Ingeniería del Software

Principios y Conceptos

G. Choque Aspiazu

La Paz, Bolivia
Diciembre de 2002

INGENIERÍA DEL SOFTWARE

Principios y Conceptos

Guillermo Choque Aspiazu

Universidad Mayor de San Andrés
Ciencias de la Computación

DEDICATORIA

Para Alison ...
En tus primeros pasos quizá comprendas el alto sentido que tiene escribir un proyecto antes de plasmarlo en la realidad

PREFACIO

En el nuevo milenio, la humanidad asiste a profundas transformaciones contextuales cuyas implicaciones, antes que presentarse como mutaciones de carácter evolutivo, representan saltos y rompimientos, que en la mayoría de los casos conllevan al desmonte de viejos paradigmas y a su sustitución abrupta.

Uno de los más significativos lo constituye la tecnología, y dentro de ella, sin duda alguna, la gerencia informática, la electrónica, las telecomunicaciones, y el desarrollo de la ingeniería de software que juega un papel importante en las organizaciones. Se avanza hacia una nueva cultura: La cultura de la información, y en consecuencia un nuevo tipo de sociedad. Organizaciones que para su desarrollo informático requieren formación en el área de la ingeniería del software.

En la era de la economía de la información y la inteligencia social en los servicios de la informática se preparan profesionales en las distintas áreas del conocimiento, con capacidad de planear, diseñar, implementar, y liderar proyectos, que apliquen técnicas modernas de la ingeniería de software, en sus diversos campos, que permitan asegurar su calidad, con base en estándares reconocidos mundialmente.

Hoy en día, las computadoras están presentes en todas las áreas de la actividad humana. En la mayoría de casos es necesario que haya intercambio de información con otras computadoras. Esto significa que para el diseño del hardware y software es importante tener en cuenta la evolución y el desarrollo de la ingeniería de software y la informática en general.

La primera versión del presente texto de ingeniería del software fue presentada a inicios del presente siglo, cuando fue ofertado en calidad de texto de lectura preliminar para la materia del mismo nombre en la Universidad Mayor de San Andrés. Con el advenimiento acelerado de las tecnologías web, se publicó el texto en el sitio web de la UMSA. Aún es posible obtener una descarga gratuita del contenido en este sitio:
www.umsanet.edu.bo\docentes\gchoque\M426texto.htm

El texto está compuesto por diez capítulos: el capitulo uno hace referencia al proceso de evolución del término software, el capitulo dos muestra las bases de la tecnología estratificada, el capitulo tres hace referencia a los conceptos y principios del análisis del software, el capitulo cuatro presenta el modelado del análisis, el capitulo cinco presenta los conceptos y principios del diseño, el capitulo seis describe los métodos de diseño, el capítulo siete hace referencia a los métodos de prueba del software, el capitulo ocho menciona las estrategias de prueba del software, el capitulo nueve describe las métricas técnicas del software y finalmente el capitulo diez muestra la ingeniería del software orientada a objetos.

AGRADECIMIENTOS

En principio agradecer al doctor Roger Pressman por el fabuloso texto, sólido y comprensivo que escribió para el área de la ingeniería del software. La claridad de sus ideas y el enfoque práctico con calidad han sido recopilados de manera sucinta en el presente texto.

Quiero agradecer a los alumnos de la materia: Ingeniería del Software, el espacio disponible no permite que nombre a los aproximadamente 300 alumnos que pasaron la materia en los anteriores años, agradecerles por las agradables clases que disfrutamos de manera conjunta.

Cuando converse con el Decano de la Facultad de Ciencias Puras y Naturales, Msc. Ing. Rolando Campuzano Arzabe, no dubitó un momento en ofrecer los servicios del Centro de Publicaciones de la Facultad para que este texto se replique y en este momento se encuentre en sus manos. En este entendido agradezco al Msc. Ing. Campuzano por su colaboración y al personal del Centro de Publicaciones por todas las atenciones.

Finalmente quiero agradecer a mi familia por haber soportado mi alejamiento virtual al comprometerme indirectamente con la computadora, mientras duro la redacción y corrección del texto. Alison adorada hijita este texto es para ti y disculpas por haber robado tus minutos de juego y la dedicación a tu rápido crecimiento.

La Paz, Bolivia.
Diciembre de 2002.

G.I.C.A.

Síntesis del Contenido

Pág.

7. Métodos de Prueba del Software
7.1. Introducción
7.2. Fundamentos de la Prueba del Software
7.3. Diseño de Casos de Prueba
7.4. Prueba de Caja Blanca
7.5. Prueba del Camino Básico
7.6. Prueba de la Estructura de Control
7.7. Prueba de Caja Negra

8. Estrategias de Prueba del Software
8.1. Introducción
8.2. Enfoque Estratégico
8.3. Aspectos Estratégicos
8.4. Prueba de Unidad
8.5. Prueba de Integración
8.6. Prueba de Validación
8.7. Prueba del Sistema
8.8. Arte de la Depuración

9. Métricas Técnicas del Software
9.1. Introducción
9.2. Calidad del Software
9.3. Estructura para las Métricas del Software
9.4. Métricas del Modelo de Análisis
9.5. Métricas del Modelo de Diseño
9.6. Métricas del Código Fuente
9.7. Métricas para Pruebas
9.8. Métricas de Mantenimiento

10. Ingeniería del Software Orientada a Objetos
10.1. Introducción
10.2. Paradigma Orientado a Objetos
10.3. Conceptos Orientados a Objetos
10.4. Gestión de Proyectos de Software OO
10.5. Análisis Orientado a Objetos
10.6. Diseño Orientado a Objetos
10.7. Pruebas Orientadas a Objetos
10.8. Métricas Técnicas para Sistemas OO

Contenido

Pág.

4.3. Mecanismos del Análisis Estructurado
Ejercicios # 4

5. Conceptos y Principios del Diseño
5.1. Introducción
5.2. Diseño e Ingeniería del Software
5.3. Proceso de Diseño
5.3.1. Diseño y Calidad del Software
5.4. Principios del Diseño
5.5. Conceptos del Diseño
5.5.1. Abstracción
5.5.2. Refinamiento
5.5.3. Modularidad
5.5.4. Arquitectura
5.5.5. Jerarquía de Control
5.5.6. Partición Estructural
5.5.7. Estructura de Datos
5.5.8. Procedimiento del Software
5.5.9. Ocultamiento de la Información
5.6. Diseño Modular
5.6.1. Independencia Funcional
5.6.2. Cohesión
5.6.3. Acoplamiento
Ejercicios # 5

6. Métodos de Diseño
6.1. Introducción
6.2. Diseño de Datos
6.3. Diseño Arquitectónico
6.4. Proceso de Diseño Arquitectónico
6.5. Diseño de la Interfaz
6.6. Diseño Procedimental
6.6.1. Programación Estructurada
6.6.2. Notaciones Gráficas de Diseño
6.6.3. Notaciones Tabulares de Diseño
6.6.4. Lenguaje de Diseño de Programas
Ejercicios # 6

7. Métodos de Prueba del Software
7.1. Introducción
7.2. Fundamentos de la Prueba del Software
7.2.1. Objetivos
7.2.2. Principios
7.2.3. Facilidad de Prueba
7.3. Diseño de Casos de Prueba
7.4. Prueba de Caja Blanca
7.5. Prueba del Camino Básico

1 EVOLUCIÓN DEL SOFTWARE

El software de la 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., la lista es casi interminable. A medida que transcurra el siglo XXI, será uno de los pilares que conduzca a grandes cambios, desde la educación elemental hasta la ingeniería genética.

1.1. DESARROLLO EN EL TIEMPO

Actualmente el software desempeña 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, en este entorno 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 complejo como una simulación en multimedia. Como vehículo utilizado para hacer entrega del producto, el software actúa como la base de control de la computadora[1], la comunicación de información (redes), y la creación y control de otros programas[2].

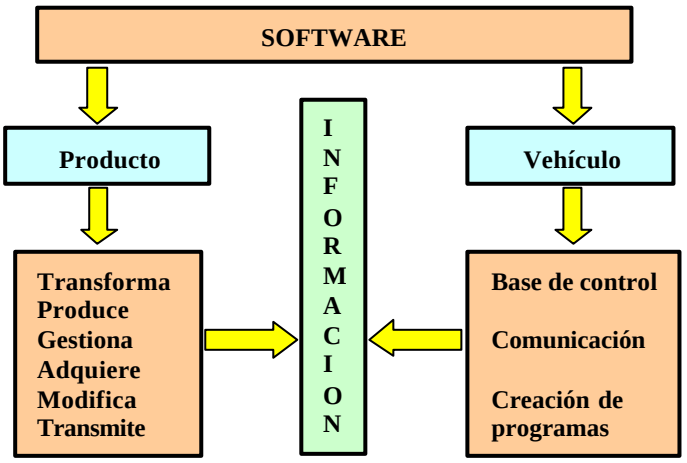

Fig. 1.1. Doble papel del software

El siguiente conjunto de problemas relacionados con el software ha persistido a través del

[1] Específicamente hace referencia a sistemas operativos.
[2] Como herramientas del software y entornos.

tiempo, en la evolución de los sistemas basados en computadoras. Estos problemas continúan en aumento:

1) Los avances del software continúan dejando atrás la habilidad de construir software para alcanzar el potencial del hardware.
2) La habilidad de construir nuevos programas no puede ir al ritmo de la demanda de nuevos programas, ni se puede construir programas lo suficientemente rápidos como para cumplir las necesidades del mercado y los negocios.
3) El uso extenso de computadoras ha hecho de 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.
4) Se lucha por construir software informático que tenga fiabilidad y alta calidad.
5) La habilidad de soportar y mejorar los programas existentes se ve amenazada por diseños pobres y recursos inadecuados.

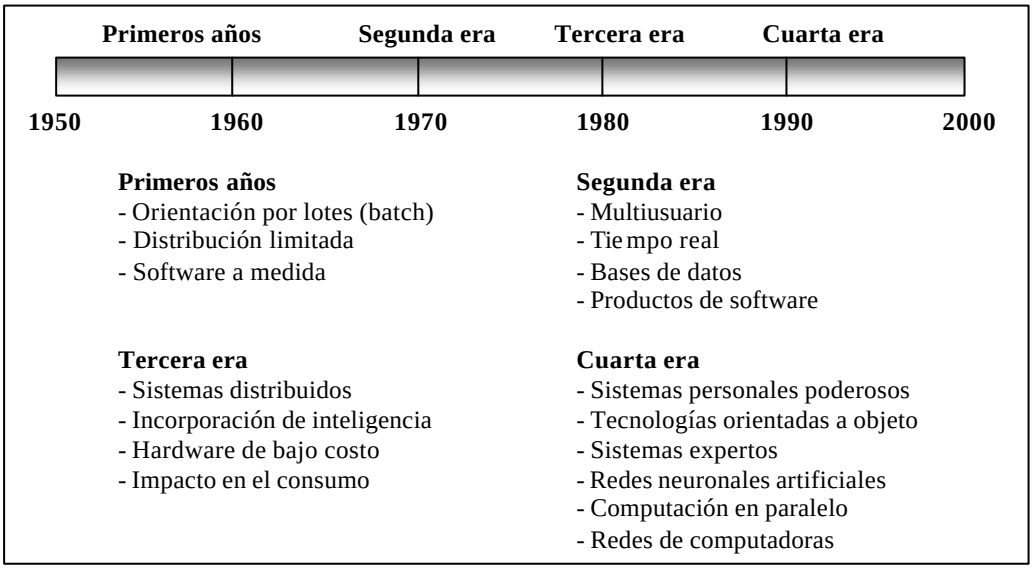

Fig. 1.2. Evolución histórica del software

1.2. CARACTERÍSTICAS

Existen en libros de texto diferentes definiciones de software:

1) instrucciones que cuando se ejecutan proporcionan la función y el rendimiento deseados.
2) estructuras de datos que permiten a los programas manejar adecuadamente la información y
3) documentos que describen la operación y uso de los programas.

El software es un elemento lógico del sistema, a diferencia del hardware que es un elemento físico, y presenta las siguientes características:

1) Se desarrolla no se fabrica en un sentido clásico.
2) No se estropea, se deteriora hasta la obsolescencia.

3) Se construye a medida, en lugar de ensamblarse a partir de componentes existentes.

1.3. COMPONENTES

La reutilización es una característica importante para un componente de software de alta calidad. 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. En el nivel más bajo el lenguaje refleja el conjunto de instrucciones del hardware, en el nivel medio los lenguajes de programación tales como Ada, C, Smalltalk, se utilizan para crear una descripción procedimental del programa, en el nivel más alto el lenguaje utiliza iconos gráficos u otra simbología para representar los requisitos de una solución.

1.4. APLICACIONES

El software puede aplicarse en cualquier situación en la que se haya definido previamente un conjunto especifico de pasos procedimentales[3]. El contenido y determinismo de la información son factores importantes a considerar para determinar la naturaleza de una aplicación de software. El contenido se refiere al significado y a la forma de la información de entrada y salida. El determinismo de la información se refiere a la predecibilidad del orden y del tiempo de llegada de los datos.

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

a) **De sistemas**. Se caracteriza por la fuerte interacción con el hardware de la computadora y su gran uso por múltiples usuarios, los más representativos son: compiladores, editores y utilitarios de gestión de archivos, utilidades para el manejo de periféricos, procesadores de telecomunicaciones.
b) **De tiempo real**. Mide, analiza o controla sucesos del mundo real conforme ocurren.
c) **De gestión**. Las aplicaciones en esta área estructuran los datos existentes para facilitar las operaciones comerciales o gestionar la toma de decisiones. El procesamiento de información comercial constituye la mayor de las áreas de aplicación del software. Los sistemas discretos[4] han evolucionado hacia el software de sistemas de información de gestión (SIG).
d) **De ingeniería y científico**. Caracterizado por los algoritmos de manejo cuantitativo (numérico). Sus aplicaciones van desde la astronomía a la vulcanología, desde el análisis de la presión de los automotores a la dinámica orbital de las lanzaderas espaciales y desde la biología molecular a la fabricación automática.
e) **Empotrado**. Reside en memoria solo de lectura y se utiliza para controlar productos y sistemas de los mercados industriales y de consumo.
f) **De computadoras personales**. El procesamiento de textos, las hojas de calculo, los gráficos por computadora, multimedia, entretenimientos, gestión de base de datos, aplicaciones financieras, acceso a bases de datos externas, son algunas de sus cientos de aplicaciones.

[3] Específicamente un algoritmo o conjunto lógico de pasos.
[4] Entre los que se cuentan las nominas o planillas de haberes, las cuentas de haberes, los inventarios, etc.

g) **De inteligencia artificial**. Utilizan conocimiento, sus aplicaciones están en las áreas de sistemas expertos, reconocimiento de patrones, pruebas de teoremas, teoría de juegos, redes neuronales, algoritmos genéticos, etc.

1.5. CRISIS DEL SOFTWARE

Al haberse alcanzado la etapa de crisis en el software, lo que realmente se tiene es una aflicción crónica, en el sentido de que es una crisis duradera y que vuelve a aparecer con frecuencia. El término crisis alude a un conjunto de problemas que aparecen en el desarrollo del software de las computadoras. Los problemas no se limitan al software que "no funciona correctamente". Es más, el mal abarca los problemas relacionados a como desarrollar software, como realizar el mantenimiento del volumen cada vez mayor de software existente y como poder atender la demanda creciente del software.

1.6. MITOS

Muchas de las causas de la crisis del software se pueden encontrar en una mitología que surge durante los primeros años del desarrollo del software. A diferencia de los mitos antiguos, que ofrecían a los hombres lecciones dignas de tener en cuenta, los mitos del software propagaron información errónea y confusión. Los mitos del software tienen varios atributos que los hacen insidiosos; por ejemplo, aparecieron como declaraciones razonables de hechos, tuvieron un sentido intuitivo y frecuentemente fueron promulgados por expertos que "estaban al día".

Existen varios mitos clasificados en tres áreas: de gestión, del cliente y de los desarrolladores.

1.6.1. De Gestión

Los gestores con responsabilidad sobre el software están normalmente bajo la presión de cumplir los presupuestos, hacer que no se retrase el proyecto y mejorar la calidad.

Mito # 1: Se tiene ya un libro que está lleno de estándares y procedimientos para construir software. ¿No le proporciona ya a los desarrolladores todo lo que necesitan saber?
Realidad: Esta muy bien que el libro exista, pero ¿Se utiliza? ¿Conocen los trabajadores de su existencia? ¿Refleja las prácticas modernas de desarrollo de software? ¿Es completo? . En muchos casos las respuestas a estas preguntas es un rotundo ¡no!.

Mito # 2: Los desarrolladores disponen de las herramientas de desarrollo de software más avanzadas; después de todo, cuentan con las computadoras más modernas.
Realidad: Se necesita mucho más que el último modelo de computadora grande para hacer desarrollo de software de gran calidad. La herramienta de ingeniería de software asistida por computadora[5] es más importante que el hardware para conseguir buena calidad y productividad.

[5] Corresponde a la sigla inglesa Computer Aided Software Engineering (CASE).

Mito # 3: Si se falla en la planificación, se puede adicionar más programadores y adelantar el tiempo perdido[6].
Realidad: El desarrollo de software no es un proceso mecánico como la fabricación. Cuando se añaden nuevas personas al proyecto de software retrasado, la necesidad de aprender y comunicarse con el equipo puede y hace que se reduzca la cantidad de tiempo gastado en el desarrollo productivo. Puede añadirse gente, pero solo de una manera planificada y bien coordinada.

1.6.2. Del Cliente

Un cliente que solicita una aplicación de software puede ser una persona del despacho contiguo, un grupo técnico de la sala de abajo, el departamento de ventas o una compañía exterior que solicita un software bajo contrato. Los mitos conducen a que el cliente se cree una falsa expectativa y finalmente, quede insatisfecho con el que desarrolla el software.

Mito # 1: Una declaración general de los objetivos es suficiente para comenzar a escribir los programas, se puede trabajar con los detalles más adelante.
Realidad: Una mala definición inicial es la causa principal del trabajo baldío en el desarrollo del software. Es esencial una descripción formal y detallada del ámbito de la información, funciones, rendimiento, interfaces, ligaduras del diseño y criterios de validación. Estas características pueden determinarse sólo después de una exhaustiva comunicación entre el cliente y el analista.

Mito # 2: Los requisitos del proyecto cambian continuamente, pero los cambios pueden acomodarse fácilmente, ya que el software es flexible.
Realidad: Es verdad que los requisitos del software cambian, pero el impacto del cambio varia según el momento en que se introduzca. Si se pone cuidado al dar la definición inicial, los cambios solicitados al principio pueden acomodarse fácilmente. El cliente puede revisar los requisitos y recomendar las modificaciones con relativamente poco impacto en los costos. Cuando los cambios se solicitan durante el diseño del software, el impacto en los costos crece rápidamente. Acordados los recursos a utilizar y establecido el esqueleto del diseño, los cambios pueden producir trastornos que requieran recursos adicionales e importantes modificaciones del diseño; es decir, costos adicionales. Cuando se solicitan cambios al final de un proyecto, pueden producir un orden de magnitud mas caro que el mismo cambio solicitado al principio.

1.6.3. De los Desarrolladores

Los mitos en los cuales aún creen los desarrolladores de software se han ido fomentando durante cuatro décadas de cultura informática. Estas viejas formas y actitudes tardan en morir.

Mito # 1: Una vez que se escribe un programa y se hace funcionar el mismo, el trabajo de programación ha terminado.
Realidad: Alguien dijo una vez "cuanto más pronto se comience a escribir código, más se

[6] Más conocido como horda mongoliana.

tardara en terminarlo". Los datos indican que entre el cincuenta y sesenta por ciento de todo el esfuerzo dedicado a un programa se realizará después de la primera entrega del software al cliente.

Mito # 2: Hasta que no se cuente con un programa ejecutable, realmente no se puede comprobar su calidad.
Realidad: Desde el inicio de un proyecto de software debe aplicarse uno de los mecanismos más efectivos para garantizar la calidad del software: la revisión técnica formal. La revisión del software es un filtro de calidad que es mucho más efectivo que la prueba, para encontrar ciertas clases de defectos en el software.

Mito # 3: Lo único que se entrega al terminar el proyecto es el programa funcionando.
Realidad: Un programa que funciona es sólo una parte de una configuración de software que incluye programas, documentos y datos. La documentación es la base de un buen desarrollo y, lo que es más importante, proporciona guías para la tarea de mantenimiento de software.

Ejercicios # 1

1. Investigar las razones por las cuales el software es el alma mater de la ciencia de las computadoras.
2. Ampliar el concepto de componentes del software mediante la escritura de un documento que resuma las ventajas y desventajas de su uso.
3. Cual es su opinión particular respecto a la crisis del software. ¿Se continuara aún en esta crisis o ya se ha superado la misma?.
4. El software es la característica que diferencia muchos productos y sistemas informáticos. Proporcionar ejemplos de al menos tres productos y de al menos un sistema en el que el software, no el hardware, sea el elemento diferenciador.
5. En los años 50 y 60 la programación de computadoras constituía un arte aprendido en un entorno básicamente experimental. ¿Cómo ha afectado esto a las practicas de desarrollo del software actualmente?.
6. Muchos autores han tratado el impacto de la era de la información. Proporcionar cinco ejemplos positivos y cinco negativos que muestren el impacto del software en nuestra sociedad.
7. A medida que el software se difunde de manera bastante rápida, los riesgos para los clientes se convierten en una preocupación significativa. Desarrollar un escenario realista de un juicio final donde el fallo de una computadora podría causar un gran daño, económico o social.
8. Revisar de manera detallada los mitos de software, adicionar un mito nuevo a cada categoría.
9. Escriba un artículo resumen de las ventajas recientes en una de las áreas de aplicación de software principales. Puede elegir las áreas de: inteligencia artificial, redes neuronales artificiales, algoritmos genéticos, agentes inteligentes, realidad virtual e interfaces humanas avanzadas.
10. Desarrollar una tabla que contenga las aplicaciones potenciales de la ingeniería del software, basándose en la clasificación proporcionada en este capítulo.

2 TECNOLOGÍA ESTRATIFICADA

La ingeniería de software se caracteriza por ser una tecnología estratificada, los diferentes enfoques conducen a las siguientes definiciones.

Según Fritz Bauer[7] "*La ingeniería del software es el establecimiento y uso de principios robustos de la ingeniería a fin de obtener económicamente software que sea fiable y que funcione eficientemente sobre máquinas reales*".

Una definición más completa desarrollada por la IEEE es: "*La ingeniería de software es la aplicación de un enfoque sistemático, disciplinado y cuantificable hacia el desarrollo, operación y mantenimiento del software*".

2.1. PROCESO, MÉTODOS Y HERRAMIENTAS

La ingeniería del software es una tecnología multicapa. Los cimientos que son la base de la ingeniería del software están orientados hacia la calidad. La gestión de calidad total y las filosofías similares fomentan una cultura continua de mejora de proceso, y es esta cultura la que conduce últimamente al desarrollo de enfoques cada vez más robustos para la ingeniería del software.

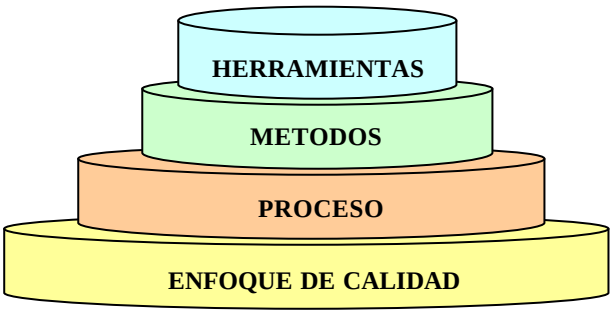

Fig. 2.1. Capas de la ingeniería del software

El fundamento de la ingeniería del software es la capa proceso. El proceso es la unión que mantiene juntas las capas de tecnología y que permite un desarrollo racional y oportuno de la ingeniería del software. Las áreas clave del proceso forman la base del control de gestión de proyectos del software y establecen en contexto en el que se aplican los métodos técnicos, se producen resultados del trabajo, se establecen hitos, se asegura la calidad y se gestiona el cambio de manera adecuada.

[7] Ver el artículo: *Software Engineering: A Report on a Conference Sponsored by the NATO Science Committee.*

Los métodos indican como construir de manera técnica el software. Los métodos abarcan una gama de tareas que incluyen análisis de requisitos, diseño, construcción de programas, pruebas y mantenimiento. Los métodos 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 proporcionan un soporte automático o semi-automático 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 de software denominada ingeniería de software asistida por computadora.

2.2. VISIÓN GENERAL

La ingeniería es el análisis, diseño, construcción, verificación y gestión de entidades técnicas o sociales. Con independencia de la entidad a la que se va aplicar ingeniería, se deben cuestionar y responder a las siguientes preguntas:

a) ¿Cuál es el problema a resolver?
b) ¿Cuales son las características de la entidad que se utiliza para resolver el problema?
c) ¿Cómo se construirá la entidad?
d) ¿Que enfoque se utilizará para evitar errores en el diseño y construcción de la entidad?
e) ¿Cómo se apoyará la entidad a las solicitudes de correcciones, adaptaciones y mejoras?

El estudio que se emprende está centrado en la entidad denominada software de computadora.

2.2.1. Fases de la Ingeniería del Software

El trabajo que se asocia a la ingeniería de software se puede dividir en tres fases genéricas:

a) **Fase de definición**. Esta centrada en resolver el qué. Durante la definición, el ingeniero de software intenta identificar qué información ha de ser procesada, qué función y rendimiento se desea, que comportamiento del sistema y que interfaces serán establecidas, que restricciones de diseño. Esta fase tiene tres tareas importantes: ingeniería de sistemas o de información, planificación del proyecto de software y análisis de requisitos.
b) **Fase de desarrollo**. Centrada en el cómo. Durante el desarrollo un ingeniero del software intenta definir como se diseñaran las estructuras de datos, la arquitectura del software, los detalles procedimentales, las interfaces, la traducción al lenguaje de programación y la prueba. Las tareas fundamentales de esta fase son: diseño del software, generación de código y prueba del software.
c) Fase de mantenimiento. Centrada en el cambio. Asociada a la corrección de errores, a las adaptaciones requeridas a medida que evoluciona el entorno de software y a

los cambios requeridos por el cliente. Durante esta fase se encuentran cuatro tipos de cambios: corrección, adaptación, mejora y prevención.

2.3. PROCESO DEL SOFTWARE

Un proceso de software se puede caracterizar de acuerdo a la Fig. 2.2.

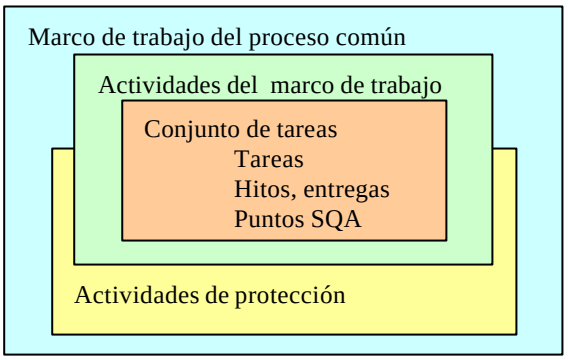

Fig. 2.2. Proceso del software

2.4. MODELOS DE PROCESO

Todo el desarrollo del software se puede caracterizar como un bucle de resolución de problemas en el que se encuentran cuatro etapas distintas mostradas en la Fig. 2.3.

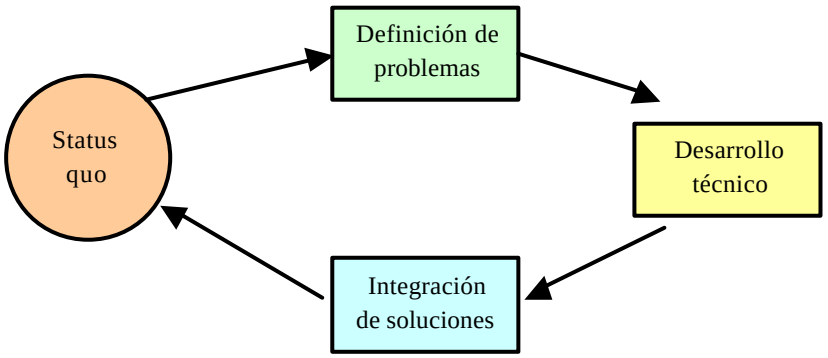

Fig. 2.3. Bucle de solución de problemas

El status quo representa el estado actual de sucesos.

2.4.1. Lineal Secuencial

Se denomina también ciclo de vida básico o modelo de cascada.

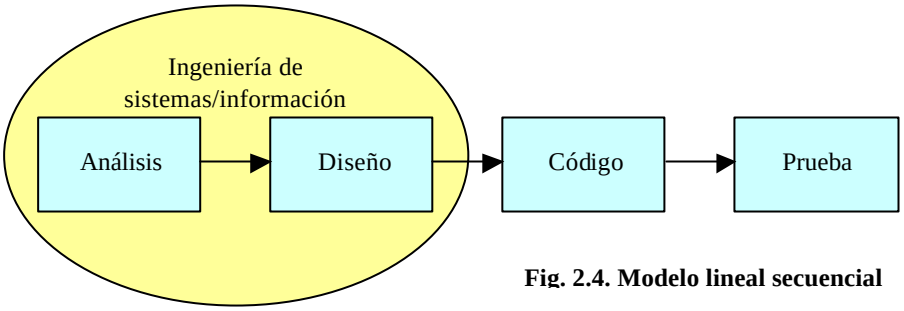

Fig. 2.4. Modelo lineal secuencial

2.4.2. Construcción de Prototipos

Ofrece su enfoque a través del paradigma de construcción de prototipos.

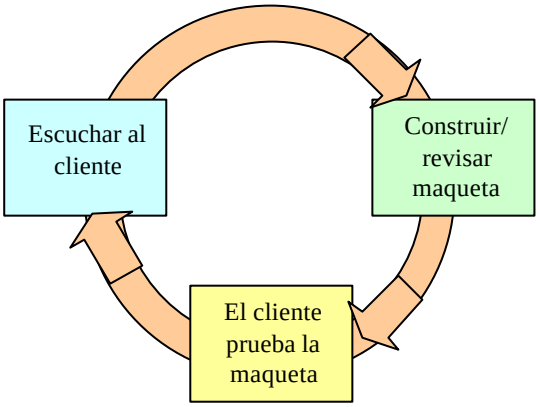

Fig. 2.5. Modelo de construcción de prototipos

Este paradigma se inicia con la recolección de requerimientos. 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. Luego aparece un "diseño rápido". El diseño rápido está centrado en una representación de los aspectos del software que serán visibles para el usuario-cliente. El diseño rápido lleva a la construcción de un prototipo. El prototipo lo evalúa el cliente-usuario y lo utiliza para refinar los requisitos del software a desarrollar. La interacción ocurre cuando el prototipo satisface las necesidades del cliente, a la vez que permite que el desarrollador comprenda mejor lo que se necesita hacer.

2.4.3. Desarrollo Rápido de Aplicaciones

El Desarrollo Rápido de Aplicaciones (DRA)[8] es un modelo de proceso del desarrollo del software lineal secuencial que enfatiza un ciclo de desarrollo extremadamente corto. El

[8] De la sigla inglesa Rapid Application Development (RAD).

modelo DRA es una adaptación a “alta velocidad” del modelo lineal secuencial en el que se logra el desarrollo rápido utilizando un enfoque de construcción basado en componentes.

Cuando se utiliza para aplicaciones de sistemas de información, el enfoque RDA comprende las fases descritas en la Fig. 2.6.

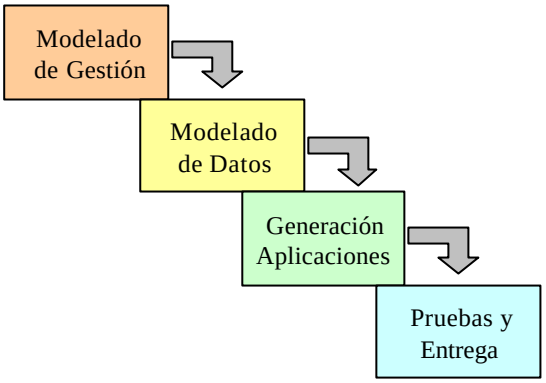

Fig. 2.6. Fases del modelo DRA

2.5. MODELOS DE PROCESO EVOLUTIVO DEL SOFTWARE

El software al igual que todos los sistemas complejos, evoluciona con el tiempo. Los requisitos de gestión y de productos a menudo cambian conforme a que el desarrollo proceda haciendo que el camino que lleva al producto final no sea real; las estrictas fechas tope del mercado hacen que no sea posible finalizar un producto completo, por lo que se debe introducir una versión limitada para cumplir la presión competitiva y de gestión; 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 haya diseñado explícitamente para acomodarse a un producto que evolucione con el tiempo.

2.5.1. Modelo Incremental

Combina elementos del modelo lineal secuencial con la filosofía interactiva de construcción de prototipos.

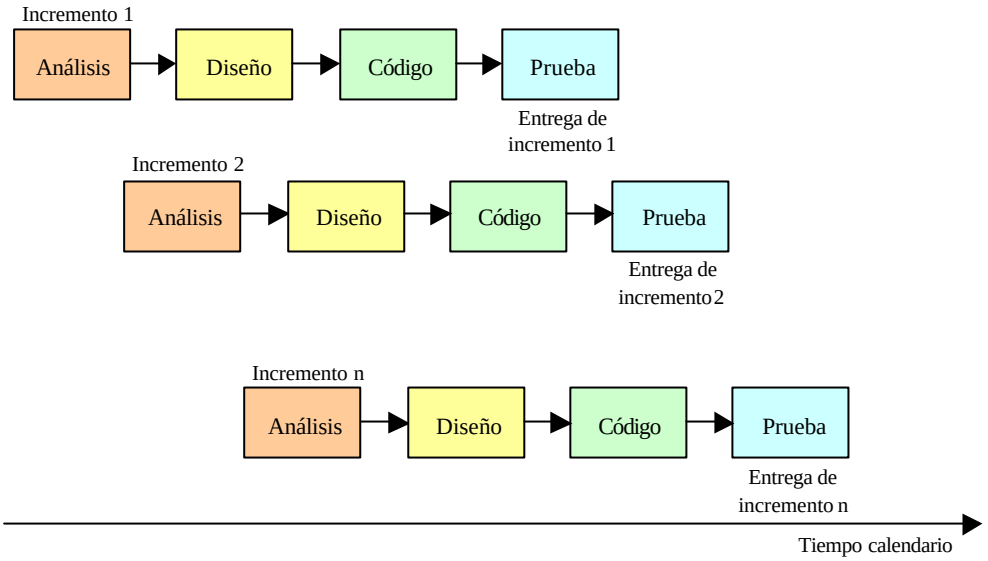

Fig. 2.7. Modelo incremental

2.5.2. Modelo en Espiral

Es un modelo que acompaña la naturaleza interactiva 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 ultimas iteraciones, se producen versiones cada vez más completas de la ingeniería del sistema.

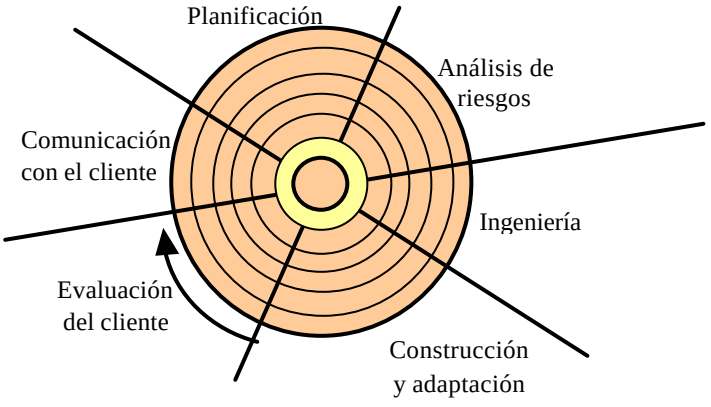

Fig. 2.8. Modelo espiral típico

Ejercicios # 2

1. Investigar tres definiciones, de ingeniería del software, diferentes a las mencionadas en el presente capitulo, confrontarlas con la definición proporcionada por el IEEE.
2. Las capas de la ingeniería del software se sitúan encima de la capa titulada "foco de calidad". Esto implica un programa de gestión de calidad total. Investigar y desarrollar un esquema de los principios clave de un programa de gestión de calidad total.
3. El modelo del caos sugiere que un bucle de resolución de problemas se puede aplicar en cualquier grado de resolución. Estudiar la forma en la que se puede aplicar el bucle para: (1) comprender los requisitos de un producto de tratamiento nuevo, (2) desarrollar un componente de corrección ortográfica y gramática avanzada para el procesamiento de textos, (3) generar el código para un módulo de programas que determine el sujeto, predicado y objeto de una oración.
4. Describir en pocas palabras el análisis comparativo que plantea la siguiente interrogante: ¿qué es más importante el producto o el proceso?.
5. Ampliar las características del modelo de desarrollo rápido de aplicaciones (DRA).
6. El modelo DRA a menudo se une a una herramienta CASE. Desarrollar el resumen de una herramienta típica CASE que soporte DRA.
7. Proporcionar cinco ejemplos de proyectos de desarrollo de software que sean adecuados para construir prototipos.
8. Mostrar un ejemplo de aplicación del modelo en espiral.
9. A medida que se sale hacia afuera por el modelo en espiral ¿qué puede decir del software, se está desarrollando o manteniendo?.
10. Muchas personas creen que la única forma de mejora de magnitud en la calidad del software y en su productividad es a través del ensamblaje de componentes. Encontrar artículos recientes sobre la temática y escribir un resumen de lo investigado.

3 CONCEPTOS Y PRINCIPIOS DEL ANALISIS

3.1. INTRODUCCIÓN

Para el éxito en el desarrollo del software es fundamental la comprensión de los requisitos necesarios para la construcción del producto software. Muchas veces es irrelevante lo bien diseñado o codificado que pueda estar un programa si es que no cuenta con una etapa de análisis previa. La tarea del análisis de requisitos es un proceso de descubrimiento, refinamiento, modelado y especificación. Se refina en detalle el ámbito del software, inicialmente establecida por el ingeniero de sistemas y refinada durante la planificación temporal del proyecto de software. Se crean modelos de los requisitos de datos, flujo de información y control, y del comportamiento operativo. Se analizan soluciones alternativas y se asignan a diferentes elementos del software.

3.2. ANÁLISIS DE REQUISITOS

Es una tarea de la ingeniería del software que cubre el hueco entre la definición del software a nivel sistema y el diseño del software. El análisis de requisitos permite refinar la definición del software y construir los modelos de los dominios de datos, funcional y de comportamiento a ser considerados en el producto software. El análisis de requisitos proporciona al diseñador del software los siguientes modelos: de datos, arquitectónico, de interfaz y procedimental. La especificación de requisitos proporciona al diseñador y al cliente los medios para valorar la calidad una vez que se ha construido el software.

El análisis de requisitos del software puede dividirse en las siguientes cinco actividades de esfuerzo:

a) **Reconocimiento del problema**. Reconocer los elementos básicos del problema, tal y como los percibe el usuario o el cliente.
b) **Evaluación y síntesis**. Definir todos los objetos de datos observables externamente, evaluar el flujo y contenido de la información y elaborar todas las funciones del software. Esta actividad continua hasta que el analista y el cliente se sienten seguros de que se puede especificar adecuadamente el software para las fases posteriores de desarrollo.
c) **Modelado**. Durante la actividad de evaluación y síntesis de la solución, el analista crea modelos del sistema en un esfuerzo de entender mejor el flujo de datos y de control, el tratamiento funcional, el comportamiento operativo y el contenido de la información.
d) **Especificación**. El modelo sirve como fundamento para el diseño del software y como base para la creación de una especificación del software.

e) **Revisión**. Consiste en llevar adelante la actividad de revisar las especificaciones del software para considerar las mas adecuadas.

3.3. TÉCNICAS DE COMUNICACIÓN

El análisis de requisitos del software siempre comienza con la comunicación entre dos o más partes. Normalmente un cliente tiene un problema que pretende sea resuelto con un resultado producido por un programa computacional. Un desarrollador responde a la solicitud de ayuda del cliente. En este paso la comunicación comienza, pero es común comprobar que el camino de la comunicación al entendimiento está a menudo lleno de baches.

3.3.1. Inicio del Proceso

La técnica de análisis más utilizada para empezar el proceso de comunicación es llevar a cabo una reunión o entrevista preliminar. Gause y Weinberg[9] sugieren que el analista comience preguntando cuestiones de contexto libre, enfocados sobre el cliente, los objetivos generales y los beneficios esperados. Por ejemplo se debería preguntar:

a) ¿Quién está detrás de la solicitud de este trabajo?
b) ¿Quién utilizará la solución?
c) ¿Cuál será el beneficio económico del éxito de una solución?
d) ¿Existe alguna otra alternativa necesaria para la solución?

El segundo conjunto de preguntas permite al analista obtener un entendimiento mejor del problema y al cliente comentar sus opiniones sobre la solución, estas preguntas son:

a) ¿Cómo caracterizaría un buen resultado generado por una buena solución?
b) ¿A que tipo de problema o problemas está dirigida la solución?
c) ¿Podría mostrarme el entorno en que se utilizará la solución?
d) ¿Existen aspectos o restricciones especiales de rendimiento que afecten a la manera de enfocar la solución?

El último conjunto de preguntas se concentra en la eficacia de la reunión, estas se denominan meta – preguntas y proponen la siguiente lista:

a) ¿Es usted la persona adecuada para responder a estas preguntas? ¿Sus respuestas son oficiales?
b) ¿Son adecuadas mis preguntas para el problema que tiene?
c) ¿Estoy preguntando demasiado?
d) ¿Hay alguien más que pueda proporcionar información adicional?
e) ¿Hay algo mas que debería preguntarle?

3.3.2. Técnica para Facilitar Especificaciones de una Aplicación (TFEA)

[9] Ver la obra: *Exploring Requirements: Quality Before Design*.

Este enfoque es partidario de la creación de un equipo de clientes y desarrolladores que trabajan juntos para identificar el problema, proponer soluciones, negociar diferentes enfoques y especificar un conjunto preliminar de requisitos de la solución.

Se han propuesto muchos enfoques diferentes de TFEA. Cada uno utiliza un escenario ligeramente diferente, pero todos aplican alguna variación en las siguientes directrices básicas:

a) La reunión se celebra en un lugar neutral y acuden tanto los clientes como los desarrolladores.
b) Se establecen normas de preparación y participación.
c) Se sugiere una agenda lo suficientemente formal como para cubrir todos los puntos importantes pero lo suficientemente informal como para animar el libre flujo de ideas.
d) Se elige un coordinador que controle la reunión.
e) Se utiliza un mecanismo de definición[10].
f) El objetivo es identificar el problema, proponer elementos de solución, negociar diferentes enfoques y especificar un conjunto preliminar de requisitos de la solución.

3.3.3. Despliegue de la Función de Calidad (DFC)

Es una técnica de gestión de calidad que traduce las necesidades del cliente en requisitos técnicos del software. DFC se concentra en maximizar la satisfacción del cliente, haciendo énfasis en entender lo que resulta valioso para el cliente y luego desplegar estos valores a lo largo del proceso de ingeniería. DFC identifica tres tipos de requisitos:

1) **Requisitos normales**. Son requisitos básicos asociados a objetivos y metas para un producto software, si estos están presentes el cliente quedará satisfecho.
2) **Requisitos esperados**. Son implícitos al producto y pueden ser tan fundamentales que el cliente no los declara explícitamente. Su ausencia puede ser motivo de una insatisfacción significativa.
3) **Requisitos innovadores**. Van más allá de las expectativas del cliente y suelen ser muy satisfactorias como valor agregado al producto.

3.4. PRINCIPIOS DEL ANÁLISIS

Todos los métodos de análisis se relacionan a través del siguiente conjunto de principios operativos:

1) La representación y el entendimiento del dominio de información de un problema. Requiere el examen del dominio de la información. Este dominio contiene tres visiones diferentes de los datos y del control: (1) contenido de la información y relaciones, (2) flujo de la información y (3) estructura de la información. El contenido es definido por los atributos necesarios para crearlo. El flujo representa

[10] Es decir hojas de trabajo, gráficos, carteles, pizarra, etc.

como cambian los datos y el control a medida que se mueven dentro de un sistema. La estructura representa la organización interna de los elementos de datos o de control.

2) La definición de funciones que debe realizar el software. Los modelos se crean para entender de mejor manera la entidad que se va construir como solución aun problema. Para transformar la información el software debe realizar al menos tres funciones genéricas: entrada, procesamiento y salida.
3) La representación del comportamiento del software. La característica estimulo respuesta forma la base del modelo de comportamiento. Un programa de computadora siempre está en un estado, en un modo de comportamiento observable exteriormente[11].
4) La división de los modelos que representan información, función y comportamiento de manera que se puedan descubrir los detalles por capas, para que las partes puedan entenderse fácilmente y establecer luego las interacciones entre las mismas de manera que se pueda conseguir la función global.
5) El proceso de análisis debería ir desde la información esencial hasta el detalle de la implementación. Una visión esencial de los requisitos del software presenta las funciones a conseguir y la información a procesar sin tener en cuenta los detalles de la implementación. La visión de implementación de los requisitos del software introduce la manifestación en el mundo real de las funciones de procesamiento y las estructuras de la información.

3.5. CREACIÓN DE PROTOTIPOS

El análisis normalmente se realiza independientemente del paradigma de ingeniería del software que se aplique. Hay circunstancias que requieren la construcción de un prototipo al principio del análisis, debido a que el modelo es el único medio a través del cual se pueden obtener eficazmente los requisitos. El modelo evoluciona después hacia la producción del software.

El paradigma para la creación de prototipos puede ser:

a) Enfoque cerrado. Se denomina a menudo prototipo desechable. Sirve únicamente para una basta demostración de los requisitos, después se desecha y se hace la ingeniería del software con un paradigma diferente.
b) Enfoque abierto. Denominado prototipo evolutivo. Emplea el prototipo como primera parte de una actividad de análisis a la que seguirá el diseño y la construcción

Para crear prototipos rápidos existen tres clases genéricas de métodos y herramientas: técnicas de cuarta generación, componentes de software reutilizables y finalmente especificaciones formales y entornos para prototipos.

3.6. ESPECIFICACION

El modo de especificación tiene mucho que ver con la calidad de la solución. La

[11] Puede ser imprimiendo, calculando, en estado de espera, etc.

especificación puede verse como un proceso de representación. Los requisitos se representan de manera que como fin ultimo lleven al éxito de la implementación del software.

Los siguientes principios de especificación son adaptados del trabajo de Balzer y Goldman[12]:

1) Separar la funcionalidad de la implementación.
2) Desarrollar un modelo del comportamiento deseado de un sistema que comprenda datos y las respuestas funcionales de un sistema a varios estímulos del entorno.
3) Establecer el contexto en que opera el software especificando la manera en que otros componentes del sistema interactúan con él.
4) Definir el entorno en que operara el sistema.
5) Crear un modelo intuitivo en lugar de un diseño o modelo de implementación.
6) Reconocer que la especificación debe ser tolerante a un posible crecimiento.
7) Establecer el contenido y la estructura de una especificación de manera que acepte cambios.

Para representar una especificación de requisitos debe seguirse el siguiente grupo de directrices:

1) El formato de representación y el contenido deben estar relacionados con el problema.
2) La información contenida dentro de la especificación debe estar escalonada.
3) La numeración de párrafos y diagramas debe indicar el nivel de detalle que se muestra.
4) Los diagramas y otras formas de notación deben restringirse en numero y ser consistentes en su empleo
5) La representación debe permitir revisiones.

Finalmente un esquema para la especificación de los requisitos del software, señalado por Pressman es la siguiente:

I. Introducción
   A. Referencia del sistema
   B. Descripción general
   C. Restricciones del proyecto de software
II. Descripción de la información
   A. Representación del contenido de la información
   B. Representación del flujo de la información
      1. Flujo de datos
      2. Flujo de control
III. Descripción funcional
   A. Partición funcional
   B. Descripción funcional

[12] Ver el artículo: *Principles of Good Specification and their Implications for Specification Languages.*

1. Descripción del procesamiento
2. Restricciones / limitaciones
3. Requisitos de rendimiento
4. Restricciones de diseño
5. Diagramas de soporte

C. Descripción del control
   i. Especificación del control
   ii. Restricciones de diseño

IV. Descripción del comportamiento
   A. Estados del sistema
   B. Eventos y acciones

V. Criterios de validación
   A. Limites del rendimiento
   B. Clases de pruebas
   C. Respuesta esperada del software
   D. Consideraciones especiales

VI. Bibliografía

VII. Apéndice

Ejercicios # 3

1. El análisis de requerimientos del software constituye la fase de comunicación más intensa del proceso de ingeniería del software ¿Por qué suele romperse frecuentemente este enlace de comunicación?
2. Suelen existir serias repercusiones políticas cuando se inicia el análisis de requerimientos del software. Por ejemplo, los trabajadores suelen creer que la seguridad de su trabajo puede verse amenazada por un nuevo sistema automático. Analizar las siguientes interrogantes: ¿Qué es lo que origina tales problemas?¿Cuáles serían las tareas que minimicen estas repercusiones?.
3. Construir una plantilla para la especificación de requerimientos que se aplique al problema del pago de haberes de una empresa de venta de computadoras.
4. Diseñar un plan de comunicación con el cliente que incluya estrategias formales que aseguren la confiabilidad de la propuesta.
5. Estudiar en una percepción propia la formación y currículo ideal de un analista de sistemas.
6. Describir al "cliente" desde el punto de vista de los desarrolladores de sistemas de información, de los constructores de productos basados en computadoras y de los constructores de sistemas.
7. Desarrollar una caja de herramientas para las técnicas de especificación de aplicaciones. La caja de herramientas debe incluir un conjunto de directrices para llevar a cabo una reunión, materiales para facilitar la creación de listas y otros elementos que pudieran ayudar en la definición de requerimientos.
8. Sería posible decir que ¿un manual preliminar de usuario es una forma de prototipo?.
9. Construir una tabla para la selección del enfoque apropiado de creación de prototipos.
10. Investigar la técnica de componentes reutilizables como enfoque para el desarrollo de prototipos.

4 MODELADO DEL ANALISIS

4.1. INTRODUCCIÓN

El modelo de análisis es la primera representación técnica de un sistema. Dos tendencias marcadas dominan el modelado del análisis:

a) Análisis estructurado. Es el método clásico que es utilizado con frecuencia para el modelado.
b) Análisis orientado a objetos. Es un método alternativo basado en la tecnología orientada a objetos.

Tom de Marco[13] describe el análisis estructurado mencionando que, para la fase de análisis es necesario añadir los siguientes fines:

a) Los productos del análisis deben ser de un mantenimiento muy sencillo, especialmente las especificaciones de los requisitos del software.
b) Los problemas de gran tamaño deben ser tratados mediante algún método efectivo de partición.
c) Siempre que sea posible, se deben utilizar gráficos.
d) Se tienen que diferenciar las consideraciones lógicas y las físicas.

Como mínimo es necesario:

a) Una ayuda para realizar una partición de los requisitos y la documentación de las divisiones antes de la especificación.
b) Algún método de seguimiento y evaluación de interfaces.
c) Nuevas herramientas para describir la lógica y la táctica, algo mejores que las descripciones narrativas.

4.2. ELEMENTOS DEL MODELO

El modelo de análisis debe lograr tres objetivos primarios:

1) Describir lo que requiere el cliente
2) Establecer una base para la creación de un diseño del software
3) Definir un conjunto de requisitos que se puedan validar después de la construcción del producto software.

[13] Ver la obra: *Structured Analysis and System Specification*.

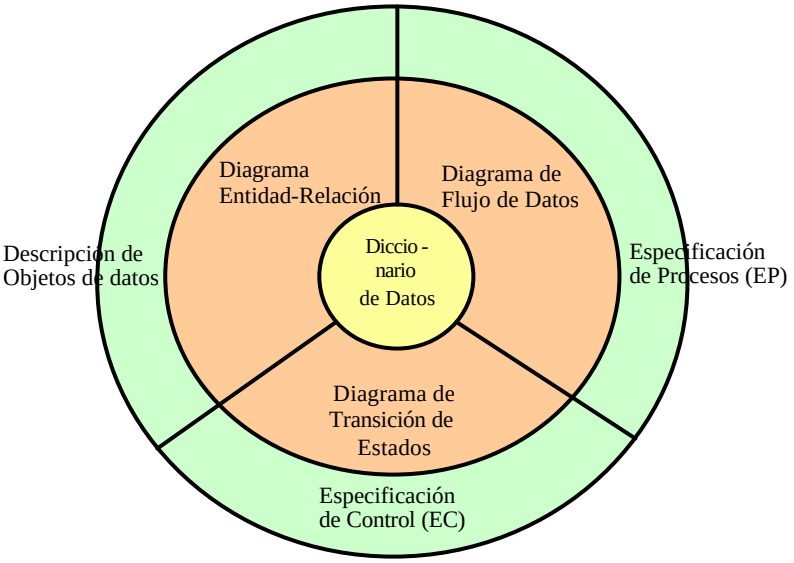

Fig. 4.1. Estructura del modelo de análisis

4.2.1. Modelado de Datos

Los métodos de modelado de datos hacen uso del diagrama entidad-relación (DER). El DER se centra solo en los datos[14], representando una "red de datos" que existe para un sistema dado.

El modelo de datos se compone de tres piezas de información interrelacionadas: el objeto de datos, los atributos que describen el objeto de datos, y la relación que conecta objetos de datos entre sí.

a) **Objetos de datos:** son una representación de cualquier composición de información compuesta que deba comprender el software. Por *composición de información* se entiende todo aquello que tiene un número de propiedades o atributos diferentes.

b) **Atributos:** definen las propiedades de un objeto de datos y se pueden usar para:
   1) Nombrar una ocurrencia del objeto de datos.
   2) Describir la ocurrencia.
   3) Hacer referencias a otra ocurrencia en otra tabla.

c) **Relaciones:** permiten una conexión entre objetos de datos.

[14] Por consiguiente satisfaciendo el primer principio operacional del análisis.

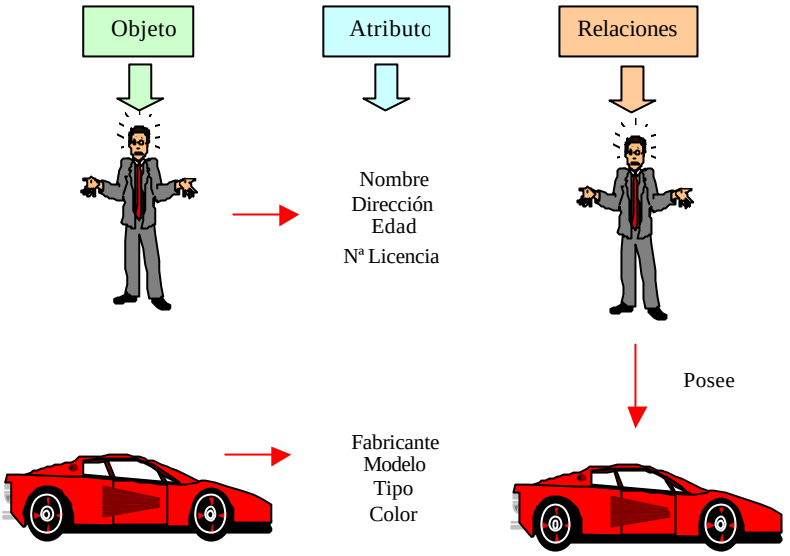

Fig. 4.2. Piezas de información del modelo de datos

Estos elementos básicos del modelado de datos proporcionan la base para el entendimiento del dominio de información de un problema. Sin embargo para comprender la información adicional que está relacionada con estos elementos es necesario definir dos conceptos complementarios: cardinalidad y modalidad.

a) **Cardinalidad**. Es la especificación del número de ocurrencias de un objeto que se relaciona con ocurrencias de otro objeto. La cardinalidad normalmente se expresa de manera simple como "uno" o "muchos". Por ejemplo, un marido puede tener una sola esposa[15], mientras un padre puede tener muchos hijos. Dos objetos se pueden relacionar como:

   - Uno a uno (1:1)
   - Uno a muchos (1:N)
   - Muchos a muchos (M:N)

b) **Modalidad**. La modalidad de una relación es cero si no hay una necesidad explícita de que ocurra una relación o de que sea opcional. La modalidad es 1 si una ocurrencia de la relación es obligatoria.

[15] Esto es discutible pero se produce en la mayoría de las culturas.

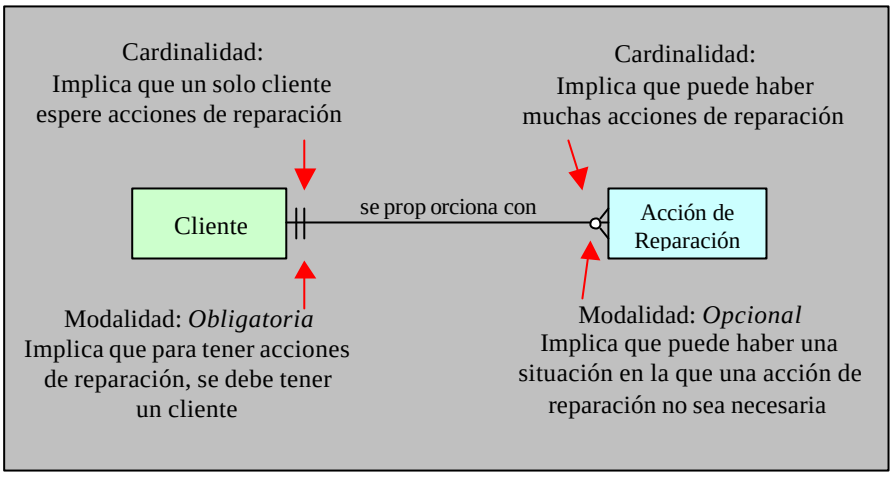

Fig. 4.3. Cardinalidad y modalidad

Por otra parte, la pareja objeto-relación es la piedra angular del modelo de datos. Estas parejas se representan gráficamente mediante el diagrama entidad-relación (DER).

4.2.2. Modelado Funcional

La información se transforma a medida que fluye por un sistema basado en computadora. El sistema acepta entradas en una gran variedad de formas, aplica elementos de hardware, software y humanos para transformar la entrada en salida, y produce salida en una gran variedad de formas. El análisis estructurado es una técnica del modelado del flujo y del contenido de la información.

El diagrama de flujo de datos (DFD) es una técnica que representa el flujo de la información y las transformaciones que se aplican a los datos al moverse desde la entrada hasta la salida. El DFD es conocido también como grafo de flujo de datos o como diagrama de burbujas.

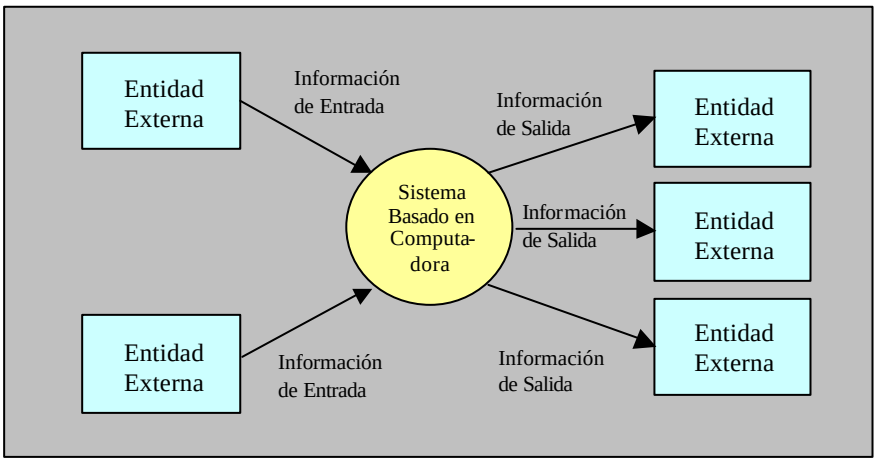

Fig. 4.3. Modelo del flujo de información

4.2.3. Modelado del Comportamiento

El modelado del comportamiento es uno de los principios fundamentales de todos los métodos de análisis de requisitos. Sin embargo, sólo algunas versiones ampliadas del análisis estructurado proporcionan una notación para este tipo de modelado.

El *diagrama de transición de estados* (DTE), representa el comportamiento de un sistema que muestra los estados y los sucesos que hacen que el sistema cambie de estado. Además el DTE indica qué acciones se llevan a cabo como consecuencia de un suceso determinado.

Un *estado* es un modo observable del comportamiento.

4.3. MECANISMOS DEL ANÁLISIS ESTRUCTURADO

Para desarrollar modelos completos y precisos mediante el análisis estructurado, normalmente se utilizan los siguientes pasos:

1) **Creación de un diagrama entidad relación**. Permite que un ingeniero del software especifique que los objetos de datos que entran y salen de un sistema, los atributos que definen las propiedades de estos objetos, y las relaciones entre objetos. Frecuentemente el DER se construye de manera interactiva.

2) **Creación de un modelo de flujo de datos**. El diagrama de flujo de datos permite al ingeniero del software desarrollar los modelos del ámbito de información y del ámbito funcional al mismo tiempo. A medida que se refina el DFD en mayores niveles de detalle, el analista lleva a cabo implícitamente una descomposición funcional del sistema a la vez que se esfuerza en conseguir los principios del análisis operacional[16]. Al mismo tiempo, el refinamiento del DFD produce un refinamiento de los datos a medida que se mueven a través de los proceso que componen la aplicación.

3) **Creación de un modelo de flujo de control**. Debido a que existe una numerosa clase de aplicaciones que se encuentran conducidas por sucesos en lugar de datos, y producen información de control más que informes o visualizaciones, además que procesan información con fuertes limitaciones de tiempo y rendimiento, estas aplicaciones requieren un modelado del flujo de control ademas del modelado del flujo de información. Para crear un diagrama de flujo de control lo primero que se hace es eliminar del modelo de flujo de datos todas las flechas del flujo de datos. Luego se añaden al diagrama los sucesos y los elementos de control[17], así como ventanas[18] a las especificaciones de control.

4) **Especificación del control**. Representa el comportamiento del sistema de dos formas diferentes. La especificación del control (EC) contiene un diagrama de

[16] Descritos en los principios del análisis, capitulo 3 sección 4.
[17] Flechas con líneas discontinuas en los diagramas de flujo de control.
[18] Hace referencia a las barras verticales de los diagramas de flujo de control.

transición de estados que es una especificación secuencial del comportamiento. También contiene una tabla de activación de procesos (TAP), que es una especificación combinatoria del comportamiento.

5) **Especificación del proceso**. Se utiliza para describir todos los procesos del modelo de flujo que aparecen en el nivel final de refinamiento. El contenido de la especificación de procesamiento pue de incluir una narrativa, una descripción en lenguaje de descripción de programas (LDP) del algoritmo del proceso, ecuaciones matemáticas, tablas, diagramas o gráficos. Al proporcionar una especificación del proceso que acompañe cada burbuja del modelo de flujo, el ingeniero del software crea una mini-especificación que se utiliza como primer paso para la creación de la especificación de requisitos del software y, constituye al mismo tiempo una guía para el diseño de componentes del programa que implementará el proceso.

6) **Diccionario de datos**. Está propuesto como una gramática casi formal, que se utiliza para describir el contenido de los objetos definidos durante el análisis estructurado. Casi siempre se implementa el diccionario de datos como parte de una herramienta CASE de análisis y diseño estructurado. Aunque el formato del diccionario varia entre las distintas herramientas, la mayoría contiene la siguiente información: nombre, alias, forma de uso, descripción del contenido e información adicional.

Ejercicios # 4

1. Investigar la historia del análisis estructurado.
2. Averiguar las similitudes y diferencias existentes entre cardinalidad y modalidad.
3. Proporcionar una visión general de otros métodos clásicos de análisis de requisitos.
4. Desarrollar los diagramas: entidad-relación, de flujo de datos y de transición de estados, para un sistema simple de facturas de una empresa pequeña.
5. Construir un sistema de registro de cursos en red para la carrera de informática de la Universidad Mayor de San Andrés.
6. Construir un sistema de facturación para una empresa pequeña de servicios.
7. Dibujar un modelo de nivel de contexto (DFD de nivel 0) para los problemas 5 y 6. Escribir una especificación de procesamiento al nivel de contexto para el sistema.
8. Mediante el DFD de nivel de contexto desarrollado en el problema 7, construir diagramas de flujo de datos de nivel 1 y nivel 2. Utilizar un analizador gramatical en la narrativa de procesamiento al nivel de contexto. Se recuerda que debe especificar el flujo de información etiquetando todas las flechas entre burbujas. Usar nombres significativos para cada transformación.
9. Desarrollar DFC, EC, EP y un diccionario de datos para el sistema de procesamiento de transacciones basadas en Internet para un almacén de venta de computadoras.
10. ¿Significará el concepto de continuidad del flujo de información que si una flecha de flujo aparece como entrada en el nivel 0, entonces en los subsiguientes niveles debe aparecer una flecha de flujo como entrada?. Explicar su respuesta.

5 CONCEPTOS Y PRINCIPIOS DEL DISEÑO

5.1. INTRODUCCIÓN

El diseño es el primer paso para el desarrollo de cualquier producto o sistema de ingeniería. Este puede definirse como el proceso de aplicar distintas técnicas y principios con el propósito de definir un dispositivo, un proceso o un sistema con suficiente detalle como para permitir su realización física.

El objetivo del diseñador es la producción de un modelo o representación de una entidad a ser construida posteriormente. El proceso de diseño desarrolla el modelo, combina la intuición y los criterios basados en la experiencia para la construcción de entidades similares, es guiado en su evolución por un conjunto de principios y heurísticos, finalmente se puede juzgar su calidad a través de un conjunto de criterios.

5.2. DISEÑO E INGENIERIA DEL SOFTWARE

El diseño se encuentra dentro del núcleo del proceso de ingeniería, independiente del paradigma de desarrollo que se utilice.

Una vez que se han establecido los requisitos del software, el diseño es la primera de tres actividades técnicas: diseño, codificación y prueba. Estas actividades son necesarias para construir y verificar el software.

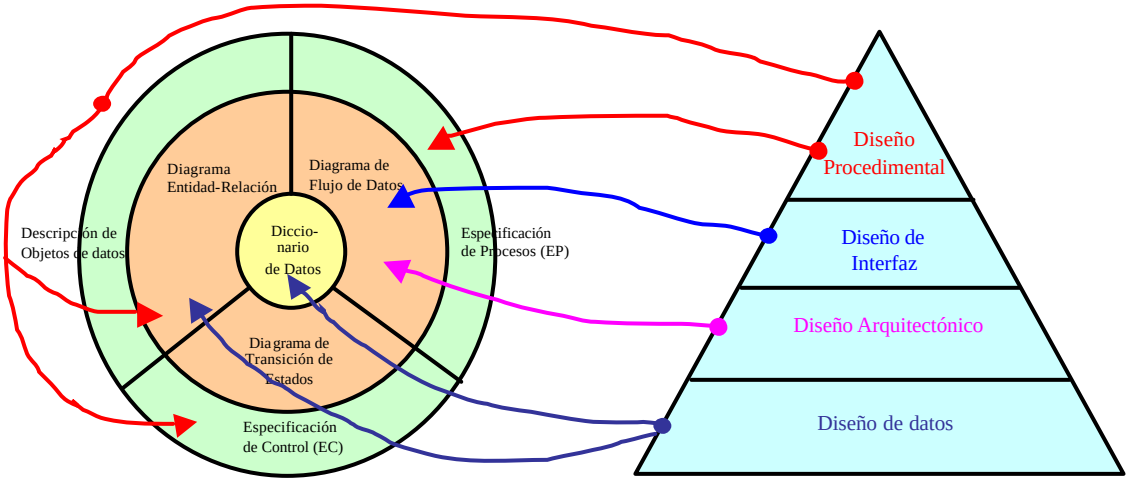

Fig. 5.1. Correspondencia entre los modelos de análisis y diseño

La fase de diseño produce el modelo de diseño con los siguientes elementos:

a) **Diseño de datos**. Transforma el modelo de dominio de la información creado durante el análisis, en las estructuras de datos necesarias para implementar el software. Los objetos de datos y las relaciones definidas en el diagrama entidad-relación y el contenido detallado de datos del diccionario de datos constituyen la base para el diseño de datos.

b) **Diseño arquitectónico**. Define la relación entre los principales elementos estructurales del programa. Se obtiene a partir del modelo de análisis y de la interacción de subsistemas definidos dentro del modelo de análisis.

c) **Diseño de interfaz**. Describe como se comunica el software consigo mismo, con los sistemas que operan con él y con los operadores que lo emplean. Los diagramas de flujo de datos y control proporcionan la información necesaria para el diseño de la interfaz.

d) **Diseño procedimental**. Transforma elementos estructurales de la arquitectura del programa en una descripción procedimental de los componentes del software. Se obtiene a partir de la especificación del proceso, la especificación del control y el diagrama de transición de estados.

MODELO DE ANÁLISIS	***MODELO DE DISEÑO***
Descripción de los objetos de datos	*Diseño procedimental*
Especificación del proceso	*Diseño procedimental*
Especificación del control	*Diseño procedimental*
Diagrama entidad relación	*Diseño de datos*
Diagrama de flujo de datos	*Diseño arquitectónico, Diseño de interfaz*
Diagrama de transición de estados	*Diseño procedimental*
Diccionario de datos	*Diseño de datos*

Tabla 5.1 Correspondencia entre los modelos de análisis y diseño

5.3. PROCESO DE DISEÑO

El diseño del software es un proceso y un modelo a la vez. Es un proceso debido a que representa un conjunto de pasos repetitivos que permiten al diseñador describir todos los aspectos del software a construir. Es modelo debido a su equivalencia con los planos de construcción de una casa para un arquitecto.

5.3.1. Diseño y Calidad del Software

McGlaughlin[19] sugiere tres características que se utilizan como directrices para evaluar un buen diseño del software, en estas el diseño debe:

a) Implementar todos los requisitos explícitos contenidos en el modelo de análisis y

[19] Ver el artículo: *Some Notes on Program Design.*

debe acomodar todos los requisitos implícitos que desea el cliente.
b) Ser una guía que puedan entender los que construyen código y los que prueban y mantienen el software.
c) Proporcionar una idea completa del producto software, enfocando los dominios de datos, funcional y de comportamiento desde la perspectiva de la implementación.

5.4. PRINCIPIOS DEL DISEÑO

Davis[20] sugiere el siguiente conjunto de principios básicos para el diseño del software:

a) Considerar enfoques alternativos basándose en los requisitos del problema.
b) Seguir los pasos del diseño hasta obtener el modelo de análisis.
c) No inventar nada que ya esté inventado.
d) Minimizar la distancia intelectual.
e) Presentar uniformidad e integración.
f) Admitir cambios.
g) No es escribir código y escribir código no es diseñar.
h) Valorar la calidad del diseño mientras se crea el software.
i) Revisar el diseño para minimizar los errores conceptuales.

5.5. CONCEPTOS DEL DISEÑO

En las últimas tres décadas se establecieron un conjunto de conceptos fundamentales para el diseño del software. Estos conceptos proporcionan al diseñador del software una base sobre la que pueden aplicar metodologías de diseño más o menos sofisticadas y ayudan a las responder preguntas acerca de:

a) Criterios que pueden ser utilizados para partir el software en componentes individuales.
b) Separar los detalles de una función o de una estructura de datos de la representación conceptual.
c) Criterios uniformes que definen la calidad técnica.

5.5.1. Abstracción

Cuando se considera una solución modular para cualquier problema, pueden formularse muchos niveles de abstracción.

a) En el nivel superior, se establece una solución en términos amplios.
b) En los niveles inferiores de abstracción, se toma una orientación más procedimental.
c) Durante el análisis de los requisitos del software, se establece la solución en términos de "lo que es familiar al entorno del problema". Conforme se avanza desde lo preliminar hacia el diseño detallado, se reduce el nivel de abstracción.
d) Se alcanza el nivel más bajo de abstracción cuando se genera el código fuente.

[20] Ver la obra: *201 Principles of Software Development.*

5.5.2. Refinamiento

El refinamiento sucesivo es una primera estrategia de diseño descendente. Se desarrolla una jerarquía descomponiendo una declaración macroscópica de una función de una manera sucesiva, hasta que se llega a las sentencias del lenguaje de programación.

El refinamiento es realmente un proceso de elaboración. El refinamiento hace que el diseñador amplíe la declaración original, dando cada vez más detalles conforme se produzcan los sucesivos refinamientos[21].

5.5.3. Modularidad

La arquitectura implica modularidad; esto es, el software se divide en componentes con nombres y ubicaciones determinados, que se denominan módulos y que se integran para satisfacer los requisitos del problema.

La modularidad es el atributo individual del software que permite a un programa ser intelectualmente manejable.

Ejemplo: Sea C(x) una función que define la complejidad de un problema x y E(x) una función que define el esfuerzo[22] requerido para resolver un problema x. Para dos problemas, p_1 y p_2, si $C(p_1) > C(p_2)$ se deduce que: $E(p_1) > E(p_2)$. Para un caso general, este resultado es obvio.

Se ha encontrado otra propiedad interesante: $C(p_1+p_2) > C(p_1)+C(p_2)$. Esta ecuación indica que la complejidad de un problema compuesto por p_1 y p_2 es mayor que la complejidad total cuando se considera cada problema por separado. Entonces, se deduce que: $E(p_1+p_2) > E(p_1)+E(p_2)$.

Esto lleva a una conclusión del tipo “divide y vencerás”. Es más fácil resolver un problema complejo cuando se divide en trozos mucho más manejables.

5.5.4. Arquitectura

La arquitectura del software se refiere a dos características importantes del software de computadora:

a) La estructura jerárquica de los módulos y
b) La estructura de los datos.

5.5.5. Jerarquía de Control

[21] Es decir las continuas elaboraciones.
[22] Tomando en consideración el tiempo.

También denominada estructura del programa, representa la organización[23] de los componentes del programa, o módulos, e implica una jerarquía de control. No representa aspectos procedimentales del software, tales como secuencias de procesos, la ocurrencia u orden de decisiones o la repetición de operaciones.

La profundidad y la amplitud son una indicación del número de niveles de control y de la amplitud global del control, respectivamente. El grado de salida es una medida del número de módulos que están directamente controlados por otros módulos. El grado de entrada indica cuantos controlan directamente a un módulo dado.

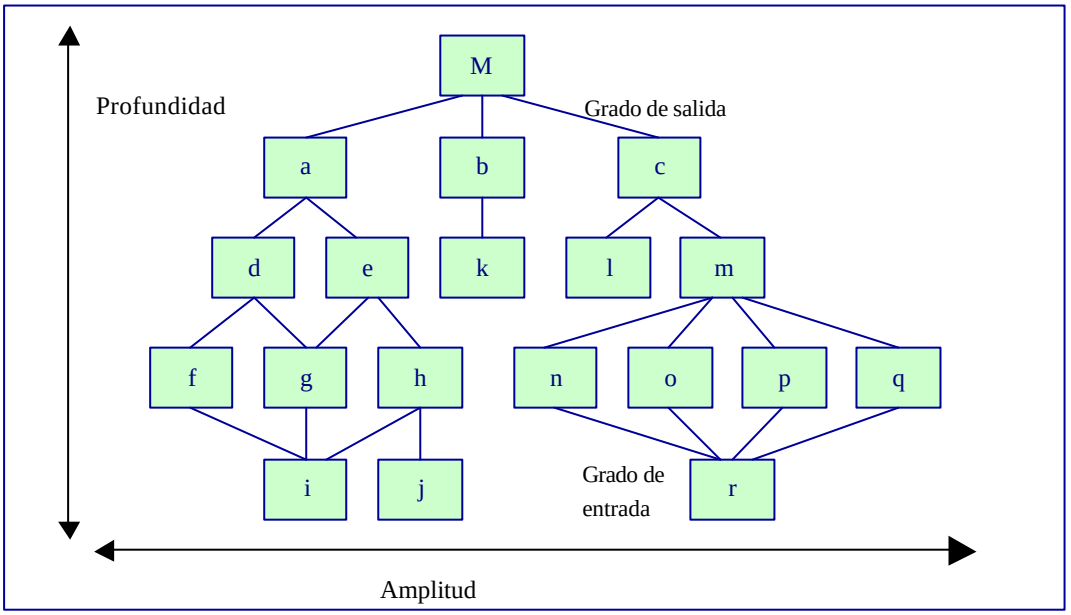

Fig. 5.2 Terminología de la estructura

5.5.6. Partición Estructural

Denominada también partición arquitectónica, hace referencia a la partición del programa tanto de manera horizontal como de manera vertical.

a) Partición horizontal. Define ramas separadas de la jerarquía modular para cada función principal del programa. El enfoque más simple define tres particiones: entrada, transformación de datos y salida.
b) Partición vertical. Denominada también descomposición en factores, sugiere que el control y el trabajo se distribuyan descendentemente.

5.5.7. Estructura de Datos

[23] Frecuentemente jerárquica.

La estructura de datos es una representación de la relación lógica existente entre los elementos individuales de datos. Presenta la organización, los métodos de acceso, el grado de asociatividad y las alternativas de procesamiento de la información.

5.5.8. Procedimiento del Software

El procedimiento del software se centra sobre los detalles de procesamiento de cada módulo individual. El procedimiento debe proporcionar una especificación precisa del procesamiento, incluyendo la secuencia de sucesos, los puntos concretos de decisiones, la repetición de operaciones e incluso la organización/estructura de los datos.

5.5.9. Ocultamiento de la Información

El principio de ocultamiento sugiere que los módulos se han de "caracterizar por decisiones de diseño que los oculten unos a otros". En otras palabras, los módulos deben especificarse y diseñarse de forma que la información[24] contenida dentro de un módulo sea inaccesible a otros módulos que no necesiten tal información.

5.6. DISEÑO MODULAR

Un diseño modular reduce la complejidad, facilita los cambios y produce como resultado una implementación más sencilla, permitiendo el desarrollo en paralelo de las diferentes partes de un sistema.

Para definir módulos en una arquitectura de software, se utiliza la abstracción y el ocultamiento de la información. Ambos atributos deben ser traducidos a características operativas del módulo, caracterizadas por:

a) el historial de incorporación,
b) el mecanismo de activación y
c) el camino de control.

5.6.1. Independencia Funcional

Este concepto es una derivación directa de modularidad, de abstracción y ocultamiento de la información. Se trata de diseñar software de forma que cada módulo se centre en una subfunción específica de los requisitos y tenga una interfaz sencilla cuando se ve desde otras partes de la estructura del software. Algunas consideraciones de la independencia funcional son:

a) El software con módulos independientes es fácil de desarrollar y fácil de mantener.
b) Se reduce la propagación de errores.
c) Se fomenta la reutilización de los módulos

[24] En este caso datos y procedimientos.

La independencia funcional es la clave de un buen diseño y un buen diseño es la clave de la calidad del software.

5.6.2. Cohesión

La cohesión es una extensión natural del concepto de ocultación de información. Un módulo cohesivo ejecuta una tarea sencilla de un procedimiento de software y requiere poca interacción con procedimientos que ejecutan otras partes de un programa. Dicho de manera sencilla, un módulo cohesivo solo hace[25] una cosa.

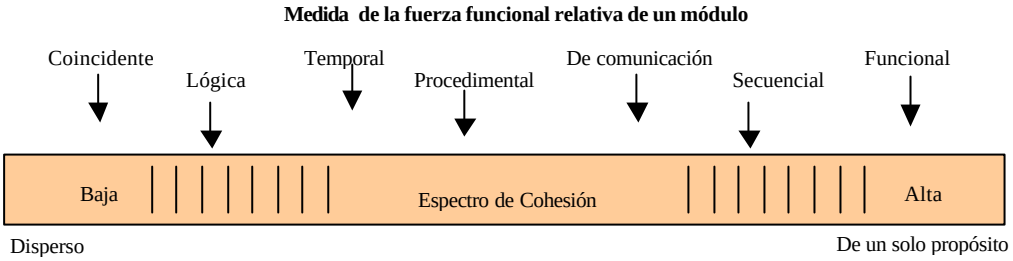

Fig. 5.3. Cohesión

5.6.3. Acoplamiento

El acoplamiento es una medida de la interconexión entre los módulos de una estructura de programa y depende de:

a) la complejidad de las interfaces entre los módulos.
b) del punto en el que se hace una entrada o referencia a un módulo y de los datos que pasan a través de la interfaz.

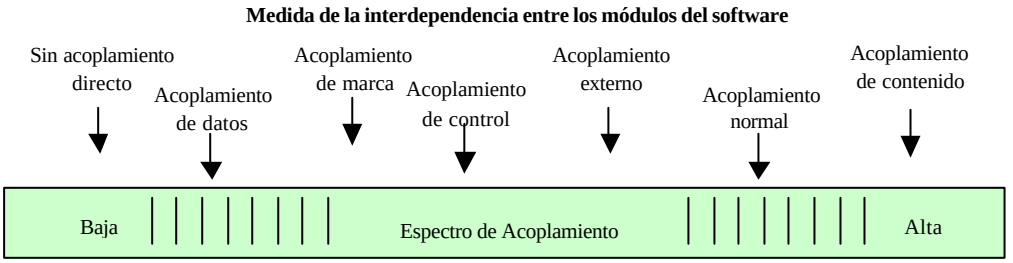

Fig. 5.4. Acoplamiento

Ejercicios # 5

1. Investigar la evolución del diseño del software.
2. ¿Será posible diseñar software cuando se escribe un programa?¿Por qué? ¿Qué es lo

[25] De manera ideal o idealmente.

que hace diferente el diseño del software de la codificación clásica?.

3. Desarrollar tres principios de diseño adicionales a los vistos en este capitulo.
4. Describir los criterios que permiten evaluar un modelo de diseño respecto a su capacidad para definir un sistema modular.
5. Desarrollar ejemplos de tres abstracciones de datos y las abstracciones procedimentales que pueden utilizarse para manipularlos.
6. Aplicar un enfoque de refinamiento paso a paso, para desarrollar tres niveles diferentes de abstracción procedimental aplicado al siguiente problema: “Resuelva iterativamente las raíces de una ecuación”.
7. Algunos lenguajes de programación de alto nivel soportan el procedimiento interno como construcción modular. ¿Cómo afecta esta construcción al acoplamiento? ¿Cómo afecta al ocultamiento de la información?.
8. ¿Cómo están relacionados los conceptos de acoplamiento y movilidad del software? Aporte ejemplos para apoyar su teoría.
9. Desarrollar el concepto de ocultamiento de la información desde un punto de vista propio.
10. Estudiar como la partición estructural puede ayudar a que el software sea más fácil de mantener.

6 METODOS DE DISEÑO

6.1. INTRODUCCIÓN

El diseño se ha descrito como un proceso multifase en el que se sintetizan representaciones de la estructura de datos, estructura del programa, características de la interfaz y detalles procedimentales desde los requisitos de la información.

Los métodos del diseño del software se obtienen del estudio de cada uno de los tres dominios del modelo de análisis: el dominio de los datos, el dominio funcional y el dominio de comportamiento sirven de elementos base para la creación de un buen diseño.

6.2. DISEÑO DE DATOS

El diseño de datos es la primera[26] de las tres actividades de diseño realizadas durante la ingeniería del software. El impacto de la estructura de datos sobre la estructura del programa y la complejidad procedimental, hace que el diseño de datos tenga una gran influencia sobre la calidad del software.

Wasserman[27] propone el siguiente conjunto de principios para especificar y diseñar datos:

1. Los principios sistemáticos de análisis aplicados a la función y el comportamiento también deben aplicarse a los datos.
2. Deben identificarse todas las estructuras de datos y las operaciones que se han de realizar sobre cada una de ellas.
3. Debe establecerse y usarse un diccionario de datos para definir el diseño de los datos y del programa.
4. Se deben posponer las decisiones de diseño de datos de bajo nivel hasta más adelante en el proceso de diseño.
5. La representación de una estructura de datos sólo debe ser conocida por los módulos que hagan un uso directo de los datos contenidos en la estructura.
6. Se debe desarrollar una biblioteca de estructuras de datos útiles y de las operaciones que se les pueden aplicar.
7. El diseño de software y el lenguaje de programación deben soportar la especificación y la realización de tipos abstractos de datos.

Estos principios forman la base de un método de diseño de datos que puede ser integrado en las fases de definición y de desarrollo del proceso de ingeniería del software.

[26] De alguna manera la más importante de las actividades de diseño.
[27] Ver el artículo: *Principles of Systematic Data Design and Implementation.*

6.3. DISEÑO ARQUITECTÓNICO

El objetivo principal del diseño arquitectónico es desarrollar una estructura de programa modular y representar las relaciones de control entre los módulos. El diseño arquitectónico combina la estructura de programas y la estructura de datos y define las interfaces que facilitan el flujo de los datos a lo largo del programa.

6.4. PROCESO DE DISEÑO ARQUITECTÓNICO

El diseño orientado al flujo de datos es un método de diseño arquitectónico que permite una transición cómoda desde el modelo de análisis a una descripción del diseño de la estructura del programa.

La transición desde el flujo de información[28] a una estructura se realiza en un proceso de cinco pasos:

1) Establecer el tipo de flujo de información.
2) Indicar los limites del flujo.
3) Convertir el DFD en la estructura del programa.
4) Definir la jerarquía de control descomponiéndola mediante particionamiento.
5) Refinar la estructura resultante utilizando medidas y heurísticas de diseño.

Los tipos de flujo de información que determinan el método de conversión son dos:

a) **Flujo de transformación**. Refleja la conversión de los datos externos a un formato interno para su procesamiento. La información entra (flujo de entrada) al sistema a lo largo de caminos que transforman (flujo de transformación) los datos externos en un formato interno (flujo de salida).
b) **Flujo de transacción** Refleja la conversión de la información del mundo exterior en una transacción.

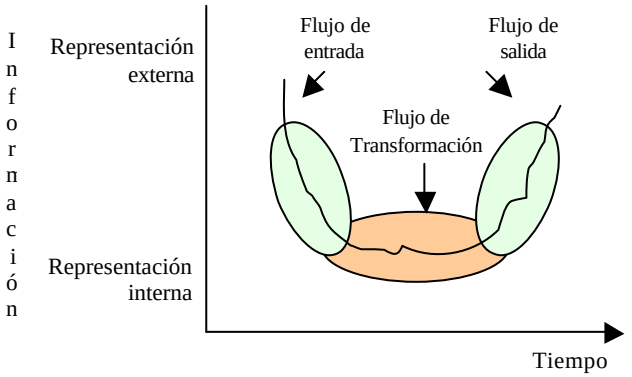

Fig. 6.1 Flujo de transformación

[28] En otras palabras del Diagrama de Flujo de Datos.

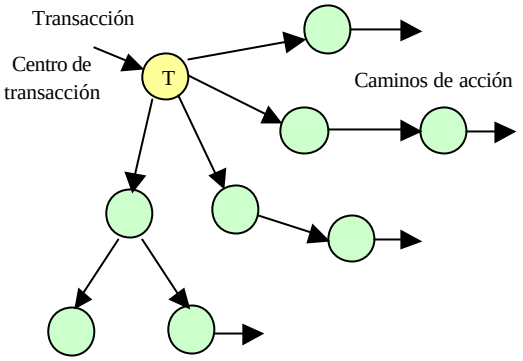

Fig. 6.2 Flujo de transacción

Dentro del DFD de un sistema grande, pueden presentarse ambos flujos de transformación y transacción.

El análisis de las transformaciones es un conjunto de pasos de diseño que permite convertir un DFD en una plantilla predefinida para la estructura del programa.

6.5. DISEÑO DE LA INTERFAZ

El diseño arquitectónico proporciona al ingeniero del software una imagen de la estructura del programa. Como el plano para una casa, el diseño general no está completo sin la representación de las puertas, ventanas y conexiones a servicios para agua, electricidad, teléfono y televisión por cable. Las puertas, ventanas y conexiones a servicios, para el software de computadora componen el diseño de la interfaz del sistema.

El diseño de la interfaz se concentra en tres áreas importantes:

1. Diseño de interfaces entre los módulos software.
2. Diseño de interfaces entre el software y otros productores y consumidores no humanos de información.
3. Diseño de interface entre el hombre y la computadora.

6.6. DISEÑO PROCEDIMENTAL

El diseño procedimental se realiza después de que se ha establecido la estructura del programa y de los datos. En un mundo ideal, la especificación procedimental que define los detalles algorítmicos debería explicarse en lenguaje natural. Desafortunadamente, existe un pequeño problema: el diseño procedimental debe explicar los detalles sin ambigüedad, y esto no es habitual en un lenguaje natural. Por eso se deben utilizar formas más restringidas para la representación.

6.6.1. Programación Estructurada

A finales de los años 60, se propuso el uso de un conjunto de construcciones lógicas con las que podría formarse cualquier programa. Cada construcción tenía una estructura lógica predecible: se entraba a ella por el principio y se salía por el final, y facilitaba el seguimiento del flujo procedimental.

Esas estructuras eran la secuencia, la condición y la repetición.

6.6.2. Notaciones Gráficas de Diseño

El diagrama de flujo es la representación gráfica más ampliamente utilizada para el diseño procedimental, aunque también es el método del que más se ha abusado.

Está constituido por gráficos muy sencillos. Se utilizan rombos, flechas y cuadros para representar las distintas construcciones:

a) Diagramas de Flujo
b) Diagramas de cajas

6.6.3. Notaciones Tabulares de Diseño

En muchas aplicaciones de software, puede ser necesario que un módulo evalúe una compleja combinación de condiciones y, de acuerdo con ellas, seleccione la acción apropiada.

Las tablas de decisión constituyen una notación que traduce las acciones y las condiciones a una forma tabular. Es difícil que se interprete mal una tabla, e incluso puede utilizarse como entrada interpretable por la máquina para un algoritmo manejado por la tabla.

6.6.4. Lenguaje de Diseño de Programas

También denominado LDP, lenguaje estructurado o pseudocódigo, es un lenguaje mezclado que utiliza el vocabulario de un lenguaje[29] y la sintaxis general de otro[30].

Un lenguaje de diseño debe tener las siguientes características:

a) Una sintaxis fija de palabras clave que permitan construir todas las construcciones estructuradas, declarar datos y establecer características de modularidad.
b) Una sintaxis libre en lenguaje natural para describir las características del procesamiento.
c) Facilidades para la declaración de datos, incluyendo estructuras de datos sencillas (arrays) y complejas (listas enlazadas o árboles.)
d) Un mecanismo de definición de subprogramas y de invocación, soportando los distintos modos de descripción de interfaces.

[29] En este caso un lenguaje natural como el español, el ingles, el francés, el alemán, etc.
[30] Este lenguaje es de programación que puede ser estructurado, orientado a objetos, orientado a eventos, etc.

Ejercicios # 6

1. Escribir un documento que contenga directrices para la selección de estructuras de datos basándose en la naturaleza del problema. Comenzar delimitando las estructuras de datos clásicas que se encuentran en el software y luego describir los criterios para la selección de estos en tipos particulares de problemas.
2. Desarrollar un diagrama de flujo de datos para un compilador sencillo, valorar sus características de flujo generales y obtener una estructura de programa. Proporcionar descripciones del procesamiento de cada módulo.
3. ¿Cómo encajaría el concepto de módulo recursivo[31] en la filosofía de diseño propuesta en este capítulo?
4. Estudiar las ventajas y dificultades relativas de aplicar un diseño orientado al flujo de datos en las siguientes áreas: aplicaciones de microprocesador empotrado, análisis de ingeniería, gráficos de computadora, diseño de sistemas operativos, aplicaciones de negocio, diseño de sistemas de gestión de bases de datos, diseño de software de comunicaciones, diseño de compiladores, aplicaciones de control de proceso, aplicaciones de inteligencia artificial.
5. Describir la peor y mejor interfaz con la que haya trabajado, haciendo una crítica en relación con los conceptos de diseño de interfaces.
6. Desarrollar un diseño procedimental para un programa que acepte un texto de longitud arbitraria como entrada y produzca una lista de palabras y su frecuencia de aparición como salida.
7. Desarrollar un enfoque que integre automáticamente los mensajes de error y las ayudas de usuario. Es decir que el sistema reconozca automáticamente el tipo de error y proporcione una ventana de ayuda con sugerencias para solucionarlo. Realice un diseño de software razonablemente completo que considere las estructuras de datos y los algoritmos apropiados.
8. Desarrollar un cuestionario para la evaluación de una interface, que contenga veinte preguntas genéricas que se puedan aplicar a la mayoría de las interfaces.
9. Se ha escrito mucho acerca de la programación estructurada. Escribir un documento que resalte las ventajas y desventajas acerca del uso de construcciones estructuradas.
10. Desarrollar un diseño procedimental para módulos que implemente las siguientes clases: Shell-Metznersort, heapsort, BSST (arbol).

[31] Entiéndase como un módulo que se invoca a si mismo.

7 METODOS DE PRUEBA DEL SOFTWARE

7.1. INTRODUCCIÓN

La prueba del software es un elemento critico para asegurar la garantía de calidad del software; representa una revisión final de las especificaciones, del diseño y de la codificación.

La creciente inclusión del software como un elemento más de muchos sistemas y la importancia de los "costos" asociados a un fallo en la especificación y/o construcción del mismo están motivando a la creación de pruebas minuciosas y bien planificadas. Normalmente las empresas de desarrollo de software emplean entre el 30 al 40 por ciento del esfuerzo total de un proyecto en la prueba del software.

7.2. FUNDAMENTOS DE LA PRUEBA DEL SOFTWARE

En la prueba del software el ingeniero crea una serie de casos de prueba que intentan "demoler" el software construido. La prueba puede ser considerada como uno de los pasos de la ingeniería del software que es de corte destructivo antes que constructivo[32]. En contraposición la gente que desarrolla software es constructiva. La prueba requiere que se descarten las ideas preconcebidas sobre la "corrección" del software y se supere cualquier conflicto de intereses que aparezcan cuando se descubren errores.

7.2.1. Objetivos

Glen Myers[33] establece varias normas que pueden servir como objetivos de la prueba del software:

a) La prueba es un proceso de ejecución de un programa con la intención de descubrir un error.
b) Un buen caso de prueba es aquel que tiene una probabilidad alta de mostrar un error no descubierto hasta entonces.
c) Una prueba tiene éxito si descubre un error no detectado hasta entonces.

Normalmente la prueba no puede asegurar la ausencia de defectos, solamente puede mostrar que existen defectos en el software.

7.2.2. Principios

[32] Al menos desde una óptica puramente psicológica.
[33] Ver la obra: *The Art of Software Testing*.

Davis[34] sugiere el siguiente conjunto de principios para las pruebas del software:

1) Todas las pruebas deberían ser capaces de hacer un seguimiento hasta los requisitos del cliente.
2) Las pruebas deberían planificarse antes de ser ejecutadas.
3) El principio de Pareto es aplicable a la prueba del software.
4) Las pruebas deben empezar por lo "pequeño" y progresar hacia lo "grande".
5) No es posible hacer pruebas exhaustivas.
6) Las pruebas deberían ser conducidas por un equipo independiente.

7.2.3. Facilidad de Prueba

De acuerdo a James Bach[35] la facilidad de prueba es: "… *simplemente lo fácil que se puede probar un programa de computadora*". Como la prueba es tan profundamente difícil, merece la pena saber que se puede hacer para hacerla más sencilla. A veces los programadores están dispuestos a hacer las cosas que faciliten el proceso de prueba y una lista de comprobación de posibles puntos de diseño, características, etc., puede ser útil a la hora de negociar con ellos.

La siguiente lista de comprobación proporciona un conjunto de características que llevan a un software fácil de probar:

a) **Operatividad**. Cuanto mejor funcione, mas eficientemente se puede probar.
b) **Observabilidad**. Lo que se ve es lo que se prueba.
c) **Controlabilidad**. Cuanto mejor se pueda controlar el software, mas se puede automatizar y optimizar.
d) **Capacidad de descomposición**. Controlando el ámbito de las pruebas se puede aislar rápidamente los problemas y llevar a cabo mejores pruebas de regresión.
e) **Simplicidad**. Cuanto menos haya que probar, se puede probar de manera más rápida.
f) **Estabilidad**. Cuanto menos cambios se produzcan, menos interrupciones sufrirán las pruebas.
g) **Facilidad de comprensión**. Cuanta más información se tenga, más inteligentes serán las pruebas.

7.3. DISEÑO DE CASOS DE PRUEBA

Es necesario diseñar pruebas que tengan la mayor probabilidad de encontrar el mayor número de errores con la mínima cantidad de esfuerzo y tiempo posible.

Cualquier producto de ingeniería puede comprobarse de alguna de las siguientes maneras:

a) Conociendo la función especifica para la que fue diseñado el producto, se pueden llevar a cabo pruebas que demuestren que cada función es completamente operativa,

[34] Ver la obra: *201 Principles of Software Development.*
[35] Complementar ingresando a Internet, localizando el grupo de noticias comp.software-eng

y al mismo tiempo busca errores en cada función. Esta prueba se denomina prueba de caja negra. Cuando se considera el software de la computadora esta prueba se refiere a las pruebas que se llevan a cabo sobre la interfaz del software.

b) Conociendo el funcionamiento del producto, se pueden desarrollar pruebas que aseguren que "todas las piezas encajan", o sea, que la operación interna se ajusta a las especificaciones y que todos los componentes internos se han comprobado de manera adecuada. Esta prueba se denomina prueba de caja blanca. Cuando se considera el software de la computadora esta prueba se basa en el examen minucioso de los detalles procedimentales.

7.4. PRUEBA DE CAJA BLANCA

Denominada también prueba de caja de cristal. Es un método de diseño de casos de prueba que utiliza la estructura de control del diseño procedimental para obtener los casos de prueba. Mediante los métodos de prueba de caja blanca, el ingeniero del software puede obtener casos de prueba que:

a) garanticen que se ejercita por lo menos una vez todos los caminos independientes de cada módulo;
b) ejerciten todas las decisiones lógicas en sus vertientes verdadera y falsa;
c) ejecuten todos los bucles en sus limites y con sus limites operacionales, y
d) ejerciten las estructuras internas de datos para asegurar su validez.

Según Jones[36] la naturaleza de los defectos del software se debe a que:

a) Los errores lógicos y las suposiciones incorrectas son inversamente proporcionales a la probabilidad de que se ejecute un camino del programa.
b) A menudo se cree que un camino lógico tiene pocas posibilidades de ejecutarse cuando, de hecho, se puede ejecutar de forma normal.
c) Los errores tipográficos son aleatorios.

Como lo establece Beizer: *"Los errores se esconden en los rincones y se aglomeran en los limites*", es mucho más fácil descubrirlos con la prueba de caja blanca.

7.5. PRUEBA DEL CAMINO BASICO

Es una técnica de prueba de caja blanca, propuesta por Tom McCabe[37], que permite al diseñador de casos de prueba obtener una medida de la complejidad lógica de un diseño procedimental y utilizar esa medida como guía para la definición de un conjunto básico de caminos de ejecución. Los casos de prueba obtenidos del conjunto básico garantizan que durante la prueba se ejecuta por lo menos una vez cada sentencia del programa.

7.5.1. Notación de Grafo de Flujo

[36] Ver la obra: *Programming Productivity: Issues for the 80´s*.
[37] Ver el artículo: *A Software Complexity Measure.*

Representa el flujo de control lógico a través de la siguiente notación:

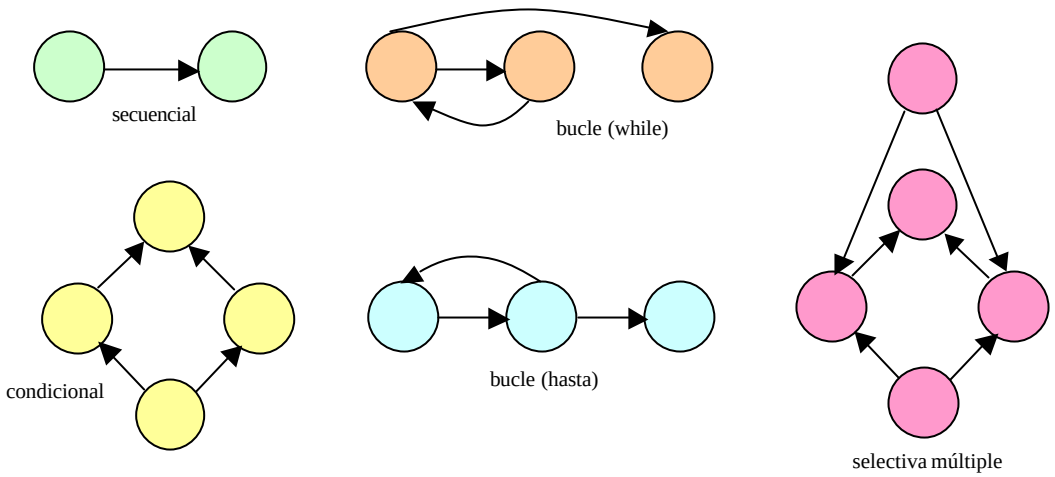

Fig. 7.1. Notación de grafo de flujo

7.5.2. Complejidad Ciclomática

Es una métrica del software que proporciona una medida cuantitativa de la complejidad lógica de un programa. El valor calculado como complejidad ciclomática define el número de caminos independientes del conjunto básico de un programa y proporciona el limite superior para el número de pruebas que se deben realizar para asegurar que se ejecuta cada sentencia al menos una vez.

Un camino independiente es cualquier camino del programa que introduce por lo menos un nuevo conjunto de sentencias de proceso o una nueva condición. En términos del grafo de flujo, un camino independiente está constituido por lo menos por una arista que no haya sido recorrida anteriormente a la definición del camino.

7.5.3. Obtención de Casos de Prueba

Un algoritmo simple para la obtención de casos de prueba se muestra en el siguiente conjunto de pasos:

Algoritmo Media

Utilizando el diseño o el código como base, se dibuja el correspondiente grafo de flujo.
Se determina la complejidad ciclomática del grafo de flujo resultante.
Se determina un conjunto básico de caminos linealmente independientes.
Se prepara los casos de prueba que forzaran la ejecución de cada camino del conjunto básico.

Fin Media

7.5.4. Matrices de Grafos

Una matriz de grafo es una matriz cuadrada cuyo tamaño es igual al número de nodos del grafo de flujo. Cada fila y cada columna corresponden a un nodo específico y las entradas de la matriz corresponden a las conexiones[38] entre los nodos.

Añadiendo un peso de enlace a cada entrada de la matriz, esta se puede convertir en una herramienta para la evaluación de la estructura de control del programa durante la prueba. El peso de enlace proporciona información adicional sobre el flujo de control.. En su forma más sencilla el peso de enlace es 1 si existe conexión, o 0 si no existe conexión.

7.6. PRUEBA DE LA ESTRUCTURA DE CONTROL

La técnica de prueba de camino básico es una de las muchas técnicas para la prueba de la estructura de control. Aunque la prueba del camino básico es sencilla y altamente efectiva, no es suficiente por si sola, algunas variantes que complementan esta prueba son:

7.6.1. Prueba de Condición

Es un método de diseño de casos de prueba que ejercita las condiciones lógicas contenidas en el módulo de un programa.

Una expresión relacional esta representada de la siguiente manera: E_1<*operador*>E_2. Donde E_1 y E_2 son expresiones aritméticas y <*operador*> es un operador relacional (>, <, >=, <=, =). Una condición simple es una variable lógica o una expresión relacional. Una condición compuesta está formada por dos o más condiciones simples, operadores lógicos y paréntesis.

Los tipos posibles de componentes en una condición pueden ser: un operador lógico,, una variable lógica, un par de paréntesis, un operador relacional o una expresión aritmética.

Si una condición es incorrecta, entonces es incorrecto al menos un componente de la condición. En este sentido los errores de una condición pueden ser los siguientes:

a) Error en operador lógico.
b) Error en variable lógica.
c) Error en paréntesis lógico.
d) Error en operador relacional.
e) Error en expresión aritmética.

El método de la prueba de condiciones está centrado en la prueba de cada una de las condiciones del programa. Las estrategias de prueba de condiciones tienen generalmente dos ventajas:

a) La cobertura de la prueba de una condición es sencilla

[38] En teoría de grafos hace referencia a las aristas.

b) Proporciona una orientación para generar pruebas adicionales del programa

El propósito de la prueba de condiciones es detectar, no solamente los errores en las condiciones de un programa, sino también otros errores en dicho programa. Se cuenta con una serie de estrategias de prueba de condiciones, de las cuales las más significativas son:

a) **Prueba de ramificaciones**. Consiste en que para una condición compuesta C, es necesario ejecutar al menos una vez las ramas verdadera y falsa de C y cada condición simple de C.
b) **Prueba del dominio**. Requiere la realización de tres o cuatro pruebas para una expresión relacional. Si es sobre el operador relacional se realiza tres pruebas; si además se incluyen las expresiones, la prueba contempla que la diferencia entre las expresiones sea la más pequeña posible.

7.6.2. Prueba de Flujo de Datos

Este método selecciona caminos de prueba de un programa de acuerdo con la ubicación de las definiciones y los usos de las variables del programa.

Debido a que las sentencias de un programa están relacionadas entre si de acuerdo con las definiciones de las variables, el enfoque de prueba de flujo de datos es efectivo para la protección contra errores. Sin embargo, los problemas de medida del cubrimiento de la prueba y de selección de caminos de ensayo para la prueba del flujo de datos son más difíciles que los correspondientes problemas para la prueba de condiciones.

7.6.3. Prueba de Bucles

Los bucles o ciclos son la piedra angula de la gran mayoría de los algoritmos implementados en el software. La prueba de bucles es una técnica de prueba de caja blanca centrada exclusivamente en la validez de las siguientes construcciones de bucles:

a) **Bucles simples.** Debe aplicarse el siguiente conjunto de pruebas, con n que representa el número máximo de pasos permitidos por el bucle:
    - ?? Pasar por alto totalmente el bucle.
    - ?? Pasar una sola vez por el bucle.
    - ?? Pasar dos veces por el bucle.
    - ?? Pasar m veces por el bucle, con m<n.
    - ?? Pasar n-1, n y n+1 veces por el bucle.

b) **Bucles anidados**. Beizer[39] sugiere el siguiente enfoque para ayudar a reducir el numero de pruebas en un bucle anidado:
    - ?? Comenzar por el bucle más interior, estableciendo o configurando los demás bucles con sus valores mínimos.
    - ?? Llevar a cabo las pruebas de bucles simples para el bucle más interior, mientras se mantienen los parámetros de iteración de los bucles externos

[39] Ver la obra: *Software Testing Techniques*.

en sus valores mínimos. Añadir otras pruebas para valores fuera de rango o excluidos.

?? Progresar hacia fuera, llevando a cabo pruebas para el siguiente bucle, pero manteniendo todos los bucles externos en sus valores mínimos y los demás bucles anidados en sus valores típicos.

?? Continuar hasta que se hayan probado todos los bucles.

c) **Bucles concatenados**. Los bucles concatenados se pueden probar mediante el enfoque definido para los bucles simples, siempre y cuando cada uno de los bucles sea independiente del resto. Sin embargo, cuando los bucles no son independientes, se recomienda utilizar el enfoque aplicado para los bucles anidados.

d) **Bucles no estructurados**. Esta clase de bucles debe ser rediseñada para que se ajusten a las construcciones de la programación estructurada.

7.7. PRUEBA DE CAJA NEGRA

Denominada también prueba de comportamiento o prueba de partición. Está centrada en los requisitos funcionales del software. Este tipo de prueba permite al ingeniero del software obtener un conjunto de condiciones de entrada que ejercitan completamente todos los requisitos funcionales de un programa. Es un enfoque complementario a las técnicas de caja blanca que intenta descubrir diferentes tipos de errores.

Las pruebas de caja negra intentan encontrar errores de las siguientes categorías:

a) Funciones incorrectas o ausentes.
b) Errores de interfaz.
c) Errores en la estructura de datos o en accesos a bases de datos externas.
d) Errores de rendimiento.
e) Errores de inicialización y de terminación.

La prueba de caja negra se aplica durante las fases posteriores de la prueba, centrando su atención en el campo de la información. Estas pruebas se diseñan para responder a las siguientes preguntas:

a) ¿Cómo se prueba la validez funcional?
b) ¿Qué clases de entrada compondrán los buenos casos de prueba?
c) ¿Es el sistema particularmente sensible a ciertos valores de entrada?
d) ¿De que forma están aislados los límites de una clase de datos?
e) ¿Qué volúmenes y niveles de datos tolerará el sistema?
f) ¿Qué efectos sobre la operación del sistema tendrán combinaciones específicas de datos?

Mediante las técnicas de prueba de caja negra se obtiene un conjunto de casos de prueba que satisfacen los siguientes criterios:

a) Casos de prueba que reducen, en un coeficiente que es mayor que uno, el número de

casos de prueba adicionales que se deben diseñar para alcanzar una prueba razonable.

b) Casos de prueba que dicen algo sobre la presencia o ausencia de clases de errores en lugar de errores asociados solamente con la prueba que se está realizando.

Ejercicios # 7

1. Myers[40] utiliza el siguiente procedimiento como autocomprobación de su capacidad para especificar una prueba adecuada: un programa lee tres valores enteros. Los tres valores se interpretan como representación de la longitud de los tres lados de un triángulo. El programa imprime un mensaje indicando si el triángulo es escaleno, isósceles o equilátero. Desarrollar un conjunto de casos de prueba que considere que probarán de manera adecuada este programa.
2. Aplicar la técnica de prueba del camino básico a un programa que haya implementado la solución al problema de cifrado y descifrado de datos mediante el algoritmo RSA (Rivier, Samir y Adleman).
3. Detallar los motivos por los cuales no es posible construir un procesador automático de pruebas de software.
4. Especificar, diseñar e implementar una herramienta del software que calcule la complejidad ciclomática para un lenguaje de programación que elija. Utilizar la matriz de grafo como estructura operativa en el diseño.
5. Investigar en que consiste la estrategia de prueba de condición OBR.
6. Desarrollar ejemplos con al menos dos métodos de prueba de caja negra.
7. Mencionar las aplicaciones principales de la prueba de entorno.
8. Proporcionar al menos tres ejemplo en los que la prueba de la caja negra pueda dar la impresión de que “todo está bien”, mientras que la prueba de caja blanca pudiera descubrir errores. Indicar por lo menos tres ejemplos en los que la prueba de caja blanca pueda dar la impresión de que “todo está bien”, mientras que la prueba de caja negra pudiera descubrir errores.
9. Seleccionar una interfaz grafica de usuario para un programa en el que esté familiarizado y diseñar una serie de pruebas para ejercitar dicha interfaz.
10. Realizar una prueba a un manual de usuario de una aplicación que utilice frecuentemente. Encontrar al menos una falla y/o un error en la documentación.

[40] Ver la obra: *The art of software testing.*

8 ESTRATEGIAS DE PRUEBA DEL SOFTWARE

8.1. INTRODUCCIÓN

Las estrategias de prueba del software integran las técnicas de diseño de casos de prueba en una serie de pasos bien planificados que tienen como resultado una construcción correcta del software. Una estrategia de prueba del software proporciona un mapa para la organización de control de calidad y del cliente, que debe seguir el responsable de desarrollo. Dicho mapa describe los pasos que se deben seguir como parte de la prueba, cuándo se debe planificar y cuanto esfuerzo, tiempo y recursos es necesario emplear. Cualquier estrategia de prueba debe incorporar la planificación de la prueba, el diseño de casos de prueba, la ejecución de las pruebas y la agrupación y evaluación de los resultados.

Una estrategia de prueba debe ser bastante flexible para promover la creatividad y la adaptabilidad necesarias para adecuar la prueba a todos los grandes sistemas basados en software. A la vez, la estrategia debe ser lo suficientemente rígida para promover un seguimiento razonable de la planificación y la gestión a medida que progresa el proyecto.

8.2. ENFOQUE ESTRATÉGICO

Los diferentes métodos de prueba están empezando a ser agrupados en distintas filosofías y enfoques diferentes. Según Pressman estas filosofías y enfoques constituyen lo que se denomina estrategia.

La prueba es un conjunto de actividades que se pueden planificar por anticipado y por consiguiente ejecutadas de manera sistémica. Las distintas estrategias de prueba del software tienen las siguientes características:

a) La prueba comienza en el nivel de modulo y trabaja “hacia fuera” integrando todo el sistema.
b) De acuerdo al momento, son apropiadas diferentes técnicas de prueba.
c) La prueba la ejecuta el responsable del desarrollo del software y un grupo independiente de pruebas.
d) La prueba y la depuración son actividades diferentes, pero la depuración se debe incluir en cualquier estrategia de prueba.

Una estrategia de prueba del software debe proporcionar pruebas de bajo nivel que verifiquen la implementación correcta de los segmentos de código fuente, así como pruebas de alto nivel que validen las principales funciones del sistema frente a los requerimientos del cliente.

8.2.1. Verificación y Validación

La prueba del software es también conocida como verificación y validación (V&V). La verificación hace referencia al conjunto de actividades que aseguran la implementación correcta de una función específica. La validación se refiere a un conjunto diferente de actividades que aseguran que el producto software se ajusta a los requerimientos del cliente. Boehm[41] define la V&V de la siguiente manera:

a) Verificación: ¿Se está construyendo el producto de manera correcta?
b) Validación: ¿Se está construyendo el producto correcto?

La V&V comprenden un rango amplio de actividades de garantía de calidad del software, estas incluyen revisiones técnicas formales, auditorias de configuración y calidad, supervisión del rendimiento, simulación, estudio de factibilidad, revisión de documentación, análisis de algoritmos, prueba de desarrollo, prueba de cualificación y prueba de instalación.

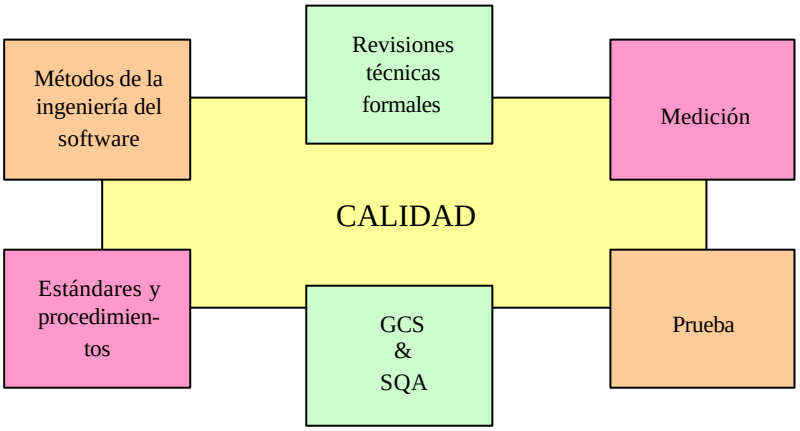

Fig.8.1. Obtención de calidad del software

8.2.2. Organización para la Prueba del Software

El responsable del desarrollo del software es el responsable de probar las unidades individuales[42] del programa, asegurándose de que cada una lleva a cabo las funciones para las cuales fue diseñada. En muchos casos, también se encargará de la prueba de integración, el paso de prueba que lleva a la construcción de la estructura total del sistema. Solamente una vez que la arquitectura del software esté completa entra en juego un grupo independiente de prueba (GIP).

El papel del GIP consiste en eliminar los problemas asociados con el hecho de permitir al

[41] Ver la obra: *Software Engineering Economics*.
[42] Es decir los módulos.

constructor que pruebe lo que ha construido. Una prueba independiente elimina el conflicto de intereses que, de otro modo, estaría siempre presente.

El responsable de desarrollo y el GIP trabajan estrechamente a lo largo del proyecto de desarrollo de software para asegurar que se realizan pruebas exhaustivas. Mientras se ejecutan las pruebas, el desarrollador debe estar disponible para corregir las fallas que se van descubriendo.

8.2.3. Estrategia de Prueba del Software

El proceso conjunto de ingeniería del software y prueba del software puede verse, de manera abstracta, como la espiral de la Fig. 8.2.

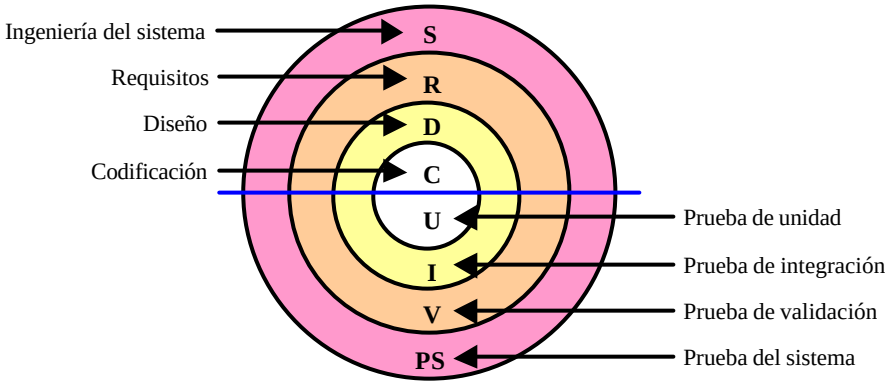

Fig. 8.2 Estrategia de prueba

El movimiento hacia el interior de la figura conduce al proceso de desarrollo propio de la ingeniería del software; la estrategia de prueba del software puede observarse a partir de la prueba de unidad hasta la prueba del sistema.

8.3. ASPECTOS ESTRATÉGICOS

Tom Gilb[43] plantea que se deben abordar los siguientes puntos si se desea implementar con éxito una estrategia de prueba del software.

a) Especificar los requisitos del producto de manera cuantificable antes de que se realicen las pruebas.
b) Establecer los objetivos de la prueba de manera explícita.
c) Comprender a los usuarios del software construido y desarrollar un perfil para cada categoría de usuario.
d) Desarrollar un plan de prueba que haga hincapié en la "prueba de ciclo rápido"
e) Construir un software "robusto" diseñado para probarse a si mismo.
f) Usar revisiones técnicas formales efectivas como filtro de la prueba.
g) Llevar a cabo revisiones técnicas formales para evaluar la estrategia de prueba y los

[43] Ver el artículo: *What we fail to do in our Current Testing Culture.*

propios casos de prueba.

h) Desarrollar un enfoque de mejora continua al proceso de prueba.

8.4. PRUEBA DE UNIDAD

La prueba de unidad está centrada en el proceso de verificación en la menor unidad del diseño del software: el módulo. Utilizando la descripción del diseño procedimental como guía, se prueban los caminos de control importantes, con el fin de descubrir errores dentro del límite del módulo. La complejidad relativa de las pruebas y de los errores descubiertos está limitada por el alcance estricto establecido por la prueba de unidad. La prueba de unidad siempre está orientada a la prueba de caja blanca y este paso se puede llevar a cabo en paralelo para múltiples módulos.

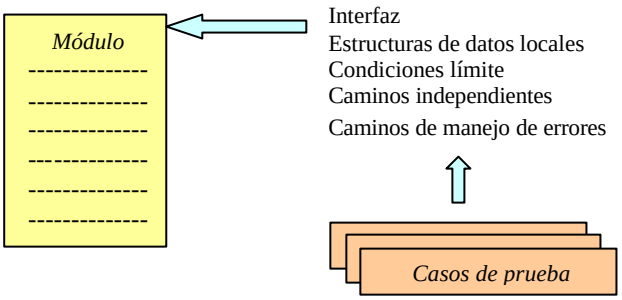

Fig. 8.3. Prueba de unidad

Primeramente se prueba la interfaz del modulo para asegurar que la información fluye de forma adecuada hacia y desde la unidad del programa que esta siendo probado. Se examinan las estructuras de datos locales para asegurar que los datos que se mantienen temporalmente conserven su integridad durante todos los pasos de ejecución del algoritmo.

Se prueban las condiciones límite para asegurar que el módulo funciona correctamente en los límites establecidos como restricciones de procesamiento. Se ejercitan todos los caminos independientes de la estructura de control con el fin de asegurar que todas las sentencias del módulo se ejecutan por lo menos una vez. Finalmente se prueban todos los caminos de manejo de errores.

8.5. PRUEBA DE INTEGRACIÓN

La prueba de integración es una técnica sistemática para construir la estructura del programa mientras que, al mismo tiempo, se llevan a cabo pruebas para detectar errores asociados con la interacción. El objetivo es tomar los módulos probados en unidad y construir una estructura de programa que esté de acuerdo con lo que dicta el diseño.

Con relación a la construcción del programa existen dos tipos de integración:

a) Integración no incremental. Se construye el programa utilizando el enfoque “big bang”. Se combinan todos los módulos por anticipado. Se prueba todo el programa

en conjunto. Normalmente se llega al caos. Se encuentra un gran conjunto de errores. La corrección se hace difícil, debido a que es complicado aislar las causas al tener la vasta extensión del programa completo. Una vez que se corrigen esos errores aparecen otros nuevos y el proceso continúa en lo que parece ser un ciclo sin fin.

b) Integración incremental. Es la antitesis del enfoque "big bang". El programa se construye y se prueba en pequeños segmentos en los que los errores son más fáciles de aislar y corregir, es más probable que se pueda probar completamente las interfaces y se pueda aplicar un enfoque de prueba sistemática.

Las estrategias de integración incremental más utilizadas son la integración descendente y la integración ascendente.

a) En la integración descendente se integran los módulos moviéndose hacia abajo por la jerarquía de control, comenzando con el módulo de control principal[44]. Los módulos subordinados al módulo de control principal se van incorporando en la estructura en profundidad o en la estructura en amplitud.
b) En la integración ascendente se empieza la construcción y la prueba con los módulos atómicos. Debido a que los módulos se integran de abajo hacia arriba, el procesamiento requerido de los módulos subordinados siempre estará disponible, eliminándose la necesidad de resguardos.

8.6. PRUEBA DE VALIDACIÓN

La validación se consigue mediante una serie de pruebas de caja negra que demuestran la conformidad con los requisitos. Un plan de prueba planifica las pruebas a ejecutarse y un procedimiento de prueba define los casos de prueba específicos que se usarán para demostrar la conformidad con los requisitos. Tanto el plan como el procedimiento estarán diseñados para asegurar que se satisfacen todos los requisitos funcionales, que se alcanzan todos los requisitos de rendimiento, que la documentación es correcta e inteligible y que se alcanzan otros requisitos[45].

Una vez que se procede con cada caso de prueba de validación, puede producirse alguna de las siguientes condiciones:

a) Las características de funcionamiento o de rendimiento están de acuerdo con las especificaciones y son aceptables o
b) Se descubre una desviación de las especificaciones y se crea una lista de deficiencias. Las desviaciones raramente se pueden corregir antes de terminar el plan, frecuentemente es necesario negociar con el cliente un método para resolver las deficiencias.

Un elemento importante del proceso de validación es el repaso de la configuración[46], este

[44] Es decir el programa principal.
[45] Entre estos portabilidad, compatibilidad, recuperación de errores y facilidad de mantenimiento.
[46] Hace referencia a la post configuración o auditoria de la configuración.

repaso intenta asegurar que todos los elementos de la configuración del software se han desarrollado de manera adecuada, están catalogados y tienen el suficiente detalle para facilitar la fase de mantenimiento dentro del ciclo de vida del software

La mayoría de los constructores de software llevan a cabo un proceso denominado prueba alfa y beta para descubrir errores que aparentar que solamente el usuario final puede descubrir.

La prueba alfa es conducida por un cliente en el lugar de desarrollo. Se utiliza el software de manera natural, con el encargado del desarrollo "mirando por encima del hombro" del usuario, registrando errores y problemas de uso. Las pruebas alfa se llevan a cabo en un ambiente controlado.

La prueba beta se lleva a cabo en uno o más lugares de clientes por los usuarios finales del software. Normalmente el encargado de desarrollo no está presente. Así la prueba beta es una aplicación "en vivo" del software en un entorno que no puede ser controlado por el equipo de desarrollo. El cliente registra todos los problemas que encuentra e informa a intervalos regulares al equipo de desarrollo, se lleva a cabo las modificaciones y se prepara una versión del producto de software para toda la base de clientes.

8.7. PRUEBA DEL SISTEMA

La prueba del sistema está constituida por una serie de pruebas diferentes cuyo propósito primordial es ejercitar profundamente el sistema basado en computadora. Aunque cada prueba tiene un propósito distinto, todas trabajan para verificar que se han integrado adecuadamente todos los elementos del sistema y que realizan las funciones apropiadas.

Algunos tipos de pruebas del sistema importantes para sistemas basados en software son:

a) **Prueba de recuperación.** Es una prueba del sistema que obliga al fallo del software de muchas formas y verifica que la recuperación se lleva a cabo apropiadamente. Si la recuperación es automática hay que evaluar la corrección de la reinicialización, de los mecanismos de recuperación de datos y del rearranque. Si la recuperación requiere la intervención humana, hay que evaluar los tiempos medios de reparación para determinar si están dentro de unos limites aceptables.

b) **Prueba de seguridad.** Intenta verificar que los mecanismos de protección incorporados en el sistema lo protegerán, de hecho, de la penetración impropia.

c) **Prueba de resistencia.** Ejecuta un sistema de forma que demande recursos en cantidad, frecuencia o volúmenes anormales. Por ejemplo: (1) diseñar pruebas especiales que generen diez interrupciones por segundo, cuando las normales son una o dos; (2) incrementar las frecuencias de datos de entrada en un orden de magnitud con el fin de comprobar cómo responden las funciones de entrada; (3) ejecutar casos de prueba que requieran el máximo de memoria o de otros recursos; (4) diseñar casos de prueba que puedan dar problemas con el esquema de gestión de memoria virtual; (5) diseñar casos de prueba que produzcan excesivas búsquedas de

datos residentes en disco. Esencialmente, el encargado de la prueba intenta tirar abajo el programa.

d) **Prueba de rendimiento.** Esta diseñada para probar el rendimiento del software en tiempo de ejecución dentro del contexto de un sistema integrado. La prueba de rendimiento se da durante todos los pasos del proceso de prueba. Incluso al nivel de unidad, se debe asegurar el rendimiento de los módulos individuales a medida que se llevan a cabo las pruebas de caja blanca.

8.8. ARTE DE LA DEPURACIÓN

La depuración aparece como consecuencia de una prueba efectiva, o sea, cuando un caso de prueba descubre un error. Aunque la depuración puede y debe ser un proceso ordenado, sigue teniendo mucho de arte. Un ingeniero del software, al evaluar los resultados de una prueba, a menudo se encuentra con una indicación "sintomática" de un problema en el software. Esto es, la manifestación externa de un error, y la causa interna del error pueden no estar relacionados de una forma obvia. El proceso mental, erróneamente comprendido, que conecta un síntoma con una causa es la depuración.

Ejercicios # 8

1. Describir las diferencias entre verificación y validación. ¿Utilizan las dos los métodos de diseño de casos de prueba y las estrategias de prueba?
2. Listar algunos problemas que puedan estar asociados con la creación de un grupo independiente de prueba (GIP). ¿Estarán formados por las mismas personas el GIP y el grupo de aseguramiento de la calidad del software (SQA)?
3. Si pudiera seleccionar solamente tres métodos de diseño de casos de prueba para aplicarlos durante la prueba de unidad ¿Cuáles serían y por qué?
4. El concepto de "antipurgado" es una manera extremadamente efectiva de proporcionar asistencia de depuración de depuración integrada cuando se descubre un error. (a) desarrollar una serie de directrices para el antipurgado, (b) Estudiar las ventajas de utilizar esta técnica, (c) estudiar las desventajas.
5. ¿Cómo podría afectar la planificación del proyecto a la prueba de integración?
6. Investigar y comentar acerca de los errores más comunes durante la prueba de unidad.
7. ¿Será posible o incluso deseable aplicar la prueba de unidad en cualquier circunstancia? Mencionar ejemplos que justifiquen su respuesta
8. ¿Quién debería llevar a cabo la prueba de validación: el desarrollador del software o el usuario? Mencionar la justificación de su respuesta.
9. Desarrollar una estrategia de prueba completa para un sistema de planillas de una empresa de servicios de comunicación. Documentar en una especificación de prueba.
10. Investigar y desarrollar acerca del modelo de fallos denominado modelo logarítmico de Poisson de tiempo de ejecución.

9 MÉTRICAS TÉCNICAS DEL SOFTWARE

9.1. INTRODUCCIÓN

Tradicionalmente se emplean medidas para entender mejor los atributos de los modelos que se crean, pero fundamentalmente se emplean las medidas para valorar la *calidad* de los productos de ingeniería o los sistemas que se construyen.

Las métricas técnicas del software proporcionan una manera sistemática de valorar la calidad basándose en un conjunto de "reglas claramente definidas". Proporcionan al ingeniero del software una visión interna en el acto en lugar de una visión a posteriori, permitiéndole descubrir y corregir problemas potenciales antes que se convierta en defectos catastróficos

9.2. CALIDAD DEL SOFTWARE

Un software de alta calidad es una de las metas más importantes en el desarrollo de un producto software. La calidad del software hace hincapié en la concordancia con los requisitos funcionales y de rendimiento explícitamente establecidos, los estándares de desarrollo explícitamente documentados y las características implícitas que se esperan de todo software desarrollado profesionalmente. Esta definición hace énfasis en los siguientes tres puntos:

a) Los requisitos del software son la base de las medidas de calidad. La falta de concordancia con los requisitos es una falta de calidad..
b) Unos estándares específicos definen un conjunto de criterios de desarrollo que guían la manera en que se hace la ingeniería del software. Si no se siguen los criterios, habrá seguramente poca calidad.
c) Existe un conjunto de requisitos implícitos que ha menudo no se nombran. Si el software cumple con sus requisitos explícitos pero falla en los implícitos, la calidad del software no será fiable.

La calidad del software es una compleja mezcla de factores que variarán a través de diferentes aplicaciones y según los clientes que las pidan.

9.2.1. Factores de Calidad de McCall

Los factores que afectan la calidad del software se pueden categorizar en dos grandes grupos:

a) Factores que se pueden medir directamente, como por ejemplo los defectos por

punto de función.

b) Factores que se pueden medir sólo indirectamente, como por ejemplo la facilidad de uso o mantenimiento.

En todos los casos debe aparecer la medición, debe ser posible comparar el software con varios datos y llegar a una conclusión sobre la calidad.

McCall[47] y sus colegas propusieron en 1997 una categorización útil de factores que afectan a la calidad del software, estos factores se concentran en tres aspectos importantes de un producto software:

a) Características operativas.
b) Capacidad de cambios
c) Adaptabilidad a nuevos entornos.

Los factores de calidad de McCall se resumen en la siguiente figura:

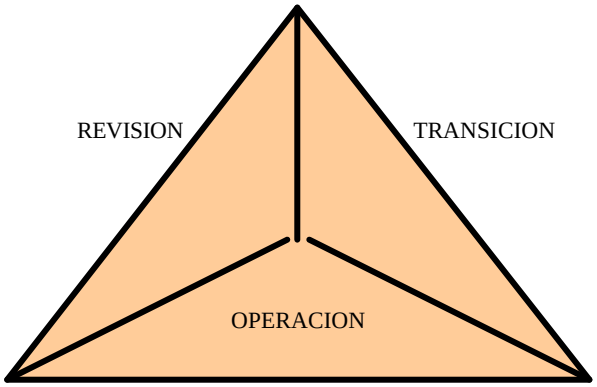

Fig. 9.1. Factores de calidad del software de McCall

Para la *operación del producto* se emplean los siguientes factores:

a) **Corrección**. Hasta donde satisface un programa su especificación y logra los objetivos de la misión del cliente.
b) **Fiabilidad**. Hasta donde se puede esperar que un programa lleve a cabo su función pretendida con la exactitud requerida.
c) **Eficiencia**. La cantidad de recursos informáticos y de código necesarios para que un programa realice su función.
d) **Integridad**. Hasta dónde se puede controlar el acceso al software o a los datos por personas no autorizadas.
e) **Usabilidad**[48]. El esfuerzo necesario para aprender, operar los datos de entrada e interpretar las salidas de un programa.

[47] Ver la obra en tres volúmenes: *Factors in Software Quality*
[48] Se puede decir que la misma representa la facilidad de manejo de la información.

Para la *revisión del producto* se emplean los siguiente factores:

a) **Facilidad de mantenimiento**. El esfuerzo necesario para localizar y arreglar un error en un programa.
b) **Flexibilidad**. El esfuerzo necesario para modificar un programa operativo.
c) **Facilidad de prueba**. El esfuerzo necesario para probar un programa asegurándose que realiza su función pretendida.

Para la *transición del producto* se utilizan los siguientes factores:

a) **Portabilidad**. El esfuerzo necesario para transferir el programa de un entorno de sistema hardware y/o software a otro.
b) **Reusabilidad**[49]. Hasta donde se puede volver a emplear un programa[50] en otras aplicaciones.
c) **Interoperatividad**. El esfuerzo necesario para acoplar un sistema con otro.

9.2.2. Métricas de McCall

Es difícil desarrollar medidas directas de los factores de calidad mencionados, por consiguiente se definen un conjunto de métricas para desarrollar expresiones que utilicen los factores de acuerdo a la siguiente relación:

$$F_q = c_1 \times m_1 + c_2 \times m_2 + \ldots + c_n \times m_n$$

donde F_q es un factor de calidad del software, c_n son coeficientes de regresión y m_n son las métricas que afectan al factor de calidad.

Lamentablemente muchas de las métricas definidas por McCall solamente pueden medirse de manera subjetiva. Las métricas se acomodan en una lista de comprobación que se emplea para "puntuar" atributos específicos del software. El esquema de puntuación propuesto por McCall es una escala del 0 (bajo) al 10 (alto).

Para el esquema de puntuación se emplean las siguientes métricas:

a) **Facilidad de auditoria**. La facilidad con la que se puede comprobar el cumplimiento de los estándares.
b) **Exactitud**. La exactitud de los cálculos y el control.
c) **Estandarización de comunicaciones**. El grado de empleo de estándares de interfaces, protocolos y anchos de banda.
d) **Compleción** El grado con el que se ha logrado la implementación total de una función.
e) **Concisión**. Lo compacto que es el programa en términos de líneas de código.
f) **Consistencia**. El empleo de un diseño uniforme y de técnicas de documentación a lo largo del proyecto de desarrollo del software.
g) **Estandarización de datos**. El empleo de estructuras y tipos de datos estándares a lo

[49] También conocida como la capacidad de reutilización.
[50] En algunos casos pueden ser solamente partes de un programa.

largo del programa.

h) **Tolerancia al error**. El daño causado cuando un programa encuentra un error.
i) **Eficiencia de ejecución**. El rendimiento del funcionamiento de un programa.
j) **Capacidad de expansión**. El grado con que se pueden ampliar el diseño arquitectónico, de datos o procedimental.
k) **Generalidad**. La amplitud de aplicación potencial de los componentes del programa.
l) **Independencia del hardware**. El grado con que se desacopla el software del hardware donde opera.
m) **Instrumentación** El grado con el que el programa vigila su propio funcionamiento e identifica los errores que ocurren.
n) **Modularidad**. La independencia funcional de componentes de programa.
o) **Operatividad**. La facilidad de operación de un programa.
p) **Seguridad**. La disponibilidad de mecanismos que controlan o protegen los programas y los datos.
q) **Autodocumentación** El grado en que el código fuente proporciona documentación significativa.
r) **Simplicidad**. El grado de facilidad con que se puede entender un programa.
s) **Independencia del sistema software**. El grado de independencia del programa respecto a las características del lenguaje de programación no estándar, características del sistema operativo y otras restricciones del entorno.
t) **Trazabilidad**. La capacidad de seguir una representación del diseño o un componente real del programa hasta los requisitos.
u) **Formación** El grado en que ayuda el software a manejar el sistema a los nuevos usuarios.

La relación entre los factores de calidad del software y las métricas nombradas se presentan en una tabla de doble entrada. Normalmente es el ingeniero del software el que asigna los pesos a cada métrica de acuerdo con las características de los productos y negocios locales.

9.2.2. FURPS[51]

Hewlett Packard[52] ha desarrollado un conjunto de factores de calidad del software al que se le ha dado el acrónimo de FURPS: funcionalidad, facilidad de empleo, fiabilidad, rendimiento y capacidad de soporte. Los factores de calidad son cinco y se definen de acuerdo al siguiente conjunto de atributos:

a) **Funcionalidad**. Se valora evaluando el conjunto de características y capacidades del programa, la generalidad de las funciones entregadas y la seguridad del sistema global.
b) **Facilidad de uso**. Se valora considerando factores humanos, la estética, consistencia y documentación general.
c) **Fiabilidad**. Se evalúa midiendo la frecuencia y gravedad de los fallos, la exactitud

[51] Abreviación de las primeras letras de las palabras iglesas: Funcionality, Usability, Reliability, Performance, Supportability.
[52] Ver la obra: *Software Metrics: Establishing a Company-Wide Program.*

de las salidas, el tiempo medio entre fallos, la capacidad de recuperación de un fallo y la capacidad de predicción del programa.

d) **Rendimiento**. Se mide por la velocidad de procesamiento, el tiempo de respuesta, consumo de recursos, rendimiento efectivo total y eficacia.
e) **Capacidad de soporte**. Combina la capacidad de ampliar el programa[53], adaptabilidad y servicios, así como la capacidad de hacer pruebas, compatibilidad, capacidad de configuración, la facilidad de instalación de un sistema y la facilidad con que se pueden localizar los problemas.

9.3. ESTRUCTURA PARA LAS MÉTRICAS DEL SOFTWARE

La medición asigna números o símbolos a atributos de entidades en el mundo real. Para conseguirlo es necesario un modelo de medición que comprenda un conjunto consistente de reglas.

Existe la necesidad de medir y controlar la complejidad del software, es bastante difícil obtener un solo valor para representar una "métrica de calidad", sin embargo es posible desarrollar medidas de diferentes atributos internos del programa como ser: modularidad efectiva, independencia funcional y otros atributos. Estas métricas y medidas obtenidas pueden utilizarse como indicadores independientes de la calidad de los modelos de análisis y diseño.

Los principios básicos de la medición, sugeridos por Roche[54], pueden caracterizarse mediante cinco actividades:

1) **Formulación**. Obtención de medidas y métricas del software apropiadas para la representación del software en cuestión.
2) **Colección**. Mecanismo empleado para acumular datos necesarios para obtener las métricas formuladas.
3) **Análisis**. Cálculo de las métricas y la aplicación de herramientas matemáticas.
4) **Interpretación**. Evaluación de los resultados de las métricas en un esfuerzo por conseguir una visión interna de la calidad de la representación.
5) **Realimentación**. Recomendaciones obtenidas de la interpretación de métricas técnicas transmitidas al equipo software.

Ejiogu[55] define un conjunto de atributos que deberían acompañar a las métricas efectivas del software. La métrica obtenida y las medidas que conducen a ello deberían ser:

a) Simple y fácil de calcular.
b) Empírica e intuitivamente persuasiva.
c) Consistente y objetiva.
d) Consistente en el empleo de unidades y tamaños.
e) Independiente del lenguaje de programación.

[53] Es decir la extensibilidad del programa.
[54] Ver el artículo: *Software Metrics and Measurement Principles.*
[55] Ver la obra: *Software Engineering with Formal Metrics.*

f) Un eficaz mecanismo para la realimentación de calidad.

9.4. MÉTRICAS DEL MODELO DE ANÁLISIS

En esta fase es deseable que las métricas técnicas proporcionen una visión interna a la calidad del modelo de análisis. Estas métricas examinan el modelo de análisis con la intención de predecir el "tamaño" del sistema resultante; es probable que el tamaño y la complejidad del diseño estén directamente relacionadas.

9.4.1. Métricas Basadas en la Función

La métrica del punto de función (PF) se puede utilizar como medio para predecir el tamaño de un sistema obtenido a partir de un modelo de análisis. Para visualizar esta métrica se utiliza un diagrama de flujo de datos, el cual se evalúa para determinar las siguientes medidas clave que son necesarias para el cálculo de la métrica de punto de función:

a) Número de entradas del usuario
b) Número de salidas del usuario
c) Número de consultas del usuario
d) Número de archivos
e) Número de interfaces externas

La cuenta total debe ajustarse utilizando la siguiente ecuación:

$$PF = \text{cuenta-total} \times (0{,}65 + 0{,}01 \times \sum F_i)$$

donde cuenta-total es la suma de todas las entradas PF obtenidas de la figura 9.2 y F_i (i=1 a 14) son los "valores de ajuste de complejidad".

Para el ejemplo descrito en la figura 9.2 se asume que[56] la $\sum F_i$ es 46, por consiguiente:

$$PF = 50 \times (0{,}65 + 0{,}01 \times 46) = 56$$

			Factor de ponderación				
Parámetro de medición	***Cuenta***		***Simple***	***Media***	***Compl.***		
Número de entradas del usuario	*3*	*X*	*3*	*4*	*6*	=	*9*
Número de salidas del usuario	*2*	*X*	*4*	*5*	*7*	=	*8*
Número de consultas del usuario	*2*	*X*	*3*	*4*	*6*	=	*6*
Número de archivos	*1*	*X*	*7*	*10*	*15*	=	*7*
Número de interfaces externas	*4*	*X*	*5*	*7*	*10*	=	*20*
Cuenta total	→						*50*

Fig. 9.2. Cálculo de puntos de función

Basándose en el valor previsto del PF obtenido del modelo de análisis, el equipo del proyecto puede estimar el tamaño global de implementación de las funciones de interacción. Asuma que los datos de los que se dispone indican que un PF supone 60 líneas de código[57] y que en un esfuerzo de un mes-persona se producen 12 PF. Estos datos históricos proporcionan al gestor del proyecto una importante información de planificación

[56] Un producto moderadamente complejo.
[57] Asumiendo también que se utilizará un lenguaje orientado a objetos.

basada en el modelo de análisis en lugar de estimaciones preliminares.

9.5. MÉTRICAS DEL MODELO DE DISEÑO

La gran ironía de las métricas de diseño para el software es que las mismas están disponibles, pero la gran mayoría de los desarrolladores de software continúan sin saber que existen. Las métricas para el diseño del software no son perfectas. Continúa el debate sobre su eficacia y como deberían aplicarse. Muchos expertos argumentan que es necesario una mayor experimentación hasta que se puedan emplear adecuadamente las métricas de diseño. Sin embargo, el diseño sin medición es una alternativa inaceptable.

9.5.1. Métricas de Diseño de Alto Nivel

Estas métricas se concentran en las características de la arquitectura del programa, con énfasis en la estructura arquitectónica y en la eficiencia de los módulos. Estas métricas son de caja negra en el sentido que no requieren ningún conocimiento del trabajo interno de un módulo en particular del sistema.

Card y Glass[58] definen las siguientes tres medidas de complejidad:

a) **Complejidad estructural**, S(i), de un módulo i se define de la siguiente manera: c. Donde $f^2_{out}(i)$ es la expansión del modulo i. La expansión indica el número de módulos inmediatamente subordinados al módulo i, es decir, el número de módulos que son invocados directamente por el módulo i.
b) **Complejidad de datos**, D(i), proporciona una indicación de la complejidad en la interfaz interna de un módulo i y se define como: $D(i)=v(i)/[f_{out}(i) + 1]$. Donde v(i) es el número de variables de entrada y salida que entran y salen del módulo i.
c) **Complejidad del sistema**, C(i), se define como la suma de las complejidades estructural y de datos: $C(i)=S(i)+D(i)$.

A medida que crecen los valores de complejidad, la complejidad arquitectónica o global del sistema también aumenta. Esto lleva a una mayor probabilidad de que aumente el esfuerzo necesario para la integración y las pruebas.

Fenton[59] sugiere varias métricas de morfología simples que permiten comparar diferentes arquitecturas mediante un conjunto de dimensiones directas.

[58] Ver la obra: *Measuring Software Design Quality*.
[59] Ver la obra: *Software metrics. A rigorous & practical approac*h

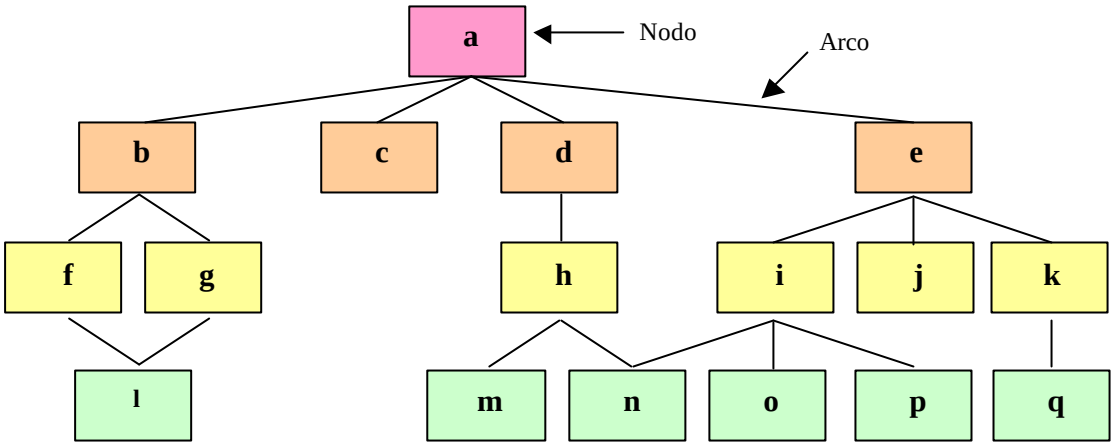

Fig. 9.3. Métricas de morfología

En la Fig. 9.3 se pueden definir las siguientes métricas:

a) tamaño = n + a. Donde n es el número de nodos (módulos) y a es el número de arcos (líneas de control). Para la arquitectura mostrada en la figura 9.3 se tiene tamaño = 17+18 = 35.
b) profundidad = camino más largo desde el nodo raíz a un nodo hoja. Para la arquitectura mostrada en la figura 9.3 la profundidad = 4.
c) amplitud = número máximo de nodos de cualquier nivel de la arquitectura. Para la arquitectura mostrada en la figura 9.3 la amplitud = 6.
d) relación arco a nodo, r=a/n, mide la densidad de conectividad de la arquitectura y proporciona una indicación sencilla de acoplamiento de la arquitectura. Para la arquitectura mostrada en la figura 9.3 r=18/17=1.06.

9.6. MÉTRICAS DEL CODIGO FUENTE

La teoría de la "ciencia del software" propuesta por Halstead[60] es probablemente la medida de complejidad mejor conocida y minuciosamente estudiada. La ciencia del software propuso la primera ley analítica y cuantitativa para el software de computadora. La ciencia del software utiliza un conjunto de medidas primitivas que pueden obtenerse una vez que se ha generado o estimado el código después de completar el diseño. Estas medidas son:

a) n_1: número de operadores diferentes que aparecen en el programa.
b) n_2: número de operandos diferentes que aparecen en el programa.
c) N_1: número total de veces que aparece el operador.
d) N_2: número total de veces que aparece el operando.

Halstead utiliza las medidas primitivas para desarrollar expresiones para la longitud global del programa; volumen mínimo potencial para un algoritmo; el volumen real (número de bits requeridos para especificar un programa); el nivel del programa (una medida de la

[60] Ver la obra: *Elements of Software Science*.

complejidad del software); nivel del lenguaje (una constante para un lenguaje dado); y otras características tales como esfuerzo de desarrollo, tiempo de desarrollo e incluso el número esperado de fallos en el software.

Halstead propone las siguientes métricas:

a) Longitud N se puede estimar como: $N = n_1 \log_2 n_1 + n_2 \log_2 n_2$.
b) Volumen de programa se define como: $V = N \log_2(n_1 + n_2)$. Tomando en cuenta que V variará con el lenguaje de programación y representa el volumen de información (en bits) necesarios para especificar un programa.

Ejemplo:

Programa de ordenación por intercambio [61]

```
      SUBROUTINE SORT(X,N)
      DIMENSION X(N)
      IF (N .LT. 2) RETURN
      DO 20 I=2, N
            DO 10 J=1, I
            IF (X(I) .GE. X(J)) GO TO 10
                  SAVE = X(I)
                  X(I) = X(J)
                  X(J) = SAVE
10    CONTINUE
20    CONTINUE
      RETURN
      END
```

Operadores del programa de ordenación por intercambio

	Operador	**Cuenta**
1	Fin de sentencia	7
2	Subíndices de arreglos	6
3	=	5
4	IF()	2
5	DO	2
6	,	2
7	Fin de programa	1
8	.LT.	1
9	.GE.	1
10	GO TO 10	1
Total	⟶	28

De esta tabla se desprenden los valores de n_1=10 y N_1=28.

Operandos del programa de ordenación por intercambio

	Operando	**Cuenta**
1	X	6
2	I	5

[61] Escrito en el lenguaje de programación Fortran.

3	*J*	*4*
4	*N*	*2*
5	*2*	*2*
6	*SAVE*	*2*
7	*1*	*1*
Total	⟶	*22*

De esta tabla se desprenden los valores de n_2=7 y N_2=22.

9.7. MÉTRICAS PARA PRUEBAS

La mayoría de las métricas para pruebas se concentran en el proceso de prueba, no en las características técnicas de las pruebas mismas. En general, los responsables de las pruebas deben fiarse en las métricas de análisis, diseño y código para que sirvan de guía en el diseño y ejecución de los casos de prueba.

El esfuerzo de las pruebas también se puede estimar utilizando métricas obtenidas de las medidas de Halstead. Usando la definición del volumen de un programa, V, y nivel de programa, NP, el esfuerzo de la ciencia del software puede calcularse como:

$NP = 1/[(n_1/2)x(N_2/n_2)]$ (Ec. 9.1)

$e = V/NP$ (Ec. 9.2)

El porcentaje del esfuerzo global de pruebas a asignar a un módulo k se puede estimar utilizando la siguiente relación:

Porcentaje de esfuerzo de pruebas(k) = e(k)/? e(i) (Ec. 9.3)

Donde e(k) se calcula para el módulo k utilizando las ecuaciones (Ec. 9.1) y (Ec. 9.2), la suma en el denominador de la ecuación (Ec. 9.3) es la suma del esfuerzo de la ciencia del software a lo largo de todos los módulos del sistema.

A medida que se van haciendo las pruebas, tres medidas diferentes proporcionan una indicación de la compleción de las pruebas:

a) **Medida de amplitud de las pruebas**. Proporciona una indicación de cuantos requisitos se han probado del numero total de ellos. Indica la compleción del plan de pruebas.
b) **Profundidad de las pruebas**. Medida del porcentaje de los caminos básicos independientes probados con relación al número total de estos caminos en el programa. Se puede calcular una estimación razonablemente exacta del número de caminos básicos sumando la complejidad ciclomática de todos los módulos del programa.
c) **Perfiles de fallos**. Se emplean para dar prioridad y categorizar los errores. La prioridad indica la severidad del problema. Las categorías de los fallos proporcionan una descripción de un error, de manera que se puedan llevar a cabo análisis estadístico de errores.

9.8. MÉTRICAS DE MANTENIMIENTO

Todas las métricas descritas en este capítulo pueden utilizarse para el desarrollo de nuevo software y para el mantenimiento del existente. El estándar IEEE 982.1-1988 sugiere el índice de madurez del software (IMS) que proporciona una indicación de la estabilidad de un producto software basada en los cambios que ocurren con cada versión del producto. Con el IMS se determina la siguiente información:

M_t = Número de módulos en la versión actual
F_c = Número de módulos en la versión actual que se han cambiado
F_a = Número de módulos en la versión actual que se han añadido
F_e = Número de módulos en la versión actual que se han eliminado

El índice de madurez del software se calcula de la siguiente manera:

$$IMS = [M_t - (F_c + F_a + F_e)]/M_t$$

A medida que el IMS se aproxima a 1 el producto se empieza a estabilizar. El IMS puede emplearse también como métrica para la planificación de las actividades de mantenimiento del software.

Ejercicios # 9

1. La teoría de la medición es un tema avanzado que tiene una gran influencia en las métricas del software. Escribir un pequeño documento que recoja los principales principios de la teoría de la medición.
2. Describir las razones por las cuales no se puede desarrollar una única métrica que lo abarque todo para la complejidad o calidad de un programa.
3. Investigar la métrica bang y las métricas de calidad de especificación del modelo de análisis.
4. Describir las métricas de diseño en los componentes.
5. Investigar las métricas de Henry-Kafura y de Morfología para el modelo de diseño.
6. Los factores de calidad de McCall se desarrollaron durante los años 70. Casi todos los aspectos de la informática han cambiado dramáticamente desde que se desarrollaron, y sin embargo, los factores de McCall todavía se aplican en el software moderno. ¿Puede sacar alguna conclusión basada en este hecho?.
7. Desarrollar una herramienta de software que calcule la complejidad ciclomática de un módulo de un lenguaje de programación de su elección.
8. Realizar una investigación y escribir un pequeño documento sobre la relación que existe entre las métricas de calidad de Halstead y las de McCabe ¿Son convincentes los datos? Recomendar algunas directrices para la aplicación de estas métricas.
9. Investigar documentos recientes en busca de métricas construidas específicamente para el diseño de casos de prueba.
10. Un sistema heredado tiene 940 módulos. La última versión requiere el cambio de 90 de estos módulos. Además se añaden 40 módulos nuevos y se retiraron 12 módulos antiguos. Calcular el índice de madurez del software del sistema.

10 INGENIERÍA DEL SOFTWARE ORIENTADA A OBJETOS

10.1. INTRODUCCIÓN

Actualmente, gran parte de los seres humanos son parte de un mundo de objetos. Los objetos existen en la naturaleza, en las entidades creadas por el ser humano, en los negocios y en los productos de uso diario. Estos objetos pueden ser clasificados, descritos, organizados, combinados, manipulados y creados. En este entendido, no es sorprendente que se proponga una visión orientada a objetos para la creación de software computacional, una abstracción, que según los expertos, modela el mundo de forma tal que ayuda a entenderlo, controlarlo y gobernarlo de mejor manera.

Las tecnologías de objetos contienen un numero de beneficios inherentes que proporcionan ventajas a los niveles de dirección y técnico. Estas tecnologías conducen a la reutilización de componentes de software, las cuales llevan a un desarrollo de software más rápido y a la obtención de programas de mejor calidad. Por otra parte el software orientado a objetos es más fácil de mantener debido a que su estructura es inherentemente descompuesta. Esto lleva a efectos colaterales menores cuando se deben hacer cambios y provoca menos frustración en el ingeniero del software y en el cliente. Adicionalmente a todo lo mencionado, los sistemas orientados a objetos son más fáciles de adaptar y escalar.

10.2. PARADIGMA ORIENTADO A OBJETOS

Un modelo evolutivo de proceso acoplado con un enfoque que fomenta el ensamblaje[62] de componentes es el mejor paradigma para la ingeniería del software orientada a objetos.

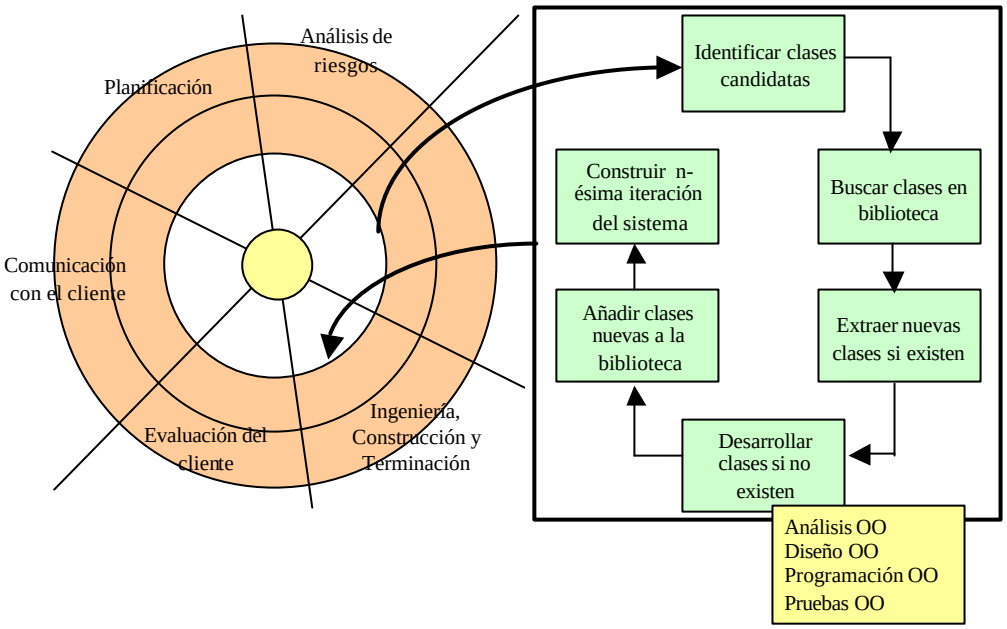

Fig. 10.1. Modelo de proceso orientado a objetos

10.3. CONCEPTOS ORIENTADOS A OBJETOS

Cualquier discusión sobre ingeniería del software orientada a objetos debe comenzar por el término "orientado a objetos", durante años han existido muchas opiniones diferentes acerca de los conceptos centrales, sin embargo las definiciones principales son las siguientes:

10.3.1. Clases y Objetos

Una clase es un concepto orientado a objetos que encapsula las abstracciones de datos y procedimientos que se requieren para describir el contenido y comportamiento de alguna entidad del mundo real. Las abstracciones de datos o atributos que describen la clase, están cerradas por una "muralla" de abstracciones procedimentales[63] capaces de manipular los datos de alguna manera. La única forma de alcanzar los atributos, y operar sobre ellos, es ir a través de alguno de los métodos que forman la muralla. Por lo tanto, la clase encapsula datos, dentro de la muralla, y el proceso que manipula los datos[64]. Esto posibilita la

[62] Entiéndase como reutilización en este contexto.
[63] Llamadas también operaciones, métodos o servicios.
[64] Es decir los métodos que componen la muralla.

ocultación de la información y reduce el impacto de los efectos colaterales asociados a cambios. Como estos métodos tienden a manipular un número limitado de atributos, esto representa una alta cohesión, y como la comunicación ocurre solamente a través de los métodos que encierra la "muralla", la clase tiene a un acoplamiento con otros elementos del sistema. Todas estas características del diseño conducen a la construcción de software de alta calidad.

En otras palabras se puede decir que una clase es una descripción generalizada que describe una colección de objetos similares. Por definición, todos los objetos que existen dentro de una clase heredan sus atributos y las operaciones disponibles para el manejo de los atributos. Una superclase es una colección de clases y una subclase es una instancia de una clase.

Las definiciones expuestas implican la existencia de una jerarquía de clases en la cual los atributos y operaciones de las superclases son heredados por subclases que pueden añadir, cada una de ellas, atributos privados y métodos.

10.3.2. Atributos

Según Champeaux[65] y sus colegas un estudio de los atributos puede ser descrito como: "Las entidades de la vida real están a menudo descritas con palabras que indican características estables. La mayoría de los objetos físicos tienen características tales como forma, peso, color y tipo de material. Las personas tienen características incluyendo fecha de nacimiento, padres, nombres y color de los ojos. Una característica puede verse como una relación binaria entre una clase y cierto dominio". La relación binaria implica que un atributo puede tomar un valor definido por un dominio enumerado. En la mayoría de los casos, un dominio es simplemente un conjunto de valores.

10.3.3. Operaciones, Métodos y Servicios

Un objeto encapsula datos[66] y los algoritmos que procesan estos datos. Estos algoritmos son denominados operaciones, métodos o servicios y pueden ser vistos como módulos en un sentido convencional.

Cada una de las operaciones encapsuladas por un objeto proporciona una representación de uno de los comportamientos del objeto. Cada vez que un objeto recibe un estimulo, éste inicia un determinado comportamiento.

10.3.4. Mensajes

Los mensajes son el medio a través del cual interactúan los objetos. Utilizando la terminología descrita, un mensaje estimula la ocurrencia de cierto comportamiento en el objeto receptor. El comportamiento de realiza cuando se ejecuta una operación.

---

[65] Ver la obra: *Object Oriented System Development.*
[66] O una colección de atributos.

Cox[67] describe el intercambio entre objetos de la siguiente manera: "Se solicita de un objeto que ejecute una de sus operaciones enviándole un mensaje que le informa acerca de lo que debe hacer. El objeto receptor responde al mensaje eligiendo primero la operación que implementa el nombre del mensaje, ejecutando dicha operación y después devolviendo el control al objeto que origina la llamada.

El paso de mensajes mantiene comunicados un sistema orientado a objetos. Los mensajes proporcionan una visión interna del comportamiento de los objetos individuales y del sistema orientado a objetos como un todo.

10.3.5. Encapsulamiento, Herencia y Polimorfismo

Las clases y los objetos derivados de ella encapsulan los datos y las operaciones que trabajan sobre esto en un único paquete.

La herencia es una de las diferencias clave entre sistemas convencionales y sistemas orientados a objetos. Una subclase Y hereda todos los atributos y operaciones asociadas con su superclase X. Esto significa que todas las estructuras de datos y algoritmos originalmente diseñados e implementados para X están inmediatamente disponibles para Y. La reutilización se realiza directamente. Cualquier cambio en los datos u operaciones contenidas dentro de una superclase se hereda inmediatamente por todas las subclases que se derivan de la superclase. Debido a esto, la jerarquía de clases se convierte en un mecanismo a través del cual los cambios pueden propagarse inmediatamente a través de todo el sistema.

El polimorfismo indica, literalmente, la posibilidad de que una entidad tome muchas formas. El polimorfismo es una característica que reduce en gran medida el esfuerzo necesario para extender un sistema orientado a objetos.

10.4. GESTION DE PROYECTOS DE SOFTWARE OO

La moderna gestión de proyectos de software puede subdividirse en las siguientes actividades:

a) Establecimiento de un marco de proceso común para el proyecto.
b) Uso del marco y de métricas históricas para desarrollar estimaciones de esfuerzo y tiempo.
c) Especificación de productos de trabajo y avances que permitirán la medición del progreso.
d) Definición de puntos de comprobación para asegurar la calidad y el control.
e) Gestión de los cambios que ocurren invariablemente al progresar el proyecto.
f) Seguimiento, monitorización y control del progreso.

El director técnico que se enfrente con un proyecto de software orientado a objetos aplica estas seis actividades. Debido a la naturaleza univoca del software orientado a objetos, cada

[67] Ver la obra: *Object Oriented Programming.*

una de estas actividades de gestión tiene un matiz levemente diferente y debe ser enfocado utilizando un modelo propio.

10.5. ANÁLISIS ORIENTADO A OBJETOS

El análisis orientado a objetos (AOO) se fundamente en un conjunto de cinco principios básicos:

1) Modelar el dominio de la información.
2) Describir la función del modulo.
3) Representar el comportamiento del modelo.
4) Dividir el modelo para mostrar mas detalles.
5) Observar que los modelos iniciales representan la esencia del problema mientras que los últimos aportan detalles de la implementación.

El propósito del AOO es definir todas las clases, además de las relaciones y comportamientos asociados a ellas, que son relevantes al problema a ser resuelto. Para cumplirlo se deben ejecutar las siguientes tareas:

1) Los requisitos básicos del usuario deben comunicarse entre el cliente y el ingeniero del software.
2) Identificar las clases, definiendo los atributos y los métodos.
3) Especificar una jerarquía de clases.
4) Representar las relaciones, conexiones, objeto a objeto.
5) Modelar el comportamiento del objeto.
6) Repetir iterativamente las tareas 1 a la 5 hasta completar el modelo.

Los conceptos asociados con el AOO son muy naturales. Coad y Yourdon[68] escriben a este respecto lo siguiente: *"El análisis orientado a objetos se basa en conceptos que aprendimos en la guardería: objetos y atributos, clases y miembros, un todo y partes. ¿Por qué nos ha llevado tanto tiempo aplicar estos conceptos al análisis y especificación de sistemas de información? se preguntan todos; quizá hemos estado muy ocupados siguiendo el flujo del tradicional análisis estructurado para considerar otras alternativas.*"

10.6. DISEÑO ORIENTADO A OBJETOS

El diseño orientado a objetos (DOO) transforma el modelo de análisis en un modelo de diseño que sirve como un anteproyecto para la construcción de software. A diferencia de los métodos convencionales de diseño del software, el DOO constituye un tipo de diseño que logra un cierto número de diferentes niveles de modularidad. Los componentes principales del sistema están organizados en "módulos" denominados subsistemas. Los datos y las operaciones que manipulan los datos están encapsulados en objetos, una forma modular que es el bloque de construcción de un sistema orientado a objetos.

[68] Ver la obra: *Object Oriented Analysis*.

El DOO debe describir la organización de datos específicos, de atributos y los detalles procedimentales de las operaciones individuales. Esta representación fragmentada de datos y algoritmos de un sistema orientado a objetos colaboran a una modularidad general.

La naturaleza única del DOO descansa en su capacidad para apoyarse en cuatro conceptos importantes del diseño clásico del software: abstracción, ocultación de la información, independencia funcional y modularidad. Todos los métodos de diseño del software se afanan en mostrar estas importantes características, pero solamente el DOO aporta un mecanismo que le permite al diseñador alcanzar estas cuatro propiedades con menor complejidad y compromiso.

Para los sistemas orientados a objetos es posible definir un diseño en pirámide con las siguientes cuatro capas:

1) **Subsistema**. Contiene una representación de cada uno de los subsistemas que le permiten al software conseguir los requisitos definidos por el cliente e implementar la infraestructura técnica que los soporta.
2) **Clases y objetos**. Contiene las jerarquías de clases que permiten crear el sistema utilizando generalizaciones y especializaciones mejor definidas incrementalmente. También contiene representaciones de diseño para cada objeto.
3) **Mensajes**. Contiene los detalles que permiten a cada objeto comunicarse con sus colaboradores. Establece las interfaces externas e internas para el sistema.
4) **Responsabilidades**. Contiene las estructuras de datos y el diseño algorítmico para todos los atributos y operaciones de cada objeto.

10.7. PRUEBAS ORIENTADAS A OBJETOS

El objetivo de la realización de pruebas es: encontrar el mayor número posible de errores con una cantidad de esfuerzo racional a lo largo de un espacio de tiempo realista. Aunque este objetivo fundamental permanece el mismo para el software orientado a objetos, la naturaleza de los programas orientados a objetos cambia la estrategia y las tácticas de prueba.

Puede argumentarse que, con la madurez del análisis y diseño orientado a objetos, el mayor grado de reusabilidad de patrones de diseño reducirá la necesidad de pruebas pesadas en los sistemas orientados a objetos. Binder[69] examina esto cuando comenta: “Cada reutilización es un nuevo contexto de uso y es prudente repetir las pruebas. Parece en consecuencia, que no se requerirán menos pruebas para obtener una alta fiabilidad en sistemas orientados a objetos”.

Para probar adecuadamente los sistemas orientados a objetos deben hacerse tres cosas:

1) Ampliar la definición de las pruebas para incluir técnicas de detección de errores aplicados a los modelos de diseño y análisis orientados a objetos.

[69] Ver el artículo: *Object-Oriented Software Testing.*

2) Cambiar de manera significativa las estrategias para las pruebas de unidad e integración.
3) Tomar en cuenta las características propias del software orientado a objetos para el diseño orientado a objetos.

10.8. MÉTRICAS TÉCNICAS PARA SISTEMAS OO

La utilización de métricas para sistemas orientados a objetos ha progresado con mucha mas lentitud que la utilización de los demás métodos orientados a objetos.

Los objetivos principales de las métricas orientadas a objetos son los mismos que los existentes para las métricas derivadas para el software convencional:

a) Comprender mejor la calidad del producto.
b) Estimar la efectividad del proceso.
c) Mejorar la calidad del trabajo realizado en el nivel del proyecto.

Cada uno de estos objetivos es importante, sin embargo para el ingeniero del software la calidad del producto debe ser lo primordial.

Ejercicios # 10

1. Haciendo uso de sus propias palabras y unos pocos ejemplos, definir los términos "clase", "encapsulamiento", "herencia" y "polimorfismo".
2. Investigar un lenguaje orientado a objetos y muestre como se implementa el paso de mensajes en la sintaxis de dicho lenguaje.
3. Investigar y comentar acerca de la importancia de la herencia para la ingeniería del software orientada a objetos.
4. Proporcionar un ejemplo concreto de herencia múltiple. Con el ejemplo muestre argumentos a favor y en contra de la herencia múltiple.
5. Describir en sus propias palabras, por qué el modelo de proceso recursivo/paralelo es apropiado para sistemas orientados a objetos.
6. Investigar acerca de la forma en la que se identifican los objetos cuando se estudian las soluciones potenciales a un problema.
7. Considere una interfaz gráfica de usuario (GUI) típica. Definir un conjunto de clases para las entidades de la interface que típicamente aparecen en la GUI. Asegure definir atributos y operaciones apropiadas.
8. Investigar las características de un objeto compuesto.
9. Se le ha asignado la tarea de producir un software nuevo para el procesamiento de textos. En este contexto se logra identificar una clase principal denominada documento. Definir un conjunto de atributos y operaciones relevantes a la clase documento.
10. Investigar el tema de herencia múltiple, proporcionando argumentos a favor y en contra de esta temática.

Bibliografía

Balzer, R. & N. Goodman. Principles of Good Specification and their Implications for Specification Languages. *Software Specification Techniques*. Gehani & McGettrick, eds. Addison-Wesley, 1986, pp. 25-39.

Beizer, B. *Software Testing Techniques*, second edition. Van Nostrand Reinhold, 1990.

Binder, R.V. Object-Oriented Software Testing. *Communications of the ACM*, volume 37, No 9, pp. 29, September 1994.

Boehm, B. *Software Engineering Economics*. Prentice–Hall, 1981.

Boehm, B.W., J.R. Brown, & M. Lipov. Quantitative Evaluation of Software Quality. pp. 592-605. *Proc. 2nd Intl. Conf., on Software Engineering*. Long Beach, Calif. IEEE Computer Society, Oct. 1976.

Card, D.N. & R.L. Glass. *Measuring Software Design Quality*. Prentice-Hall, 1990.

Champeaux, D., D. Lea & P. Faure. *Object Oriented System Development.* Addison-Wesley, 1993.

Chidamber, S.R. and C.F. Kemerer. A metrics suite for object oriented design, *IEEE Trans. Software Eng*., vol. 20, pp. 476-493, 1994.

Coad, P. & E. Yourdon. *Object Oriented Analysis*, second edition. Prentice-Hall, 1991.

Cox, B.J. *Object Oriented Programming*. Addison-Wesley, 1986.

Davis, A. *201 Principles of Software Development.* McGraw-Hill, 1995.

De Marco, T. *Controlling Software Projects*. Prentice-Hall, 1982.

De Marco, T. *Structured Analysis and System Specification.* Prentice-Hall, 1979.

Dogru, A., Jololian, L., Kurfess, F., & M. Tanik. *Component-Based Technology for the Engineering of Virtual Enterprises and Software*, Computer Engineering Department, METU, TR-98-7, Turkey, 1998.

Drommey, R. A Model for Software Product Quality», *IEEE Transactions on Software Engineering*, Vol. 21, No. 2, February 1995, pp. 146-162.

Ejiogu, L. *Software Engineering with Formal Metrics*. QED Publishing, 1991.

Fenton, N.E. & S.L. Pfleeger. *Software metrics. A rigorous & practical approach*. International Thompson Computer Press. 1997.

Frakes W. & C. Terry. Software Reuse: Metrics and Models, *ACM Computing Surveys*, Vol. 28, No. 2, June 1996, pp. 415-435.

Freeman, P. Requirements Analysis and Specification, *Proc. Intl. Computer Technology Conf., ASME*, San Francisco, August, 1980.

Gause, D.C. & G.M. Weinberg. *Exploring Requirements: Quality Before Design*. Dorset House, 1989.

Gilb, T. What we fail to do in our Current Testing Culture. *Testing Techniques Newsletter.* Software Research Inc. San Francisco, January 1995.

Grady, R.B. & D.L. Caswell. *Software Metrics: Establishing a Company-Wide Program*. Prentice-Hall, 1987.

Halstead, M. *Elements of Software Science*. North Holland, 1977.

Jones, T.C. *Programming Productivity: Issues for the 80´s*. IEEE Computer Society Press, 1981.

Kan, S.H. *Metrics and models in Software Quality Engineering*, Addison-Wesley, 1995.

Kobryn, C. UML 2001: A Standardization Odyssey, *Communications of ACM*, Volume 42, No. 10, pp. 29-37, 1999.

Krieger D. & Adler R. The Emergence of Distributed Component Platforms. *IEEE Computer*, Volume 31 No. 3, pp. 43-53, 1998.
McCabe, T. A Software Complexity Measure. *IEEE Trans on Software Engineering*, volume 2, December 1976, pp. 308-320.
McCall, J., P. Richards & G. Walters. *Factors in Software Quality*, three volumes. NTIS AD-A049-014, 015, 055. November 1977.
McGlaughlin, R. Some Notes on Program Design. *Software Engineering Notes*, ACM, volume 16, No 4, October 1991, pp. 53-54.
McIlroy, M.D. Mass-produced Software Components. In *Software Engineering Concepts and Techniques*, 1968, NATO Conf. Software Eng.
Myers, G. *The Art of Software Testing*. Wiley, 1979.
Naur, P. & B. Randall (eds) *Software Engineering: A Report on a Conference Sponsored by the NATO Science Committee*, NATO, 1969.
Oman P. & S. Pfleeger. *Applying Software Metrics*, IEEE Press. 1996.
Pressman, R. *Software Engineering: A Practitioners Approach, 5th edition*. McGraw-Hill. 1997.
Pressman, R. *Ingeniería del Software: Un enfoque práctico, segunda edición*. Editorial McGraw Hill, 1990.
Roche, J.M. Software Metrics and Measurement Principles. *Software Engineering Notes*, ACM, volume 19, No 1, January 1994, pp. 76-85.
Rodríguez, M.L. Garvi E. & Granja J.C., Calidad y reusabilidad del software: estudio de la funcionalidad, *Novatic*a, No. 125, Enero-Febrero de 1997, pp. 67-70.
Wasserman, A. Principles of Systematic Data Design and Implementation. *Software Design Techniques*. P. Freeman & A. Wasserman eds. Third edition. IEEE Computer Society Press, 1980, pp. 287-293.

Referencias Electrónicas

Complete list of "software engineering" sites.
http://www.search.com/search?q=%22software+engineering%22&channel=1&ref=wf

Comprehensive technology info on "software engineering tutorial".
http://cnet.search.com/search?q=%22software+engineering%22&ref=wf

Computer Science
http://www.looksmart.com/eus1/eus53832/eus155852/r?comefrom=webferret-eus155852

E-Base Interactive offers programming and software design/development services. Clientele ranges from mid-sized to Fortune 500 companies.
http://www.ebaseinteractive.com/

Involved primarily in research and development, this software engineering tutorial group assists Canadian software companies with improved techniques.
http://wwwsel.iit.nrc.ca/

Midwest Software Engineering Tutorial, Inc.
http://www.mselink.com/

Powerful C++ & Java Software Engineering Tutorial Tools for Object Behavior Management
http://www.mastersys.com/

European Organization for Nuclear Research defines terms associated with software engineering tutorial. Links are also provided.
http://dxsting.cern.ch/sting/glossary.html

About the SEI. Welcome. Our Purpose. How to Reach Us. Our Organization. Director's Office. Customer Sectors. Complete Technical Project List
http://www.sei.cmu.edu/director/aboutSEI.html

WebSoft. Building a Global Software Engineering Tutorial Environment. The WebSoft project is a concerted effort to understand the software engineering tutorial
http://www.ics.uci.edu/pub/websoft/

GLOSARIO

Abstracción (Abstraction)
(1) nivel de detalle técnico de alguna representación del software, (2) modelo cohesivo de datos o un procedimiento algorítmico.

Alcance del proyecto (Project scope)
Declaración de los requerimientos básicos del software a ser construido.

Análisis (Analysis)
Conjunto de actividades que intentan entender y modelar las necesidades y restricciones de los clientes.

Análisis de requerimientos (Requirements analysis)
Actividad de modelado cuyo objetivo es entender lo que el cliente realmente necesita.

Análisis de riesgos (Risk analysis)
Técnicas para identificar y evaluar riesgos.

Análisis orientado a objetos (Object-oriented analysis (OOA))
Técnica para clases definidas de objetos, sus relaciones y estructura básica.

Aseguramiento de la calidad del software (Software quality assurance (SQA))
Serie de actividades que ayudan a una organización a producir software de alta calidad.

Auditoria de configuración (Configuration audit)
Actividad ejecutada por un grupo SQA con el intento de asegurar que el proceso de control de cambios se encuentre trabajando.

Calidad (Quality)
Grado en el cual el producto conforma tanto los requerimientos explícitos como los requerimientos implícitos.

Calidad del software (Software quality)
Ver calidad

Casos de prueba (Test cases)
Creación de datos que pueden ser utilizados para descubrir errores en el software.

Ciclo de vida clásico (Classic life cycle)
Enfoque secuencial y lineal para el modelado de procesos.

Clases (Classes)
Una construcción básica en los métodos orientados a objeto que categoriza los elementos del problema.

Cliente (Customer)
Persona o grupo que hace el requerimiento del software y que se hará cargo del pago por su desarrollo.
Codificación (Coding)
Generación de código fuente.

Complejidad (Complexity)
Medida cuantitativa de la complejidad de un programa.

Complejidad ciclomática (Cyclomatic complexity)
Medida de la complejidad lógica de un algoritmo utilizado en la prueba de caja blanca.

Componentes reusables (Reusable components)
Ítems de configuración que son reutilizables.

Configuración (Configuration)
Colección de programas, documentos y datos que deben ser controlados cuando los cambios son realizados.

Control de configuración (Configuration control)
Control de cambios a los programas, documentos o datos.

Control del proyecto (Project control)
Control de la calidad y los cambios.

Depuración (Debugging)
Actividad asociada con la búsqueda y corrección de un error o un defecto.

Descomposición funcional (Functional decomposition)
Técnica utilizada durante la planificación, análisis y diseño; se encarga de crear una jerarquía funcional para el software.

Diagrama de flujo de datos (Data flow diagram (DFD))
Notación de modelado que representa la descomposición funcional de un sistema.

Diagrama de transición de estados (State transition diagram (STD))
Notación para el modelado del comportamiento.

Diccionario de datos (Data dictionary)
Base de datos que contiene definiciones de todos los ítems de datos definidos durante el análisis.

Diseño (Design)
Actividad que traslada el modelo de requerimientos en un modelo mas detallado que constituye la guía para la implementación del software.

Diseño arquitectónico (Architectural design)

Actividad que intenta trazar el modulo “de cimientos” para la construcción del software.

Diseño de casos de prueba (Test case design)
Conjunto de técnicas para derivar casos de prueba efectivos.

Diseño de datos (Data design)
Actividad que traslada el modelo de datos desarrollado durante el análisis en estructuras de datos implementables.

Diseño detallado (Detail design)
Actividad de diseño que se enfoca en la creación de un algoritmo.

Diseño modular (Modular design)
Enfoque de diseño que hace énfasis en la modularidad.

Diseño orientado a objetos (Object-oriented design (OOD))
Técnica para trasladar el modelo OOA en un modelo de implementación.

Diseño preliminar (Preliminary design)
Creación de la representación de datos y arquitectura.

Diseño procedimental (Procedural design) Crea representaciones de detalles algorítmicos al interior del modulo.

Documentación (Documentation)
Información descriptiva.

Documentos (Documents)
Entregas produc idas como parte del proceso de ingeniería del software.

Efectos laterales (Side effects)
Errores que ocurren como consecuencia de los cambios.

Especificación de diseño (Design specification)
Documento que describe el diseño.

Especificación de requerimientos del software (Software Requirements Specification)
Entrega que describe todos los datos, los requerimientos funcionales y de comportamiento, todas las restricciones y los requerimientos de validación para el software.

Estimación (Estimation)
Actividad de planificación de un proyecto que intenta proyectar esfuerzos y costos para un proyecto de software.

Generación automática de código (Automatic code generation)
Herramientas que generan código fuente a partir de una representación de software que no es código fuente.

Grafico de causa efecto (Cause-effect graphing)
Método de prueba de caja negra.

Grupos (Clusters)
Colección de componentes de programas (módulos) que son probados como un grupo.

Grupos de prueba independientes (Independent test group (ITG))
Grupo de personas cuya responsabilidad primaria es la prueba del software.

Herramientas (Tools)
Software de aplicación utilizado para ejecutar tareas de la ingeniería del software (herramientas de diseño, herramientas de prueba, etc.)

Ingeniería del software (Software engineering)
Disciplina que abarca el proceso asociado con el desarrollo del software, los métodos utilizados para analizar, diseñar y probar software de computadoras, las técnicas de administración asociadas con el control y monitoreo de los proyectos de software y las herramientas utilizadas para soportar procesos, métodos y técnicas.

Ingeniería del software asistida por computadoras (CASE - Computer-aided software engineering)
Ver Herramientas.

Interoperabilidad (Interoperability)
Grado con el cual una aplicación se comunica con otra.

Lenguaje de diseño de programas (PDL - program design language)
Combinación de lenguaje natural con lenguaje de programación en la fase de construcción del programa.

Lenguaje de diseño de programas (Program design language, PDL)
Lenguaje para la creación inicial de algoritmos.

Líneas de código (LOC - lines of code)
Numero de líneas de código de un programa.

Mantenibilidad (Maintainability)
Grado en el cual un programa es tratable al cambio.

Mantenimiento (Maintenance)
Actividades asociadas con los cambios realizados al software después de que este ha sido liberado a los usuarios finales.

Mantenimiento adaptativo (Adaptive maintenance)
Actividad asociada con los cambios realizados a una aplicación para hacer que las mismas sean acordes a los cambios de su medio externo.

Mantenimiento correctivo (Corrective maintenance)
Búsqueda y ajuste de los defectos reportados por los usuarios.

Mantenimiento del software (Software maintenance)
Ver mantenimiento.

Mantenimiento perfectivo (Perfective maintenance)
Mejoras.

Medida (Measurement)
Colección de datos cuantitativos acerca del software o el proceso de ingeniería del software.

Métodos de análisis (Analysis methods)
Notación y heurísticas para la creación de modelos de las necesidades y limitaciones de los clientes.

Métrica (Metrics)
Medida específica.

Métricas de calidad (Quality metrics)
Medidas de calidad.

Métricas de línea de código (Line-of-code metrics)
Medidas de calidad o productividad que son normalizadas utilizando líneas de código producidas.

Métricas del software (Software metrics)
Medidas cuantitativas del proceso o del producto.

Modelado de datos (Data modeling)
Método de análisis que modela objetos de datos y sus relaciones.

Modelo de comportamiento (Behavioral modeling)
Representa el modelo de comportamiento (llamado estados) de una aplicación y los eventos que causan transiciones de estado a estado.

Modelo espiral (Spiral model)
Paradigma evolucionario de la ingeniería del software.

Modularidad (Modularity) Atributo de un diseño que conduce a la creación de componentes de programas de alta calidad.

Niveles de abstracción (Levels of abstraction)
Grado de detalle con el cual alguna representación del software es presentada.

Objetos (Objects)
Elemento nombrado del dominio del problema que contiene datos y procedimientos para procesar los datos.

Objetos de datos (Data objects)
Entrada o salida que es visible por el usuario.

Orientado a objetos (Object-oriented)
Enfoque para el desarrollo del software que hace uso de un enfoque de clasificación y empaqueta datos y procedimientos de manera conjunta.

Paradigmas (Paradigms)
Modelo de procesos.

Plan del proyecto (Project Plan)
Descripción del enfoque de administración de un proyecto.

Planificación del proyecto (Project planning)
Actividad que crea el plan del proyecto.

Portabilidad (Portability)
Habilidad para transportar software de un medio ambiente objetivo a otro.

Programación estructurada (Structured programming)
Método de diseño que limita la construcción del diseño a solamente tres formas básicas y a un flujo de programa restringido para una mejor calidad.

Prototipación (Prototyping)
Creación de la maqueta de una aplicación.

Prueba beta (Beta testing)
Prueba conducida por los usuarios.

Prueba de caja blanca (White box testing)
Técnica de diseño de casos de prueba que utiliza el conocimiento de la lógica interna del programa.

Prueba de caja negra (Black box testing)
Prueba que no se enfoca en los detalles internos del programa sino utiliza los requerimientos externos.

Prueba de camino básico (Basis path testing)
Técnica de diseño de casos de caja blanca que utiliza el flujo algorítmico del programa para diseñar las pruebas.

Prueba de ciclos (Loop testing)
Técnica de prueba de caja blanca que ejercita los ciclos del programa.

Prueba de integración (Integration testing)
Paso de prueba que construye el software mientras se prueba el mismo.

Prueba de regresión (Regression testing)
Pruebas que son conducidas de manera repetida para asegurar que un cambio no introduzca efectos colaterales.

Prueba de unidad (Unit testing)
Parte de la estrategia de prueba que se concentra en pruebas de componentes individuales de los programas.

Pruebas (Testing)
Conjunto de actividades que intentan encontrar errores en el software.

Pruebas del software (Software testing)
Conjunto de actividades realizadas con la intención de encontrar errores en el software.

Pruebas selectivas (Selective testing)
Pruebas de un conjunto selecto de programas y entrada de datos.

Puntos de función (Function points)
Medida para la utilidad producida por una aplicación.

Reingeniería (Re-engineering)
Serie de actividades que transforman sistemas legado (con mantenimiento pobre) en software que exhibe alta calidad.

Reporte de problemas del software (Software problem report)
Reporte de los defectos asociados al software.

Reporte del estado de configuración (Configuration status reporting (CSR))
Actividad que ayuda al desarrollador del software a entender que cambios han sido realizados y por qué.

Requerimiento de cambio (Change request)
Proporciona detalles del tipo de cambio que es requerido.

Restricciones (Constraints)
Restricciones o limitaciones colocadas en los requerimientos o en el diseño.

Reusabilidad (Reusability)
Habilidad para volver a utilizar un componente de programa existente en otra aplicación.

Reuso de componentes (Component reuse)
Habilidad para reutilizar una parte de un modelo, de un código fuente, casos de uso, etc.

Revisiones técnicas formales (Formal technical reviews)
Reunión estructurada conducida por el ingeniero del software con el intento de descubrir errores en algunas entregas o productos de trabajo.

Riesgo (Risk)
Ocurrencia o problema potencial que coloca a un proyecto en riesgo de no ser llevado a cabo.

Riesgo de negocios (Business risks)
Conjunto de problemas potenciales de negocio u ocurrencias que pueden causar la falla del proyecto.

Riesgos del proyecto (Project risks)
Conjunto de problemas potenciales u ocurrencias no previstas que pueden causar la falla del proyecto.

Software
Programas, documentos y datos.

Tamaño del proyecto (Project size)
Indicación del esfuerzo completo a ser consumido o el número de personas trabajando en el proyecto.

Usuario (User)
Persona que actualmente utiliza el software o el producto que el software tiene empotrado en el mismo.

Validación (Validation)
Pruebas para asegurar que el software se encuentra conforme de acuerdo a sus requerimientos.