lecciones demasiado tardías para aplicarse y evitarlas repetir en otras partes del sistema. Se consideró algo importante llevar también un registro de los errores y tipificar dichos errores para tener posteriormente una retroalimentación, así como también tener una retroalimentación y aprendizaje de los riesgos tanto superados como no superados, ya que estos dos aspectos son los que nos sirven como indicadores de cuáles problemáticas se pueden suscitar en un proyecto y al ir documentando en las siguientes fases o en posteriores proyectos puede uno adelantarse a los posibles problemas que uno puede ir encontrando y mejorando al aprender las soluciones que se dieron a determinadas dificultades e inclusive se pueden analizar dichas complicaciones e ir encontrando las mejores soluciones a las mismas pero no cuando se presenten sino antes de que sucedan.

Capítulo 4. Mediciones y Resultados

4.1 Mediciones del sistema

A lo largo de cada ciclo de desarrollo y de cada etapa de estos ciclos, se pueden establecer métricas de productividad, costos, errores entre otras, para nuestro proyecto se escogió tan sólo establecer métricas para errores ya que se consideró que son éstos los que más afectan a que el sistema concluya de una forma exitosa.

Normalmente se da prioridad a cierto grupo de métricas ya que por un lado el estar generando información y formándolas implica esfuerzo y el esfuerzo tiene un costo y por otro lado también depende que utilidad se les dé a las mismas. No tiene sentido agregar costos a un proyecto en reunir métricas que no tendrán utilidad en la mejora de los procesos de desarrollo de sistemas.

Como se mencionó para este caso se hizo una serie de mediciones para ayudarnos a ver los errores que se tuvieron y poder mejorar en este caso los ciclos subsecuentes de desarrollo.

La siguiente figura muestra los resultados de las métricas para el ciclo de desarrollo en que se creó la generación de órdenes de la que anteriormente hemos estado tratando.

Etapa	Resultado
Análisis	
Funciones faltantes por especificar	1
Porcentaje de requerimientos defectuosos	(2/12)= 16%
Porcentaje de requerimientos que se modificaron	1/12 = 8%
Diseño	
Errores severos de diseño encontrados	2
Errores medios de diseño encontrados	1
Errores pequeños de diseño encontrados	4
Promedio de errores por número de métodos	.2
Programación	
Requerimientos no programados	1
Errores graves encontrados	3
Errores medianos encontrados	7
Errores pequeños encontrados	4
Promedio de errores por número de métodos	.4
Pruebas	
Errores graves encontrados	9
Errores medianos encontrados	12
Errores pequeños encontrados	15
Promedio de errores por número de métodos	1.02
Puesta integración (3 meses)	
Errores graves encontrados	1
Errores medianos encontrados	3
Errores pequeños encontrados	12
Promedio de errores por número de métodos	.46

Tabla 20.- Resultados de las Métricas de Caso de Uso Generación de Ordenes.

4.2 Alcance de metas

El sistema fue puesto en producción y demostró ser una herramienta eficaz que automatiza varios procesos y permite generar información que posteriormente puede utilizarse para generar reportes útiles para el área. Además de que interactúa con otros sistemas y eso genera que la información fluya de un área a otra de una forma transparente y automática, esto es debido a que se tiene un repositorio unificado de información. Esto como herramienta nos indica de una forma cualitativa que se cumplió la meta de generar una herramienta útil.

De manera cuantitativa, podemos decir que fueron eliminados al menos 3 procesos por lotes para cargar la información generada a otras áreas. Dichos procesos tardaban una hora al mes cada uno mas el tiempo invertido en estar verificando la información y corrigiendo la misma. Por supuesto, lo más importante en este caso no son estas horas ahorradas, sino la disponibilidad de la información al momento de ser

generada por el área; ya no se tiene que esperar a fin de mes para poder obtener en ciertos reportes la información de cualquier indicador que se desee.

La base de datos fue diseñada para guardar toda la información transaccional tanto del área como de la empresa estableciendo en el diseño el modelo entidad-relación, por lo cual existen dentro de esta base mecanismos que nos permiten mantener la consistencia de la información como son las relaciones entre tablas, las llaves primarias, las llaves foráneas, los índices, los valores por defecto, etcétera, cumpliéndose la meta de generar la infraestructura transaccional de datos.

En general, las aplicaciones actuales en ambientes gráficos son muy intuitivas y fáciles de usar y esta no fue la excepción, así que desde este punto de vista también los usuarios lograron obtener una herramienta con mucho más recursos que les ayudaran a realizar su trabajo en forma más fácil.

El conjunto de problemáticas y desventajas de los sistemas usados anteriormente y puntualizadas en los antecedentes también fue solucionado. Al tener una base de datos unificada para todas las aplicaciones de la empresa, se acabaron los procesos aislados ya que por ejemplo el sistema de producción implementa la entrada y salida de los productos producidos al almacén, esto es en la parte concerniente a la información que se maneja; pero también se puede trasladar a la funcionalidad ya que los objetos usados para realizar este almacenamiento de información son los mismos que se usan en el sistema de almacén. Esto se puede llamar comunicación entre procesos o reutilización de código. Otros puntos que también se solucionaron al unificar la base de datos en una sola para todos los sistemas y que en los antecedentes se mencionaron son la utilización de catálogos estándares ya que un catálogo se usa para todas las aplicaciones como puede ser el catálogo de productos que es utilizado en los sistemas de producción, almacén, compras, etcétera. Ya no se necesita reprocesar información debido a que toda está en un solo repositorio y cada sistema hace uso de ella de una forma transparente y tampoco hay que hacerle cambios manualmente ni está aislada.

En cuanto a tiempos, hubo pequeños desfasamientos ya que hubo algunos problemas imprevistos, como fue la salida de 1 integrante del equipo y algunos problemas técnicos que en algún momento se dieron y que se tuvieron que negociar y llegar a acuerdos de ajustes del plan de trabajo. Básicamente se pueden jugar con 3 factores que cuando se planifica un proyecto se deben tomar en cuenta y que son: el personal o recursos humanos, los alcances del proyecto o características y funciones a desarrollar del mismo y por último registrar en la agenda las actividades. Estas son las variables que al combinarse nos pueden dar el equilibrio para el cumplimiento de metas del proyecto, por supuesto con sus debidas limitaciones y consideraciones de no linealidad, por ejemplo, el incrementar el número de personas en un equipo nos puede ayudar en algún momento a bajar en cierta medida las fechas de entrega o a incrementar el alcance del proyecto pero hasta un grado razonable, porque esto no quiere decir que si duplicamos el personal necesariamente el tiempo de desarrollo se disminuya a la mitad, incluso el incrementar el número de personas en un equipo de desarrollo puede llegar a ser contraproducente porque la productividad del equipo puede disminuir al haber demasiados participantes, pudiendo presentarse problemas en la comunicación, así como también se debe tomar en cuenta que al ingresar un nuevo integrante tiene forzosamente una curva de aprendizaje del proyecto antes de ser productivo.

Como se dijo anteriormente, hubo pequeños retrasos pero realmente en un porcentaje bastante bajo, de acuerdo a lo estimado en los planes de trabajo. Así que desde este punto de vista se cumplieron también las metas en tiempo, porque además hubo en este proyecto una planificación, una prevención de riesgos, una comunicación estrecha con el cliente que permitió transmitir además de confianza en lo que

estábamos haciendo una plena conciencia de que las metas se fueron cumpliendo conforme las fases y las iteraciones del proyecto fueron avanzando, además de la mejora en cada una de estas fases conforme se fueron superando los obstáculos.

Considero que esto último de que el cliente siempre esté enterado de cuál es el estado de nuestro proyecto y que vaya utilizando los primeros módulos o las primeras versiones conforme vayan saliendo es algo que puede considerarse como algo de poca importancia ya que el sistema irá cambiando conforme pasen las iteraciones y el resultado final puede llegar a ser diferente y puede resultar más laborioso crear un ambiente donde el usuario pueda interactuar, pero desde la perspectiva del usuario y del cliente de que puede en todo momento ver cómo el sistema va progresando y llenándose de funcionalidad, es algo importante desde esa perspectiva, ya que con ello el cliente y el usuario van viendo los avances y no tienen la percepción de no saber que sucede y en qué estado está el proyecto hasta que éste acaba.

Conclusiones

De este trabajo puedo concluir que se obtuvieron resultados satisfactorios debido a que el área puede generar de una forma más eficiente sus órdenes de producción al poder revisar en el momento si existen los insumos para la orden, al generar en el momento de crear la orden de producción la solicitud al almacén para la salida de las materias primas, con la seguridad de las existencias y los lotes adecuados.

También la administración de las mismas se mejoró ya que se tiene mucha información de las órdenes y se puede saber en cualquier momento que estatus tiene cualquier orden, que materias primas faltan para producir algún medicamento y en que tiempos se logran completar los subprocesos para generar una orden de producción.

Además, los procesos son más fluidos ya que en el momento en el que se genera la orden se genera también la solicitud de salida de materiales en el almacén y con esto se hace más fluido el trabajo del área, otra mejora a este flujo es que cuando hace falta algún material en ese momento el sistema lo indica para que se haga una solicitud al área de compras.

El incremento en la eficiencia es notable ya que al saber con exactitud los materiales que hacen falta se pueden ahorrar hasta 2 días en hacer la petición lo cual si se considera que un material puede llegar a tardar en promedio 15 días debido a que las sustancias que se manejan son importadas se tiene un ahorro de 12%.

Por otra parte el sistema utiliza un repositorio unificado de datos al compartir con los demás sistemas una base de datos única.

Las ventajas que esto conlleva son las siguientes:

- Diseño único de un repositorio de datos para todas las áreas
- Catálogos estándares para las diferentes áreas
- Fuente de datos única para la diferente información manejada

Esto nos beneficia en varios aspectos, en primer lugar prácticamente se eliminaron los errores en el flujo de información de un área a otra, los procesos son más robustos y fluidos ya que por ejemplo una salida de materiales es registrada una sola vez para el área de almacén, de producción, contabilidad etcétera. No existen diferentes claves para el mismo producto o entidad, es decir, un material tiene la misma clave para todas las áreas y finalmente se registran todas las transacciones tanto del área como de la empresa en una sola base de datos.

Esto además permitió que dejara de haber procesos aislados entre sistemas y por tanto el tiempo que consumían estos procesos se redujo al 100%. Con el beneficio adicional de que la posibilidad de introducir errores debido a estos procesos también se descarta.

Considero se utilizó una tecnología adecuada, que contiene los avances necesarios para hacer un sistema eficiente. Por ejemplo, la tecnología utilizada soporta el diseño orientado a objetos, tiene interfases gráficas, la base de datos está basada en el modelo relacional, tiene un sistema integrado de seguridad y administración de roles que nos permite delegar y apoyarnos en estos esquemas para generar desarrollos más fácilmente.

También se aprovecharon las licencias que ya se tenían para poder disminuir costos del proyecto, esto es importante señalarlo porque considero que aunque normalmente es deseable que los sistemas contengan la tecnología más actualizada para disminuir el impacto del rápido cambio en la tecnología, también es importante considerar los costos que nos trae usar nueva tecnología y el aprovechamiento que se le dará a la misma.

El sistema fue desarrollado utilizando las técnicas de Ingeniería de Software que consideramos más adecuadas dado el problema que teníamos que resolver y también tomando en cuenta nuestra experiencia en alguna de estas técnicas. No dudo que otro equipo de desarrollo pudiera resolver la problemática de desarrollar este sistema utilizando otros métodos en algunos casos, como por ejemplo utilizando el modelo en cascada como el proceso de desarrollo del sistema, pero como lo mencioné anteriormente, hay muchas formas de solucionar un problema y las técnicas finalmente son opciones que se nos brindan para que podamos lograr nuestras metas. Lo importante aquí es construir de una manera eficiente un sistema útil.

El utilizar un proceso de desarrollo, generó también que las reglas del negocio se definieran en caso de no existir y esto a su vez ayudó a robustecer los procesos utilizados en la empresa, no sólo los productivos sino los administrativos. Al generar una lista de requerimientos y un análisis del sistema no sólo nos ayudó a nosotros a definir de una forma precisa el funcionamiento del sistema, también al área le ayudó a tener una retroalimentación de sus procesos y volver a plantearlos y redefinir algunos a detalle y como consecuencia se robustecieron las reglas de negocio y los procesos del área.

Un factor que concluyo es importantísimo en este trabajo es el humano. A pesar de que este tipo de trabajos son muy técnicos y puede uno perderse en ese clase de aspectos; considero de suma importancia se cuiden los procesos de comunicación, tanto los que existen dentro del equipo de desarrollo como los que hay entre éste y los clientes, usuarios, expertos del negocio etcétera. El cuidado que se tuvo en este sentido considero, fue pieza clave en el éxito del sistema ya que del equipo hacia el exterior, hubo mucho apoyo por parte de los usuarios y expertos en el negocio, al ser una empresa pequeña todas las personas se conocen y llevan un trato más cercano, esto nos ayudó y aunado a un esfuerzo grande en la gestión, permitieron que la comunicación fluyera adecuadamente. Internamente dentro del equipo de trabajo también fue muy importante que las personas que estuvieron a cargo del proyecto ejercieran su rol de gestión y liderazgo de una manera adecuada.

Con el paso que se dio el área y la empresa misma robustecieron su infraestructura informática y a partir de estos avances se pueden dar pasos tendientes a continuar optimizando el trabajo y también a obtener más información que nos dé las bases para la toma de decisiones como podría ser generar un sistema OLAP (On-Line Analytical Processing) o que en algún momento ciertos datos se publiquen en Internet. Contando con un sistema que tenga información confiable se pueden comenzar a dar otros pasos en la automatización de los procesos y también en la generación de nuevos sistemas que nos ayuden a satisfacer otras necesidades del área y de la empresa.

Finalmente concluyo que los sistemas son herramientas muy complejas que tienen que estar basadas en un conjunto de metodologías sólidas para permitir llegar a una solución adecuada

Glosario

Abstracción	Acción de concentrar las cualidades esenciales o generales de cosas similares. También, las características esenciales resultantes de una cosa.
Acoplamiento	Dependencia entre elementos (generalmente tipos, clase y subsistemas), normalmente debida a la colaboración entre ellos para prestar un servicio.
Actor	Un papel específico adoptado por el usuario de una aplicación mientras participa en un caso de uso.
Administración de la Configuración (AC)	Proceso de mantener y administrar las diferentes versiones de los distintos artefactos de un proyecto de software
Administración del proyecto	Proceso de satisfacer la responsabilidad de una terminación exitosa de un proyecto.
Agregación	Propiedad de una asociación que representa una relación todo-parte y donde (normalmente) se le contiene por toda la vida.
Análisis	Investigación de un dominio, la cual da origen a modelos que describen sus características estáticas y dinámicas. Se centra en cuestiones de "qué" más que de "cómo".
Análisis de requerimientos	Proceso de obtener una declaración escrita completa de la funcionalidad, apariencia, desempeño y comportamiento que requiere la aplicación.
Application Programming Interface (API)	Lista de prototipos clases de funciones y elementos proporcionados para beneficio de los programadores. La información de la función consiste en nombre, tipos de parámetros, tipos de resultados y las excepciones incluidas.
Arquitectura	Conjunto de decisiones significativas acerca de la organización de un sistema software, la selección de los elementos estructurales (y sus interfaces) de los que se compone el sistema, junto con su comportamiento tal y como se especifica en las colaboraciones entre esos elementos, la composición de esos elementos estructurales y de comportamiento en subsistemas cada vez mayores y el estilo arquitectónico que orienta esta organización, (esos elementos y sus interfaces, sus colaboraciones y su composición). La arquitectura del software no sólo tiene que ver con la estructura y el comportamiento, sino también con las restricciones y los compromisos entre uso, funcionalidad, rendimiento, flexibilidad, reutilización, comprensibilidad, economía y tecnología, y con intereses estéticos.
Arquitectura de Software	Un diseño global de una aplicación que incluye su descomposición en partes.

Artefacto	Cualquier tipo de datos, código fuente o información producida o usada por un desarrollador durante el proceso de desarrollo; se usa en particular al describir el proceso de desarrollo de software unificado *(Unified Software Development Process).*
Aseguramiento de la Calidad (AQ)	Proceso de asegurar que se logrará un nivel especificado de calidad en la ejecución de un proyecto; también se puede usar para hacer referencia a la organización que realiza esta función, en lugar de a la función misma.
Association for Computing Machinery (ACM)	Organización de profesionales involucrados con la computación, con hincapié en el software.
Atributo	Variable de una clase como un todo (no una variable local de un método).
Capability Maturity Model (CMM)	Modelo sistemático que evalúa las capacidades globales de una organización para desarrollar software; desarrollado por el Software Engineering Institute en Pittsburgh, PA.
Cascada	Proceso de desarrollo de software en el cual primero se recolectan los requerimientos, se desarrolla un diseño, el diseño se convierte en código y después se prueba. Esto se hace en secuencia, con un pequeño traslape entre las etapas sucesivas.
Casos de uso	Secuencia de acciones, algunas realizadas por una aplicación y otras por el usuario, que son comunes al usar la aplicación; el usuario tiene un papel particular en esta interacción y recibe el nombre de "actor" respecto al caso de uso.
Clase	En el lenguaje UML, "descripción de un conjunto de objetos que comparten los mismos atributos, operaciones, métodos, relaciones y significado" [BJR97]. A menudo se usa como sinónimo de una "clase de implementación" de UML: el mecanismo de software con que se definen e implementan los atributos y métodos de un tipo en particular.
Common Object Request Broker Architecture (CORBA)	Un estándar con el cual las aplicaciones pueden llamar a funciones que residen en plataformas remotas, sin importar el lenguaje en que están escritas.
Computer Aided Design / Computer Aided Manufacturing (CAD/CAM)	Software intensivo en gráficas que ayuda en el diseño y manufactura de productos electrónicos, de construcción o mecánicos.
Computer Aided Software Engineering (CASE)	Proceso de ingeniería de software ayudado por un conjunto coordinado de herramientas de software. Estas herramientas son de diseño especial para las diferentes etapas del desarrollo de software.
Constructive Cost Model (COCOMO)	Fórmulas de Barry Boehm para calcular los requerimientos probables de mano de obra, en personas mes, para construir una aplicación, y el tiempo probable que tomará, con base en las líneas de código estimadas.

Contrato	Define las responsabilidades y poscondiciones que se aplican al uso de una operación o método. También designa el conjunto de las condiciones relacionadas con una interfaz.
Desarrollo de Aplicaciones Rápido (RAD)	Proceso de desarrollar con rapidez una aplicación, o parte de ella, que puede implicar sacrificar la documentación o el diseño adecuados o la posibilidad de ampliarla.
Diagrama	Representación gráfica de un conjunto de elementos, representado la mayoría de las veces como un grafo conexo de nodos (elementos) y arcos (relaciones).
Diagrama de Estado	Forma de una máquina de estado finito con que se describe el comportamiento dinámico de un tipo.
Diagrama de Actividades	Diagrama que muestra el flujo de control entre actividades.
Diagrama de Casos de Uso	Diagrama que muestra un conjunto de casos de uso y actores y sus relaciones; los diagramas de casos de uso cubren la vista de casos de uso estática de un sistema.
Diagrama de Clases	Diagrama que muestra un conjunto de clases, interfaces y colaboraciones y sus relaciones; los diagramas de clases cubren la vista de diseño estática de un sistema; diagrama que muestra una colección de elementos declarativos (estáticos).
Diagrama de Colaboración	Diagrama de interacción que resalta la organización estructural de los objetos que envían y reciben señales; diagrama que muestra interacciones organizadas alrededor de instancias y los enlaces de unas a otras.
Diagrama de Componentes	Diagrama que muestra la organización y las dependencias entre un conjunto de componentes; los diagramas de componentes cubren la vista de implementación estática de un sistema.
Diagrama de Despliegue	Diagrama que muestra la configuración en tiempo de ejecución de los nodos de procesamiento y los componentes que residen en ellos; un diagrama de despliegue cubre la vista de despliegue estática de un sistema.
Diagrama de Estados (Statechart)	Diagrama que muestra una máquina de estados; los diagramas de estados cubren la vista dinámica de un sistema.
Diagrama de Flujo de Datos (DFD)	Diagrama que muestra cómo fluyen los datos al entrar, dentro y al salir de una aplicación. Los datos fluyen entre las aplicaciones del usuario, los almacenes de datos y los elementos de procesamiento interno de la aplicación.
Diagrama de Interacción	Diagrama que muestra una interacción, que consta de un conjunto de objetos y sus relaciones, incluyendo los mensajes que pueden enviarse entre ellos; los diagramas de interacción cubren la vista dinámica de un sistema; término genérico que se aplica a varios tipos de diagramas que resaltan las interacciones entre objetos, incluyendo los diagramas de colaboración, los diagramas de secuencia y los diagramas de actividades.

Diagrama de Objetos	Diagrama que muestra un conjunto de objetos y sus relaciones en un momento dado; los diagramas de objetos cubren la vista de diseño estática o la vista de procesos estática de un sistema.
Diagrama de secuencia	Diagrama de interacción que resalta la ordenación temporal de los mensajes; diagrama formado por los objetos de la aplicación que muestra una secuencia de llamadas de funciones entre los objetos; los diagramas de secuencia suelen dar detalles de los casos de uso.
Dirigido por Casos de Uso	En el contexto del ciclo de vida del desarrollo de software, proceso en el que se utilizan los casos de uso como artefactos principales para establecer el comportamiento deseado del sistema, para verificar y validar la arquitectura del sistema, para las pruebas y para comunicación entre los usuarios del proyecto.
Dirigido por el Riesgo	En el contexto del ciclo de vida del desarrollo de software, proceso en el que cada nueva versión se ocupa principalmente de acometer y reducir los riesgos más significativos para el éxito del proyecto.
Herencia	Mecanismo por el que elementos más específicos incorporan la estructura y comportamiento de elementos más generales.
Diseño Orientado a Objetos	Especificación de una solución lógica de software a partir de objetos de software: clases, atributos, métodos y colaboraciones.
Documentación de pruebas del software (DPS)	Documento que especifica todos los aspectos del proceso de pruebas para una aplicación.
Documentación personal de software (DPS)	Documentación que mantiene un individuo acerca del estado actual de su código.
Dominio	Límite formal que define determinado tema o área de interés.
Encapsulamiento	Mecanismo con que se ocultan los datos, la estructura interna y los detalles de la implementación de un objeto. La interacción con un objeto se realiza a través de una interfaz pública de las operaciones.
Especificación de Requerimientos de software (ERS)	Documento que establece lo que una aplicación debe lograr.
Estado	El estado de un objeto; la definición formal es el conjunto de valores de las variables de un objeto (por ejemplo, se puede decir que un objeto "automóvil" está en un estado "clásico", si el año de fabricación es menor que 1955 y la variable de su condición es "bien" o mejor que eso.
Evaluación de capacidades	Proceso mediante el cual se miden de manera cuantitativa y objetiva las capacidades de una organización, grupo o persona para producir software.
Evento	Una ocurrencia que afecta un objeto, iniciado desde el exterior del objeto.

Herencia	Característica de los lenguajes de programación orientados a objetos, en virtud de la cual las clases pueden especializarse a partir de superclases más generales. La subclase adquiere automáticamente las definiciones de atributos y clases hechas a partir de las superclases.
Ingeniería de sistemas	Proceso de analizar y diseñar un sistema completo; incluye hardware y software.
Ingeniería directa	Proceso de transformar un modelo en código a través de una correspondencia con un lenguaje de implementación específico.
Ingeniería inversa	Proceso de deducir el contenido de una etapa de desarrollo de software a partir de los artefactos de una etapa subsecuente (por ejemplo, deducir el diseño a partir del código).
Ingeniería inversa	Proceso de transformación de código en un modelo a través de una correspondencia con un lenguaje de implementación específico.
Institute of Electrical and Electronic Engineers (IEEE)	Organización de profesionales dedicada a la ingeniería relacionada con la electrónica, la electricidad y el software.
Instancia	Miembro individual de un tipo o de una clase.
Integración	La fusión de los módulos que forman una aplicación.
Integridad	Relación correcta y consistente de unas cosas con otras.
Interactive Development Environment (IDE)	Aplicación de software que ayuda a los desarrolladores a crear, editar, compilar y ejecutar el código.
Interfaz	Una interfaz para un sistema es una especificación de un conjunto de funciones que proporciona el sistema; la especificación incluye los nombres de las funciones, sus tipos de parámetros, los tipos de resultados y las excepciones. También puede definirse como un conjunto de representaciones de operaciones públicas.
Interfaz Gráfica de Usuario (GUI)	Despliegue gráfico, a menudo interactivo, mediante el cual el usuario interactúa con una aplicación.
Invariante	Relación entre las variables que no cambia, dentro del contexto especificado (sin embargo, los valores de las variables individuales pueden cambiar).
Involucrados	Persona, grupo u organización que tienen algo que ver en el resultado de una aplicación en desarrollo.
Iteración	Conjunto bien definido de actividades, con un plan base y unos criterios de evaluación, que produce una versión, ya sea interna o externa. Proceso de agregar repetidas veces requerimientos, diseño, implementación y pruebas a la construcción parcial de una aplicación.
Iterativo	En el contexto del ciclo de vida del desarrollo de software, proceso que implica la gestión de un flujo de versiones ejecutables.

Mantenimiento	Proceso de reparar y mejorar una aplicación que ya se entregó.
Mapa conceptual	Lista de actividades que se obtiene para lograr una meta específica.
Marco de trabajo	Una colección de clases generales que constituye la base de varias aplicaciones. Las clases de cada aplicación agregan o heredan las clases del marco de trabajo.
Método de la caja negra	Método casi siempre asociado con las pruebas, aplicado a un código implementado, que toma en cuenta sólo la entrada y la salida (es decir, no la manera interna en que el opera el código).
Métodos formales	Métodos rigurosos para especificar los requerimientos, el diseño o la implementación, de naturaleza matemática o lógica.
Métricas	Especificación para cómo medir un artefacto de ingeniería de software. Por ejemplo, (líneas de código) es una medición para el código fuente.
Modelo	Simplificación de la realidad, creada para comprender mejor el sistema que se está creando; abstracción semánticamente cerrada de un sistema. Descripción de las características estáticas, dinámicas o ambas de un tema, presentada en varias vistas (generalmente diagramáticas o textuales). El modelo de una aplicación es un panorama de su diseño desde una perspectiva particular, como la combinación de sus clases, o su comportamiento manejado por los eventos.
Object Management Group (OMG)	Organización no lucrativa de compañías que establece los estándares para la computación de objetos distribuidos.
Orientado a Objetos (OO)	Organización de los diseños y el código en clases de instancias ("objetos"). Se asigna un conjunto de funciones especificadas por una clase dada a cada objeto de esa clase; cada objeto tiene su copia de un conjunto de variables especificadas para la clase.
Paquete	Mecanismo de propósito general para organizar elementos en grupos.
Paradigma	Una manera de pensar, como el paradigma de la orientación a objetos.
Parámetro	Especificación de una variable que puede cambiarse, pasarse o ser devuelta.
Patrón	Solución común a un problema común en un contexto determinado. Es la descripción etiquetada de un problema, de la solución, de cuándo aplicar la solución y la manera de hacerlo dentro de otros contextos.
Patrón de diseño	Patrón de clases que ocurren al mismo tiempo, con las relaciones entre ellas y los algoritmos que las acompañan.
Persistencia	Almacenamiento duradero del estado de un objeto.

Personal Software Process℠ (PSP)	Proceso desarrollado por Watts Humphrey en el Software Engineering Institute, para mejorar y medir las capacidades de la ingeniería de software de los ingenieros individuales de software.
Plan de Administración de la Configuración del Software (PACS)	Documento que especifica cómo debe administrarse el código y la documentación de un proyecto, y todas sus versiones.
Plan de Administración del Proyecto de Software (PAPS)	Plan que establece quién desarrollará qué partes de una aplicación y en qué orden lo harán.
Polimorfismo	Concepto según el cual dos o más tipos de objetos pueden responder a un mismo mensaje en formas diferentes, usando para ello operaciones polimórficas. También, capacidad de definir las operaciones polimórficas.
Poscondición	Estricción que debe ser cierta al finalizar una operación. Restricción que debe cumplirse una vez terminada una operación.
Precondición	Restricción que debe ser cierta cuando se invoca una operación. Restricción que debe cumplirse antes que se solicite una operación.
Proceso	"Proceso de software" es el orden en que se realizan las actividades de desarrollo.
Proceso de caja blanca	Método, casi siempre de prueba, aplicado al código desarrollado, que tiene en cuenta la intención de funcionamiento del código.
Producto	Artefacto del desarrollo, tal como los modelos, el código, la documentación y los planes de trabajo.
Prototipo	Aplicación que ilustra o muestra algunos aspectos de la aplicación que se construye.
Proyección	Correspondencia de un conjunto hacia un subconjunto de él.
Pruebas de integración	Proceso de probar el éxito de la fusión de los módulos.
Pruebas de regresión	Proceso de validar el hecho de que agregar código a una aplicación en desarrollo no reduce las capacidades que poseía la aplicación antes de la adición.
Pruebas de unidades	Proceso para probar una parte de una aplicación aislada del resto de la aplicación.
Pruebas del sistema	Proceso de probar una aplicación completa (no sus partes).
Pseudocódigo	Lenguaje parecido a inglés o español que tiene la suficiente formalidad para describir un algoritmo.
Punto de función (PF)	Medida de la complejidad de una aplicación.
Requerimientos D	Requerimientos para el desarrollador, una forma de requerimientos adecuada principalmente para que los desarrolladores trabajen con ella, pero que también forma parte de los requerimientos de los clientes.

Restricción	Limitación específica.
Retiro del riesgo	Proceso de manejar una amenaza percibida (un riesgo) para la ejecución exitosa de un proyecto, ya sea encontrando la manera de evitarlo o actuando para eliminar su impacto.
Rol	Comportamiento de una entidad que participa en un contexto particular.
Sistema	Posiblemente descompuesto en una colección de subsistemas, conjunto de elementos organizados para lograr un propósito específico y que se describe por un conjunto de modelos, posiblemente desde diferentes puntos de vista.
Software Engineering Institute (SEI)	Instituto fundado en su inicio para mejorar la calidad del software desarrollado para el Ministerio de Defensa de Estados Unidos. Su trabajo se usa también en muchas organizaciones civiles.
Team Software Process℠ (TSP)	Proceso desarrollado por Watts Humphrey en el Software Engineering Institute para evaluar y mejorar el desempeño de los equipos de desarrollo de software.
Tiempo medio entre fallas (MTBF)	(Tiempo que está en uso una aplicación) / (número de veces que falla la aplicación durante ese tiempo); debe proporcionarse la definición de "falla".
Unified Modeling Language (UML)	Lenguaje de modelado unificado o notación gráfica que sirve para expresar los diseños orientados a objetos.
Validación	Proceso para asegurar que una aplicación de software realiza las funciones para las que se creó de la manera especificada.
Verificación	Proceso para asegurar que una aplicación de software se está construyendo de la manera planeada.
Versión	Conjunto relativamente completo y consistente de artefactos entregado a un usuario externo o interno.
Versión alfa	Versión preliminar de una aplicación dada a clientes altamente confiables y/o a usuarios internos para obtener retroalimentación.
Versión beta	Versión preliminar de una aplicación, dada a clientes seleccionados para ayudar a detectar defectos y obtener retroalimentación.
Vista	Proyección de un modelo, que se ve desde una perspectiva o un punto de vista dado, y que omite entidades que no son relevantes desde esa perspectiva.
Vista de Casos de Uso	Vista de la arquitectura de un sistema que incluye los casos de uso que describen el comportamiento del sistema tal y como es visto por sus usuarios finales, los analistas y los encargados de las pruebas.

Vista de despliegue Vista de la arquitectura de un sistema que incluye los nodos que forman la topología hardware sobre la cual se ejecuta el sistema; una vista de despliegue que cubre la distribución, entrega e instalación de las partes que configuran el sistema físico.

Vista de diseño Vista de la arquitectura de un sistema que incluye las clases, interfaces y colaboraciones que forman el vocabulario del problema y su solución; una vista de diseño se ocupa de los requisitos funcionales de un sistema.

Vista de implementación Vista de la arquitectura de un sistema que incluye los componentes que se utilizan para ensamblar y hacer disponible el sistema físico; una vista de implementación cubre la gestión de configuraciones de las versiones del sistema, formada por componentes relativamente independientes que se pueden ensamblar de varias formas para producir un sistema ejecutable.

Vista de procesos Vista de la arquitectura de un sistema que incluye los hilos y procesos que forman los mecanismos de concurrencia y sincronización de un sistema; una vista de procesos cubre el funcionamiento, capacidad de adaptación y rendimiento de un sistema.

Vista dinámica Aspecto de un sistema que destaca su comportamiento.

Vista estática Aspecto de un sistema que destaca su estructura.

Bibliografía

ERIC J. BRAUDE Ingeniería de Software**,** Una perspectiva orientada a objetos. Alfa Omega Grupo Editor, S.A. de C. V. México 2003.

CRAIG LARMAN UML y Patrones, Introducción al análisis y diseño orientado a objetos. Prentice Hall Hispanoamericana, S. A. México 1999.

ROGER S. PRESSMAN Ingeniería de Software, Un enfoque práctico. Mc GRAW Hill/Interamericana de España, S. A. de C. V. Madrid 2002.

GRADY BOOCH, JAMES RUMBAUGH, IVAR JACOBSON El lenguaje unificado de modelado. Addison Wesley Iberoamericana, Madrid 1999.

PIERRE-ALAIN MULLER Modelado de Objetos con UML. Ediciones Gestión 2000, S.A. Barcelona, 1997.

CRAIG UTLEY Desarrollo de aplicaciones Web con SQL Server™ 2000. McGraw Hill Interamericana de España, S. A. U. Madrid, 2001.

IVAR JACOBSON, GRADY BOOCH, JAMES RUMBAUGH El Proceso Unificado de Desarrollo de Software. Pearson Educación, S. A. Madrid, 2000.

WILLIAM R. VAUGHN Programación de SQL Server 7.0 con Visual Basic 6.0. McGraw Hill Interamericana Editores, S. A. de C. V. México, D. F., 2000.

SHARI LAWRENCE PFLEEGER Ingeniería de Software, Teoría y Práctica – 1ª Edición. Pearson Educación. Buenos Aires, Argentina, 2002.

CERTIFICACION MICROSOFT MCDBA SQL Server 7.0 Exámenes Prácticos. Syngress Media, Inc. Mc Graw Hill Interamericana de España, S. A. U. Madrid, 2000

PROJECT MANAGEMENT INSTITUTE A Guide to the Project Management Body of Knowledge. Third Edition. ANSI/PMI 99-001-2004.

SCHACH, STEPHEN R. Ingeniería De Software Clásica Y Orientada A Objetos. Mc Graw Hill Interamericana de España, S. A. U. Madrid, 2006.

PAUL HARMON AND BRIAN SAWYER UML For Visual Basic 6.0 Developers. Morgan Kaufmann Pub; 1st edition (November 9, 1998).

Mesografía

http://es.wikipedia.org

http://www.elprisma.com/apuntes/curso.asp?id=7625

http://www.microsoft.com/spanish/msdn/comunidad/mtj.net/voices/MTJ_2828.asp#M2

http://www.universidadabierta.edu.mx/Biblio/B/Badilla-ANALISIS%20Y%20DISENO.html

http://www.monografias.com/trabajos/anaydisesis/anaydisesis.shtml

http://www.microsoft.com/spanish/msdn/comunidad/mtj.net/voices/art151.asp

http://www.mug.org.ar/Patrones/ArticPatrones/304.aspx

http://www.epidataconsulting.com/tikiwiki/tiki-read_article.php?articleId=31

http://www.monografias.com/trabajos34/ingenieria-software/ingenieria-software.shtml

http://www.elrincondelprogramador.com/default.asp?pag=articulos/leer.asp&id=29

http://www.sparxsystems.com.au/UML_Tutorial.htm

http://www.microsoft.com/spanish/msdn/comunidad/mtj.net/voices/MTJ_2295.asp

http://www.clikear.com/manuales/uml/

Referencias

[Al] Albrecht, A. J., "Measuring Application Development Productivity", Proceedings of the Joint SHARE/GUIDE/IBM Application Development Symposium, octubre 1979, pp. 83-92.

[Am] Ambler, Scott, www.ambysoft.com (1999).

[ASR02] Pekka Abrahamsson, Outi Salo, Jussi Ronkainen y Juhani Warsta. "Agile Software Development Methods". VTT Publications 478, Universidad de Oulu, Suecia, 2002.

[Baer03] Martha Baer. "The new X-Men". Wired 11.09, Septiembre de 2003.

[Bat03] Don Batory. "A tutorial on Feature Oriented Programming and Product Lines". Proceedings of the 25th International Conference on Software Engineering, ICSE'03, 2003.

[Bauer, 1972] Bauer, F.L.: Software Engineering, Information Processing, 71, North Holland Publishing Co., Amsterdam, 1972.

[BC89] Kent Beck y Ward Cunningham. "A laboratory for teaching Object-Oriented thinking". OOPSLA'89 Conference Proceedings, SIGPLAN Notices, 24(10), Octubre de 1989.

[Beck99a] Kent Beck. Extreme Programming Explained: Embrace Change. Reading, Addison Wesley, 1999.

[Bel] Berry, Daniel M. y Lawrence, Brian, "Requirements Engineering", IEEE Software, vol. 15, núm. 2, marzo/abril 1998.

[Ber03] Edward Berard. "Misconceptions of the Agile zealots". The Object Agency, http://www.svspin.org/Events/Presentations/MisconceptionsArticle20030827.pdf, 2003.

[BMP98] Grady Booch, Robert Martin y James Newkirk. Object Oriented Analysis and Design With Applications, 2a edición, Addison-Wesley, 1998

[Bo] Boehm, Barry, Software Engineering Economies, Englewood Cliffs, NJ: Prentice Hall, 1981.

[BOE88] Boehm, B., «A Spiral Model for Software Development and Enhancement", Computer, vol. 21, n.^{e} 5, Mayo 1988, pp. 61-72.

[BOE96] Boehm, B., «Anchoring the Software Process», IEEE Software, vol. 13, n.$^{\circ}$ 4, Julio 1996, pp.73-82.

[BOE98] Boehm, B., «Using the WINWIN Spiral Model: A Case Study», Computer, vol. 31, n.^{s} 7, Julio 1998, pp. 33-44.

[Bohem, 1976] Boehm, B. W.: Software Engineering, IEEE Transactions on Computers, C-25, Núm. 12, diciembre, pp. 1226-1241.

[Br] Brackett, J., ftp://ftp.sei.cmu.edu/pub/edu-cation/cm!9.ps, (enero, 1990).

[BT75] Vic Basili y Joe Turner. "Iterative Enhancement: A Practical Technique for Software Development", IEEE Transactions on Software Engineering, 1(4), pp. 390-396, 1975.

[BUT94] Butler, J., «Rapid Application Development in Action», Managing System Development, Applied Computer Research, vol. 14, n.° 5, Mayo 1995, pp. 6-8.

[Cha04] Robert Charette. "The decision is in: Agile versus Heavy Methodologies". Cutter Consortium, Executive Update, 2(19),

http://www.cutter.com/freestuff/apmupdate.html, 2004.

[CLD00] Peter Coad, Eric Lefebvre y Jeff DeLuca. Java modeling in color with UML: Enterprise components and process. Prentice Hall, 2000.

[COU80] Cougar, J., y R. Zawacky, Managing and Motivating Computer Personnel, Wiley, 1980.

[Cu] Cucumano, A. y Selby, R. W., "How Microsoft Builds Software", Communications of the ACM, vol. 40, núm. 6, junio 1997, pp. 53-61.

[CUR94] Curtis, B., et al., People Management Capability Maturity Model, Software Engineering Institute, Pittsburgh, PA, 1994.

[DEM98] DeMarco, T., y T. Lister, Peopleware, 2.ª edición, Dorset House, 1998.

[Di] Dijkstra, E., A Discipline of Programming, Englewood Cliffs, NJ: Prentice Hall, 1976.

[DS03] DSDM Consortium y Jennifer Stapleton. DSDM: Business Focused Development. 2a edición, Addison-Wesley, 2003.

[DYE92] Dyer, M., The Cleanroom Approach to Quality Software Development, Wiley, 1992.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke y Don Roberts. Refactoring: Improving the design of existing code. Addison Wesley, 1999.

[Ga] Gamma, Erich; Helm, Richard; Johnson, Ralph, y Vlissides, John, Design Patterns: Elements of Reusable Object-Oriented Software (Addison-Wesley Professional Computing), Addison-Wesley, agosto de 1999.

[Hig00a] Jim Highsmith. Adaptive software development: A collaborative approach to managing complex systems. Nueva York, Dorset House, 2000.

[Hig00b] Jim Highsmith. "Extreme Programming". EBusiness Application Development, Cutter Consortium, Febrero de 2000.

[Hig01] Jim Highsmith. "The Great Methodologies Debate. Part 1". Cutter IT Journal, 14(12), diciembre de 2001.

[Hig02a] Jim Highsmith. "What is Agile Software Development". Crosstalk, http://www.stsc.hill.af.mil/crosstalk/2002/10/highsmith.html, Octubre de 2002.

[Hol95] John Holland. Hidden Order: How Adaptation builds Complexity. Addison Wesley, 1995.

[Hu3] http://www.stsc.hill.af.mil/crosstalk/ 1998/apr/dimensions.html (12/99).

[Hu7] Humphrey, Watts S., Introduction to the Team Software Process (The SEI Series in Software Engineering), Addison-Wesley, agosto de 1999.

[IEEE 982] Guide to the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software, ANSÍ/ IEEE 982.2-1988.

[IEEE, 1993] Standards Collection: Software Engineering, IEEE Estándar 610.12- 1990, IEEE, 1993.

[IF] International Function Point Users' Group, http//www.ifpug.org/ (12/99).

[Ja] Jacobson, Ivar, Object-Oriented Software Engineering: A Use Case Driven Approach (Addison-Wesley Object Technology Series), Reading, MA: Addison-Wesley, 1994.

[Jac02] Ivar Jacobson. "A resounding Yes to Agile Processes – But also to more". The Rational Edge, Marzo de 2002.

[Jacobson 92] Jacobson I., et al. 1992. Object-Oriented Software Engineering: A Use Case Driven Approach. Reading, MA: Addison-Wesley.

[JAH01] Ron Jeffries, Ann Anderson, Chet Hendrikson. Extreme Programming Installed. Upper Saddle River, Addison-Wesley, 2001.

[Jal] Jacobson, Ivar; Rumbaugh, James, y Booch, Grady, The Unified Software Devél-opment Process (Addison-Wesley Object Technology Series), Reading, MA: Addison-Wesley, 1999.

[Ke] Keil, M.; Cule, P.; Lyytinen, K, y Schmidt, R., "A Framework for Identifying Software Project Risks", Communications of the ACM, vol. 41, núm. 11, noviembre, 1998.

[Kee03] Gerold Keefer. "Extreme Programming considered harmful for reliable software development 2.0". AVOCA GmbH, Documento público, 2001.

[KER94] Kerr, J., y R. Hunter, Inside RAD, McGraw-Hül, 1994.

[Ki] Kit, Edward, Software Testing in the Real World: Improving the Process, Reading, MA: Addison-Wesley, 1995.

[Kr] Kruchten, P., The Rational Unified Process, Reading, MA: Addison-Wesley, 1998.

[Kru95] Philippe Kruchten. "The 4+1 View Model of Architecture." IEEE Software 12(6), pp. 42-50, Noviembre de 1995.

[Lar04] Craig Larman. Agile & Iterative Development. A Manager's Guide. Reading, Addison-Wesley 2004.

[McB02] Pete McBreen. Questioning Extreme Programming. Addison Wesley, 2002.

[McC02] Craig McCormick. "Programming Extremism". Upgrade, 3(29, http://www.upgrade-cepis.org/issues/2002/2/up3-2McCormick.pdf, Abril de 2002. (Original en Communications of the ACM, vol. 44, Junio de 2001).

[McC93] Steve McConnell. Code complete. Redmond, Microsoft Press, 1993.

[McC96] Steve McConnell. Rapid Development. Taming wild software schedules. Redmond, Microsoft Press, 1996.

[MDE93] McDermid, J. y P. Rook, «Software Developement Process Models», en Software Ingineer's Reference Book, CRC Press, 1993, pp. 15/26-15/28.

[Mel03] Stephen Mellor. "What's wrong with Agile?", Project Technology, http://www.projtech.com/pdfs/shows/2003/wwwa.pdf, 2003.

[NA99] Joe Nandhakumar y David Avison. "The fiction of methodological development: a field study of information systems development". Information Technology & People, 12(2), pp. 176-191, 1999.

[MIL87] Mills, H.D., M. Dyer y R. Linger, «Cleanroom Software Engineering», IEEE Software, vol. 4, n.° 5, Septiembre de 1987, pp. 19-24.

[MIL87] Mills, H.D., M. Dyer y R. Linger, «Clearoom Software Engineering», IEEE Software, Septiembre 1987, pp. 19-25.

[My] Myers, Glenford J., The Art of Software Testing, Nueva York, John Wiley & Sons, 1979.

[NIE92] Nierstrasz, «Component-Oriented Software Development», CACM, vol. 35, n.° 9, Septiembre 1992, pp. 160-165.

[Op92] William F. Opdyke. Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks. Tesis de Doctorado, University of Illinois at Urbana-Champaign, 1992.

[PAU93] Paulk, M., et al., «Capability Maturity Model for Software», Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1993.

[PMB04] PMBOK® Guide – 2000 Edition. Project Management Institute, http://www.pmi.org/prod/groups/public/documents/info/pp_pmbok2000welcome.asp, 2004.

[PW92] Dewayne E. Perry y Alexander L. Wolf. "Foundations for the study of software architecture". ACM SIGSOFT Software Engineering Notes, 17(4), pp. 40–52, Octubre de 1992.

[Rak01] Steven Rakitin. "Manifesto elicits cynicism". Computer, pp. 4-7, Diciembre de 2001.

[REE99] Reel, J.S., «Critical Success Factors in Software Projects», IEEE Software, Mayo de 1999, pp. 18-23.

[RIG 81] Riggs, J., Production Systems Planning, Análisis and Control, 3a ed., Wiley, 1981.

[Rol] Software Specification: A Framework, http://www.sei.cmu.edu/topics/publications /documents/cms/cm.011 .html (12/99).

[RS01] Bernhard Rumpe y Astrid Schröder. "Quantitative survey on Extreme Programming Projects". Informe, Universidad de Tecnología de Munich, 2001.

[SHA95a] Shaw, M., y D. Garlan, «Formulations and For-malisms in Software Architecture», Volume 1000-Lectu-re Notes in Computer Science, Springer Verlag, 1995.

[She97] Sarah Sheard. "The frameworks quagmire, a brief look". Software Productivy Consortium, NFP, http://www.software.org/quagmire/frampapr/FRAMPAPR.HTML, 1997.

[Shl] Shnayerson, Michael, The Car That Could; The Inside Story of General Motors' Revolutionary Electric Vehicle, ASÍN: 067942105X.

[SR03] Matt Stephens y Doug Rosenberg. Extreme Programming Refactored: The case against XP. Apress, 2003.

[Sta97] Jennifer Stapleton. Dynamic Systems Development Method – The method in practice. Addison Wesley, 1997.

[Wie98] Karl Wiegers. "Read my lips: No new models!". IEEE Software. Setiembre-Octubre de 1998.

[Wie99] Karl Wiegers. "Molding the CMM to your organization". Software Development, Mayo de 1998.

[WIT94] Whitaker, K., Managing Software Maníaos, Wiley, 1994.

[ZAH 95] Zahniser, R., Time-Boxing for Top Team Performance, Software Developent, Marzo 1995, pp. 34-38

[Zelkovitz, 1978 Zelkovitz, M. V., Shaw, A. C. y Gannon, J.D.: Principles of Software Engineering and Design. Prentice-Hall, Eaglewoods Clif, 1979

Indice de Figuras

Indice de Tablas