INDICE

Unidad Académica I

TEMAS GENERALES DE INGENIERIA DEL SOFTWARE

La ingeniería de Software fue definida por Bauer a finales de los 60s como el establecimiento y uso de principios de ingeniería para obtener software que fuera confiable y que funcionara eficientemente con las máquinas reales. A pesar de ser vieja, esta definición da el sentimiento correcto detrás de la disciplina.

En la presente unidad entenderemos lo que abarca esta disciplina y como ha evolucionado a través del tiempo.

INDICADORES DE LOGRO

Al finalizar el estudio de la presente unidad temática el estudiante:

1. Conoce los principios básicos de la ingeniería del software
2. Define conceptualmente a la ingeniería del software
3. Conoce la evolución de la ingeniería del software
4. Entiende el proceso básico que involucra la ingeniería del software

1. ¿Que es Ingeniería de Software?

La Ingeniería del Software, término utilizado por primera vez por Fritz Bauer en la primera conferencia sobre desarrollo de software patrocinada por el Comité de Ciencia de la OTAN celebrada en Garmisch, Alemania, en Octubre de 1968, puede definirse según Alan Davis[1] como "la aplicación inteligente de principios probados, técnicas, lenguajes y herramientas para la creación y mantenimiento, dentro de un coste razonable, de software que satisfaga las necesidades de los usuarios"...

La ingeniería de software, como las disciplinas tradicionales de ingeniería, tiene que ver con el costo y la confiabilidad. Algunas aplicaciones de software contienen millones de líneas de código que se espera que se desempeñen bien en condiciones siempre cambiantes.

Desde ese punto de vista podemos definir a la Ingeniería de software como la rama de la ingeniería que crea y mantiene las aplicaciones de software aplicando tecnologías y prácticas de las ciencias computacionales, manejo de proyectos, ingeniería, el ámbito de la aplicación, y otros campos.

2. Historia de la ingeniería del software.

2.1 Crisis del Software.

Muchos observadores de la industria han caracterizado los problemas asociados con el desarrollo del software como una crisis, sin embargo, lo que realmente tenemos puede ser algo bastante diferente. La palabra **crisis** se define en el diccionario de Webster como un punto decisivo en el curso de algo; momento, etapa o evento decisivo o crucial. Sin embargo, para el software no ha habido ningún punto decisivo ningún momento decisivo, solamente un lento cambio evolutivo. En la industria del software hemos tenido una crisis que ha estado con nosotros cerca de 30 años y eso es una contradicción para el término.

[1] **Alan Davis,** Presidente de Ommi-Vista Corporation, profesor de la Universidad de Colorado en Colorado Springs -USA- y Editor Emérito de IEEE Software., reconocido como uno de los primeros expertos en la gestión de requerimientos de software y planeación del producto.

Quizás el término mas apropiado es mal del software, o aflicción crónica que como se define en el diccionario Webster es algo que causa pena o desastre. Pero la clave de este argumento es la definición de la palabra crónica que puede entenderse como muy duradero o que vuelve a aparecer con frecuencia continuando indefinidamente. Es bastante más preciso describir lo que hemos estado aguantando las tres últimas décadas como una aflicción crónica que como una crisis.
Tanto si lo llamamos crisis del software como mal del software, el término alude a una serie de problemas que aparecen en el desarrollo del software de computadoras. Los problemas no se limitan al software que no funciona correctamente, es más, el mal abarca los problemas asociados a cómo desarrollar software, como mantener el volumen cada vez mayor de software existente y como poder esperar mantenernos al corriente de la demanda creciente de software. Aunque las referencias a crisis y aflicción pueden sonar melodramáticas, las frases resultan útiles por referirse a verdaderos problemas que se encuentran en todas las áreas del desarrollo del software.
En los gráficos siguientes ilustraremos una de las principales causas por la que el software no funciona correctamente:

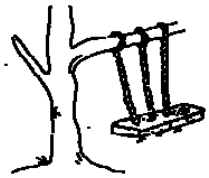
En el proceso de la Ingeniería de Software:

- Requerimientos del software...
  - Lo que el usuario realmente desea...[2]

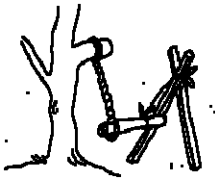
  - Lo que el usuario realmente reflejó...

  - Cómo fue la percepción del analista...

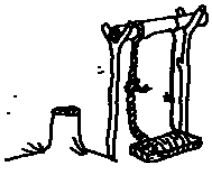
- Diseño del software
  - Cómo fue diseñado el sistema...



- Construcción del software
  - Cómo el programador lo construyó...



- Despliegue / Instalación del software
  - Cómo funcionó realmente el sistema...



[2] Gráficos por: Craig Murphy (craig.murphy@currieb.co.uk) Enterprise Developer Currie & Brown

2.2 Metas

El software se ha convertido en el elemento clave de la evolución de los sistemas y productos basados en computadoras, así como en una de las tecnologías más importantes en el ámbito mundial. En los pasados 50 años, el software ha evolucionado desde ser una herramienta para la solución de problemas especializados y el análisis de información, hasta convertirse en una industria por sí mismo. Todavía se tienen problemas al desarrollar software de alta calidad a tiempo y dentro del presupuesto. El software (programas, datos y documentos) se dirige a un amplio espectro de tecnologías y áreas de aplicación. En la actualidad el software evoluciona de acuerdo con un conjunto de leyes que han permanecido inalteradas a lo largo de 30 años. La intensión de la ingeniería de software es proporcionar un marco general para construir software con una calidad mucho mayor.

El propósito de la ingeniería de software es generar y mantener sistemas de software dentro de las restricciones de tiempo, funcionalidad y costos acordados con el cliente. Las metas de esta disciplina tecnológica son mejorar la calidad de los productos desarrollados y aumentar la productividad de los ingenieros de software. El grado de formalidad y el tiempo asignado al proyecto de software varía de acuerdo al tamaño y complejidad del producto que será desarrollado.

2.3 Mitos

Muchas de las causas de la crisis del software se pueden encontrar en una mitología que surge durante los primeros años del desarrollo de 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 (algunas veces conteniendo elementos verdaderos); tuvieron un sentido intuitivo y frecuentemente fueron promulgados por expertos que estaban al día.

Hoy, la mayoría de los profesionales competentes consideran a los mitos por lo que son: actitudes erróneas que han causado serios problemas, tanto a los gestores como a los técnicos. Sin embargo, las viejas actitudes y hábitos son difíciles de modificar y, cuando vamos hacia la quinta década del software, todavía se cree en algunos restos de los mitos del software.

Existen mitos que vienen del lado de los administradores del proyecto y mitos del lado del cliente como veremos a continuación:

- **Mitos de la administración.** Los administradores con responsabilidades sobre el software, al igual que sus pares en la mayoría de las disciplinas, están normalmente bajo la presión de cumplir lo prosupuestos, evitar que no se retrace el proyecto y mejorar la calidad. Con frecuencia el administrador del software se aferra a un mito del software, si siente que esta creencia disminuirá la presión (aún en forma temporal)
- **Mitos del cliente.** El cliente que solicita un software de computadora puede ser la persona del escritorio de al lado, un grupo técnico en el piso de abajo, el departamento de ventas o de mercadotecnia, o una compañía que ha solicitado el software bajo contrato. En muchos casos, el cliente cree en mitos acerca del software porque los profesionales y administradores del software hacen muy poco para corregir la desinformación. Los mitos conducen a expectativas falsas (del cliente) y en definitiva a insatisfacción con el desarrollador

En el siguiente cuadro tenemos algunos mitos y realidades del software:

Mito	Realidad
Tenemos ya un libro que esta lleno de estándares y procedimientos para construir software: ¿Esto proporcionara a mi gente todo el conocimiento necesario?	Tal vez sea verdad que el libro de estándares existe, pero ¿se usa? ¿Los encargados de la construcción del software saben de su existencia? ¿El libro refleja la práctica moderna de la ingeniería de software? ¿Esta completo? ¿Es adaptable? ¿Esta dirigido al mejoramiento del tiempo de entrega sin dejar de enfocarse en al calidad? En muchos casos la respuesta a todas estas preguntas es no.
Si se está atrasado en el itinerario es posible contratar más programadores para así terminar a	El desarrollo de software no es un proceso mecánico como la manufactura. En palabras de

tiempo.	Brooks. "Agregar gente a un proyecto de software atrasado lo atrasa más". De inicio, este enunciado podría parecer contrario a la intuición. Sin embargo, cuando se agregan nuevos integrantes a un equipo la gente que ya estaba trabajando debe invertir tiempo en la enseñanza a los recién llegados, lo cual reduce el tiempo dedicado al esfuerzo para el desarrollo productivo. Se puede agregar gente pero solo de manera planeada y bien coordinada
Si decido subcontratar el proyecto de software a un tercero, puedo relajarme y dejar que esa compañía lo construya.	Si una organización no entiende como administrar y controlar internamente los proyectos de software, de manera invariable entrará en conflicto al subcontratar este tipo de proyectos

2.4 La ingeniería de software en nuestros días

En los últimos 20 años, los cambios en hardware han sido enormes. Aparentemente, los cambios en software también: Por ejemplo:

- Las grandes infraestructuras (energía, comunicaciones, transporte) descansan sobre sistemas muy complejos y en general muy fiables.
- El auge del Internet y las aplicaciones relacionadas se han ampliado grandemente
- Se dispone de una enorme variedad de tecnologías (p.e. J2EE, .NET, EJB, SAP, BPEL4WS, SOAP, CBSE) para construir aplicaciones (como las aplicaciones web) que pueden ser desplegadas mucho más rápidamente que en el pasado

Es así que la ingeniería de software tiene un efecto sobre la economía y la sociedad como veremos a continuación:

Económicamente: En los EEUU, el software contribuyó a 1/4 de todo el incremento del PIB durante los 90's (alrededor de 90,000 millones de dólares por año), y 1/6 de todo el crecimiento de productividad durante los últimos años de la década (alrededor de 33,000 millones de dólares por año). La ingeniería de software contribuyó a $1 billón de crecimiento económico y productividad en esa década. Alrededor del globo, el software contribuye al crecimiento económico en formas similares, aunque es difícil de encontrar estadísticas fiables.

Socialmente: La ingeniería de software cambia la cultura del mundo debido al extendido uso de la computadora. El correo electrónico (E-mail), la www y la mensajería instantánea permiten a la gente interactuar en nuevas formas. El software baja el costo y mejora la calidad de los servicios de salud, los departamentos de bomberos, las dependencias gubernamentales y otros servicios sociales. Los proyectos exitosos donde se han usado métodos de ingeniería de software incluyen a Linux, el software del transbordador espacial, los cajeros automáticos y muchos otros.

Lo que no ha cambiado…
Sin embargo, más allá de la tecnología, si miramos los procesos de ingeniería del software, desgraciadamente muchas cosas permanecen igual:

- El modelo en cascada sigue siendo utilizado por más del 40% de las empresas[3], a pesar de que sus serios problemas fueron identificados hace más de 20 años.
- La prueba es la técnica de validación predominante, a pesar de que otras técnicas, como la inspección de programas, han sido usados más eficientemente desde los años 70.
- Las herramientas CASE son todavía en su mayoría, simplemente editores de diagramas con algunas funcionalidades de chequeo y generación de código.
- Todavía muchos proyectos terminan tarde, exceden el presupuesto o no entregan el software que esperaban los clientes.
- En muchas áreas sigue sin existir un conjunto de estándares que se use ampliamente, no existen suficientes datos - guía (estadísticas)

[3] IEEE Software, Dic. 2003

2.5 Aproximaciones prometedoras de la Ingeniería del Software:

- Se ha establecido UML (Lenguaje Unificado de Modelado) como una notación estándar de análisis y diseño OO.
- Aparecen métodos ágiles como Extreme Programming[4].
- Algunas universidades han comenzado a ofrecer un título en ingeniería del software.[5]
- La llegada de SWEBOK (Software Engineering Body of Knowledge) que plasma un nuevo concepto de validación internacional de la disciplina de ingeniería de software. El proyecto que dio lugar a este concepto fue impulsado por el IEEE y el documento es actualmente un informe técnico de ISO e IEC con la referencia ISO/IEC TR 19759:2005.
- Se ha cristalizado la integración de CSAB (Computer Science Accreditation Board) dentro de ABET (Accreditation Board for Engineeringand Technology), constituyendo CSAB la sociedad que acredita los programas de Informática, Sistemas de Información e Ingeniería del Software.[6]
- El CMM (Capability Maturity Model) del SEI (Software Engineering Institute) y la familia de estándares ISO 9000 son usados para valorar la capacidad de una organización de ingeniería del software.
- En EE UU, el Colegio de Ingenieros Profesionales de Texas (Texas Board of Professionals Engineers) ha comenzado a licenciar ingenieros del software.
- ACM e IEEE-CS han desarrollado y adoptado conjuntamente un Código de ética para Profesionales en Ingeniería del Software.

3. El proceso de Ingeniería de Software

La ingeniería de software requiere llevar a cabo muchas tareas, sobre todo las siguientes:

a. Análisis de requisitos

Extraer los requisitos de un producto de software es la primera etapa para crearlo. Mientras que los clientes piensan que ellos saben lo que el software tiene que hacer, se requiere de habilidad y experiencia en la ingeniería de software para reconocer requisitos incompletos, ambiguos o contradictorios. El resultado del análisis de requisitos con el cliente se plasma en el documento ERS, *Especificación de Requerimientos del Sistema*, cuya estructura puede venir definida por varios estándares, tales como CMM-I. Asimismo, se define un diagrama de Entidad/Relación, en el que se plasman las principales entidades que participarán en el desarrollo del software.

b. Especificación

Es la tarea de describir detalladamente el software a ser escrito, en una forma matemáticamente rigurosa. En la realidad, la mayoría de las buenas especificaciones han sido escritas para entender y afinar aplicaciones que ya estaban desarrolladas. Las especificaciones son más importantes para las interfaces externas, que deben permanecer estables.

c. Diseño y arquitectura

Se refiere a determinar como funcionará de forma general sin entrar en detalles. Yourdon dice que consiste en incorporar consideraciones de la implementación tecnológica, como el hardware, la red, etc. Se definen los casos de uso para cubrir las funciones que realizará el sistema, y se transforman las entidades definidas en el análisis de requisitos en clases de diseño, obteniendo un modelo cercano a la programación orientada a objetos.

---

[4] SWEBOK (http://www.idg.es/computerworld/articulo.asp?id=175227)

[5] http://www.upc.edu.pe/plantillas/0_0.asp?ARE=0&PFL=0&CAT=3&SUB=38&SSC=3910
http://webprojects.utp.edu.pe/portals/Ingenier%C3%ADadeSoftware/tabid/81/Default.aspx

[6] http://www.csab.org/

d. Programación

Reducir un diseño a código puede ser la parte más obvia del trabajo de ingeniería de software, pero no es necesariamente la porción más larga.

e. Prueba

Consiste en comprobar que el software realice correctamente las tareas indicadas en la especificación. Una técnica de prueba es probar por separado cada módulo del software, y luego probarlo de forma integral.

f. Documentación

Realización del manual de usuario, y posiblemente un manual técnico con el propósito de mantenimiento futuro y ampliaciones al sistema.

g. Mantenimiento

Mantener y mejorar el software para enfrentar errores descubiertos y nuevos requisitos. Esto puede llevar más tiempo incluso que el desarrollo inicial del software. Alrededor de 2/3 de toda la ingeniería de software tiene que ver con dar mantenimiento. Una pequeña parte de este trabajo consiste en arreglar errores, o bugs. La mayor parte consiste en extender el sistema para hacer nuevas cosas. De manera similar, alrededor de 2/3 de toda la ingeniería civil, arquitectura y trabajo de construcción es dar mantenimiento.

4. Elementos de la Ingeniería de software.

La Ingeniería de Software surge dentro de la ingeriría de sistemas y de hardware. Abarca un conjunto de 3 elementos: métodos, herramientas y procedimientos; que facilitan al gestor controlar el proceso de desarrollo del software y suministrar a los que practiquen dicha ingeniería, las bases para construir software de alta calidad de una forma productiva. A continuación examinaremos brevemente cada uno de esos elementos:

a. **Los métodos** de la ingeniería de software indican *como* construir técnicamente el software. Los métodos abarcan un amplio espectro de tareas que incluyen: planificación y estimación de proyectos, análisis de los requisitos del sistema y del software, diseño de estructuras de datos, arquitectura de programas y procedimientos algorítmicos, codificación, prueba y mantenimiento. Los métodos de la ingeniería del software introducen frecuentemente una notación especial orientada a un lenguaje o grafica y un conjunto de criterios para la calidad del software.

b. **Las herramientas** de la ingeniería del software suministran un soporte automático o semiautomático para los métodos. Hoy existen herramientas para soportar cada uno de los métodos mencionados anteriormente. Cuando se integran las herramientas de forma que la información creada por una herramienta pueda ser usada por otra, se establece un sistema para el soporte del desarrollo del software, llamado ingeniería de software asistida por computadora (del inglés, CASE). CASE combina software, hardware y bases de datos sobre ingeniería del software (una estructura de datos que contenga la información relevante sobre el análisis, diseño, codificación y prueba) para crear un entorno de ingeniería de software, por ejemplo, análogo al diseño asistido por computadora, CAD/CAE (de las siglas en inglés) para el hardware.

c. **Los procedimientos** de la ingeniería de software son el pegamento que junta los métodos y las herramientas y facilita su desarrollo racional y oportuno del software de computadora. Los procedimientos definen la secuencia en la que se aplican los métodos, las entregas (documentos, informes, formas, etc.) que se requieren, los controles que ayudan a

asegurar la calidad y coordinar los cambios, y las directrices que ayudan a los gestores del software a evaluar el progreso.

La ingeniería del software esta compuesta por una serie de pasos que abarcan los métodos, las herramientas y los procedimientos antes mencionados. Estos pasos se denominan frecuentemente modelos o paradigmas de la ingeniería del software (involucran el ciclo de vida del software). La elección de un paradigma para la ingeniería del software se lleva a cabo de acuerdo con la naturaleza del proyecto y de la aplicación, los métodos y herramientas a usar y los controles y entregas requeridos.

5. Herramientas CASE (Computer Aided Assisted Automated Software Systems Engineering)

Se puede definir a las herramientas CASE como un conjunto de programas y ayudas que dan asistencia a los analistas, ingenieros de software y desarrolladores, durante todos los pasos del Ciclo de Vida de desarrollo de un Software (investigación preliminar, análisis, diseño, implementación e instalación.).
CASE es también definido como el Conjunto de métodos, utilidades y técnicas que facilitan el mejoramiento del ciclo de vida del desarrollo de sistemas de información, completamente o en alguna de sus fases. Se puede ver al CASE como la unión de las herramientas automáticas de software y las metodologías de desarrollo de software formales. Existe también el CASE integrado que fue comenzando a tener un impacto muy significativo en los negocios y sistemas de información de las organizaciones, además con este CASE integrado las compañías pueden desarrollar rápidamente sistemas de mejor calidad para soportar procesos críticos del negocio y asistir en el desarrollo y promoción intensiva de la información de productos y servicios.

5.1 Clasificación de las Herramientas Case

No existe una única clasificación de herramientas CASE y, en ocasiones, es difícil incluirlas en una clase en común. Sin embargo, podríamos clasificarlas según:

- Las plataformas que soportan.
- Las fases del ciclo de vida del desarrollo de sistemas que abarca.
- La arquitectura de las aplicaciones que produce.
- Su funcionalidad.

Las herramientas CASE, en función de las fases del ciclo de vida que cubre, se pueden agrupar de la forma siguiente:

a. **Herramientas integradas**, I-CASE (Integrated CASE, CASE integrado): abarcan todas las fases del ciclo de vida del desarrollo de sistemas. Son llamadas también CASE workbench.

b. **Herramientas de alto nivel**, U-CASE (Upper CASE - CASE superior), orientadas a la automatización y soporte de las actividades desarrolladas durante las primeras fases del desarrollo: análisis y diseño.

c. **Herramientas de bajo nivel**, L-CASE (Lower CASE - CASE inferior), dirigidas a las últimas fases del desarrollo: construcción e implantación.

d. **Juegos de herramientas o Tools-Case**, son el tipo más simple de herramientas CASE. Automatizan una fase dentro del ciclo de vida. Dentro de este grupo se encontrarían las herramientas de reingeniería, orientadas a la fase de mantenimiento.

5.2 Ejemplos de Herramientas Case más utilizadas.

a) ERwin:

PLATINUM ERwin es una herramienta para el diseño de base de datos, que brinda productividad en su diseño, generación, y mantenimiento de aplicaciones. Desde un modelo lógico de los requerimientos de información, hasta el modelo físico perfeccionado para las características específicas de la base de datos diseñada, además ERwin permite visualizar la estructura, los elementos importantes, y optimizar el diseño de la base de datos. Genera

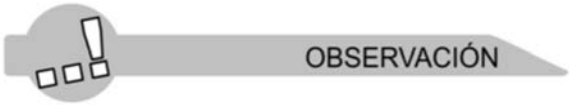
automáticamente las tablas y miles de líneas de store procedures[7] y triggers[8] para los principales tipos de base de datos.
ERwin hace fácil el diseño de una base de datos. Los diseñadores de bases de datos sólo apuntan y pulsan un botón para crear un gráfico del modelo E-R (Entidad - Relación) de todos sus requerimientos de datos y capturar las reglas de negocio en un modelo lógico, mostrando todas las entidades, atributos, relaciones, y llaves importantes.
ERwin soporta principalmente bases de datos relacionales: Oracle, Microsoft SQL Server, Sybase. El mismo modelo puede ser usado para generar múltiples bases de datos, o convertir una aplicación de una plataforma de base de datos a otra.

b) EasyCASE

EasyCASE Profesional es un producto para la generación de esquemas de base de datos e ingeniería reversa - trabaja para proveer una solución comprensible para el diseño, consistencia y documentación del sistema en conjunto.
Esta herramienta permite automatizar las fases de análisis y diseño dentro del desarrollo de una aplicación, para poder crear las aplicaciones eficazmente – desde el procesamiento de transacciones a la aplicación de bases de datos de cliente/servidor, así como sistemas de tiempo real.
EasyCASE Profesional, una herramienta multi-usuario, es ideal para aquellos que necesitan compartir datos y trabajar en un proyecto con otros departamentos. El equipo completo puede acceder proyectos localizados en el servidor de la red concurrentemente. Para asegurar la seguridad de los datos, existe el diagrama y diccionario de los datos que bloquean por niveles al registro, al archivo y al proyecto, y niveles de control de acceso.

c) Oracle Designer:

Oracle Designer es un conjunto de herramientas para guardar las definiciones que necesita el usuario y automatizar la construcción rápida de aplicaciones cliente/servidor gráficas.
Todos los datos ingresados por cualquier herramienta de Oracle Designer, en cualquier fase de desarrollo, se guardan en un repositorio central, habilitando el trabajo fácil del equipo y la dirección del proyecto.



Existen innumerables herramientas CASE que pueden ser de ayuda en todo el proceso de desarrollo de software. Sin embargo es difícil encontrar una única herramienta CASE que pueda acompañarnos en todo el proceso

1. Defina con sus propias palabras a la Ingeniería del software
2. Enumere cinco causas de la crisis del software
3. Enumere tres mitos del software del lado del cliente
4. Determine la clasificación de las herramientas CASE mencionadas como ejemplos
5. Investigue acerca de Rational Rose y defina si es una herramienta CASE y en que parte del proceso de desarrollo de software puede ayudarnos

[7] Los store procedures son una colección precompilada de instrucciones de programación, que realizan operaciones en la base de datos y que se almacenan bajo un nombre y se procesan como una unidad (http://msdn2.microsoft.com/es-es/library/k2e1fb36(VS.80).aspx)

[8] Un trigger es un tipo especial de procedimiento almacenado, que se activa al modificar datos en una tabla especificada mediante una o más de las operaciones de modificación de datos: UPDATE, INSERT o DELETE (http://msdn2.microsoft.com/es-es/library/k2e1fb36(VS.80).aspx)

Hemos definido a la Ingeniería del software como la aplicación inteligente de principios probados, técnicas, lenguajes y herramientas para la creación y mantenimiento, dentro de un coste razonable, de software que satisfaga las necesidades de los usuarios.
También hemos comprendido la necesidad de seguir una metodología adecuada para todo el proceso de desarrollo de software y así evitar los problemas generados durante la llamada "crisis del software". Entendemos que solo siguiendo los procedimientos adecuados acabaremos también con los mitos del software y realizaremos proyectos en condiciones más realistas y con mayor probabilidad de éxito.

[1] PRESSMAN, Roger S. **Ingeniería del Software. Un enfoque práctico.** Mc Graw Hill, 6ta Ed, Interamericana de España S.A.U, 2006

[2] CAMPDERRICH FALGUERAS, Benet. **Ingeniería del Software. Abril 2002.** Ed UOC, 1ra Ed. España, 2002

Bibliografía electrónica:

- SWEBOK: Guía sobre Ingeniería del Software
  http://www.swebok.org/

En la siguiente unidad temática detallaremos aspectos relacionados al proceso de desarrollo de software y su ciclo de vida

AUTOEVALUACIÓN FORMATIVA

INGENIERIA DEL SOFTWARE
UNIDAD ACADÉMICA Nº 1

NOMBRE:_____

APELLIDOS:_____FECHA; ____/_____/_____

CIUDAD:_____SEMESTRE:_____

PARTE I: Marque la alternativa correcta:

1. Podemos definir a la Ingeniería de software como:
   a) El desarrollo de software a medida
   b) La rama de la ingeniería que se preocupa por mejorar la programación
   c) La aplicación de técnicas y herramientas para la creación y mantenimiento, de software
   d) La rama de la ingeniería que se ocupa del mantenimiento de software dentro de un tiempo determinado y reduciendo costos
   e) N.A.

2. Una meta de la Ingeniería de software es:
   a) Analizar y diseñar sistemas de software dentro de las restricciones de tiempo, funcionalidad y costos acordados con el cliente
   b) Definir métodos y técnica para la construcción de software
   c) Definir estándares para la programación
   d) Dar mantenimiento al software de acuerdo a las necesidades del cliente
   e) Mejorar la calidad de los productos desarrollados y aumentar la productividad de los ingenieros de software

3. El proceso de Ingeniería del software esta dado por:
   a) Análisis, diseño, programación, pruebas, documentación y mantenimiento.
   b) Análisis del diseño, arquitectura, programación, pruebas, documentación y mantenimiento
   c) Análisis de requisitos, diseño, programación, pruebas, mantenimiento y diseño de la arquitectura
   d) Análisis de requisitos, programación, pruebas, mantenimiento y documentación
   e) N.A.

4. No es una herramienta CASE:

   a) Oracle Designer
   b) Erwin
   c) Easy Case
   d) Rational Rose
   e) N.A.

5. “El proyecto de software estaba programado para concluir en Noviembre. Faltan escasos dos meses y el proyecto está retrasado. Contrataré mas programadores para así terminar a tiempo”. La afirmación anterior constituye un mito del software. Esto es:

   a) Verdadero
   b) Falso

PARTE II: Responda brevemente:

6. ¿Qué entiende por crisis del software?

_____
_____
_____

7. ¿Cree que puede obviar algunos pasos del proceso de desarrollo de software? De ser así ¿Cuáles y por qué?

_____
_____
_____
_____
_____
_____

Unidad Académica II

EL PRODUCTO SOFTWARE Y SU CICLO DE VIDA

El desarrollo de aplicaciones informáticas (proyectos medios-grandes) se ha convertido en una tarea compleja que involucra un gran número de recursos (tanto humanos como materiales); resulta pues de vital importancia la adopción de métodos a fin de guiar la construcción, mantenimiento y evolución de la aplicación a través de su ciclo de vida.

Sommerville define modelo de proceso de software o ciclo de vida como "Una representación simplificada de un proceso de software, representada desde una perspectiva específica. Por su naturaleza los modelos son simplificados, por lo tanto un modelo de procesos del software es una abstracción de un proceso real."

Los modelos genéricos no son descripciones definitivas de procesos de software; sin embargo, son abstracciones útiles que pueden ser utilizadas para explicar diferentes enfoques del desarrollo de software.

Al finalizar el estudio de la presente unidad temática el estudiante:

5. Conoce el proceso de desarrollo de software
6. Enlaza el proceso de desarrollo de software con el ciclo de vida
7. Analiza las ventajas y desventajas del los diferentes modelos de ciclo de vida del software
8. Selecciona el modelo de ciclo de vida más adecuado de acuerdo a las necesidades del proyecto de software.

1. El proceso de desarrollo del software

Un proceso de desarrollo de software tiene como propósito la producción eficaz y eficiente de un producto software que reúna los requisitos del cliente. Dicho proceso, en términos globales se muestra en la Figura 2.1 [4]. Este proceso es intensamente intelectual, afectado por la creatividad y juicio de las personas involucradas [3]. Aunque un proyecto de desarrollo de software es equiparable en muchos aspectos a cualquier otro proyecto de ingeniería, en el desarrollo de software hay una serie de desafíos adicionales, relativos esencialmente a la naturaleza del producto obtenido. A continuación se explican algunas particularidades asociadas al desarrollo de software y que influyen en su proceso de construcción.

Un producto software en sí es complejo, es prácticamente inviable conseguir un 100% de confiabilidad de un programa por pequeño que sea. Existe una inmensa combinación de factores que impiden una verificación exhaustiva de las todas posibles situaciones de ejecución que se puedan presentar (entradas, valores de variables, datos almacenados, software del sistema, otras aplicaciones que intervienen, el hardware sobre el cual se ejecuta, etc.).

Un producto software es intangible y por lo general muy abstracto, esto dificulta la definición del producto y sus requisitos, sobre todo cuando no se tiene precedentes en productos software similares. Esto hace que los requisitos sean difíciles de consolidar tempranamente. Así, los cambios en los requisitos son inevitables, no sólo después de entregado en producto sino también durante el proceso de desarrollo.

Además, de las dos anteriores, siempre puede señalarse la inmadurez de la ingeniería del software como disciplina, justificada por su corta vida comparada con otras disciplinas de la ingeniería. Sin embargo, esto no es más que un inútil consuelo.

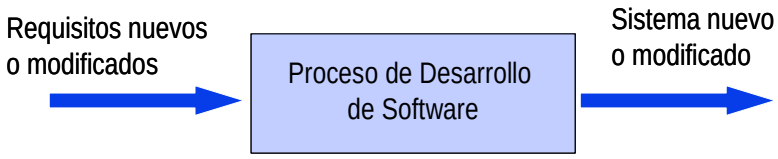


Figura 1.1: *Proceso de desarrollo de software.*

El proceso de desarrollo de software no es único. No existe un proceso de software universal que sea efectivo para todos los contextos de proyectos de desarrollo. Debido a esta diversidad, es difícil automatizar todo un proceso de desarrollo de software.
A pesar de la variedad de propuestas de proceso de software, existe un conjunto de actividades fundamentales que se encuentran presentes en todos ellos [3]:

1. **Especificación de software**: Se debe definir la funcionalidad y restricciones operacionales que debe cumplir el software.
2. **Diseño e Implementación**: Se diseña y construye el software de acuerdo a la especificación.
3. **Validación**: El software debe validarse, para asegurar que cumpla con lo que quiere el cliente.
4. **Evolución**: El software debe evolucionar, para adaptarse a las necesidades del cliente.

Además de estas actividades fundamentales, Pressman [1] menciona un conjunto de "actividades protectoras", que se aplican a lo largo de todo el proceso del software. Ellas se señalan a continuación:

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

Pressman [1] caracteriza un proceso de desarrollo de software como se muestra en la Figura 2.2. Los elementos involucrados se describen a continuación:

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

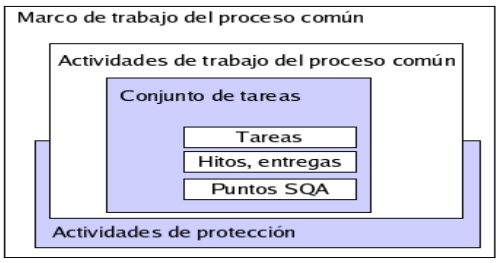


Figura 2.2: *Elementos del proceso del software*

Otra perspectiva utilizada para determinar los elementos del proceso de desarrollo de software es establecer las relaciones entre elementos que permitan responder **Quién** debe hacer **Qué**, **Cuándo** y **Cómo** debe hacerlo [4].

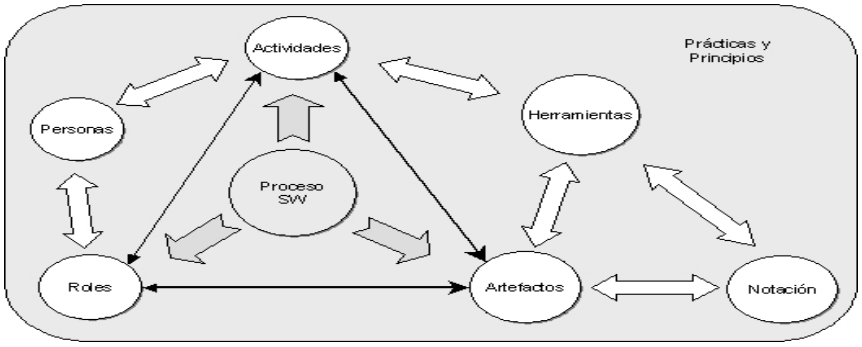


Figura 2.3: *Relación entre elementos del proceso del software*

En la Figura 2.3 se muestran los elementos de un proceso de desarrollo de software y sus relaciones. Así las interrogantes se responden de la siguiente forma:

- **Quién:** Las Personas participantes en el proyecto de desarrollo desempeñando uno o más Roles específicos.
- **Qué:** Un Artefacto[9] es producido por un Rol en una de sus Actividades. Los Artefactos se especifican utilizando Notaciones específicas. Las Herramientas apoyan la elaboración de Artefactos soportando ciertas Notaciones.
- **Cómo y Cuándo:** Las Actividades son una serie de pasos que lleva a cabo un Rol durante el proceso de desarrollo. El avance del proyecto está controlado mediante hitos que establecen un determinado estado de terminación de ciertos Artefactos.

2. El Ciclo de Vida

Todo proyecto de ingeniería tiene unos fines ligados a la obtención de un producto, proceso o servicio que es necesario generar a través de diversas actividades. Algunas de estas actividades pueden agruparse en fases porque globalmente contribuyen a obtener un producto intermedio, necesario para continuar hacia el producto final y facilitar la gestión del proyecto. Al conjunto de las fases empleadas se le denomina "ciclo de vida".

La ISO, Internacional Organization for Standarization, en su norma 12207 define al ciclo de vida de un software como un marco de referencia que contiene las actividades y las tareas involucradas en el desarrollo, la explotación y el mantenimiento de un producto software, abarcando desde la definición hasta la finalización de su uso

Sin embargo, la forma de agrupar las actividades, los objetivos de cada fase, los tipos de productos intermedios que se generan, etc. pueden ser muy diferentes dependiendo del tipo de producto o proceso a generar y de las tecnologías empleadas.

La complejidad de las relaciones entre las distintas actividades crece exponencialmente con el tamaño, con lo que rápidamente se haría inabordable si no fuera por la vieja táctica de "divide y vencerás". De esta forma la división de los proyectos en fases sucesivas es un primer paso para la reducción de su complejidad, tratándose de escoger las partes de manera que sus relaciones entre sí sean lo más simples posibles.

La definición de un ciclo de vida facilita el control sobre los tiempos en que es necesario aplicar recursos de todo tipo (personal, equipos, suministros, etc.) al proyecto. Si el proyecto incluye subcontratación de partes a otras organizaciones, el control del trabajo subcontratado se facilita en la medida en que esas partes encajen bien en la estructura de las fases. El control de calidad también se ve facilitado si la separación entre fases se hace corresponder con puntos en los que ésta deba verificarse (mediante comprobaciones sobre los productos parciales obtenidos).

De la misma forma, la práctica acumulada en el diseño de modelos de ciclo de vida para situaciones muy diversas permite que nos beneficiemos de la experiencia adquirida utilizando el enfoque que mejor de adapte a nuestros requerimientos.

2.1 Elementos Del Ciclo De Vida

Un ciclo de vida para un proyecto se compone de **fases sucesivas** compuestas por tareas planificables. Según el modelo de ciclo de vida, la sucesión de fases puede ampliarse con **bucles de realimentación**, de manera que lo que conceptualmente se considera una misma fase se pueda ejecutar más de una vez a lo largo de un proyecto, recibiendo en cada pasada de ejecución aportaciones de los resultados intermedios que se van produciendo (realimentación).

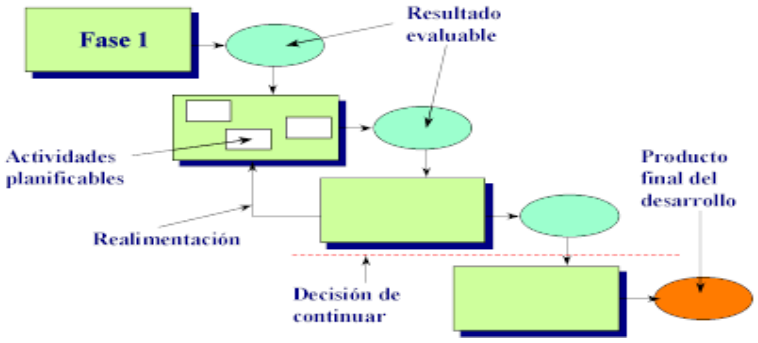


Figura 2.4: *Elementos del ciclo de vida*

Para un adecuado control de la progresión de las fases de un proyecto se hace necesario especificar con suficiente precisión los resultados evaluables, o sea, productos intermedios que deben resultar de las tareas incluidas en cada fase. Normalmente estos productos marcan los hitos entre fases.

A continuación presentamos los distintos elementos que integran un ciclo de vida:

- **Fases**. Una fase es un conjunto de actividades relacionadas con un objetivo en el desarrollo del proyecto. Se construye agrupando tareas (actividades elementales) que pueden compartir un tramo determinado del tiempo de vida de un proyecto. La agrupación temporal de tareas impone requisitos temporales correspondientes a la asignación de recursos (humanos, financieros o materiales).
  Cuanto más grande y complejo sea un proyecto, mayor detalle se necesitará en la definición de las fases para que el contenido de cada una siga siendo manejable. De esta forma, cada fase de un proyecto puede considerarse un "*micro-proyecto*" en sí mismo, compuesto por un conjunto de micro-fases.

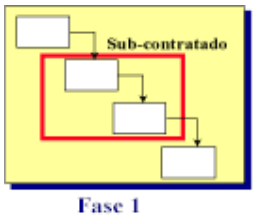


Figura 2.5: Descomposición de fases

Otro motivo para descomponer una fase en subfases menores puede ser el interés de separar partes temporales del proyecto que se subcontraten a otras organizaciones, requiriendo distintos procesos de gestión.

Cada fase viene definida por un conjunto de elementos observables externamente, como son las **actividades** con las que se relaciona, los **datos de entrada** (resultados de la fase anterior, documentos o productos requeridos para la fase, experiencias de proyectos anteriores), los **datos de salida** (resultados a utilizar por la fase posterior, experiencia acumulada, pruebas o resultados efectuados) y la **estructura interna** de la fase.

Figura 2.6: *Esquema general de operación de una fase*

Entregables ("*deliverables*"). Son los productos intermedios que generan las fases. Pueden ser materiales (componentes, equipos) o inmateriales (documentos, software). Los entregables permiten evaluar la marcha del proyecto mediante comprobaciones de su adecuación o no a los requisitos funcionales y de condiciones de realización previamente establecidos. Cada una de estas evaluaciones puede servir, además, para la toma de decisiones a lo largo del desarrollo del proyecto.

2.2 Modelos del Ciclo de Vida

a) Codificar y corregir (Code-and-Fix)

Este es el modelo básico utilizado en los inicios del desarrollo de software. Contiene dos pasos:

- Escribir código.
- Corregir problemas en el código.

Se trata de primero implementar algo de código y luego pensar acerca de requisitos, diseño, validación, y mantenimiento.

Este modelo tiene tres problemas principales [5]:

- Después de un número de correcciones, el código puede tener una muy mala estructura, hace que los arreglos sean muy costosos.
- Frecuentemente, aún el software bien diseñado, no se ajusta a las necesidades del usuario, por lo que es rechazado o su reconstrucción es muy cara.
- El código es difícil de reparar por su pobre preparación para probar y modificar.

b) Desarrollo en cascada

En ingeniería de software el desarrollo en cascada, también llamado modelo en cascada, es el enfoque metodológico que ordena rigurosamente las etapas del ciclo de vida del software, de forma tal que el inicio de cada etapa debe esperar a la finalización de la inmediatamente anterior.

De esta forma, cualquier error de diseño detectado en la etapa de prueba conduce necesariamente al rediseño y nueva programación del código afectado, aumentando los costes del desarrollo. La palabra *cascada* sugiere, mediante la metáfora de la fuerza de la gravedad, el esfuerzo necesario para introducir un cambio en las fases más avanzadas de un proyecto

El modelo en cascada consta de las siguientes fases:

1. **Definición de los requisitos:** Se analizan las necesidades de los usuarios finales del software para determinar qué objetivos debe cubrir. De esta fase surge una memoria llamada SRD (Documento de Especificación de Requisitos), que contiene la especificación completa de lo que debe hacer el sistema sin entrar en detalles internos. Es importante señalar que en esta etapa se deben consensuar todo lo que se requiere del sistema y será aquello lo que seguirá en las siguientes etapas, no pudiéndose requerir nuevos resultados a mitad del proceso de elaboración del software
2. **Diseño del sistema y software:** Se descompone y organiza el sistema en elementos

que puedan elaborarse por separado, aprovechando las ventajas del desarrollo en equipo. Como resultado surge el SDD (Documento de Diseño del Software), que contiene la descripción de la estructura relacional global del sistema y la especificación de lo que debe hacer cada una de sus partes, así como la manera en que se combinan unas con otras.
Es la fase de diseño del software es en donde se realizan los algoritmos necesarios para el cumplimiento de los requerimientos del usuario así como también los análisis necesarios para saber que herramientas usar en la etapa de Codificación.

3. **Implementación y pruebas unitarias:** Es la fase de programación propiamente dicha. Aquí se desarrolla el código fuente, haciendo uso de prototipos así como pruebas y ensayos para corregir errores. Se realizan pruebas de cada unidad
Dependiendo del lenguaje de programación y su versión se crean las librerías y componentes reutilizables dentro del mismo proyecto para hacer que la programación sea un proceso mucho más rápido.

4. **Integración y pruebas del sistema:** Se integran todas las unidades. Se prueban en conjunto. Se entrega el conjunto probado al cliente.

5. **Operación y mantenimiento:** Generalmente es la fase más larga. El sistema es puesto en marcha y se realiza la corrección de errores descubiertos. Se realizan mejoras de implementación, incluso pueden surgir cambios, bien para corregir errores o bien para introducir mejoras. Todo ello se recoge en los Documentos de Cambios

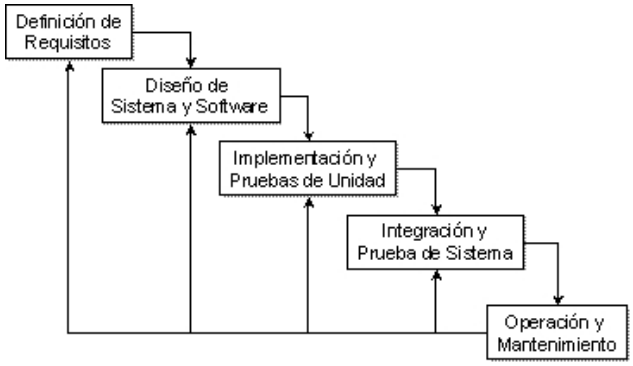


Figura 2.7: *Modelo de desarrollo en cascada.*

La interacción entre fases puede observarse en la Figura 2.7. Cada fase tiene como resultado documentos que deben ser aprobados por el usuario.
Una fase no comienza hasta que termine la fase anterior y generalmente se incluye la corrección de los problemas encontrados en fases previas.
En la práctica, este modelo no es lineal, e involucra varias iteraciones e interacción entre las distintas fases de desarrollo. Algunos problemas que se observan en el modelo de cascada son:

- Las iteraciones son costosas e implican rehacer trabajo debido a la producción y aprobación de documentos.
- Aunque son pocas iteraciones, es normal congelar parte del desarrollo y continuar con las siguientes fases.
- Los problemas se dejan para su posterior resolución, lo que lleva a que estos sean ignorados o corregidos de una forma poco elegante.
- Existe una alta probabilidad de que el software no cumpla con los requisitos del usuario por el largo tiempo de entrega del producto.
- Es inflexible a la hora de evolucionar para incorporar nuevos requisitos. Es difícil responder a cambios en los requisitos.

Este modelo sólo debe usarse si se entienden a plenitud los requisitos. Aún se utiliza como parte de proyectos grandes.

c) Desarrollo Evolutivo o Modelo de Prototipos

En Ingeniería de software el desarrollo con prototipación, también llamado modelo de prototipos o modelo de desarrollo evolutivo, se inicia con la definición de los objetivos globales para el software, luego se identifican los requisitos conocidos y las áreas del esquema en donde es necesaria más definición. Entonces se plantea con rapidez una iteración de construcción de prototipos y se presenta el modelado (en forma de un diseño rápido).

El diseño rápido se centra en una representación de aquellos aspectos del software que serán visibles para el cliente o el usuario final (por ejemplo, la configuración de la interfaz con el usuario y el formato de los despliegues de salida). El diseño rápido conduce a la construcción de un prototipo, el cual es evaluado por el cliente o el usuario para una retroalimentación; gracias a ésta se refinan los requisitos del software que se desarrollará. La iteración ocurre cuando el prototipo se ajusta para satisfacer las necesidades del cliente. Esto permite que al mismo tiempo el desarrollador entienda mejor lo que se debe hacer y el cliente vea resultados a corto plazo.

Entre los puntos favorables de este modelo están:

- Este modelo es útil cuando el cliente conoce los objetivos generales para el software, pero no identifica los requisitos detallados de entrada, procesamiento o salida.
- También ofrece un mejor enfoque cuando el responsable del desarrollo del software está inseguro de la eficacia de un algoritmo, de la adaptabilidad de un sistema operativo o de la forma que debería tomar la interacción humano-máquina.

La construcción de prototipos se puede utilizar como un modelo del proceso independiente, se emplea más comúnmente como una técnica susceptible de implementarse dentro del contexto de cualquiera de los modelos del proceso expuestos. Sin importar la forma en que éste se aplique, el paradigma de construcción de prototipos ayuda al desarrollador de software y al cliente a entender de mejor manera cuál será el resultado de la construcción cuando los requisitos estén satisfechos.

A pesar de que tal vez surjan problemas, la construcción de prototipos puede ser un paradigma efectivo para la ingeniería del software. La clave es definir las reglas del juego desde el principio; es decir, el cliente y el desarrollador se deben poner de acuerdo en:

- Que el prototipo se construya y sirva como un **mecanismo para la definición de requisitos**
- Que el prototipo se **descarte**, al menos en parte
- Que después se desarrolle el software real con un enfoque hacia la **calidad**

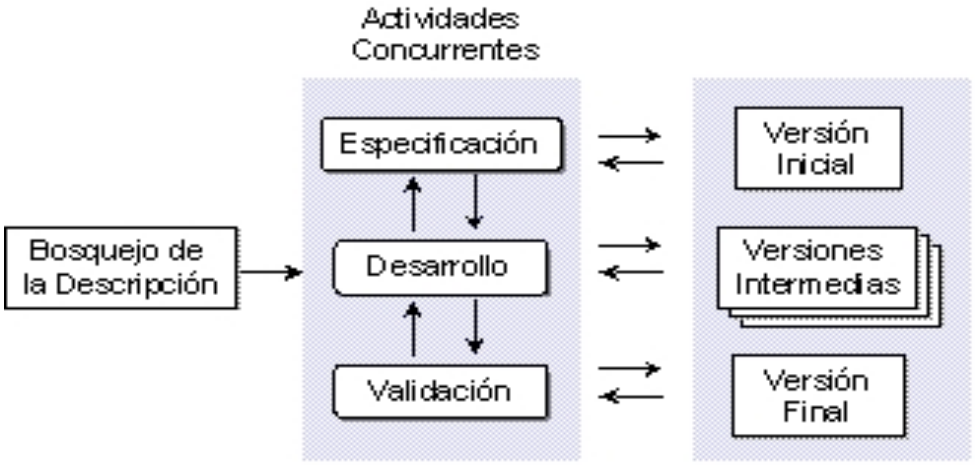


Figura 2.8: *Modelo de desarrollo evolutivo.*

d) Desarrollo formal de sistemas

Este modelo se basa en transformaciones formales de los requisitos hasta llegar a un programa ejecutable.

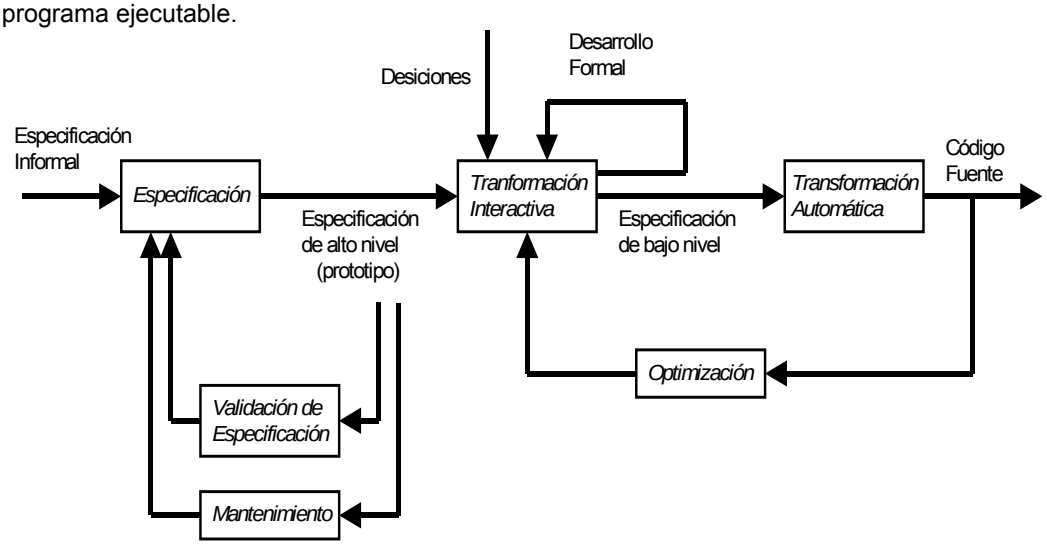


Figura 2.9: *Paradigma de programación automática.*

La Figura 2.9 ilustra un paradigma ideal de programación automática. Se distinguen dos fases globales: especificación (incluyendo validación) y transformación. Las características principales de este paradigma son: la especificación es formal y ejecutable constituye el primer prototipo del sistema), la especificación es validada mediante prototipación. Posteriormente, a través de transformaciones formales la especificación se convierte en la implementación del sistema, en el último paso de transformación se obtiene una implementación en un lenguaje de programación determinado, el mantenimiento se realiza sobre la especificación (no sobre el código fuente), la documentación es generada automáticamente y el mantenimiento es realizado por repetición del proceso (no mediante parches sobre la implementación).
Observaciones sobre el desarrollo formal de sistemas:

- Permite demostrar la corrección del sistema durante el proceso de transformación. Así, las pruebas que verifican la correspondencia con la especificación no son necesarias.
- Es atractivo sobre todo para sistemas donde hay requisitos de seguridad y confiabilidad importantes.
- Requiere desarrolladores especializados y experimentados en este proceso para llevarse a cabo.

e) Desarrollo basado en reutilización

Como su nombre lo indica, es un modelo fuertemente orientado a la reutilización. Este modelo consta de 4 fases ilustradas en la Figura 2.9. A continuación se describe cada fase:

1. **Análisis de componentes**: Se determina qué componentes pueden ser utilizados para el sistema en cuestión. Casi siempre hay que hacer ajustes para adecuarlos.

2. **Modificación de requisitos**: Se adaptan (en lo posible) los requisitos para concordar con los componentes de la etapa anterior. Si no se puede realizar modificaciones en los requisitos, hay que seguir buscando componentes más adecuados (fase 1).
3. **Diseño del sistema con reutilización**: Se diseña o reutiliza el marco de trabajo para el sistema. Se debe tener en cuenta los componentes localizados en la fase 2 para diseñar o determinar este marco.
4. **Desarrollo e integración**: El software que no puede comprarse, se desarrolla. Se integran los componentes y subsistemas. La integración es parte del desarrollo en lugar de una actividad separada.

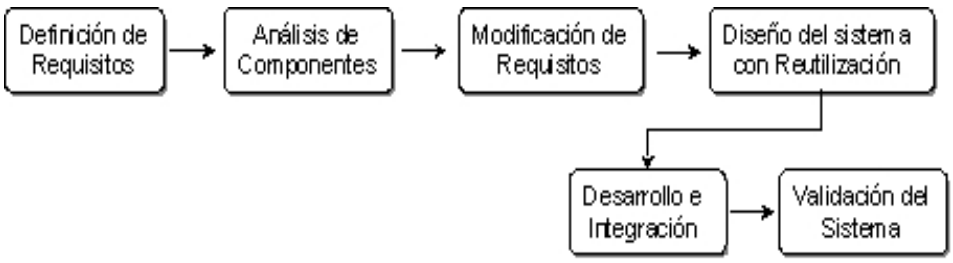


Figura 2.9: *Desarrollo basado en reutilización de componentes*

Las ventajas de este modelo son:

- Disminuye el costo y esfuerzo de desarrollo.
- Reduce el tiempo de entrega.
- Disminuye los riesgos durante el desarrollo.

Desventajas de este modelo:

- Los "compromisos" en los requisitos son inevitables, por lo cual puede que el software no cumpla las expectativas del cliente.
- Las actualizaciones de los componentes adquiridos no están en manos de los desarrolladores del sistema.

f) Procesos iterativos

A continuación se expondrán dos enfoques híbridos, especialmente diseñados para el soporte de las iteraciones:

- Desarrollo Incremental.
- Desarrollo en Espiral.

f.1 Desarrollo incremental

Mills [6] sugirió el enfoque incremental de desarrollo como una forma de reducir la repetición del trabajo en el proceso de desarrollo y dar oportunidad de retrasar la toma de decisiones en los requisitos hasta adquirir experiencia con el sistema (ver Figura 2.10). Es una combinación del Modelo de Cascada y Modelo Evolutivo.

Reduce el rehacer trabajo durante el proceso de desarrollo y da oportunidad para retrasar las decisiones hasta tener experiencia en el sistema.

Durante el desarrollo de cada incremento se puede utilizar el modelo de cascada o evolutivo, dependiendo del conocimiento que se tenga sobre los requisitos a implementar. Si se tiene un buen conocimiento, se puede optar por cascada, si es dudoso, evolutivo.

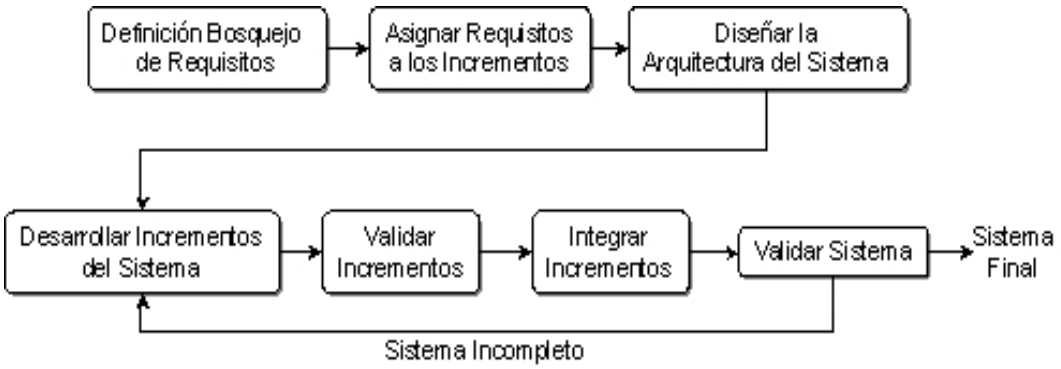


Figura 2.10: Modelo de desarrollo iterativo incremental.

Entre las ventajas del modelo incremental se encuentran:

- Los clientes no esperan hasta el fin del desarrollo para utilizar el sistema. Pueden empezar a usarlo desde el primer incremento.
- Los clientes pueden aclarar los requisitos que no tengan claros conforme ven las entregas del sistema.
- Se disminuye el riesgo de fracaso de todo el proyecto, ya que se puede distribuir en cada incremento.
- Las partes más importantes del sistema son entregadas primero, por lo cual se realizan más pruebas en estos módulos y se disminuye el riesgo de fallos.

Algunas de las desventajas identificadas para este modelo son:

- Cada incremento debe ser pequeño para limitar el riesgo Cada incremento debe aumentar la funcionalidad.
- Es difícil establecer las correspondencias de los requisitos contra los incrementos.
- Es difícil detectar las unidades o servicios genéricos para todo el sistema.

f.2 Desarrollo en espiral

El modelo de desarrollo en espiral (ver Figura 2.11) es actualmente uno de los más conocidos y fue propuesto por Boehm [5]. El ciclo de desarrollo se representa como una espiral, en lugar de una serie de actividades sucesivas con retrospectiva de una actividad a otra.

Cada ciclo de desarrollo se divide en cuatro fases:

1. **Definición de objetivos**: Se definen los objetivos. Se definen las restricciones del proceso y del producto. Se realiza un diseño detallado del plan administrativo. Se identifican los riesgos y se elaboran estrategias alternativas dependiendo de estos.
2. **Evaluación y reducción de riesgos**: Se realiza un análisis detallado de cada riesgo identificado. Pueden desarrollarse prototipos para disminuir el riesgo de requisitos dudosos. Se llevan a cabo los pasos para reducir los riesgos.
3. **Desarrollo y validación**: Se escoge el modelo de desarrollo después de la evaluación del riesgo. El modelo que se utilizará (cascada, sistemas formales, evolutivo, etc.) depende del riesgo identificado para esa fase.
4. **Planificación**: Se determina si continuar con otro ciclo. Se planea la siguiente fase del proyecto.

Este modelo a diferencia de los otros toma en consideración explícitamente el riesgo, esta es una actividad importante en la administración del proyecto.

El ciclo de vida inicia con la definición de los objetivos. De acuerdo a las restricciones se determinan distintas alternativas. Se identifican los riesgos al sopesar los objetivos contra las alternativas. Se evalúan los riesgos con actividades como análisis detallado, simulación, prototipos, etc. Se desarrolla un poco el sistema. Se planifica la siguiente fase.

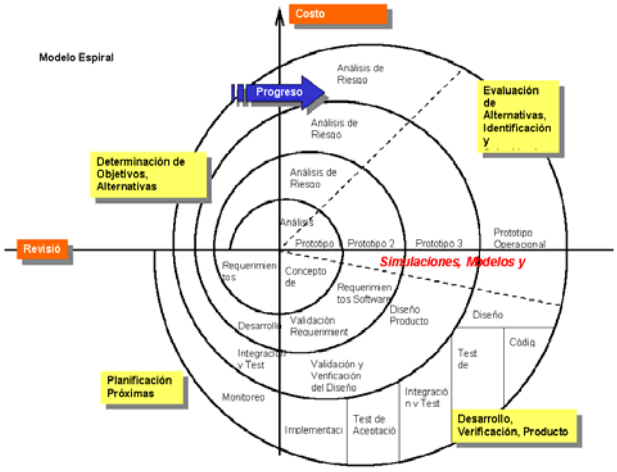
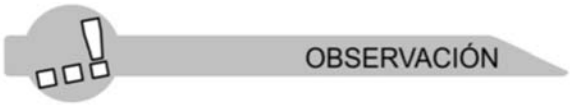


Figura 2.11: *Modelo de desarrollo en Espiral*



Existen innumerables herramientas CASE que pueden ser de ayuda en todo el proceso de desarrollo de software. Sin embargo es difícil encontrar una única herramienta CASE que pueda acompañarnos en todo el proceso

1. Elabore un cuadro comparativo de los diferentes modelos de ciclo de vida tomando en cuenta los siguientes criterios:
   - Funciona con requisitos y arquitectura no predefinidos
   - Produce software altamente fiable
   - Permite la gestión de riesgos
   - Permite correcciones sobre la marcha
   - Visión del progreso por el Cliente y el Jefe del proyecto

2. Se supone que se va desarrollar una aplicación relativa a la gestión de pedidos de una empresa. En este caso el cliente no tiene todavía muy claro qué es lo que quiere. Además, el personal informático va a utilizar una tecnología que le resulta completamente nueva. Discútase qué tipo de ciclo de vida es más apropiado y qué procesos se deberían utilizar para desarrollar esta aplicación.

Se ha visto las fases por las que pasa un producto de software a lo largo de su "vida" y la forma en que se relacionan dichas fases.

El desarrollo de software va unido a un ciclo de vida compuesto por una serie de etapas que comprenden todas las actividades, desde el momento en que surge la idea de crear un nuevo producto software, hasta aquel en que el producto deja definitivamente de ser utilizado por el último de sus usuarios.

[1] PRESSMAN, Roger S. **Ingeniería del Software. Un enfoque práctico.** Mc Graw Hill, Ed. Interamericana de España S.A.U, 2006

[2] LETELIER, Patricio., Proyecto Docente e Investigador, DSIC, 2003

[3] SOMMERVILLE, I., **Ingeniería de Software**, Ed. Pearson Educación, 2002

[4] JACABOSON, I., BOOCH, G., RUMBAUGH J., **El Proceso Unificado de Desarrollo de Software**, Ed. Addison Wesley 2000.

[5] BOEHM, B. W., **A Spiral Model of Software Develpment and Enhancement**, IEEE Computer, 1988.

[6] MILLS, H., O´NEILL, D., **The Management of Software Engineering**, IBM Systems, 1980.

[7] ROYCE, W., **Managing the development of large software systems: concepts and technique**, IEEE Westcon, 1970.

Bibliografía electrónica:

- Guía sobre el ciclo de vida del Software
  http://www.tectimes.com/lbr/Graphs/revistas/lpcu097/capitulogratis.pdf

- Fundamentos de Ingeniería del Software
  http://dis.um.es/~jnicolas/09BK_FIS.html

En la siguiente unidad temática detallaremos aspectos relacionados a las metodologías que facilitan el proceso de desarrollo de software

INGENIERIA DEL SOFTWARE
UNIDAD ACADÉMICA Nº 2

NOMBRE:_____

APELLIDOS:_____FECHA; ____/_____/_____

CIUDAD:_____SEMESTRE:_____

PARTE I: Marque la alternativa correcta:

1. El ciclo de vida:
   a) Comienza con una idea o necesidad que satisfacer y acaba con las pruebas satisfactorias del producto.
   b) No existe ningún estándar que describa sus procesos y actividades.
   c) No se trata sólo de realizar el análisis, diseño, codificación y pruebas; también incluye, entre otros, procesos de soporte.
   d) El mantenimiento lo constituyen las actividades para mantener sin cambios el sistema.
   e) En la actividad de análisis de los requisitos software los desarrolladores obtienen de los futuros usuarios los requisitos que piden al sistema

2. Cuando el cliente conoce los objetivos generales para el software, pero no identifica los requisitos detallados de entrada, procesamiento o salida, es recomendable utilizar el siguiente modelo de ciclo de vida:
   a) Modelo evolutivo
   b) Code & Fix
   c) Desarrollo en cascada
   d) Modelo incremental
   e) Modelo en espiral

PARTE II: Responda brevemente:

3. ¿Cree Ud. que seguir un modelo de ciclo de vida nos garantiza el éxito del desarrollo? ¿Por que?
   _____
   _____
   _____
   _____

4. ¿Qué factores influyen a la hora de elegir un ciclo de vida para resolver un problema dado?
   _____
   _____
   _____
   _____

5. ¿Qué ciclo de vida elegiría para resolver un problema que se comprende bien desde el principio y está muy estructurado?
   _____
   _____
   _____
   _____

6. Una vez elegido el ciclo de vida, ¿qué procesos escogería para dicho ciclo de vida, teniendo en cuenta que el desarrollo informático para resolver el problema anterior lo realiza una única persona?

_____
_____
_____
_____

7. La generación de programas prototipo es exclusiva de un modelo de ciclo de vida? Fundamente su respuesta

_____
_____
_____
_____

Unidad Académica III

METODOLOGÍAS PARA DESARROLLO DE SOFTWARE

Las metodologías de desarrollo de software son un conjunto de procedimientos, técnicas y ayudas a la documentación para el desarrollo de productos software.

Es como un libro de recetas de cocina, en el que se van indicando paso a paso todas las actividades a realizar para lograr el producto informático deseado, indicando además qué personas deben participar en el desarrollo de las actividades y qué papel deben de tener. Además detallan la información que se debe producir como resultado de una actividad y la información necesaria para comenzarla.

Al finalizar el estudio de la presente unidad temática el estudiante:

9. Entiende la importancia del uso de una metodología en el proceso de desarrollo de software
10. Conoce la clasificación de metodologías de desarrollo de software
11. Elige la metodología de desarrollo mas adecuada de acuerdo al proyecto que tenga en manos
12. Conoce a detalle mas metodologías de desarrollo más usadas en la actualidad

3.1 Categorías de Metodologías de Desarrollo de Software

a) Metodologías estructuradas

Comenzaron a desarrollarse a fines de los 70's con la Programación Estructurada, luego a mediados de los 70's aparecieron técnicas para el Diseño (por ejemplo: el Diagrama de Estructura) primero y posteriormente para el Análisis (por ejemplo: Diagramas de Flujo de Datos). Estas metodologías son particularmente apropiadas en proyectos que utilizan para la implementación lenguajes de 3ra y 4ta generación.

Ejemplos de metodologías estructuradas de ámbito gubernamental: MERISE[10] (Francia), MÉTRICA[11](España), SSADM[12] (Reino Unido). Ejemplos de propuestas de metodologías estructuradas en el ámbito académico: Gane & Sarson[13], Ward & Mellor[14], Yourdon & DeMarco[15] e Information Engineering[16].

b) Metodologías orientadas a objetos

Su historia va unida a la evolución de los lenguajes de programación orientados a objetos, los más representativos: a fines de los 60's SIMULA, a fines de los 70's Smalltalk-80, la primera versión de C++ por Bjarne Stroustrup en 1981 y actualmente Java o C# de Microsoft. A fines de los 80's comenzaron a consolidarse algunas metodologías Orientadas a Objetos.

[10] http://perso.club-internet.fr/brouardf/SGBDRmerise.htm
[11] http://www.map.es/csi/metrica3/
[12] http://www.comp.glam.ac.uk/pages/staff/tdhutchings/chapter4.html
[13] http://portal.newman.wa.edu.au/technology/12infsys/html/dfdnotes.doc
[14] http://www.yourdon.com/books/coolbooks/notes/wardmellor.html
[15] http://wombat.doc.ic.ac.uk/foldoc/foldoc.cgi?Yourdon%2FDemarco
[16] http://gantthead.com/Gantthead/process/processMain/1,1289,2-12009-2,00.html

En 1995 Booch y Rumbaugh proponen el Método Unificado con la ambiciosa idea de conseguir una unificación de sus métodos y notaciones, que posteriormente se reorienta a un objetivo más modesto, para dar lugar al Unified Modeling Language (UML)[17], la notación OO más popular en la actualidad.
Algunas metodologías OO con notaciones predecesoras de UML son: OOAD (Booch), OOSE (Jacobson), Coad & Yourdon, Shaler & Mellor y OMT (Rumbaugh).
Algunas metodologías orientadas a objetos que utilizan la notación UML son: Rational Unified Process (RUP)[18], OPEN[19], MÉTRICA (que también soporta la notación estructurada).

c) Metodologías tradicionales (no ágiles)

Las metodologías no ágiles son aquellas que están guiadas por una fuerte planificación durante todo el proceso de desarrollo; llamadas también metodologías tradicionales o clásicas, donde se realiza una intensa etapa de análisis y diseño antes de la construcción del sistema.
Todas las propuestas metodológicas antes indicadas pueden considerarse como metodologías tradicionales. Aunque en el caso particular de RUP, por el especial énfasis que presenta en cuanto a su adaptación a las condiciones del proyecto (mediante su configuración previa a aplicarse), realizando una configuración adecuada, podría considerarse Ágil.

d) Metodologías ágiles

Un proceso es ágil cuando el desarrollo de software es incremental (entregas pequeñas de software, con ciclos rápidos), cooperativo (cliente y desarrolladores trabajan juntos constantemente con una cercana comunicación), sencillo (el método en sí mismo es fácil de aprender y modificar, bien documentado), y adaptable (permite realizar cambios de último momento)
Algunas metodologías ágiles son: Extreme Programming, Scrum, Familia de Metodologías Cristal, Feature Driven Development, Proceso Unificado Rational, una configuración ágil, Dynamic Systems Development Method, Adaptive Software Development, Open Source Software Development.

3.2 Conociendo algunas Metodologías…

a) Métrica V3

La metodología MÉTRICA Versión 3 ofrece a las Organizaciones un instrumento útil para la sistematización de las actividades que dan soporte al ciclo de vida del software dentro del marco que permite alcanzar los siguientes objetivos:

- Proporcionar o definir Sistemas de Información que ayuden a conseguir los fines de la Organización mediante la definición de un marco estratégico para el desarrollo de los mismos.
- Dotar a la Organización de productos software que satisfagan las necesidades de los usuarios dando una mayor importancia al análisis de requisitos.
- Mejorar la productividad de los departamentos de Sistemas y Tecnologías de la Información y las Comunicaciones, permitiendo una mayor capacidad de adaptación a los cambios y teniendo en cuenta la reutilización en la medida de lo posible.
- Facilitar la comunicación y entendimiento entre los distintos participantes en la producción de software a lo largo del ciclo de vida del proyecto, teniendo en cuenta su papel y responsabilidad, así como las necesidades de todos y cada uno de ellos.
- Facilitar la operación, mantenimiento y uso de los productos software obtenidos.

[17] http://www.uml.org/
[18] http://www.rational.com/products/rup/index.jsp
[19] http://www.open.org.au/

Procesos Principales De Métrica V3

MÉTRICA Versión 3 tiene un enfoque orientado al proceso, ya que la tendencia general en los estándares se encamina en este sentido y por ello se ha enmarcado dentro de la norma ISO 12.207, que se centra en la clasificación y definición de los procesos del ciclo de vida del software. Como punto de partida y atendiendo a dicha norma, MÉTRICA Versión 3 cubre el Proceso de Desarrollo y el Proceso de Mantenimiento de Sistemas de Información.
MÉTRICA Versión 3 ha sido concebida para abarcar el desarrollo completo de Sistemas de Información sea cual sea su complejidad y magnitud
La metodología descompone cada uno de los procesos en actividades, y éstas a su vez en tareas. Para cada tarea se describe su contenido haciendo referencia a sus principales acciones, productos, técnicas, prácticas y participantes.
Así los procesos de la estructura principal de MÉTRICA Versión 3 son los siguientes:

Planificación de Sistemas de Información (PSI)
El objetivo de un Plan de Sistemas de Información es proporcionar un marco estratégico de referencia para los Sistemas de Información de un determinado ámbito de la Organización.
El resultado del Plan de Sistemas debe, por tanto, orientar las actuaciones en materia de desarrollo de Sistemas de Información con el objetivo básico de apoyar la estrategia corporativa, elaborando una arquitectura de información y un plan de proyectos informáticos para dar apoyo a los objetivos estratégicos.
Por este motivo es necesario un proceso como el de Planificación de Sistemas de Información, en el que participen, por un lado los responsables de los procesos de la organización con una visión estratégica y por otro, los profesionales de SI capaces de enriquecer dicha visión con la aportación de ventajas competitivas por medio de los sistemas y tecnologías de la información y comunicaciones.
Como productos finales de este proceso se obtienen los siguientes, que podrán constituir la entrada para el siguiente proceso de Estudio de Viabilidad del Sistema:

a. **Catálogo de requisitos de PSI** que surge del estudio de la situación actual en el caso de que sea significativo dicho estudio, del diagnóstico que se haya llevado a cabo y de las necesidades de información de los procesos de la organización afectados por el plan de sistemas.
b. **Arquitectura de información** que se compone a su vez de los siguientes productos:
    - Modelo de información.
    - Modelo de sistemas de información.
    - Arquitectura tecnológica.
    - Plan de proyectos.
    - Plan de mantenimiento del PSI.

Un Plan de Sistemas de Información proporcionará un marco de referencia en materia de Sistemas de Información. En ocasiones podrá servir de palanca de cambio para los procesos de la Organización, pero su objetivo estará siempre diferenciado del de un análisis de dichos procesos por sí mismos. Dicho en otras palabras, no se debe confundir el resultado que se persigue con un Plan de Sistemas de Información, con el de una mejora o reingeniería de procesos, ya que los objetivos en ambos casos no son los mismos, aunque el medio para conseguirlos tenga puntos en común (estudio de los procesos y alineamiento con los objetivos estratégicos).

Desarrollo de Sistemas de Información
El proceso de Desarrollo de MÉTRICA Versión 3 contiene todas las actividades y tareas que se deben llevar a cabo para desarrollar un sistema, cubriendo desde el análisis de requisitos hasta la instalación del software. Además de las tareas relativas al análisis, incluye dos partes en el diseño de sistemas: arquitectónico y detallado. También cubre las pruebas unitarias y de integración del sistema, aunque siguiendo la norma ISO 12.207 no propone ninguna técnica específica y destaca la importancia de la evolución de los requisitos. Este proceso es, sin duda, el más importante de los identificados en el ciclo de vida de un sistema y se relaciona con todos los demás.

Las actividades y tareas propuestas por la norma se encuentran más en la línea de un desarrollo clásico, separando datos y procesos, que en la de un enfoque orientado a objetos.
En MÉTRICA Versión 3 se han abordado los dos tipos de desarrollo: estructurado y orientado a objeto, por lo que ha sido necesario establecer actividades específicas a realizar en alguno de los procesos cuando se utiliza la tecnología de orientación a objetos. Para este último caso se ha analizado alguna de las propuestas de otras metodologías orientadas a objetos y se han tenido en cuenta la mayoría de las técnicas que contempla UML 1.2 (Unified Modeling Language).
El desarrollo en MÉTRICA Versión 3 lo constituyen los procesos:

Estudio de Viabilidad del Sistema (EVS)
El propósito de este proceso es analizar un conjunto concreto de necesidades, con la idea de proponer una solución a corto plazo. Los criterios con los que se hace esta propuesta no serán estratégicos sino tácticos y relacionados con aspectos económicos, técnicos, legales y operativos.
Los resultados del Estudio de Viabilidad del Sistema constituirán la base para tomar la decisión de seguir adelante o abandonar.
.

Análisis del Sistema de Información (ASI)
El propósito de este proceso es conseguir la especificación detallada del sistema de información, a través de un catálogo de requisitos y una serie de modelos que cubran las necesidades de información de los usuarios para los que se desarrollará el sistema de información y que serán la entrada para el proceso de Diseño del Sistema de Información.
.

Diseño del Sistema de Información (DSI)
El propósito del Diseño del Sistema de Información (DSI) es obtener la definición de la arquitectura del sistema y del entorno tecnológico que le va a dar soporte, junto con la especificación detallada de los componentes del sistema de información. A partir de dicha información, se generan todas las especificaciones de construcción relativas al propio sistema, así como la especificación técnica del plan de pruebas, la definición de los requisitos de implantación y el diseño de los procedimientos de migración y carga inicial.
El diseño de la arquitectura del sistema dependerá en gran medida de las características de la instalación, de modo que se ha de tener en cuenta una participación activa de los responsables de Sistemas y Explotación de las Organizaciones para las que se desarrolla el sistema de información.

Construcción del Sistema de Información (CSI)
La construcción del Sistema de Información (CSI) tiene como objetivo final la construcción y prueba de los distintos componentes del sistema de información, a partir del conjunto de especificaciones lógicas y físicas del mismo, obtenido en el Proceso de Diseño del Sistema de Información (DSI). Se desarrollan los procedimientos de operación y seguridad y se elaboran los manuales de usuario final y de explotación.

Implantación y Aceptación del Sistema (IAS)
Este proceso tiene como objetivo principal, la entrega y aceptación del sistema en su totalidad, que puede comprender varios sistemas de información desarrollados de manera independiente, según se haya establecido en el proceso de Estudio de Viabilidad del Sistema (EVS), y un segundo objetivo que es llevar a cabo las actividades oportunas para el paso a producción del sistema.

Se establece el plan de implantación, una vez revisada la estrategia de implantación y se detalla el equipo que lo realizará.

Mantenimiento de Sistemas de Información (MSI)

El objetivo de este proceso es la obtención de una nueva versión de un sistema de información desarrollado con MÉTRICA, a partir de las peticiones de mantenimiento que los usuarios realizan con motivo de un problema detectado en el sistema o por la necesidad de una mejora del mismo.
Como consecuencia de esto, sólo se considerarán en MÉTRICA Versión 3 los tipos de Mantenimiento Correctivo y Evolutivo.

Interfaces De Métrica Versión 3

La estructura de MÉTRICA Versión 3 incluye también un conjunto de interfaces que definen una serie de actividades de tipo organizativo o de soporte al proceso de desarrollo y a los productos, que en el caso de existir en la organización se deberán aplicar para enriquecer o influir en la ejecución de las actividades de los procesos principales de la metodología y que si no existen habrá que realizar para complementar y garantizar el éxito del proyecto desarrollado con MÉTRICA Versión 3.
La aplicación de MÉTRICA Versión 3 proporciona sistemas con calidad y seguridad, no obstante puede ser necesario en función de las características del sistema un refuerzo especial en estos aspectos, refuerzo que se obtendría aplicando la interfaz.
Las interfaces descritas en la metodología son:

Gestión de Proyectos
La Gestión de Proyectos tiene como finalidad principal la planificación, el seguimiento y control de las actividades y de los recursos humanos y materiales que intervienen en el desarrollo de un Sistema de Información. Como consecuencia de este control es posible conocer en todo momento qué problemas se producen y resolverlos o paliarlos lo más pronto posible, lo cual evitará desviaciones temporales y económicas.

Seguridad
El análisis de los riesgos constituye una pieza fundamental en el diseño y desarrollo de sistemas de información seguros. Si bien los riesgos que afectan a un sistema de información son de distinta índole: naturales (inundaciones, incendios, etc.) o lógicos (fallos propios, ataques externos, virus, etc.) son estos últimos los contemplados en la interfaz de Seguridad de MÉTRICA Versión 3.
La interfaz de Seguridad hace posible incorporar durante la fase de desarrollo las funciones y mecanismos que refuerzan la seguridad del nuevo sistema y del propio proceso de desarrollo, asegurando su consistencia y seguridad, completando el plan de seguridad vigente en la organización o desarrollándolo desde el principio, utilizando MAGERIT[20] como metodología de análisis y gestión de riesgos en el caso de que la organización no disponga de su propia metodología.

Gestión de la Configuración
La interfaz de gestión de la configuración consiste en la aplicación de procedimientos administrativos y técnicos durante el desarrollo del sistema de información y su posterior mantenimiento. Su finalidad es identificar, definir, proporcionar información y controlar los cambios en la configuración del sistema, así como las modificaciones y versiones de los mismos. Este proceso permitirá conocer el estado de cada uno de los productos que se hayan definido como elementos de configuración, garantizando que no se realizan cambios incontrolados y que todos los participantes en el desarrollo del sistema disponen de la versión adecuada de los productos que manejan.

Aseguramiento de la Calidad
El objetivo de la interfaz de Aseguramiento de la Calidad de MÉTRICA Versión 3 es proporcionar un marco común de referencia para la definición y puesta en marcha de planes específicos de aseguramiento de calidad aplicables a proyectos concretos.

[20] Metodología de Análisis y Gestión de Riesgos de los Sistemas de Información (http://www.csi.map.es/csi/pg5m20.htm)

b) Programación Extrema (eXtreme Programming, XP)

XP[21] es una metodología ágil centrada en potenciar las relaciones interpersonales como clave para el éxito en desarrollo de software, promoviendo el trabajo en equipo, preocupándose por el aprendizaje de los desarrolladores, y propiciando un buen clima de trabajo. XP se basa en realimentación continua entre el cliente y el equipo de desarrollo, comunicación fluida entre todos los participantes, simplicidad en las soluciones implementadas y coraje para enfrentar los cambios. XP es especialmente adecuada para proyectos con requisitos imprecisos y muy cambiantes, y donde existe un alto riesgo técnico.

Características esenciales de XP:

Las Historias de Usuario

Las historias de usuario son la técnica utilizada en XP para especificar los requisitos del software. Se trata de tarjetas de papel en las cuales el cliente describe brevemente las características que el sistema debe poseer, sean requisitos funcionales o no funcionales. El tratamiento de las historias de usuario es muy dinámico y flexible, en cualquier momento historias de usuario pueden romperse, reemplazarse por otras más específicas o generales, añadirse nuevas o ser modificadas. Cada historia de usuario es lo suficientemente comprensible y delimitada para que los programadores puedan implementarla en unas semanas.

Roles XP

Aunque en otras fuentes de información aparecen algunas variaciones y extensiones de roles XP, en este apartado describiremos los roles de acuerdo con la propuesta original de Beck.

Programador

El programador escribe las pruebas unitarias y produce el código del sistema.

Cliente

El cliente escribe las historias de usuario y las pruebas funcionales para validar su implementación. Además, asigna la prioridad a las historias de usuario y decide cuáles se implementan en cada iteración centrándose en aportar mayor valor al negocio.

Encargado de pruebas (Tester)

El encargado de pruebas ayuda al cliente a escribir las pruebas funcionales. Ejecuta las pruebas regularmente, difunde los resultados en el equipo y es responsable de las herramientas de soporte para pruebas.

Encargado de seguimiento (Tracker)

El encargado de seguimiento proporciona realimentación al equipo en el proceso XP. Su responsabilidad es verificar el grado de acierto entre las estimaciones realizadas y el tiempo real dedicado, comunicando los resultados para mejorar futuras estimaciones. También realiza el seguimiento del progreso de cada iteración y evalúa si los objetivos son alcanzables con las restricciones de tiempo y recursos presentes.

Entrenador (Coach)

Es responsable del proceso global. Es necesario que conozca a fondo el proceso XP para proveer guías a los miembros del equipo de forma que se apliquen las prácticas XP y se siga el proceso correctamente.

Consultor

Es un miembro externo del equipo con un conocimiento específico en algún tema necesario para el proyecto. Guía al equipo para resolver un problema específico.

Gestor (Big boss)

Es el vínculo entre clientes y programadores, ayuda a que el equipo trabaje efectivamente creando las condiciones adecuadas. Su labor esencial es de coordinación.

Proceso XP

[21] www6..extremeprogramming.org, www.xprogramming.com, c2.com/cgi/wiki?ExtremeProgramming

Un proyecto XP tiene éxito cuando el cliente selecciona el valor de negocio a implementar basado en la habilidad del equipo para medir la funcionalidad que puede entregar a través del tiempo. El ciclo de desarrollo consiste (a grandes rasgos) en los siguientes pasos:

a. El cliente define el valor de negocio a implementar.
b. El programador estima el esfuerzo necesario para su implementación.
c. El cliente selecciona qué construir, de acuerdo con sus prioridades y las restricciones de tiempo.
d. El programador construye ese valor de negocio.
e. Vuelve al paso 1.

En todas las iteraciones de este ciclo tanto el cliente como el programador aprenden. No se debe presionar al programador a realizar más trabajo que el estimado, ya que se perderá calidad en el software o no se cumplirán los plazos. De la misma forma el cliente tiene la obligación de manejar el ámbito de entrega del producto, para asegurarse que el sistema tenga el mayor valor de negocio posible con cada iteración.
El ciclo de vida ideal de XP consiste de seis fases [2]: Exploración, Planificación de la Entrega (Release), Iteraciones, Producción, Mantenimiento y Muerte del Proyecto.

Fase I: Exploración

En esta fase, los clientes plantean a grandes rasgos las historias de usuario que son de interés para la primera entrega del producto. Al mismo tiempo el equipo de desarrollo se familiariza con las herramientas, tecnologías y prácticas que se utilizarán en el proyecto. Se prueba la tecnología y se exploran las posibilidades de la arquitectura del sistema construyendo un prototipo.

Fase II: Planificación de la Entrega

En esta fase el cliente establece la prioridad de cada historia de usuario, y correspondientemente, los programadores realizan una estimación del esfuerzo necesario de cada una de ellas. Se toman acuerdos sobre el contenido de la primera entrega y se determina un cronograma en conjunto con el cliente. Una entrega debería obtenerse en no más de tres meses.

Fase III: Iteraciones

Esta fase incluye varias iteraciones sobre el sistema antes de ser entregado. El Plan de Entrega está compuesto por iteraciones de no más de tres semanas. En la primera iteración se puede intentar establecer una arquitectura del sistema que pueda ser utilizada durante el resto del proyecto. Esto se logra escogiendo las historias que fuercen la creación de esta arquitectura, sin embargo, esto no siempre es posible ya que es el cliente quien decide qué historias se implementarán en cada iteración (para maximizar el valor de negocio). Al final de la última iteración el sistema estará listo para entrar en producción.

Fase IV: Producción

La fase de producción requiere de pruebas adicionales y revisiones de rendimiento antes de que el sistema sea trasladado al entorno del cliente. Al mismo tiempo, se deben tomar decisiones sobre la inclusión de nuevas características a la versión actual, debido a cambios durante esta fase.

Fase V: Mantenimiento

Mientras la primera versión se encuentra en producción, el proyecto XP debe mantener el sistema en funcionamiento al mismo tiempo que desarrolla nuevas iteraciones. Para realizar esto se requiere de tareas de soporte para el cliente. De esta forma, la velocidad de desarrollo puede bajar después de la puesta del sistema en producción.

Fase VI: Muerte del Proyecto

Es cuando el cliente no tiene más historias para ser incluidas en el sistema. Esto requiere que se satisfagan las necesidades del cliente en otros aspectos como rendimiento y confiabilidad del sistema. Se genera la documentación final del sistema y no se realizan más cambios en la arquitectura. La muerte del proyecto también ocurre cuando el sistema no genera los beneficios esperados por el cliente o cuando no hay presupuesto para mantenerlo.

Prácticas XP

La principal suposición que se realiza en XP es la posibilidad de disminuir la mítica curva exponencial del costo del cambio a lo largo del proyecto, lo suficiente para que el diseño

evolutivo funcione. XP apuesta por un crecimiento lento del costo del cambio y con un comportamiento asintótico. Esto se consigue gracias a las tecnologías disponibles para ayudar en el desarrollo de software y a la aplicación disciplinada de las prácticas que describiremos a continuación.

El juego de la planificación
Es un espacio frecuente de comunicación entre el cliente y los programadores. El equipo técnico realiza una estimación del esfuerzo requerido para la implementación de las historias de usuario y los clientes deciden sobre el ámbito y tiempo de las entregas y de cada iteración. Esta práctica se puede ilustrar como un juego, donde existen dos tipos de jugadores: Cliente y Programador. El cliente establece la prioridad de cada historia de usuario, de acuerdo con el valor que aporta para el negocio. Los programadores estiman el esfuerzo asociado a cada historia de usuario.

Entregas pequeñas
La idea es producir rápidamente versiones del sistema que sean operativas, aunque obviamente no cuenten con toda la funcionalidad pretendida para el sistema pero si que constituyan un resultado de valor para el negocio. Una entrega no debería tardar más de 3 meses.

Metáfora
En XP no se enfatiza la definición temprana de una arquitectura estable para el sistema. Dicha arquitectura se asume evolutiva y los posibles inconvenientes que se generarían por no contar con ella explícitamente en el comienzo del proyecto se solventan con la existencia de una metáfora. El sistema es definido mediante una metáfora o un conjunto de metáforas compartidas por el cliente y el equipo de desarrollo. Una metáfora es una historia compartida que describe cómo debería funcionar el sistema.

Diseño simple
Se debe diseñar la solución más simple que pueda funcionar y ser implementada en un momento determinado del proyecto. La complejidad innecesaria y el código extra debe ser removido inmediatamente.

Pruebas
La producción de código está dirigida por las pruebas unitarias. Las pruebas unitarias son establecidas antes de escribir el código y son ejecutadas constantemente ante cada modificación del sistema. Los clientes escriben las pruebas funcionales para cada historia de usuario que deba validarse.

Refactorización (Refactoring)
La refactorización es una actividad constante de reestructuración del código con el objetivo de remover duplicación de código, mejorar su legibilidad, simplificarlo y hacerlo más flexible para facilitar los posteriores cambios.

Programación en parejas
Toda la producción de código debe realizarse con trabajo en parejas de programadores. Según Cockburn y Williams [2], los problemas de programación se resuelven más rápido, se posibilita la transferencia de conocimientos de programación entre los miembros del equipo, varias personas entienden las diferentes partes sistema, los programadores conversan mejorando así el flujo de información y la dinámica del equipo, y finalmente, los programadores disfrutan más su trabajo.

Propiedad colectiva del código
Cualquier programador puede cambiar cualquier parte del código en cualquier momento. Esta práctica motiva a todos a contribuir con nuevas ideas en todos los segmentos del sistema, evitando a la vez que algún programador sea imprescindible para realizar cambios en alguna porción de código.

Integración continua
Cada pieza de código es integrada en el sistema una vez que esté lista. Así, el sistema puede llegar a ser integrado y construido varias veces en un mismo día. Todas las pruebas son ejecutadas y tienen que ser aprobadas para que el nuevo código sea incorporado definitivamente..

40 horas por semana
Se debe trabajar un máximo de 40 horas por semana. No se trabajan horas extras en dos semanas seguidas. El trabajo extra desmotiva al equipo. Los proyectos que

requieren trabajo extra para intentar cumplir con los plazos suelen al final ser entregados con retraso. En lugar de esto se puede realizar el juego de la planificación para cambiar el ámbito del proyecto o la fecha de entrega.

Cliente in-situ

El cliente tiene que estar presente y disponible todo el tiempo para el equipo. Gran parte del éxito del proyecto XP se debe a que es el cliente quien conduce constantemente el trabajo hacia lo que aportará mayor valor de negocio y los programadores pueden resolver de manera inmediata cualquier duda asociada.

Estándares de programación

XP enfatiza la comunicación de los programadores a través del código, con lo cual es indispensable que se sigan ciertos estándares de programación (del equipo, de la organización u otros estándares reconocidos para los lenguajes de programación utilizados). Los estándares de programación mantienen el código legible para los miembros del equipo, facilitando los cambios.

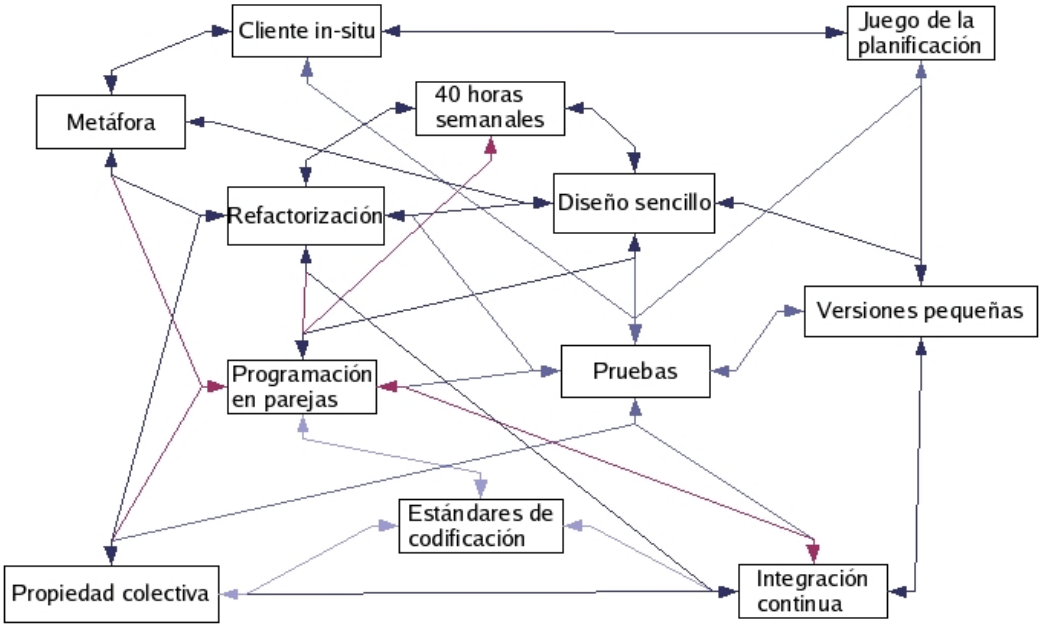


***Figura 3.1**. Las prácticas se refuerzan entre sí*

El mayor beneficio de las **prácticas** se consigue con su aplicación conjunta y equilibrada puesto que se apoyan unas en otras. Esto se ilustra en la Figura 1 de [1], donde una línea entre dos prácticas significa que las dos prácticas se refuerzan entre sí.

c) Rational Unified Process (RUP)

RUP es un producto comercial desarrollado y comercializado por Rational Software, una compañía de IBM.

Historia

La Figura 3.2 ilustra la historia de RUP. El antecedente más importante se ubica en 1967 con la Metodología Ericsson (Ericsson Approach) elaborada por Ivar Jacobson, una aproximación de desarrollo basada en componentes, que introdujo el concepto de Caso de Uso. Entre los años de 1987 a 1995 Jacobson fundó la compañía Objectory AB y lanza el proceso de desarrollo Objectory (abreviación de Object Factory).

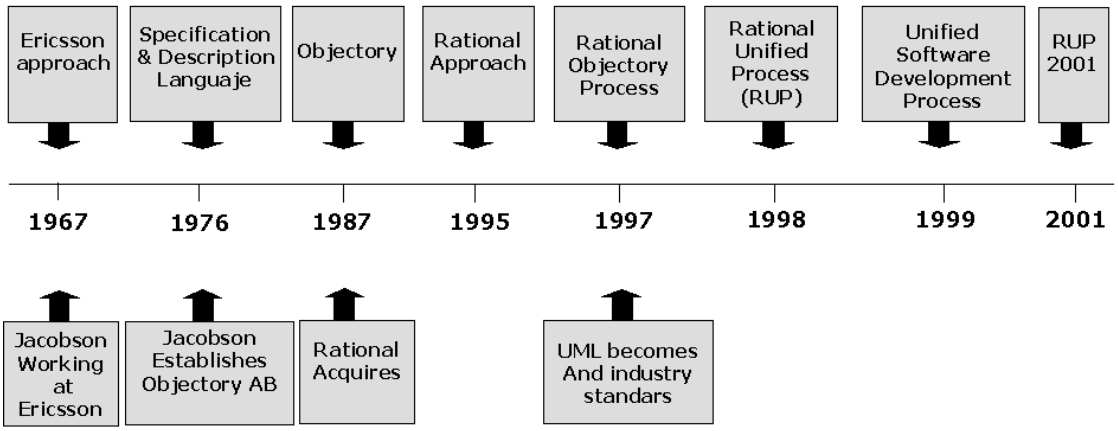


***Figura** 3.2: Historia de RUP*

Posteriormente en 1995 Rational Software Corporation adquiere Objectory AB y entre 1995 y 1997 se desarrolla Rational Objectory Process (ROP) a partir de Objectory 3.8 y del Enfoque Rational (Rational Approach) adoptando UML como lenguaje de modelado.

Desde ese entonces y a la cabeza de Grady Booch, Ivar Jacobson y James Rumbaugh, Rational Software desarrolló e incorporó diversos elementos para expandir ROP, destacándose especialmente el flujo de trabajo conocido como modelado del negocio. En junio del 1998 se lanza Rational Unified Process.

Características esenciales

-

Proceso dirigido por Casos de Uso

Los Casos de Uso son una técnica de captura de requisitos que fuerza a pensar en términos de importancia para el usuario y no sólo en términos de funciones que seria bueno contemplar. Se define un Caso de Uso como un fragmento de funcionalidad del sistema que proporciona al usuario un valor añadido. Los Casos de Uso representan los requisitos funcionales del sistema.

En RUP los Casos de Uso no son sólo una herramienta para especificar los requisitos del sistema. También guían su diseño, implementación y prueba. Los Casos de Uso constituyen un elemento integrador y una guía del trabajo como se muestra en la Figura 2.

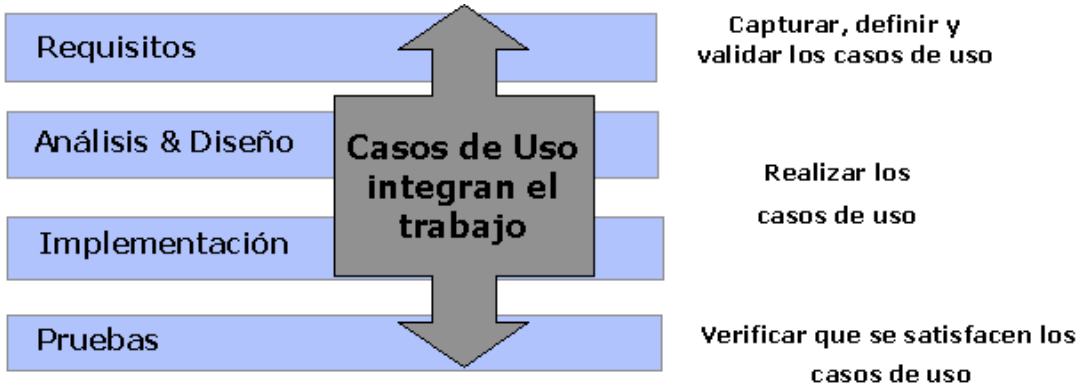


***Figura 3.3**: Los Casos de Uso integran el trabajo*

Los Casos de Uso no sólo inician el proceso de desarrollo sino que proporcionan un hilo conductor, permitiendo establecer trazabilidad entre los artefactos que son generados en las diferentes actividades del proceso de desarrollo.

Como se muestra en la Figura 3, basándose en los Casos de Uso se crean los modelos de análisis y diseño, luego la implementación que los lleva a cabo, y se verifica que efectivamente el producto implemente adecuadamente cada Caso de Uso. Todos los modelos deben estar sincronizados con el modelo de Casos de Uso.

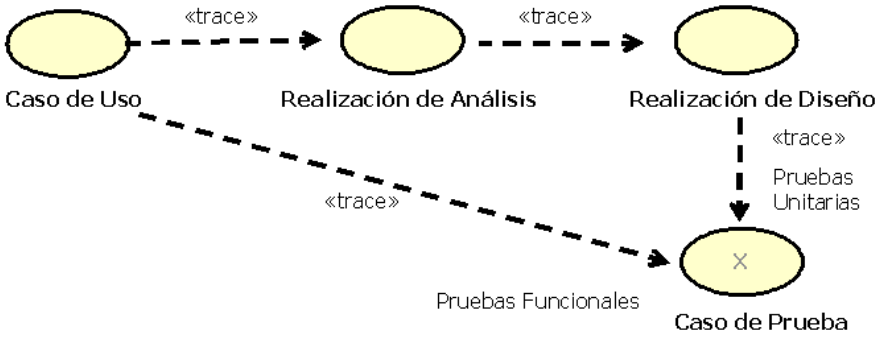


***Figura 3.4**: Trazabilidad a partir de los Casos de Uso*

Proceso centrado en la arquitectura

La arquitectura de un sistema es la organización o estructura de sus partes más relevantes, lo que permite tener una visión común entre todos los involucrados (desarrolladores y usuarios) y una perspectiva clara del sistema completo, necesaria para controlar el desarrollo

La arquitectura involucra los aspectos estáticos y dinámicos más significativos del sistema, está relacionada con la toma de decisiones que indican cómo tiene que ser construido el sistema y ayuda a determinar en qué orden. Además la definición de la arquitectura debe tomar en consideración elementos de calidad del sistema, rendimiento, reutilización y capacidad de evolución por lo que debe ser flexible durante todo el proceso de desarrollo. La arquitectura se ve influenciada por la plataforma software, sistema operativo, gestor de bases de datos, protocolos, consideraciones de desarrollo como sistemas heredados. Muchas de estas restricciones constituyen requisitos no funcionales del sistema.

Existe una interacción entre los Casos de Uso y la arquitectura, los Casos de Uso deben encajar en la arquitectura cuando se llevan a cabo y la arquitectura debe permitir el desarrollo de todos los Casos de Uso requeridos, actualmente y en el futuro.

En la Figura 3.5 se ilustra la evolución de la arquitectura durante las fases de RUP. Se tiene una arquitectura más robusta en las fases finales del proyecto. En las fases iniciales lo que se hace es ir consolidando la arquitectura por medio de baselines [22] y se va modificando dependiendo de las necesidades del proyecto.

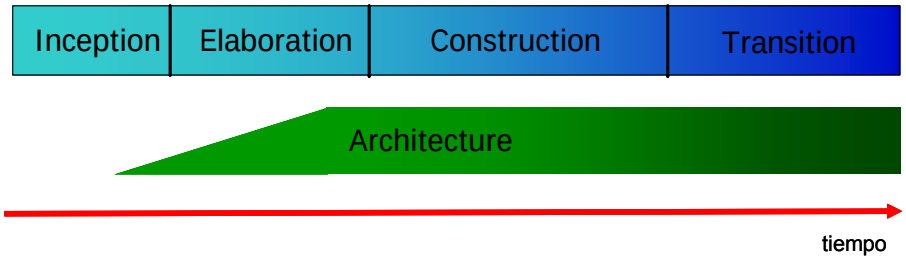


***Figura 3.5:** Evolución de la arquitectura del sistema*

22 Una baseline es una instantánea del estado de todos los artefactos del proyecto, registrada para efectos de gestión de configuración y control de cambios.

Es conveniente ver el sistema desde diferentes perspectivas para comprender mejor el diseño por lo que la arquitectura se representa mediante varias vistas que se centran en aspectos concretos del sistema, abstrayéndose de los demás. Para RUP, todas las vistas juntas forman el llamado modelo 4+1 de la arquitectura, el cual recibe este nombre porque lo forman las vistas lógica, de implementación, de proceso y de despliegue, más la de Casos de Uso que es la que da cohesión a todas.

Proceso iterativo e incremental

El equilibrio correcto entre los Casos de Uso y la arquitectura es algo muy parecido al equilibrio de la forma y la función en el desarrollo del producto, lo cual se consigue con el tiempo. Para esto, la estrategia que se propone en RUP es tener un proceso iterativo e incremental en donde el trabajo se divide en partes más pequeñas o mini proyectos. Permitiendo que el equilibrio entre Casos de Uso y arquitectura se vaya logrando durante cada mini proyecto, así durante todo el proceso de desarrollo. Cada mini proyecto se puede ver como una iteración (un recorrido más o menos completo a lo largo de todos los flujos de trabajo fundamentales) del cual se obtiene un incremento que produce un crecimiento en el producto.

Una iteración puede realizarse por medio de una cascada como se muestra en la Figura 3.6. Se pasa por los flujos fundamentales (Requisitos, Análisis, Diseño, Implementación y Pruebas), también existe una planificación de la iteración, un análisis de la iteración y algunas actividades específicas de la iteración. Al finalizar se realiza una integración de los resultados con lo obtenido de las iteraciones anteriores.

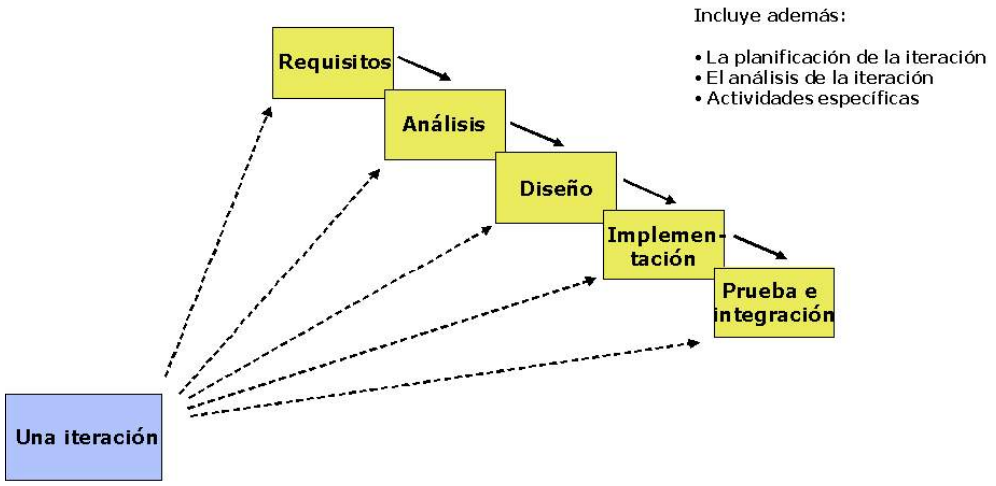


***Figura 3.6**: Una iteración RUP*

Mejores prácticas para RUP

RUP identifica 6 *best practices* con las que define una forma efectiva de trabajar para los equipos de desarrollo de software.

Gestión de requisitos

RUP brinda una guía para encontrar, organizar, documentar, y seguir los cambios de los requisitos funcionales y restricciones. Utiliza una notación de Caso de Uso y escenarios para representar los requisitos.

Desarrollo de software iterativo

Desarrollo del producto mediante iteraciones con hitos bien definidos, en las cuales se repiten las actividades pero con distinto énfasis, según la fase del proyecto.

Desarrollo basado en componentes

La creación de sistemas intensivos en software requiere dividir el sistema en componentes con interfaces bien definidas, que posteriormente serán ensamblados para

generar el sistema. Esta característica en un proceso de desarrollo permite que el sistema se vaya creando a medida que se obtienen o se desarrollan sus componentes.

Modelado visual (usando UML)

UML es un lenguaje para visualizar, especificar, construir y documentar los artefactos de un sistema software. Es un estándar de la OMG. Utilizar herramientas de modelado visual facilita la gestión de dichos modelos, permitiendo ocultar o exponer detalles cuando sea necesario. El modelado visual también ayuda a mantener la consistencia entre los artefactos del sistema: requisitos, diseños e implementaciones.

Verificación continua de la calidad

Es importante que la calidad de todos los artefactos se evalúe en varios puntos durante el proceso de desarrollo, especialmente al final de cada iteración. En esta verificación las pruebas juegan un papel fundamental y se integran a lo largo de todo el proceso. Para todos los artefactos no ejecutables las revisiones e inspecciones también deben ser continuas.

Gestión de los cambios

El cambio es un factor de riesgo crítico en los proyectos de software. Los artefactos software cambian no sólo debido a acciones de mantenimiento posteriores a la entrega del producto, sino que durante el proceso de desarrollo, especialmente importantes por su posible impacto son los cambios en los requisitos.

Estructura del proceso

El proceso puede ser descrito en dos dimensiones o ejes:

Eje horizontal: Representa el tiempo y es considerado el eje de los aspectos dinámicos del proceso. Indica las características del ciclo de vida del proceso expresado en términos de fases, iteraciones e hitos.

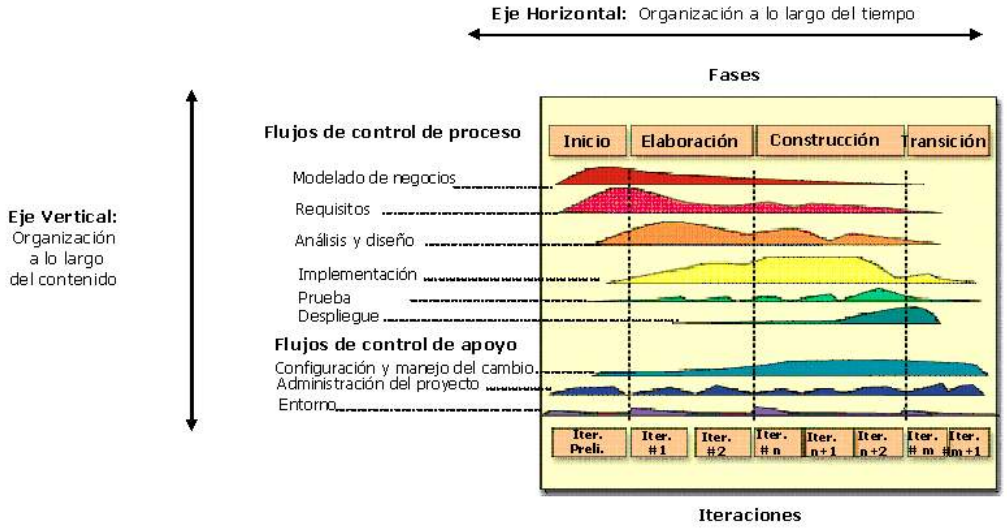


Figura 3.7: *Estructura de RUP*

Eje vertical: Representa los aspectos estáticos del proceso. Describe el proceso en términos de componentes de proceso, disciplinas, flujos de trabajo, actividades, artefactos y roles.

Estructura Dinámica del proceso. Fases e iteraciones

RUP se repite a lo largo de una serie de ciclos que constituyen la vida de un producto. Cada ciclo concluye con una generación del producto para los clientes. Cada ciclo consta de cuatro fases: Inicio, Elaboración, Construcción y Transición. Cada fase se subdivide a la vez en iteraciones, el número de iteraciones en cada fase es variable.

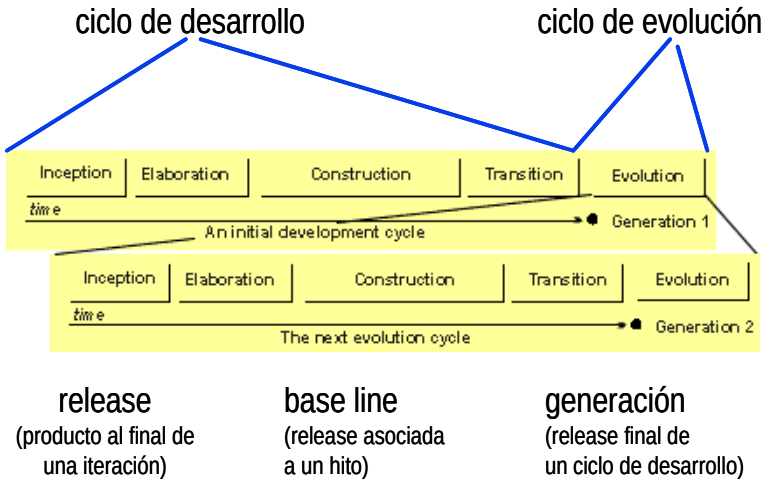


Figura 3.8: *Ciclos, releases, baseline*

La duración y esfuerzo dedicado en cada fase es variable dependiendo de las características del proyecto. Sin embargo, la Figura 3.9 ilustra porcentajes frecuentes al respecto. Consecuente con el esfuerzo señalado, la Figura 3.10 ilustra una distribución típica de recursos humanos necesarios a lo largo del proyecto.

	Inicio	Elaboración	Construcción	Transición
Esfuerzo	**5 %**	**20 %**	**65 %**	**10%**
Tiempo Dedicado	**10 %**	**30 %**	**50 %**	**10%**

Figura 3.9: *Distribución típicas de esfuerzo y tiempo*

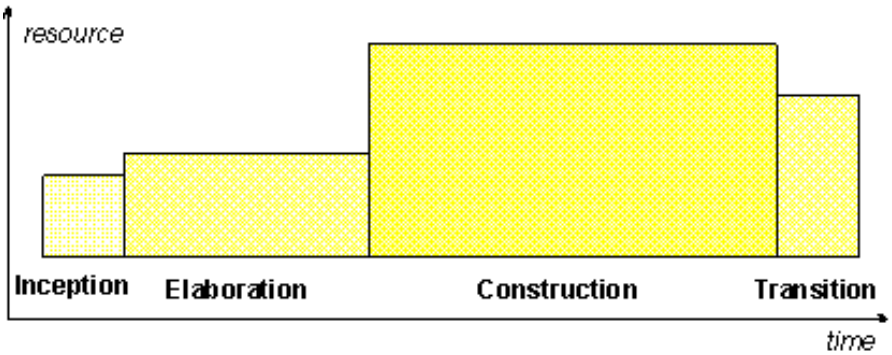


Figura 3.10: *Distribución típica de recursos humanos*

Inicio

Durante la fase de inicio se define el modelo del negocio y el alcance del proyecto. Se identifican todos los actores y Casos de Uso, y se diseñan los Casos de Uso más esenciales

(aproximadamente el 20% del modelo completo). Se desarrolla, un plan de negocio para determinar que recursos deben ser asignados al proyecto.

Elaboración
El propósito de la fase de elaboración es analizar el dominio del problema, establecer los cimientos de la arquitectura, desarrollar el plan del proyecto y eliminar los mayores riesgos.

En esta fase se construye un prototipo de la arquitectura, que debe evolucionar en iteraciones sucesivas hasta convertirse en el sistema final. Este prototipo debe contener los Casos de Uso críticos identificados en la fase de inicio. También debe demostrarse que se han evitado los riesgos más graves.

Construcción
La finalidad principal de esta fase es alcanzar la capacidad operacional del producto de forma incremental a través de las sucesivas iteraciones. Durante esta fase todos los componentes, características y requisitos deben ser implementados, integrados y probados en su totalidad, obteniendo una versión aceptable del producto.

Transición
La finalidad de la fase de transición es poner el producto en manos de los usuarios finales, para lo que se requiere desarrollar nuevas versiones actualizadas del producto, completar la documentación, entrenar al usuario en el manejo del producto, y en general tareas relacionadas con el ajuste, configuración, instalación y facilidad de uso del producto.

Estructura Estática del proceso. Roles, actividades, artefactos y flujos de trabajo

Un proceso de desarrollo de software define quién hace qué, cómo y cuándo. RUP define cuatro elementos: los roles, que responden a la pregunta ¿Quién?, las actividades que responden a la pregunta ¿Cómo?, los productos, que responden a la pregunta ¿Qué? y los flujos de trabajo de las disciplinas que responde a la pregunta ¿Cuándo? (ver Figura 3.11 y 3.12.

Figura3.11: *Relación entre roles, actividades, artefactos*

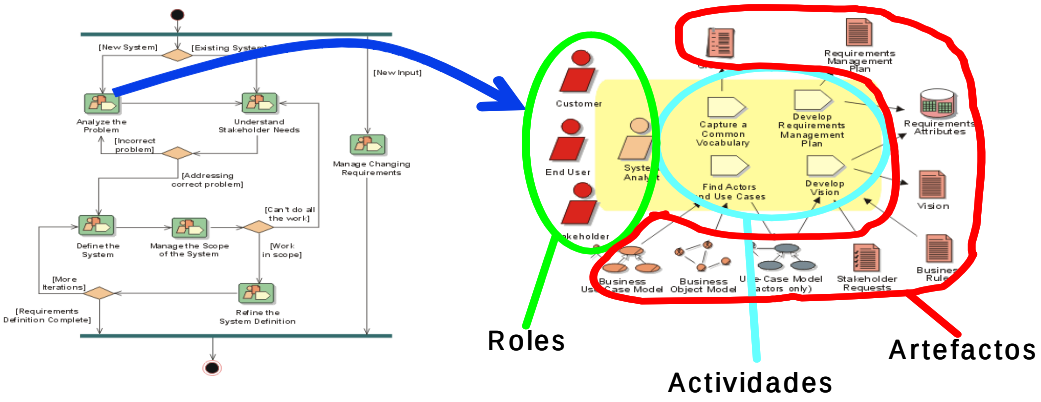


Figura 3.12: *Detalle de un workflow mediante roles, actividades y artefactos*

Roles

Un rol define el comportamiento y responsabilidades de un individuo, o de un grupo de individuos trabajando juntos como un equipo. Una persona puede desempeñar diversos roles, así como un mismo rol puede ser representado por varias personas.

Las responsabilidades de un rol son tanto el llevar a cabo un conjunto de actividades como el ser el dueño de un conjunto de artefactos.

RUP define los siguientes grupos de roles: Analistas, Desarrolladores, gestores, apoyo, especialista en pruebas, etc.

Actividades

Una actividad en concreto es una unidad de trabajo que una persona que desempeñe un rol puede ser solicitado a que realice. Las actividades tienen un objetivo concreto, normalmente expresado en términos de crear o actualizar algún producto.

Artefactos

Un producto o artefacto es un trozo de información que es producido, modificado o usado durante el proceso de desarrollo de software. Los productos son los resultados tangibles del proyecto, las cosas que va creando y usando hasta obtener el producto final

Un artefacto puede ser cualquiera de los siguientes:

- Un documento, como el documento de la arquitectura del software.
- Un modelo, como el modelo de Casos de Uso o el modelo de diseño.
- Un elemento del modelo, un elemento que pertenece a un modelo como una clase, un Caso de Uso o un subsistema.

Flujos de trabajo

Con la enumeración de roles, actividades y artefactos no se define un proceso, necesitamos contar con una secuencia de actividades realizadas por los diferentes roles, así como la relación entre los mismos. Un flujo de trabajo es una relación de actividades que nos producen unos resultados observables. A continuación se dará una explicación de cada flujo de trabajo.

Modelado del negocio

Con este flujo de trabajo pretendemos llegar a un mejor entendimiento de la organización donde se va a implantar el producto.

Para esto, el modelo de negocio describe como desarrollar una visión de la nueva organización, basado en esta visión se definen procesos, roles y responsabilidades de la organización por medio de un modelo de Casos de Uso del negocio y un Modelo de Objetos del Negocio. Complementario a estos modelos, se desarrollan otras especificaciones tales como un Glosario.

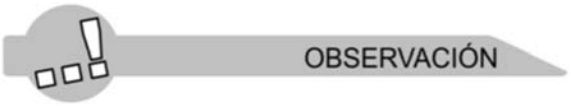
Requisitos
Este es uno de los flujos de trabajo más importantes, porque en él se establece qué tiene que hacer exactamente el sistema que construyamos. En esta línea los requisitos son el contrato que se debe cumplir, de modo que los usuarios finales tienen que comprender y aceptar los requisitos que especifiquemos.
Los requisitos se dividen en dos grupos. Los requisitos funcionales representan la funcionalidad del sistema. Se modelan mediante diagramas de Casos de Uso. Los requisitos no funcionales representan aquellos atributos que debe exhibir el sistema, pero que no son una funcionalidad específica. Por ejemplo requisitos de facilidad de uso, fiabilidad, eficiencia, portabilidad, etc.
En este flujo de trabajo, y como parte de los requisitos de facilidad de uso, se diseña la interfaz gráfica de usuario. Para ello habitualmente se construyen prototipos de la interfaz gráfica de usuario que se contrastan con el usuario final.

Análisis y Diseño
El objetivo de este flujo de trabajo es traducir los requisitos a una especificación que describe cómo implementar el sistema.

El análisis consiste en obtener una visión del sistema que se preocupa de ver qué hace, de modo que sólo se interesa por los requisitos funcionales. Por otro lado el diseño es un refinamiento del análisis que tiene en cuenta los requisitos no funcionales, en definitiva cómo cumple el sistema sus objetivos.

Implementación
En este flujo de trabajo se implementan las clases y objetos en ficheros fuente, binarios, ejecutables y demás. Además se deben hacer las pruebas de unidad: cada implementador es responsable de probar las unidades que produzca.
Pruebas
Este flujo de trabajo es el encargado de evaluar la calidad del producto que estamos desarrollando, pero no para aceptar o rechazar el producto al final del proceso de desarrollo, sino que debe ir integrado en todo el ciclo de vida.
Despliegue
El objetivo de este flujo de trabajo es producir con éxito distribuciones del producto y distribuirlo a los usuarios. Este flujo de trabajo se desarrolla con mayor intensidad en la fase de transición, ya que el propósito del flujo es asegurar una aceptación y adaptación sin complicaciones del software por parte de los usuarios..
Gestión del proyecto
La Gestión del proyecto es el arte de lograr un balance al gestionar objetivos, riesgos y restricciones para desarrollar un producto que sea acorde a los requisitos de los clientes y los usuarios.

Configuración y control de cambios
La finalidad de este flujo de trabajo es mantener la integridad de todos los artefactos que se crean en el proceso, así como de mantener información del proceso evolutivo que han seguido.
Entorno
La finalidad de este flujo de trabajo es dar soporte al proyecto con las adecuadas herramientas, procesos y métodos. Brinda una especificación de las herramientas que se van a necesitar en cada momento, así como definir la instancia concreta del proceso que se va a seguir.



Elegir una metodología de desarrollo de software es algo muy difícil porque dependerá de los objetivos principales del proyecto de software. Sin embargo, podríamos tomar en cuenta los siguientes criterios para elegir alguna de ellas:

- La metodología debe ajustarse a los objetivos
- La metodología debe cubrir el ciclo entero de desarrollo de software
- La metodología debe integrar las distintas fases del ciclo de desarrollo
- La metodología debe soportar la eventual evolución del sistema
- La metodología debe estar soportada por herramientas CASE

1. Compare las diversas metodologías de software y discuta sobre cual de ellas seria la mas adecuada para los siguientes casos:
   a. Para desarrollar un sistema de venta de libros por Internet considerando que la arquitectura de software debe ser de 3 capas.
   b. Para desarrollar un sistema de ventas de productos de primera necesidad si se ha definido que el lenguaje de desarrollo será Visual Fox Pro

2. Investigue acerca de las metodologías de desarrollo de software mas usadas en Perú. Elabore una lista de éstas metodologías, los nombres del producto software obtenidos y las empresas donde han sido desarrollados.

El desarrollo de software no es una tarea fácil. Prueba de ello es que existen numerosas propuestas metodológicas que inciden en distintas dimensiones del proceso de desarrollo. Las metodologías se basan en una combinación de los modelos de proceso genéricos (cascada, evolutivo, incremental, etc.). Adicionalmente una metodología debería definir con precisión los artefactos, roles y actividades involucrados, junto con prácticas y técnicas recomendadas, guías de adaptación de la metodología al proyecto, guías para uso de herramientas de apoyo, etc

La comparación y/o clasificación de metodologías no es una tarea sencilla debido a la diversidad de propuestas y diferencias en el grado de detalle, información disponible y alcance de cada una de ellas. A grandes rasgos, si tomamos como criterio las notaciones utilizadas para especificar artefactos producidos en actividades de análisis y diseño, podemos clasificar las metodologías en dos grupos: Metodologías Estructuradas y Metodologías Orientadas a Objetos. Por otra parte, considerando su filosofía de desarrollo, aquellas metodologías con mayor énfasis en la planificación y control del proyecto, en especificación precisa de requisitos y modelado, reciben el apelativo de Metodologías Tradicionales (o peyorativamente denominada Metodologías Pesadas, o Peso Pesado).

Otras metodologías, denominadas Metodologías Ágiles, están más orientadas a la generación de código con ciclos muy cortos de desarrollo, se dirigen a equipos de desarrollo pequeños, hacen especial hincapié en aspectos humanos asociados al trabajo en equipo e involucran activamente al cliente en el proceso.

BIBLIOGRAFÍA RECOMENDADA

[1] Abrahamsson, P., Salo, O., Ronkainen, J., Warsta, J. **"Agile software development methods Review and analysis".** VTT Publications. 2002.
[2] Beck, K.. **"Extreme Programming Explained. Embrace Change",** Pearson Education, 1999. Traducido al español como: "Una explicación de la programación extrema. Aceptar el cambio", Addison Wesley, 2000.
[3] Jacaboson, I., Booch, G., Rumbaugh J. **El Proceso Unificado de Desarrollo de Software,** Addison Wesley. 2000
[4] Kruchten, P., **The Rational Unified Process: An Introduction,** Addison Wesley. 2000

Bibliografía electrónica:

- Guía sobre METRICA V3
  http://www.csi.map.es/csi/metrica3/

NEXO

En la siguiente unidad utilizaremos RUP como guía para el desarrollo de un caso práctico.

INGENIERIA DEL SOFTWARE
UNIDAD ACADÉMICA Nº 3

NOMBRE:_____

APELLIDOS:_____FECHA; ____/_____/_____

CIUDAD:_____SEMESTRE:_____

PARTE I: Marque la alternativa correcta:

1. Un proceso es ágil cuando el desarrollo de software es:
   a) Incremental, cooperativo, sencillo, y adaptable
   b) No incremental, pero si cooperativo, sencillo, y adaptable
   c) Rápido aunque tenga muchas fallas porque solo sirve como prototipo
   d) Incremental, sencillo pero no adaptable
   e) N.A.

2. METRICA v3 es una metodología de desarrollo de software:
   a) Estructurada
   b) Orientada a Objetos
   c) Tradicional
   d) a y b
   e) N.A.

3. Se dice que RUP es una metodología de desarrollo de software dirigido por caso de uso porque:
   a) El resultado final del análisis y diseño es un diagrama de casos de uso
   b) Son sólo una herramienta para especificar los requisitos del sistema.
   c) Además de ser una herramienta para especificar los requisitos del sistema, también guían su diseño, implementación y prueba
   d) Los casos de uso representan la arquitectura del software
   e) N.A.

4. XP enfatiza la definición temprana de una arquitectura estable para el sistema. Esta afirmación es
   a) Verdadera
   b) Falsa

PARTE II: Responda brevemente:

5. ¿Cree Ud.. necesario el uso de una metodología de desarrollo de software? ¿Por qué?
   _____
   _____
   _____
   _____

6. ¿Qué recomendaría Ud. para una mejor elección de una metodología de desarrollo de software?

7. ¿Cuál es la relación que existe entre una metodología de análisis y diseño de sistemas y una metodología de desarrollo de software?

8. ¿Cuál es la diferencia primordial entre una metodología de desarrollo de software ágil y una tradicional?

Unidad Académica IV

RATIONAL UNIFIED PROCESS (RUP): CASO PRÁCTICO

En la presente unidad se pretende mostrar un ejemplo de desarrollo de software basado en la metodología de desarrollo Rational Unified Process (RUP). El proyecto trata del desarrollo de un sistema de gestión clínica para un hospital. El ejemplo esta centrado principalmente en el modelado del negocio, obtención de requerimientos y el modelo de análisis y diseño por considerarse la parte substancial del RUP sin desmerecer de ningún modo las otras actividades concernientes a dicho proceso. Hacia el final de la unidad se dará una breve guía de cómo continuar con el proceso hasta llegar a la implementación del sistema y su posterior despliegue.

INDICADORES DE LOGRO

Al finalizar el estudio de la presente unidad temática el estudiante:

13. Conoce y utiliza RUP
14. Conoce los artefactos que se obtendrán como resultado del proceso
15. Entiende la diferencia entre el modelo del negocio y la obtención de requerimientos
16. Entiende la diferencia entre el modelo de análisis y diseño del proceso
17. Aplica RUP a un caso practico

Modelamiento de un Sistema de Gestión Clínica Utilizando El Proceso Unificado

1. Disciplinas y artefactos: Modelo del Negocio

DISCIPLINA	ACTIVIDAD	DOCUMENTOS	MODELOS	DIAGRAMA
Modelado del Negocio	• Evaluar el estado del negocio • Describir el negocio actual • Identificar los procesos del negocio • Refinando las definiciones del proceso del negocio· • Diseñar las realizaciones de los procesos del negocio • Refinando roles y responsabilidades • Explotando procesos de automatización • Desarrollo de un modelo del Dominio	• Evaluación del estado de la Organización • Plan de Desarrollo de Software • Reglas del Negocio • Especificaciones de casos de uso del Negocio • Arquitectura del Negocio • Visión del Negocio	Modelo del Negocio	• Diagrama de Casos de Uso del Negocio • Modelo de Objetos del Negocio
			Modelo del dominio	Modelo del Dominio del problema

Tabla 4-1: *Disciplinas y artefactos del modelo del negocio*

Artefactos que se obtendrán en el ejemplo:

Al término del modelo del negocio se debe lograr:

- El diagrama de de Casos de Uso del Negocio (Business Use Case diagrams).
- Uno o más diagramas de actividad para cada caso de uso del negocio.
- Un Diagrama de Objetos del Negocio.
- Un diagrama de clases para todo el modelo del negocio. Este diagrama es también llamado modelo del dominio.

Modelado del Negocio para el Sistema de Gestión Clínica

a) Construir 4 Paquetes:

Se recomienda que los dos primeros siempre se creen, mientras que los dos últimos solo se crean con fines de tener organizado nuestro modelo:

- **Unidades Organizacionales.-** Contendrá los trabajadores del negocio (**bussiness worker**) y otras unidades organizacionales según la jerarquía establecida por el negocio (organigrama de la empresa).
- **Casos de Uso del Negocio.-** Contendrá los casos de uso del negocio (**bussiness use case -** procesos que permiten cumplir los objetivos del negocio) así como los trabajadores del negocio (**bussiness worker**) y los actores del negocio (**bussines actor**).
- **Actores del Negocio.-** Permite agrupar en un solo sitio todos los actores que participan en los casos de uso del negocio.
- **Trabajadores del Negocio.-** Permite agrupar a los empleados de la organización.

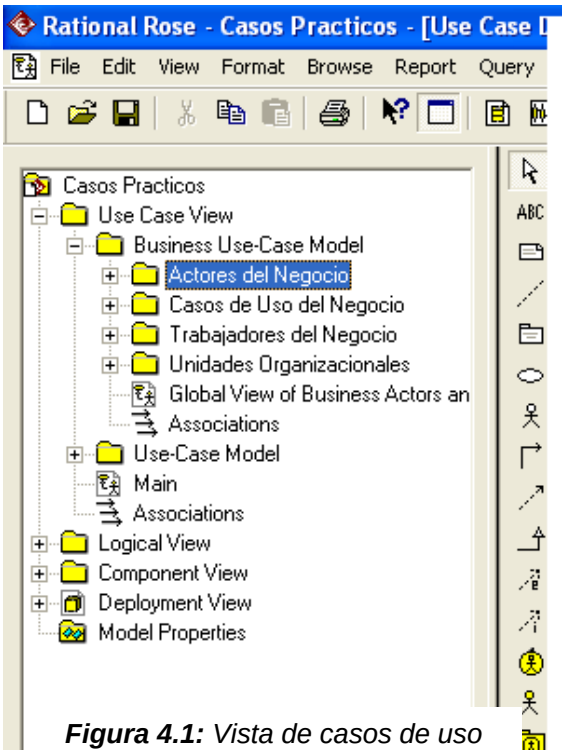


Figura 4.1: *Vista de casos de uso*

b) Identificar las unidades organizacionales

Nuestro software de gestión clínico de ejemplo va a correr en un área específica de la empresa, pero necesitamos establecer el contexto más general para luego detallar solo lo que nos interesa. Así, nuestro negocio es una empresa que está dividida en áreas tales como: Producción, Finanzas, Recursos Humanos, Ventas, Contabilidad y Logística. Estas áreas las podemos obtener directamente desde el organigrama de la empresa. Usando la notación UML estereotipada para el modelo del negocio tendremos:

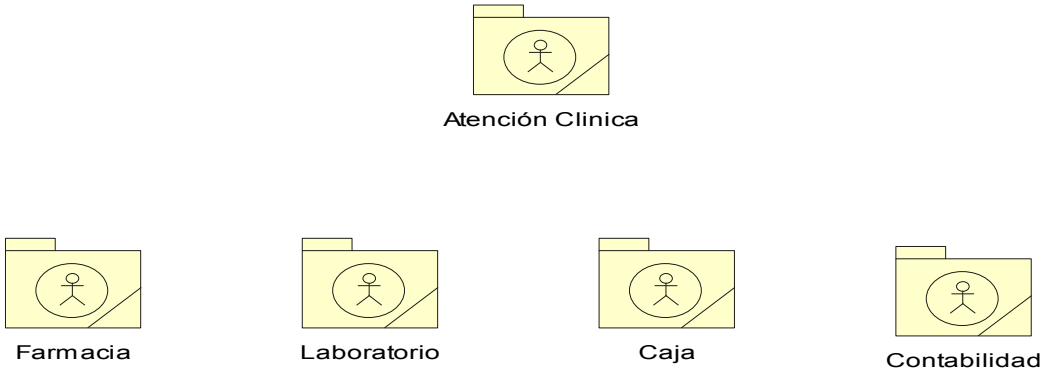


Figura 4.2: *Unidades organizacionales*

Se irá detallando cada unidad organizacional. Para la unidad organizacional de nuestro interés (atención clínica) tenemos:

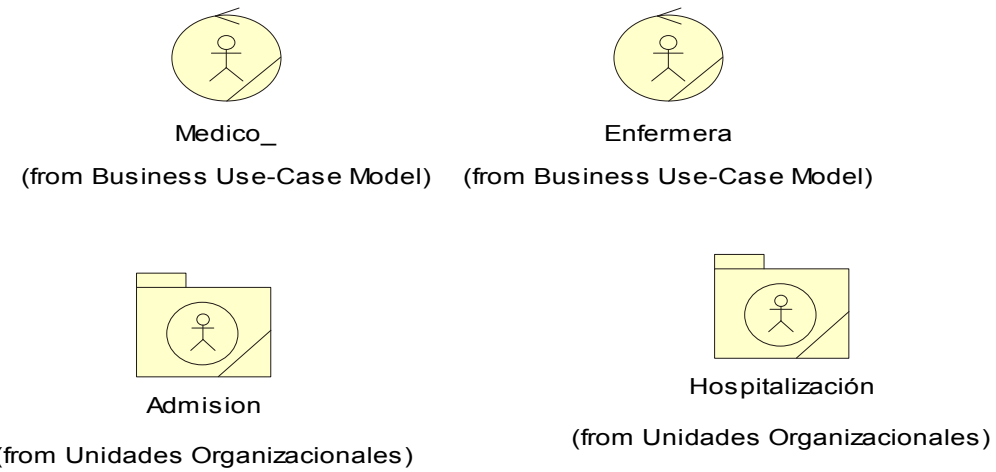


Figura 4.3: *Actores del negocio con sus respectivas unidades organizacionales*

Para que el modelo de la organización este completo será necesario que describamos las funciones principales de cada área así como las funciones de cada **business worker**.

c) Identificar los Casos de Uso del Negocio

El diagrama de casos de uso del negocio debe representar los principales objetivos de la organización, y en nuestro caso particular los objetivos del área. Veamos, según la descripción que encontramos al construir nuestro modelo organizacional, tenemos que Atención Clínica tiene como objetivos fundamentales: La admisión de pacientes, atención de pacientes, hospitalización, servicios de análisis clínicos y la atención en farmacia. Por lo que nuestro Diagrama de Casos de uso será:

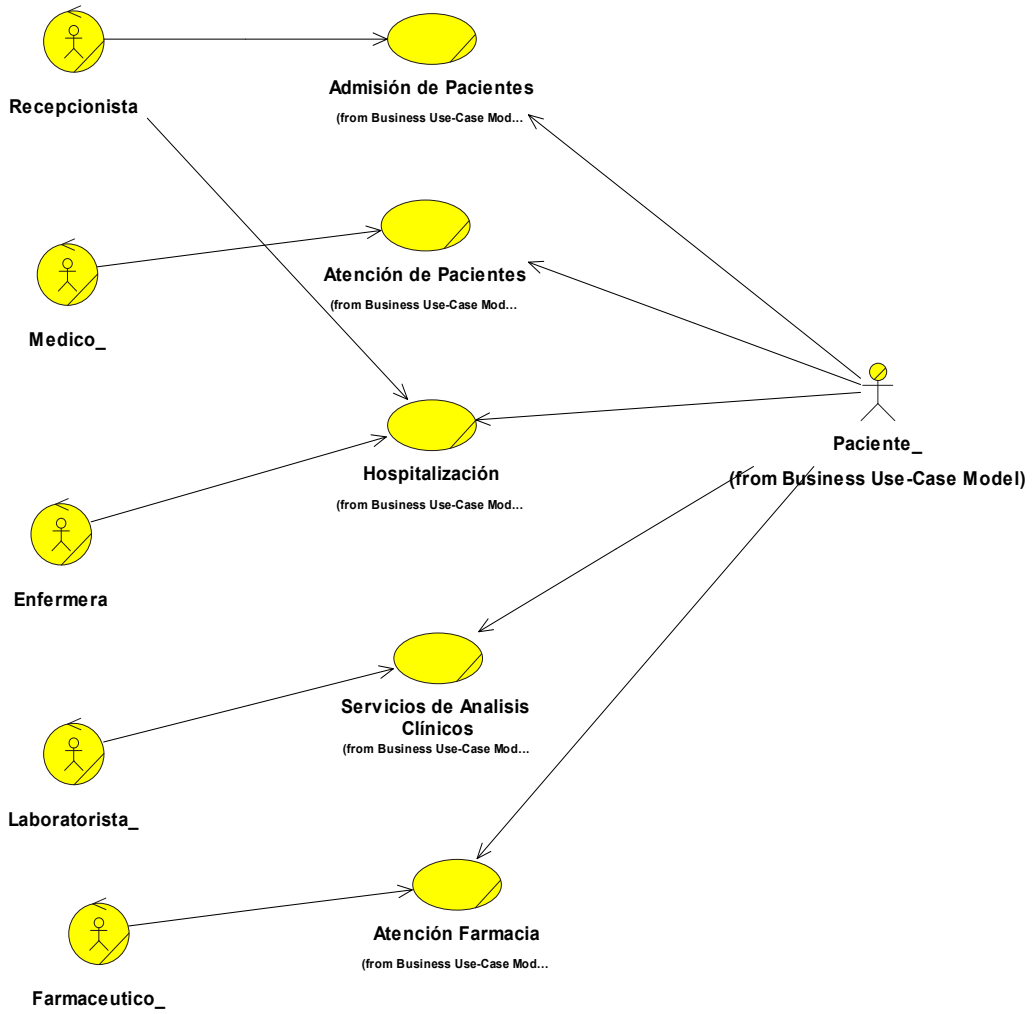


Figura 4.4: *Diagrama de casos de uso del negocio*

Los trabajadores del negocio ya han sido identificados cuando investigamos las unidades organizaciones, mientras que los actores del negocio se identifican conforme vamos desarrollando los casos de uso del negocio. Debemos indicar que algunos autores consideran que solo debemos mostrar los actores externos pues se dan límites al sistema y los workers son "actores internos" y están dentro de la organización.

d) Se crean los paquetes del modelo de objetos del negocio

Para esto hay que ubicarse en la Vista lógica y encontrar el paquete Business Object Model, dentro del cual se crean 3 paquetes:

- **Diagrama de Objetos del Negocio:** Que contendrá el diagrama de objetos.
- **Diagrama del Dominio:** El cual contiene las principales entidades que se encontraron en el negocio.
- **Realización de los Casos de Uso del Negocio:** Aquí detallamos los casos de uso del negocio mediante un diagrama de actividad (u otro).

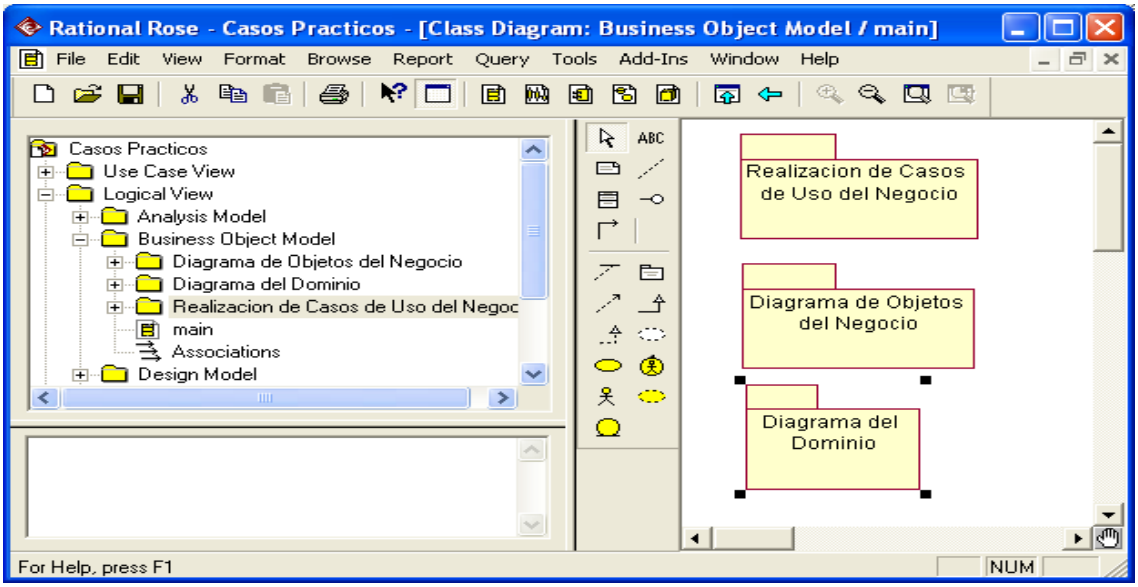


Figura 4.5: *Vista lógica del negocio*

e) Se construye la realización de los casos de uso del negocio

Los casos de uso del negocio se pueden especificar de diversas formas, como son: pseudocódigo, lenguaje natural, diagramas de actividades, diagramas de secuencia, diagramas de colaboración, diagramas de estado, etc. Sin embargo nosotros utilizaremos los diagramas de actividad pues son los más indicados para representar procesos (**actividades**) mostrando las áreas, personas y objetos en general (**carriles**) que llevan a cabo procesos secuenciales o en paralelo (**barra de sincronización**) y objetos fluyen entre procesos (**flujo de objetos**), así también representan decisiones de bifurcación.
Cada uno de los casos de uso tendrá uno o más diagramas de actividades (o de cualquier otro tipo, según los necesitemos en un caso específico).
El paquete realización de los casos de uso del negocio contendrá lo siguiente:

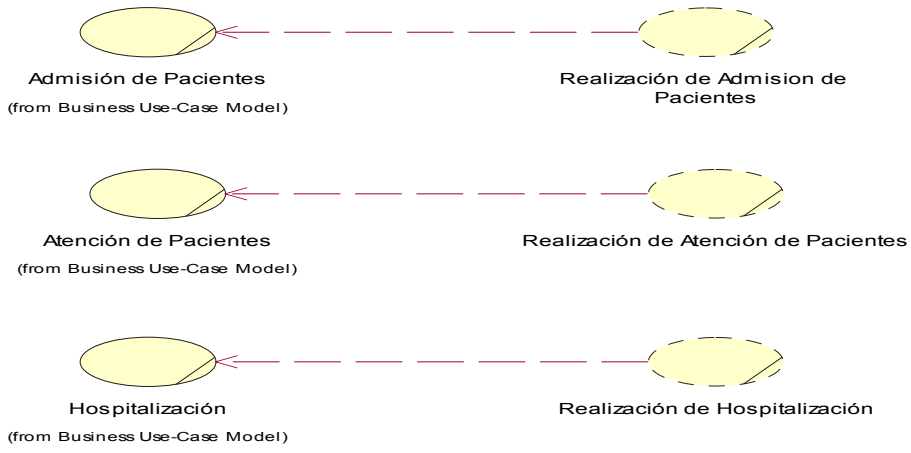


Figura 4.6: *Realizaciones de los casos de uso*

f) Para cada realización de los casos de uso del negocio construimos un diagrama de actividad y un diagrama de clases.

El diagrama de actividad nos va ha permitir especificar como se lleva a cabo el caso de uso del negocio y si en ese diagrama incluimos el flujo de objetos, estaremos identificando los objetos participantes, a partir de lo cual podremos derivar un pequeño diagrama de objetos para cada realización.

Realización del Caso de Admisión Paciente

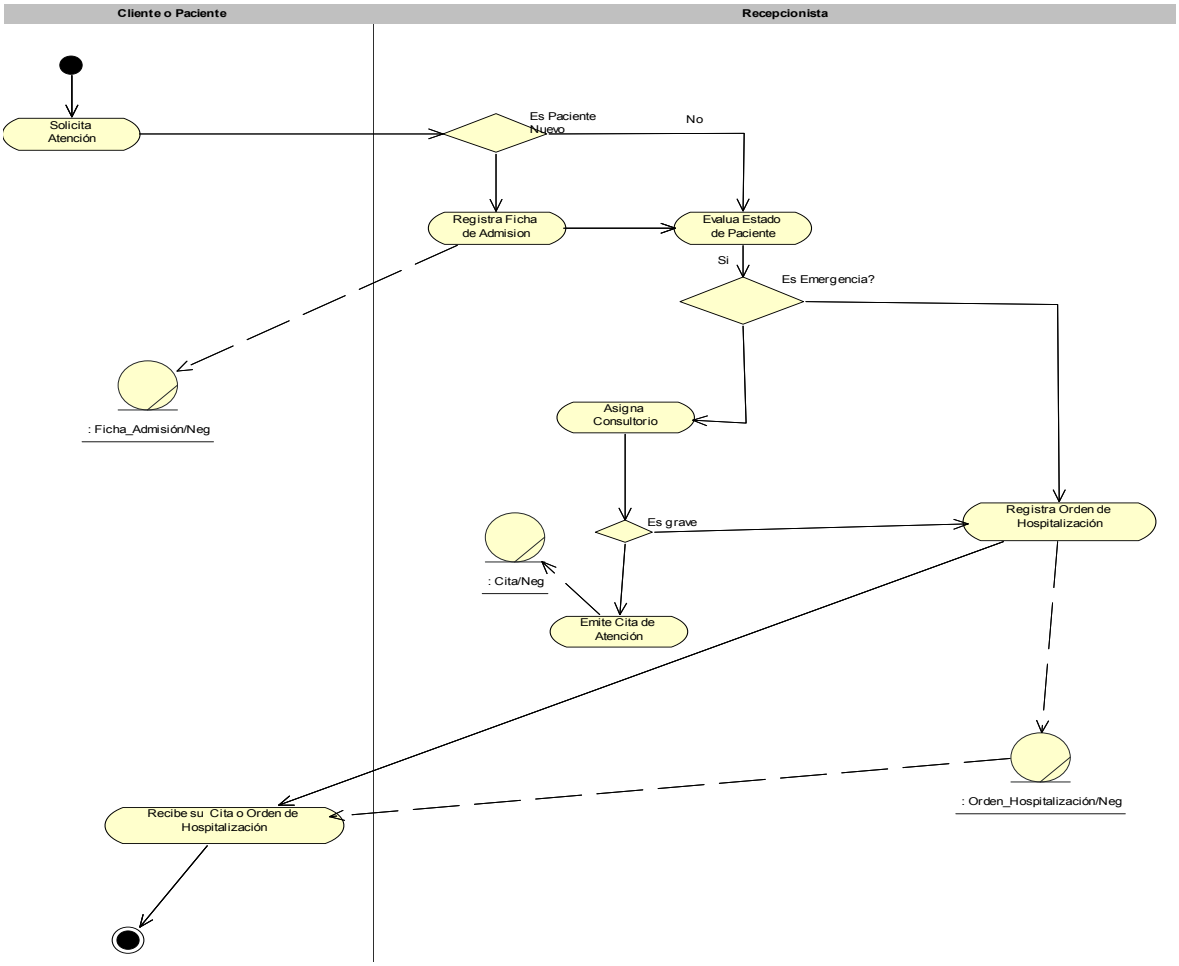


Figura 4.7: *Diagrama de Actividad: Admisión Paciente*

Diagrama de Objetos participantes: Admisión Pacientes

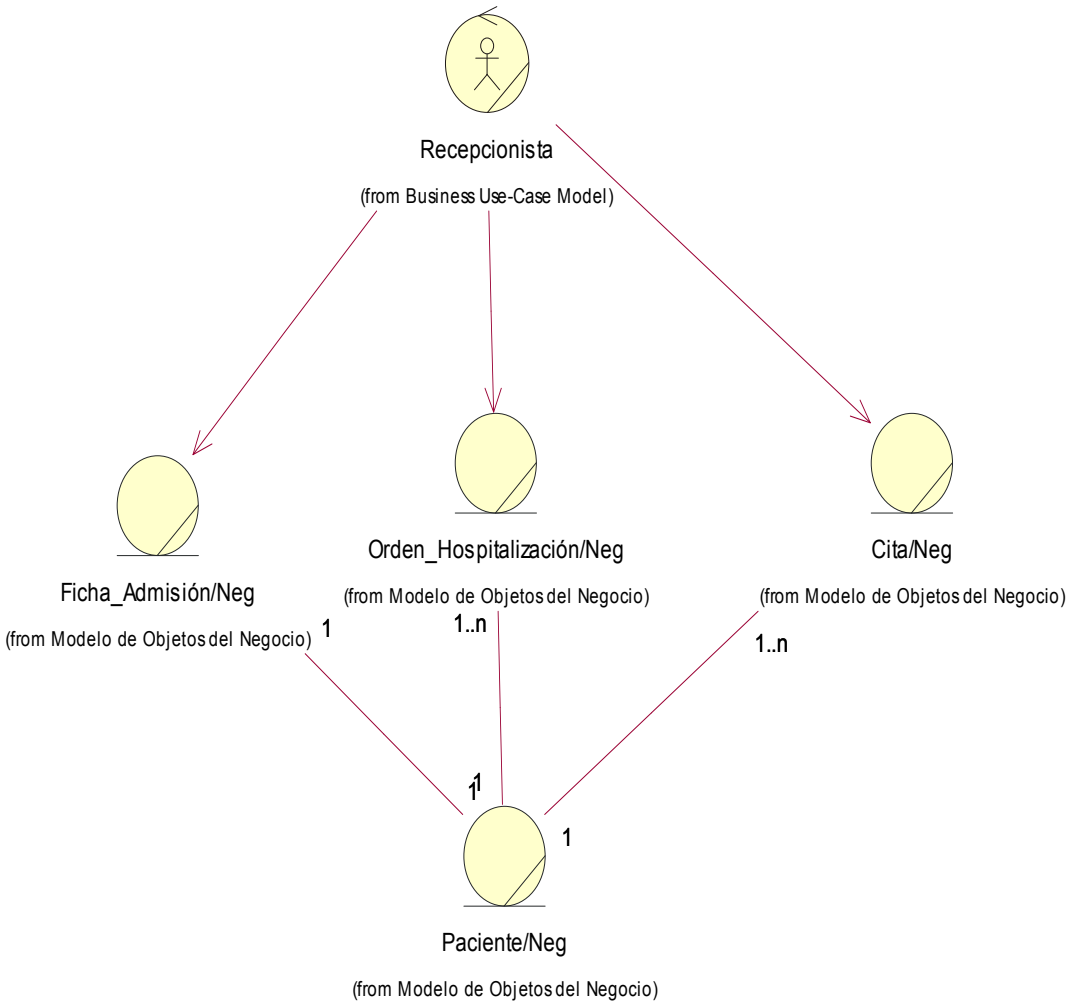


Figura 4.8: *Diagrama de Objetos: Admisión Paciente*

Para efectos de ejemplo en el presente libro solo se mostrará la realización de casos de uso de Admisión de Pacientes. El lector debe entender que las realizaciones se hacen por cada caso de uso del negocio.

g) Se unen los diagramas de objetos de cada caso de uso y así se obtiene el diagrama de Objetos del Negocio

Como ya tenemos identificados los objetos participantes en cada caso de uso, podemos unirlos en un único diagrama y debemos refinar este modelo completando detalles como multiplicidad y otros, asimismo nos daremos cuenta si hay objetos redundantes.

Modelo de Objetos del Negocio

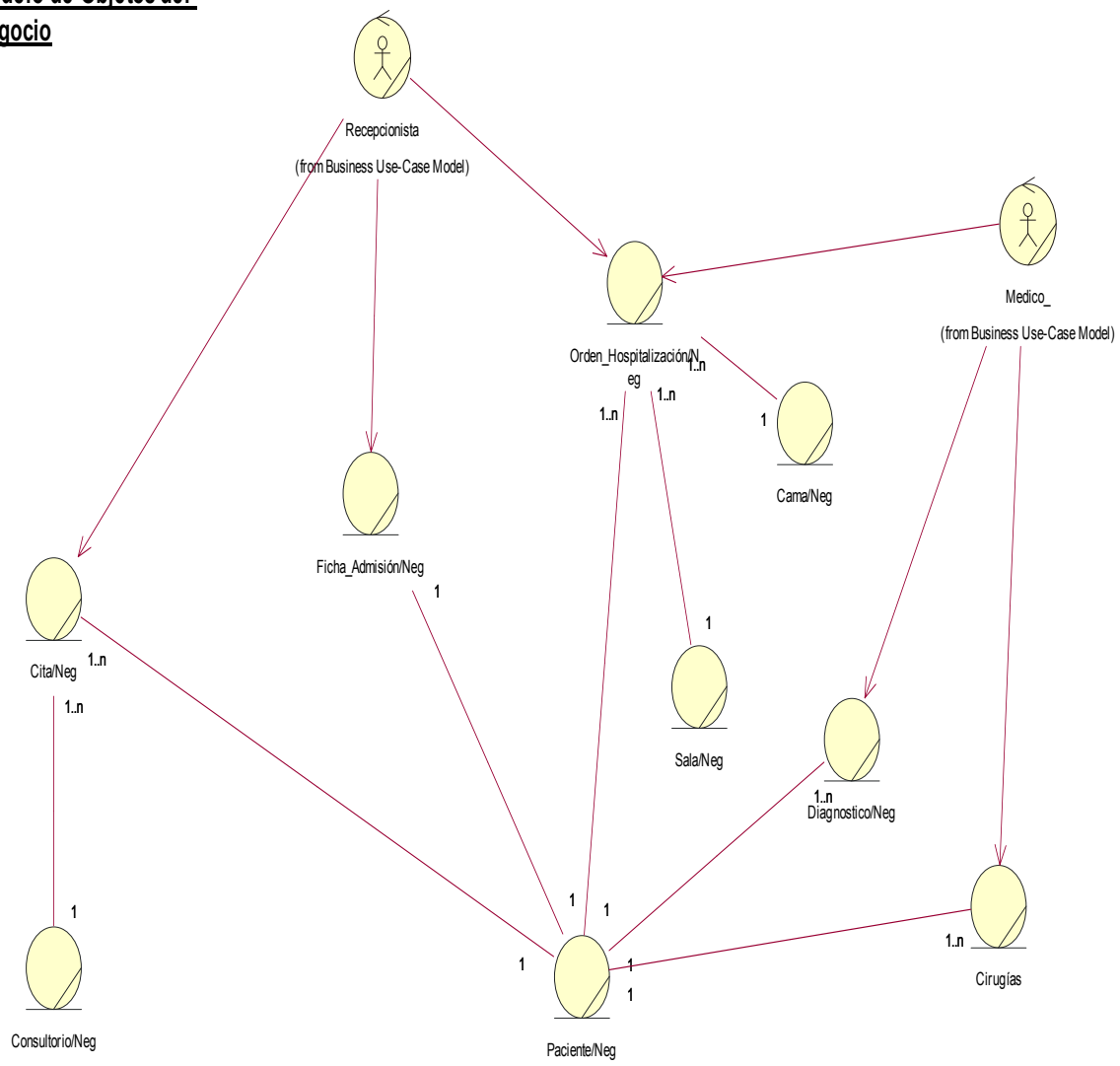


Figura 4.9: *Modelo de Objetos del Negocio*

h) De ser necesario se construye el Modelo del Dominio

En realidad solo es necesario construir uno de los dos modelos: el modelo del dominio o el modelo de objetos. El modelo del dominio se construye cuando se desea efectuar un Modelamiento del Negocio muy breve y solo identifica las clases más importantes en la organización para brindar un panorama de la misma. Sin embargo, cuando se desarrolla un Modelo del Negocio más amplio, se construye un diagrama de objetos del Negocio. A pesar de haber desarrollado un Modelo del Negocio amplio, se incluye aquí un Modelo del Dominio como ilustración y para que el modelo resulte completo.

Modelo del Dominio del Sistema

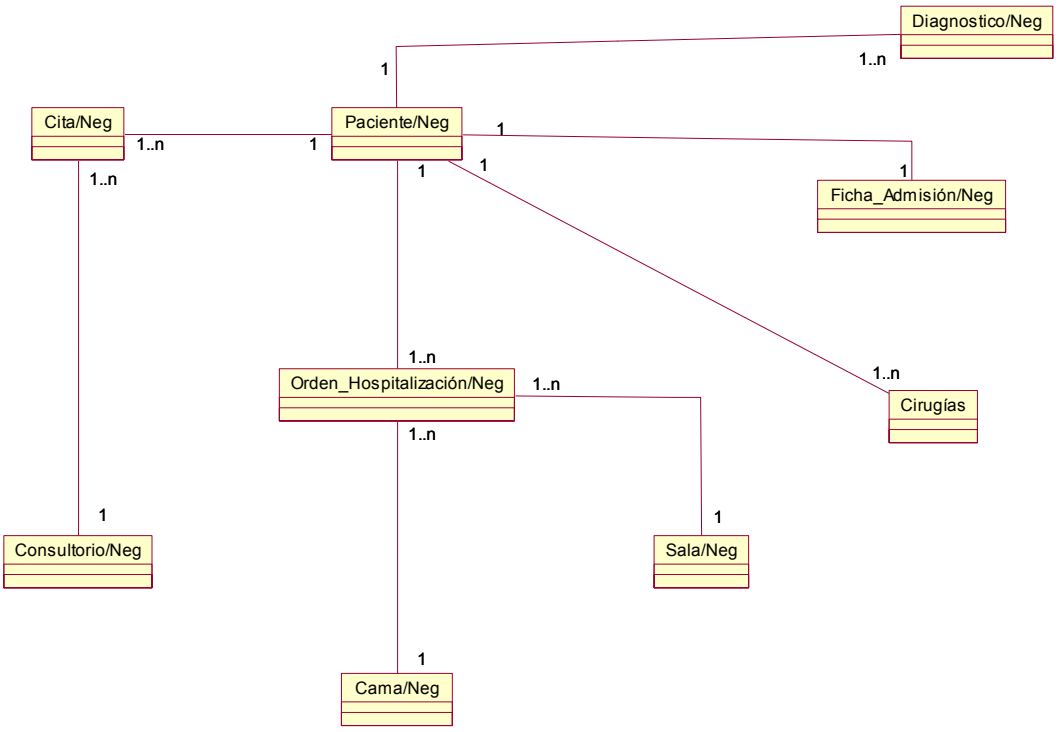


Figura 4.10: *Modelo del dominio*

2. Disciplinas y artefactos: Modelado de Requerimientos

DISCIPLINA	ACTIVIDAD	DOCUMENTOS	MODELOS	DIAGRAMA
Requerimiento	• Analizar el problema • Conocer las necesidades de los stakeholders • Definir el sistema • Administrar el Alcance del Sistema • Administrar los cambios de Requerimiento.	• Especificación de Casos de Uso. • Visión. • Plan de Administración de requerimientos	Modelo de requerimiento	• Diagrama de Casos de Uso

Tabla 4-2: *Disciplinas y artefactos del modelo de requerimientos*

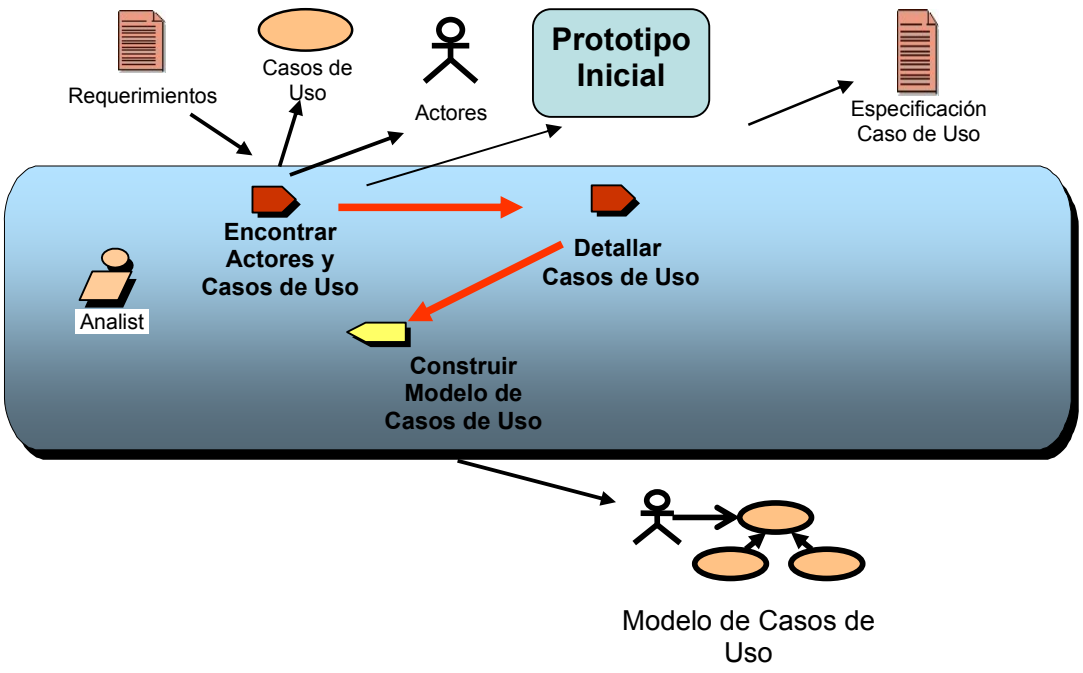


Figura 4.10: *Proceso de captura de requerimientos*

Requerimientos para el Sistema de Gestión Clínica

a) Ir a la Vista de Casos de Uso dentro del Modelo de Casos de Uso y crear los paquetes

- **Actores:** Servirá para agrupar todos los actores del modelo de casos de uso
- **Casos de Uso:** Servirá para agrupar todos los casos de uso
- **Casos de Uso Arquitecturalmente significativos:** Es un subconjunto de casos de uso que influyen fuertemente en la arquitectura.

En la parte derecha del diagrama siguiente, se observa como el Modelo de Casos de Uso es dependiente del modelo de casos de uso del negocio, ya que algunos de sus elementos se derivan de este, y los casos de uso del software deben soportar a los casos de uso del negocio.

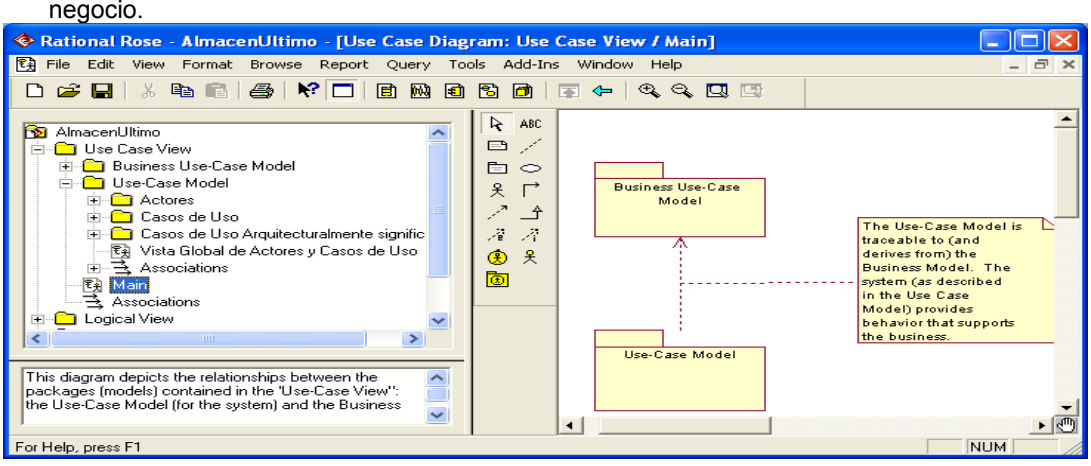
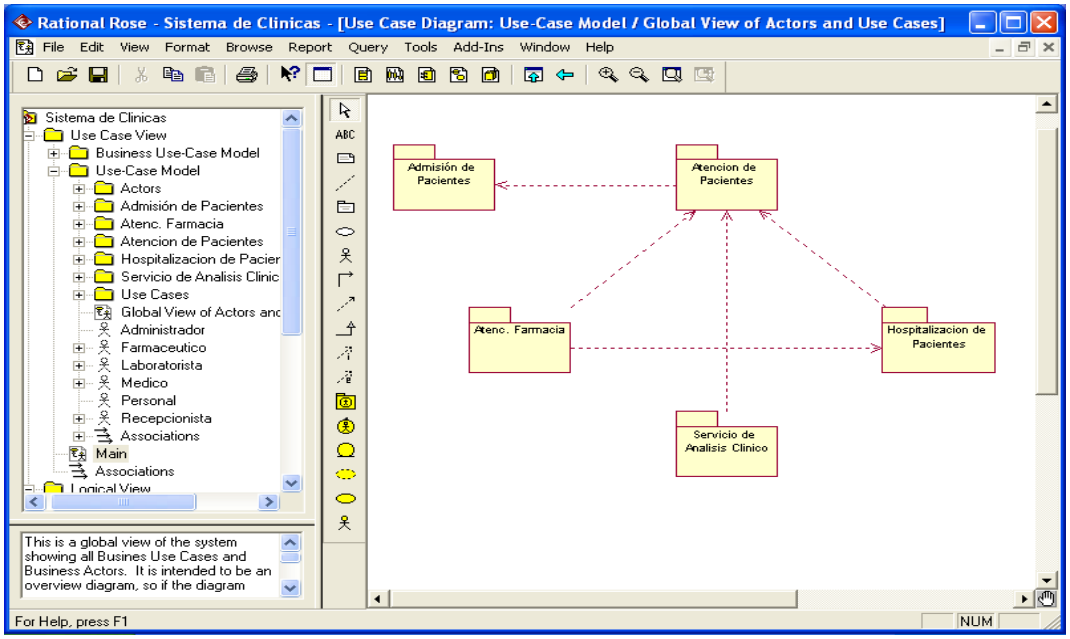


Figura 4.11: *Vista de casos de uso. Dependencia entre el Business use-case model y el use-case model*

b) Ir al paquete de Casos de Uso e identificar como organizar los casos de uso.

En este paso se comienza identificando los casos de uso, sus actores y relaciones entre ellos. Luego se procede a agruparlos según algún criterio. Por ejemplo si el sistema



involucra varias áreas se debería crear un paquete para cada área. Si se desea agrupar según alguna funcionalidad se debe crear un paquete para cada una, o si es un sistema pequeño no se necesita ningún agrupamiento. En éste caso se ha optado por agruparlos en los paquetes: Admisión de Pacientes, Atención de Pacientes, Hospitalización, Atención Farmacia y Servicio de Análisis Clínico. En la mayoría de casos se empieza sin ningún paquete y conforme se avance se va agrupando según el criterio del analista.

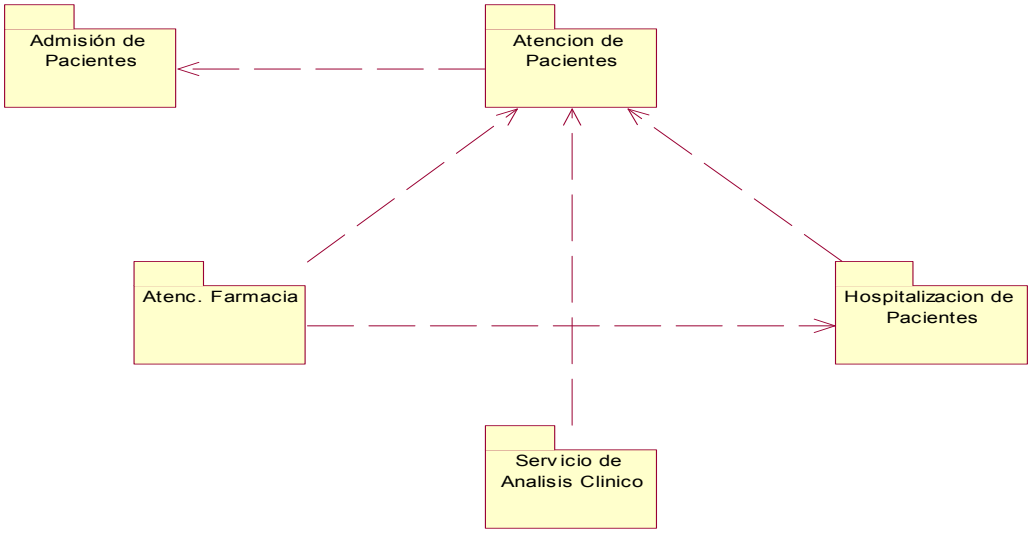


Figura 4.12: *Paquetes de casos de uso*

c) Construir diagramas de Casos de Uso para cada paquete de Casos de Uso

Modelo de Casos de Uso de Admisión de Pacientes

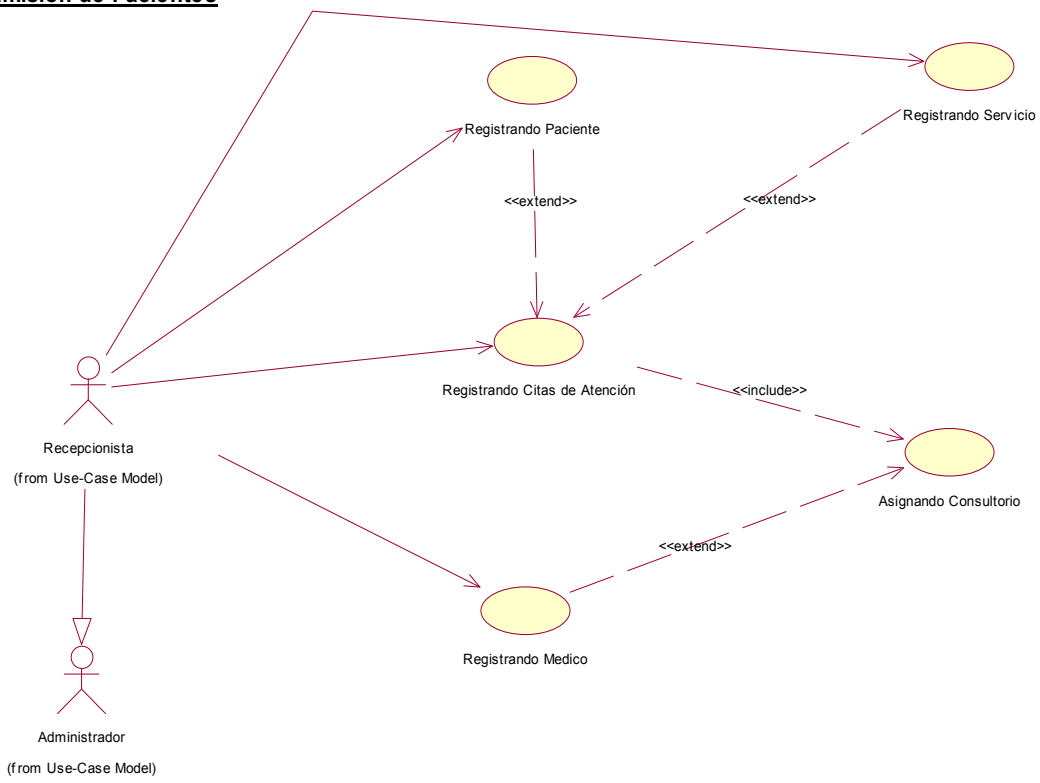


Figura 4.13: *Casos de uso de Admisión pacientes*

En la pestaña general de cada caso de uso se puede escribir su especificación según el formato mostrado. Si la especificación es larga se puede enlazar un archivo en word que contenga la especificación (Anexo 1).

También puede realizarse una descripción de otros elementos como los actores.

Figura 4.14: *Especificación de casos de uso*

d) Ir al paquete de Actores y arrastrar los actores identificados

Todos los actores identificados en los casos de uso deben ser arrastramos hacia el paquete de actores con el fin de tener una vista completa de todos los actores, entonces se puede añadir algunas relaciones de generalización.

e) Identificar los casos de uso arquitecturalmente significativos
A partir de los casos de uso identificados se busca a aquellos que son más relevantes en el sistema y que influyen en su arquitectura.

f) Construcción de Prototipo

3. Disciplinas y artefactos: Modelado de Análisis y Diseño

DISCIPLINA	ACTIVIDAD	DOCUMENTOS	MODELOS	DIAGRAMA
Análisis Y Diseño	• Definir una arquitectura candidata • Analizar la conducta del Sistema • Diseñar Componentes • Refinar la arquitectura • Diseñar la Base de Datos.	• Flujo de Eventos • ·Especificaciones suplementarias • Arquitectura del Software	• Modelo de Análisis • Modelo del Diseño	• Diagrama de Colaboración . • Diagrama de secuencia • Diagrama de Clases • Diagrama de Despliegue

Tabla 4-3: *Disciplinas y artefactos de análisis y diseño*

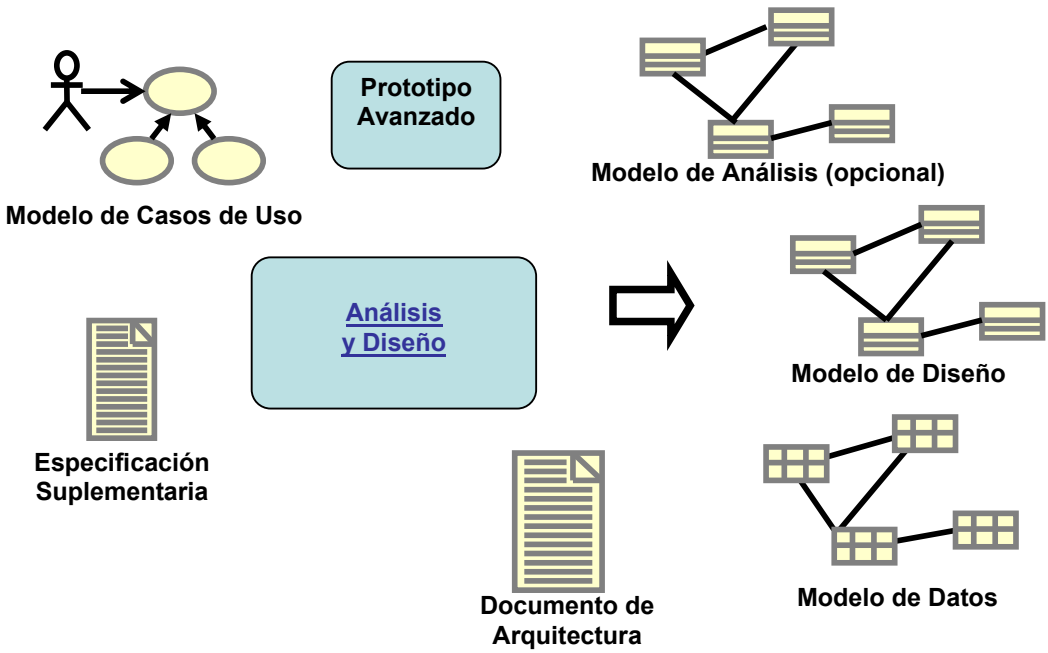
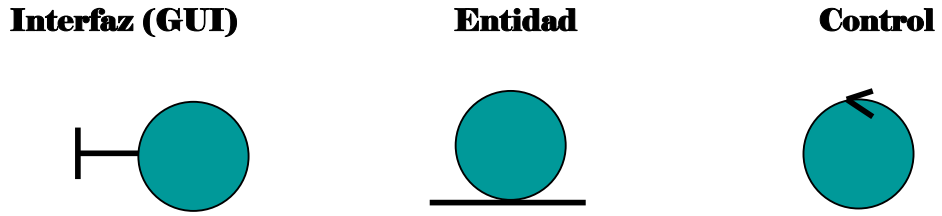


Figura 4.15: *Proceso de análisis y diseño*

Estereotipos Principales



Modelo de Análisis del Sistema de Gestión Clínica

a) Ir a la Vista Lógica y en el Modelo de Análisis crear los paquetes:

- Paquete Diagrama de Clases del Análisis
- Paquete Realización de los Casos de Uso

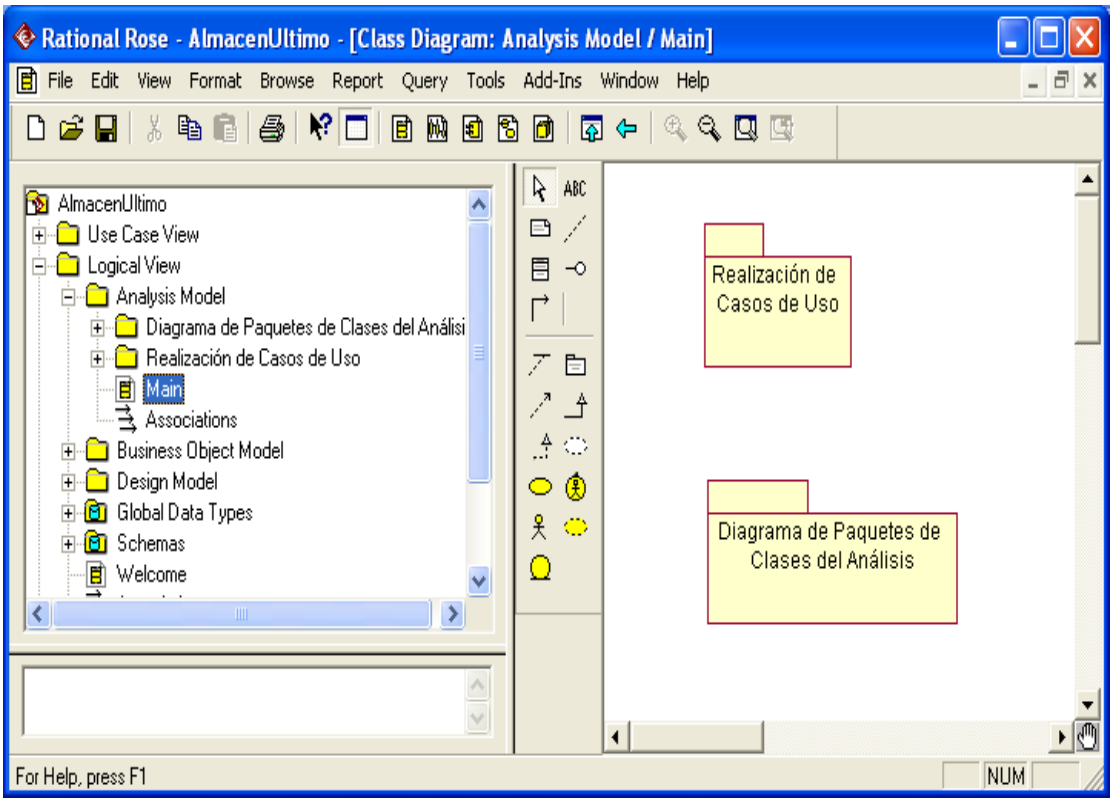


Figura 4.16: *Paquetes del Modelo de Análisis*

b) Ir al Paquete de Realización de Casos de Uso y crear los paquetes correspondientes.
Para cada paquete del modelo de casos de uso se le crea su paquete de realizaciones donde justamente se mostrará como cada caso de uso se enlaza con su realización (la cual esta especificada mediante un diagrama)

c) Mostrar cada Realización de Casos de Uso

Para mostrar la realización de cada caso de uso utilizamos los símbolos del UML dependencia y Colaboración, así cada caso de uso estará enlazado a su correspondiente realización (colaboración) y dentro de este símbolo construiremos el diagrama respectivo.

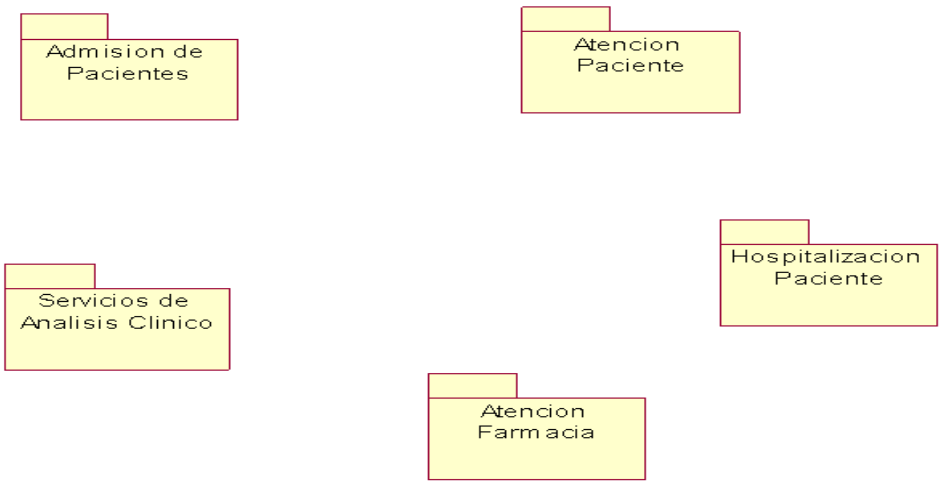


Figura 4.17: *Paquetes de las realizaciones de casos de uso*

Debe elaborarse realizaciones para cada paquete. Para el ejemplo solo se mostrara la realización de uno de los paquetes.

Realización del Paquete de Admisión de Pacientes

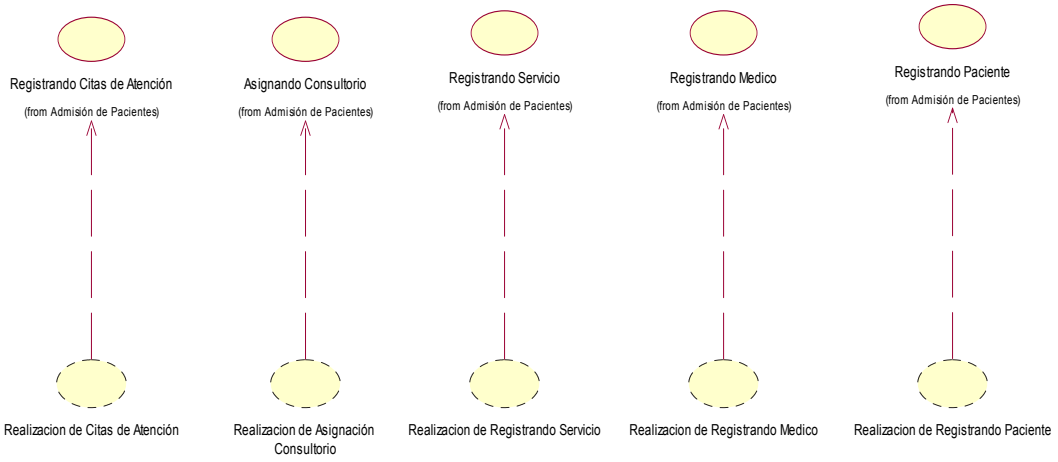


Figura 4.18: *Realización de casos de uso para Admisión pacientes*

A continuación debe elaborarse un diagrama de colaboración por cada realización de caso de uso. Nuevamente, solo se muestra un diagrama de colaboración para una sola realización de caso de uso.

Diagrama de Colaboración de Citas de Atención

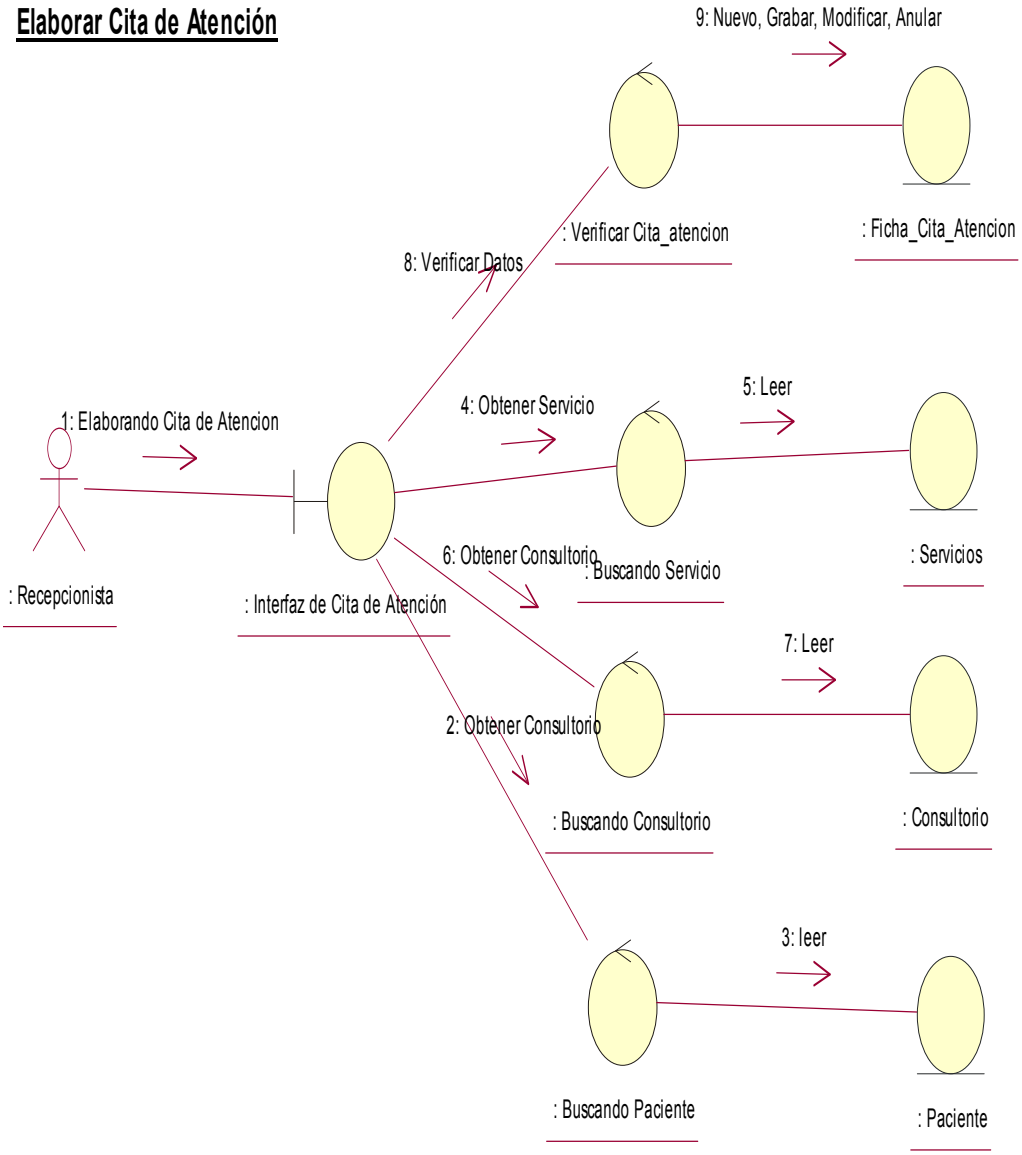


Figura 4.19: *Diagrama de colaboración para la realización: Elaborar citas de Atención*

Con cada entidad generada en el diagrama de colaboración se crea el diagrama de clases para cada realización

Diagrama de Clases de Citas de Atención

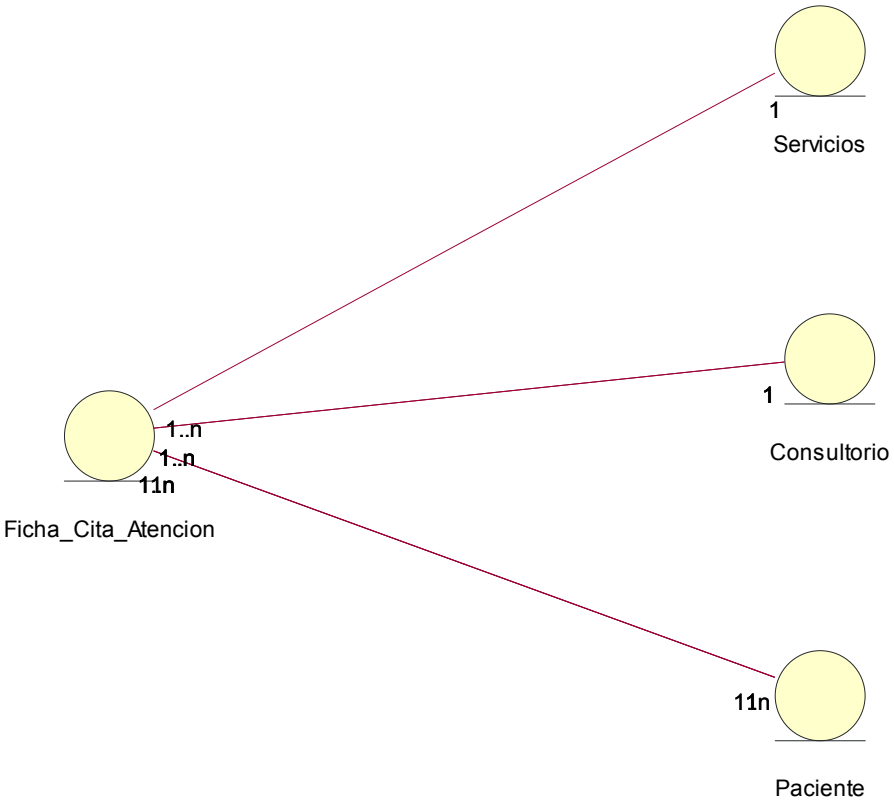


Figura 4.20: *Diagrama de clases de citas de atencion*

d) Ir al Paquete de Clases del Análisis y Crear otros paquetes

Según el criterio adoptado podemos crear paquetes agrupando alguna posible funcionalidad, área, o capas. Para el ejemplo se crean tres paquetes: interfaz, control y entidad.

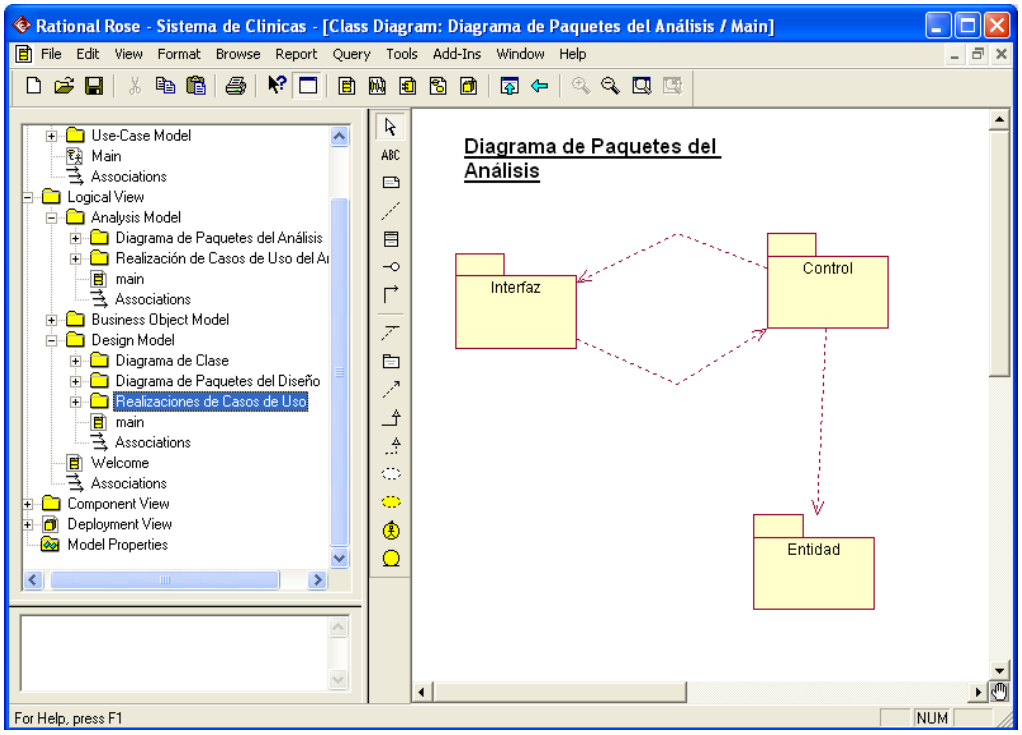


Figura 4.21: *Diagrama de paquetes del análisis*

e) Arrastrar cada clase del Análisis a su respectivo paquete

Diseño del Sistema de Gestión Clínica

a) Ir a la Vista Lógica y dentro de ella a la vista de diseño y crear los paquetes siguientes:

- Paquete de diseño
- Realización de casos de uso diseño.

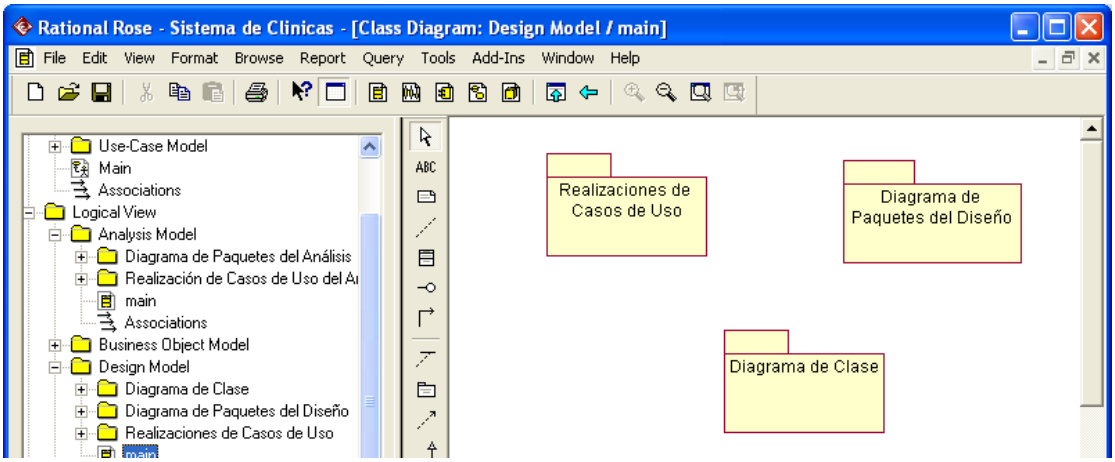
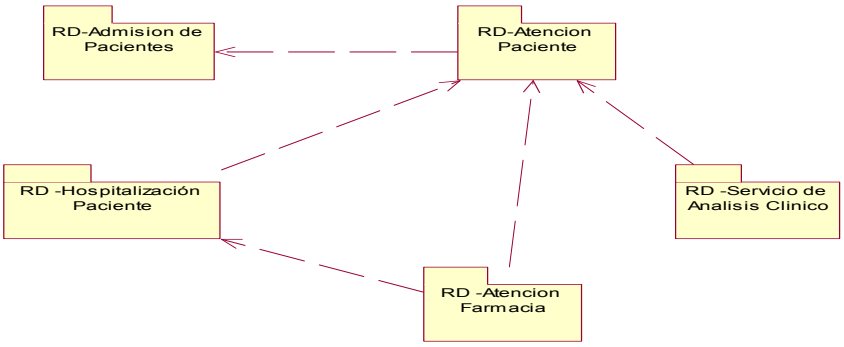


Figura 4.22: *Diagrama de paquetes del modelo del diseño*

b) Construir la realización de los casos de uso diseño

A partir de los paquetes de la realización de los casos de uso del análisis construimos los correspondientes paquetes de realización de casos de uso de diseño.



Cada uno de estos paquetes contiene los casos de uso de requerimientos enlazados con sus correspondientes realizaciones del análisis los que a su vez se enlazan con sus realizaciones del diseño.

Realización de casos de uso de Admisión de Pacientes

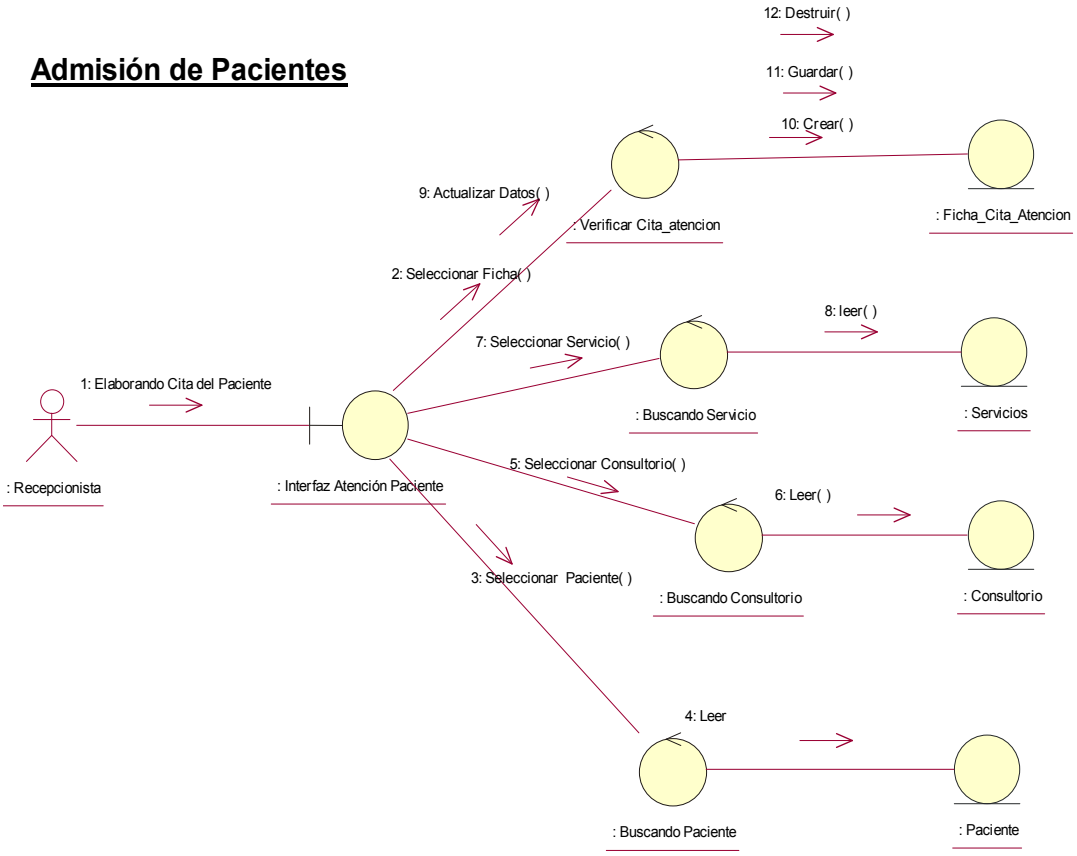


Figura 4.23: *Diagrama de* ***colaboración*** *para la realización de Citas de Atención*

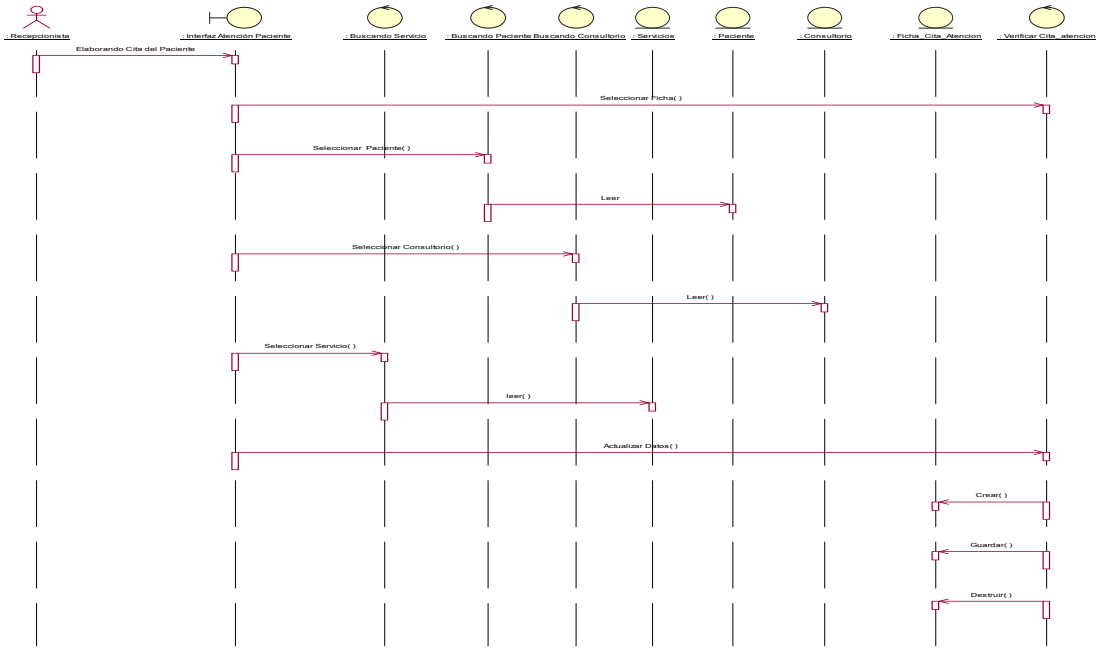


Figura 4.24: *Diagrama de* ***secuencia*** *para la realización de Citas de Atención*

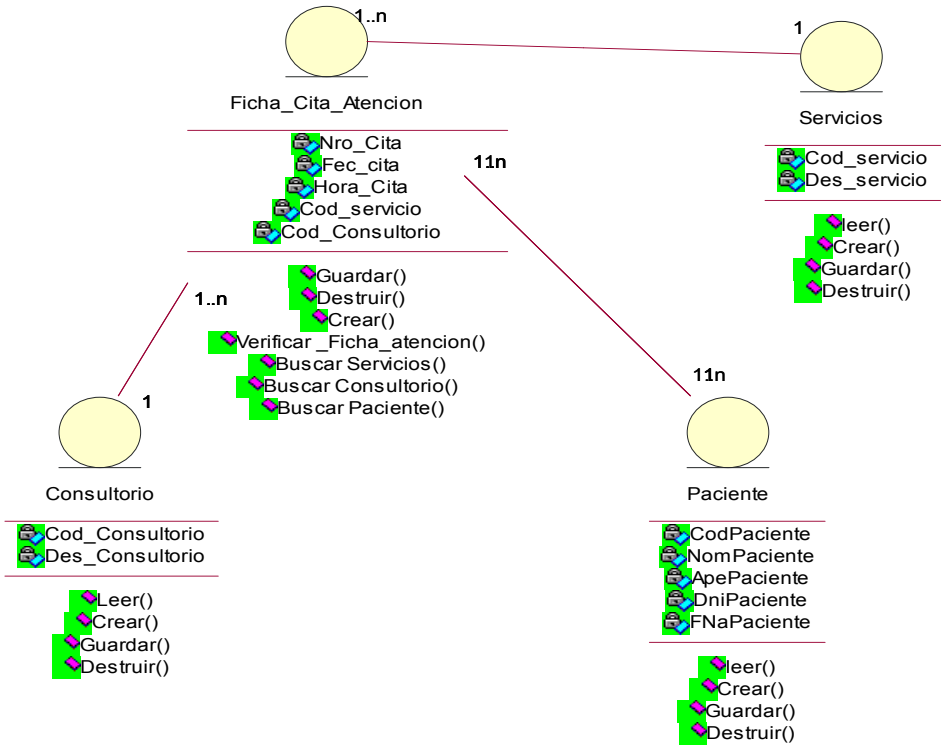


Figura 4.24: *Diagrama de* ***clases*** *para la realización de Citas de Atención*

Debemos hacer una realización del diseño por cada una de las realizaciones del análisis. Cada realización del diseño se realiza mediante un diagrama de secuencia (o colaboración) pero indicando mayor detalle que en el análisis. Luego identificamos los objetos que participan y obtendremos nuestro diagrama de Clases del Diseño.

c) Construir paquetes del diseño que agruparan a las Clases del Diseño

Como la arquitectura escogida es de tres capas el diseño reflejará eso.

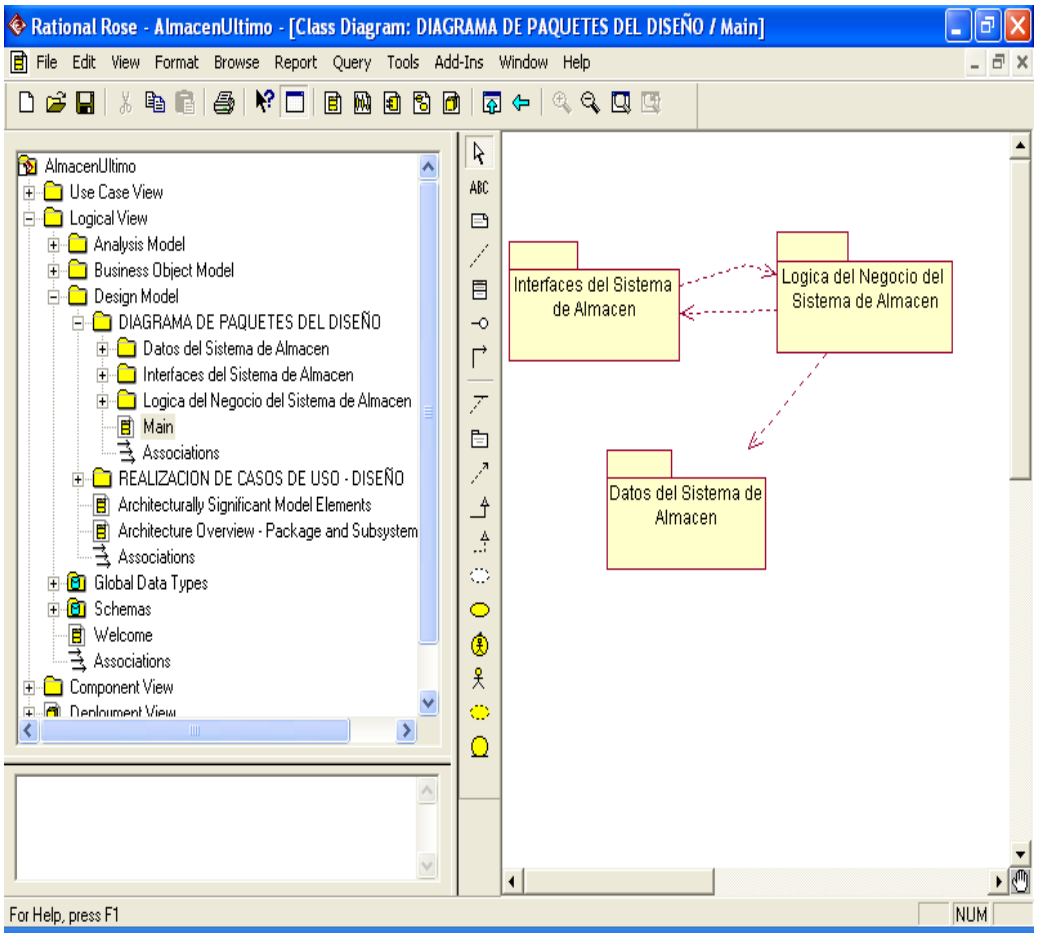
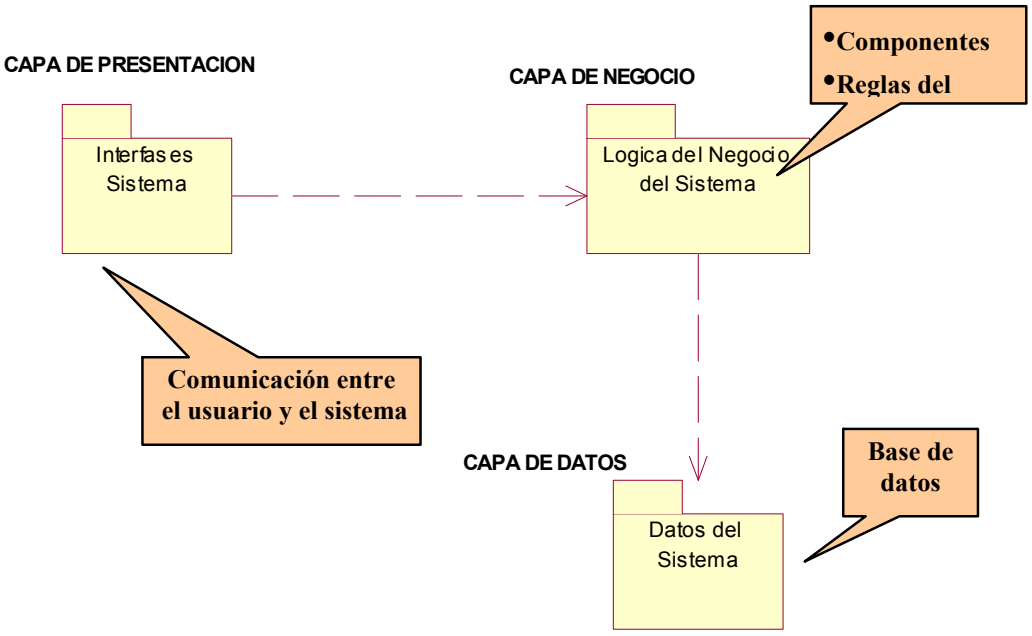


Figura 4.25: *Diagrama de paquetes del diseño*



d) Arrastrar las Clases del Diseño a sus respectivos paquetes.

e) El Modelo de Datos

Mapeo de la Base de Datos

Mapeo de Clases a Tablas
Mapeo de atributos a columnas
Mapeo de la Herencia en una base de datos relacional
Mapeo de relaciones de Uno a uno, uno a muchos, muchos a muchos
Mapeo de agregaciones y composiciones
Mapeo de Clase asociación

También podemos realizar el mapeo de varias clases a una única tabla

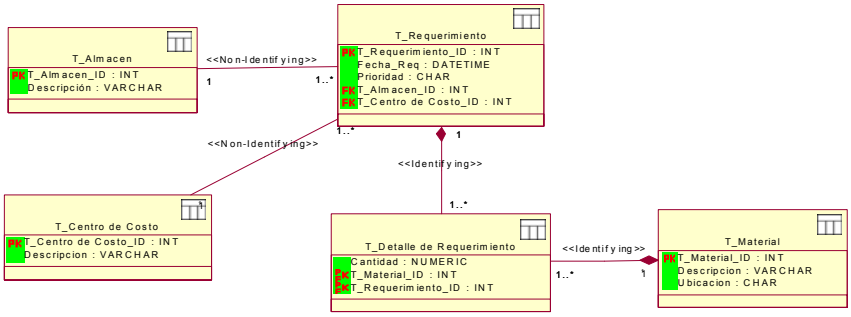


Figura 4.26: *Diagrama de clases*

Modelo físico de la Base de Datos (Hacia el modelo relacional)

- Crear el Componente de Base de Datos
  - Vista de Componentes:
  - Data Modeler / New/ Database

- Verificar que los Datos sean Persistentes

- En el Paquete de Datos
  - Data Modeler/ Transform to Data Model
  - Seleccionar el Nombre de la BD y el Schema creado

- En el Shema
  - Crear data Model Diagram
  - Data Modeler /New/ Datamodel Diagram

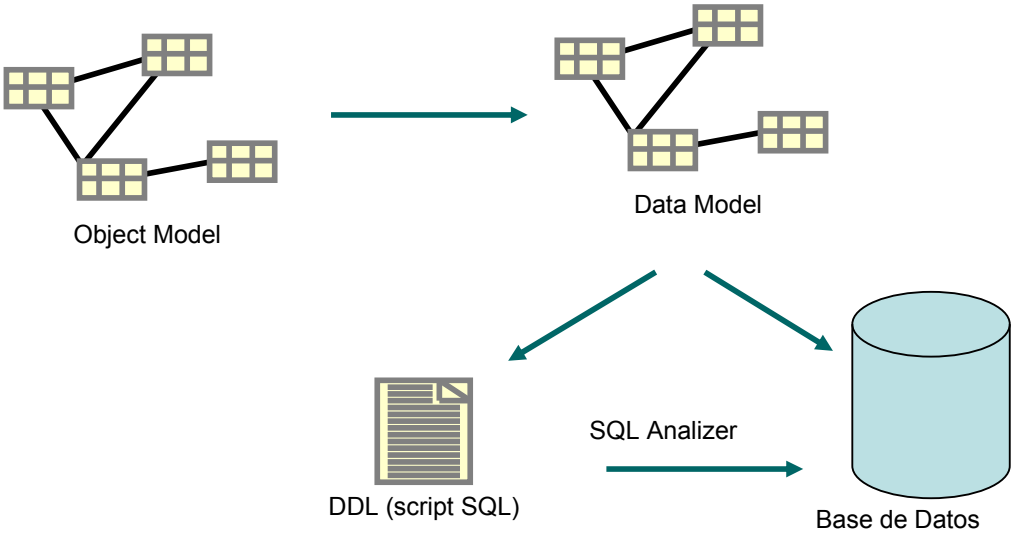


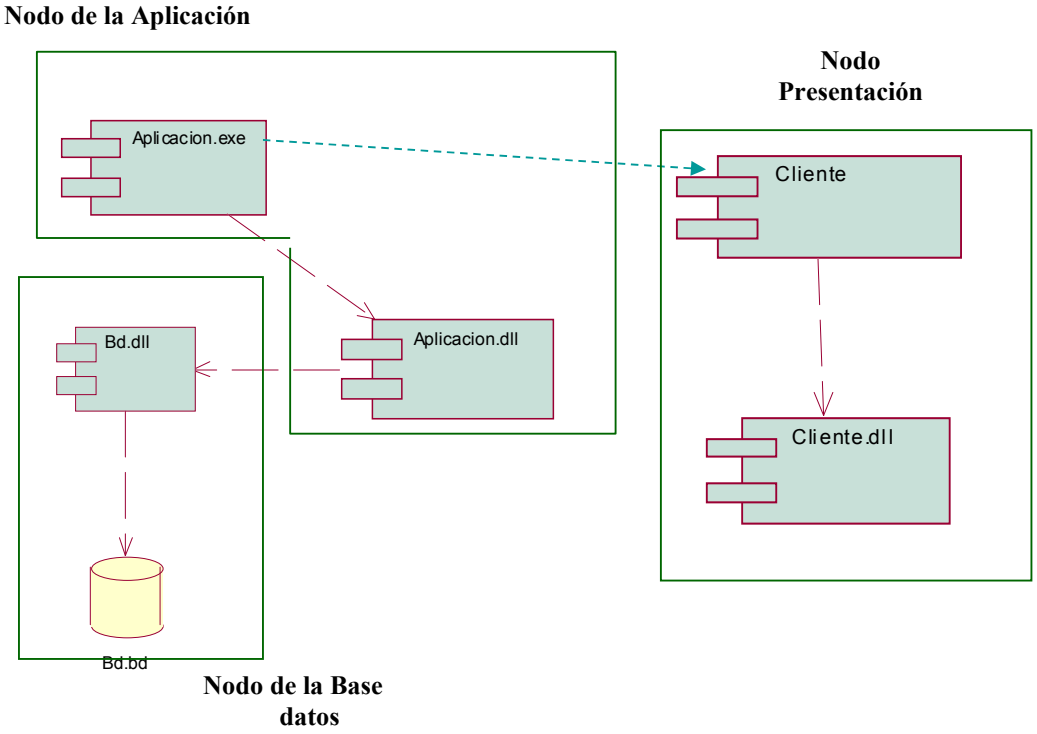
Figura 4.27: *Generación del modelo de datos*

4. Disciplinas y Artefactos: Modelado de Implementación

DISCIPLINA	ACTIVIDAD	DOCUMENTOS	MODELOS	DIAGRAMA
Implementación	• Estructurar el modelo de implementación • Plan de integración • Implementación de Componentes • Integrar cada subsistema • Integrar el Sistema		• Modelo de Implementación • Prototipo del Sistema	• Diagrama de Componentes

Tabla 4-4: *Disciplinas y artefactos del modelo de implementación*

Vista de Componentes



La capa de lógica del negocio: Generación de Código

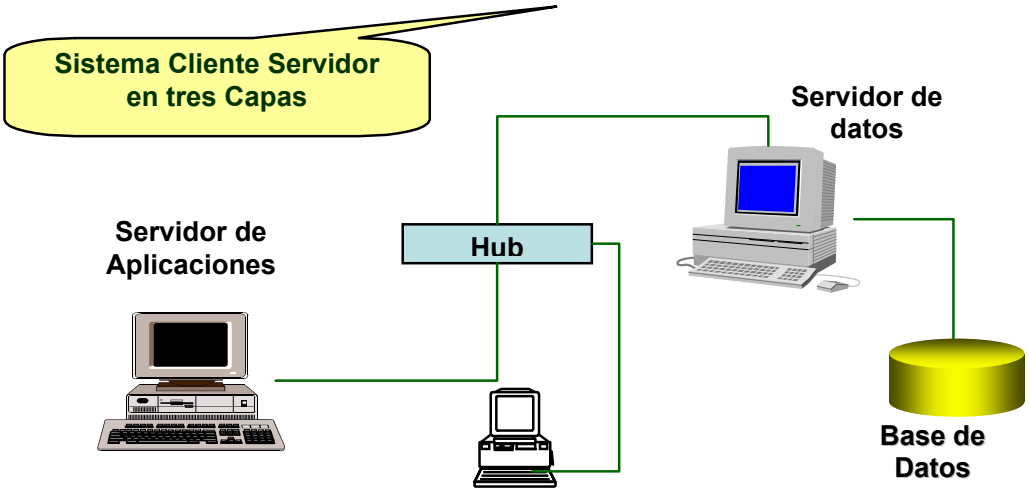
- Verificar el Modelo
- Crear un componente
- Asignar Clases a Componente
- Generar código

Disciplinas y Artefactos: Despliegue

DISCIPLINA	ACTIVIDAD	DOCUMENTOS	MODELOS	DIAGRAMA
Despliegue	• Planificar el Despliegue • Desarrollar el Material de Soporte • Administrar la aceptación de las pruebas • Producir las unidades de Desarrollo • Paquete del producto • Proveer accesos a sitios descargables • Evaluar los productos beta.	• Flujo de Eventos • ·Especificaciones suplementarias • Arquitectura del Software	• Manual de Instalación • Manual de Usuario	

Tabla 4-4: *Disciplinas y artefactos del modelo de implementación*

Vista de despliegue



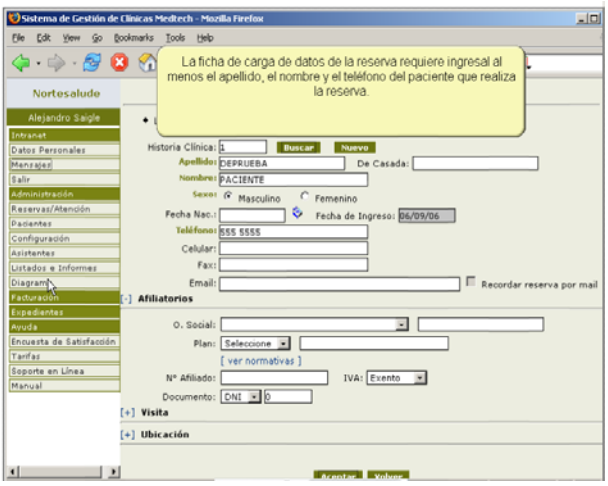
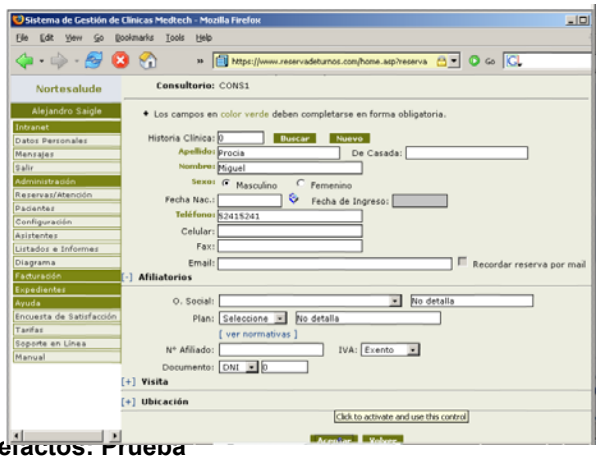
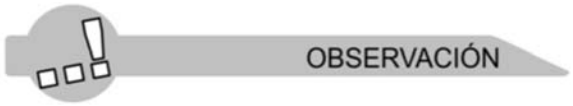


Figura 4.27: *Prototipo del sistema*



5. Disciplinas y Artefactos: Prueba

DISCIPLINA	ACTIVIDAD	DOCUMENTOS	MODELOS	DIAGRAMA
• Prueba	• Definir los objetivos de evaluación • ·Verificar el enfoque de la prueba • Validar la estabilidad de la construcción • Probar y evaluar el sistema • ·Establecer el nivel de aceptación de las pruebas • Evaluar los resultados de las pruebas		• Modelo de Pruebas	• Pruebas de caja Negra.



El ejemplo anterior se ha desarrollado con la ayuda de Rational Rose[23].
La elección del lenguaje de programación es libre.

1. Investigue acerca del modelado para aplicaciones web con RUP y elabore un informe resaltando las actividades necesarias para tal fin
2. Desarrollar un sistema para una empresa del medio donde pueda llevar a cabo todo el proceso de desarrollo de software con RUP

El Proceso Unificado de Rational (Rational Unified Process en inglés, habitualmente resumido como RUP) es un proceso de desarrollo de software y junto con el Lenguaje Unificado de Modelado UML, constituye la metodología estándar más utilizada para el análisis, implementación y documentación de sistemas orientados a objetos. RUP es en realidad un refinamiento realizado por Rational Software del más genérico Proceso Unificado.

En el ejemplo se ha puesto especial énfasis en el modelado del negocio, la obtención de requerimientos y el proceso de análisis y diseño del sistema

[1] JACABOSON, I., BOOCH, G., RUMBAUGH J., **El Proceso Unificado de Desarrollo de Software**, Ed. Addison Wesley 2000.

Bibliografía electrónica:

- Ejemplo de desarrollo software con RUP:

[23] IBM Rational Rose Enterprise es uno de los productos más completos de la familia Rational Rose. Todos los productos de Rational Rose dan soporte a Unified Modeling Language (UML),

http://www.dsic.upv.es/asignaturas/facultad/lsi/ejemplorup/

- Desarrollo basado en RUP con Rational Rose: http://lml.ls.fi.upm.es/mdp/si/
- Tutorial RUP : requisitos, análisis, diseño
  http://www.dsi.uclm.es/asignaturas/42541/Practicas/Requisitos.pdf
  http://www.dsi.uclm.es/asignaturas/42541/Practicas/Analisis.pdf
  http://www.dsi.uclm.es/asignaturas/42541/Practicas/Disenyo.pdf

En la siguiente unidad temática trataremos aspectos relacionados a la gestión de proyectos de software

INGENIERIA DEL SOFTWARE
UNIDAD ACADÉMICA Nº 4

NOMBRE:_____

APELLIDOS:_____FECHA; ____/_____/_____

CIUDAD:_____SEMESTRE:_____

1. CASO DE ESTUDIO: Hotel

El dueño de un hotel le pide a usted desarrollar un programa para consultar sobre las habitaciones disponibles, reservar habitaciones de su hotel y ofrecer otros servicios asociados.

El hotel posee tres tipos de habitaciones: simple, doble y matrimonial, y los tipos de clientes pueden ser nacionales o extranjeros. Una reservación almacena datos del cliente, de la habitación reservada, la fecha de inicio de hospedaje y el número de días que será ocupada la habitación.

El recepcionista del hotel debe poder hacer las siguientes operaciones:

- Obtener un listado de las habitaciones disponibles de acuerdo a su tipo
- Preguntar por el precio de una habitación de acuerdo a su tipo
- Preguntar por el descuento ofrecido a los clientes habituales (es necesario saber quienes son los clientes habituales y quienes los esporádicos)
- Obtener listados de otros servicios que ofrece el hotel (lavandería, cochera, restaurant) y sus respectivos precios
- Preguntar por el precio total del servicio para un cliente dado
- Dibujar en pantalla la foto de un habitación de acuerdo a su tipo
- Reservar una habitación
- Eliminar una reserva

El administrador puede usar el programa para:

- Cambiar el precio de una habitación de acuerdo a su tipo
- Cambiar el precio de otros servicios ofrecidos por el hotel
- Cambiar el valor del descuento ofrecido a los clientes habituales
- Calcular las ganancias que tendrán en un mes especificado

El diseño a desarrollar debe facilitar la extensibilidad de nuevos tipos de habitación o clientes y a su vez permitir agregar nuevos servicios y generar nuevas consultas.

PARA EL CASO:

- Elaborar el modelo del negocio
- Obtener la captura de requerimientos
- Elaborar los modelos de análisis y diseño del sistema

Unidad Académica V

GESTION DE PROYECTOS DE SOFTWARE

En la realización de cualquier proyecto de desarrollo de software, el director del proyecto tiene una importancia vital ya que será responsable de que el proyecto se encamine adecuadamente encargándose así de proveer del personal necesario, y de tomar las decisiones que ayuden a que el proyecto cumpla con los objetivos propuestos. Es por esta razón que es muy importante que quien se encargue de la dirección del proyecto este familiarizado con la gestión de proyectos y todas las técnicas y herramientas que la componen, y que le facilitarán su labor al trabajar en un proyecto.

En esta unidad intentaremos explicar algunas de las principales directrices y herramientas que componen la Gestión de Proyectos y que ayudarán al director a tomar de una manera más precisa las decisiones para el proyecto.

Al finalizar el estudio de la presente unidad temática el estudiante:

- Clasifica y relaciona las actividades de la gestión de proyectos con sus respectivas áreas de conocimiento
- Conoce las actividades que involucra la gestión de proyectos
- Entiende la importancia del rol que desempeña el director del proyecto
- Entiende el rol que desempeñan las herramientas CASE en el desarrollo de un proyecto de software

5.1 ¿Qué es un Proyecto?

Un proyecto es una empresa temporal que se asume con el fin de crear un producto o servicio único.

Temporal quiere decir que cada proyecto tiene un comienzo y un término definitivos. Único quiere decir que el producto o servicio es distintivamente diferente de todos los demás productos o servicios. Para muchas organizaciones, los proyectos son una forma de responder a aquellas solicitudes que no se pueden abordar dentro de los límites operacionales normales de la organización.

Los proyectos se llevan a cabo a todo nivel de la organización. Estos pueden involucrar a una sola persona o bien a varios miles de individuos. Su duración va de unas cuantas semanas a más de cinco años. Los proyectos pueden involucrar a una sola unidad de una organización o bien pueden traspasar las fronteras organizacionales, en la forma de sociedades contractuales (joint ventures[24]) y sociedades. Los proyectos son críticos para el cumplimiento de la estrategia de negocios de la organización que los ejecuta, debido a que los proyectos son una forma de implementar la estrategia.

5.2 Las Áreas de Conocimiento de la Gestión de Proyectos

Las Áreas de Conocimiento de la Gestión de Proyectos, describe el conocimiento y la práctica de la gestión de proyectos en términos de sus procesos integrados. Estos procesos se han

[24] Join Venture: Asociación temporal de empresas para acceder a un proyecto determinado

organizado en nueve áreas de conocimiento, como se describe más abajo y se ilustran en la Figura 5.1

a) **Gestión de Integración de Proyectos**, describe los procesos requeridos para asegurar que se coordinen adecuadamente los distintos elementos del proyecto. Esta consiste en el desarrollo de un plan de proyecto, la ejecución del plan de proyecto y en el control integrado de cambios.

b) **Gestión del Alcance del Proyecto**, describe los procesos requeridos para asegurar que el proyecto incluya todo el trabajo requerido, y sólo el trabajo requerido, a fin de completar el proyecto exitosamente. Esta consiste en la iniciación, planificación del alcance, definición del alcance, verificación del alcance y control de cambios en el alcance.

c) **Gestión de Duración (Tiempo) del Proyecto**, describe los procesos requeridos para asegurar el término a tiempo del proyecto. Esta consiste en la definición de las actividades, la secuencia de las actividades, estimación de la duración de las actividades, desarrollo del programa y control del programa.

d) **Gestión de Costos del Proyecto**, describe los procesos requeridos para asegurar la ejecución total del proyecto dentro del presupuesto aprobado. Esta consiste en la planificación de los recursos, estimación de los costos, preparación de presupuestos de costos y control de costos.

e) **Gestión de Calidad del Proyecto**, describe los procesos requeridos para asegurarse de que el proyecto satisfará las necesidades para las cuales fue ejecutado. Esta consiste en la planificación de la calidad, aseguramiento de la calidad y control de calidad.

f) **Gestión de Recursos Humanos del Proyecto**, describe los procesos requeridos para realizar un uso más eficiente y eficaz de las personas involucradas con el proyecto. Esta consiste en la planificación organizacional, la adquisición de personal, y en el desarrollo del equipo.

g) **Gestión de Comunicaciones del Proyecto**, describe los procesos requeridos para asegurar la generación, recopilación, diseminación, almacenamiento y disposición final de la información del proyecto en forma adecuada y a tiempo. Esta consiste en la planificación de las comunicaciones, distribución de la información, reporte del rendimiento/desempeño y cierre administrativo.

h) **Gestión de Riesgos del Proyecto**, describe los procesos que tienen que ver con la identificación, análisis y respuesta al riesgo del proyecto. Esta consiste en la planificación de la gestión de riesgos, identificación de los riesgos, análisis cualitativo de los riesgos, análisis cuantitativo de los riesgos, planificación de las respuestas a los riesgos, y monitoreo y control de los riesgos.

i) **Gestión de Abastecimiento de Proyectos**, describe los procesos requeridos para adquirir bienes y servicios desde fuera de la organización ejecutante. Esta consiste en la planificación de la adquisición, planificación del requerimiento, requisición, selección de la fuente, administración del contrato y término del contrato.

3. ¿Qué es la gestión de proyectos de software?

La gestión de proyectos es el proceso por el cual se planifica, dirige y controla el desarrollo de un sistema aceptable con un costo mínimo y dentro de un período de tiempo especifico.
La gestión de proyectos involucra la aplicación del conocimiento, habilidades, herramientas y técnicas a las actividades del proyecto de forma tal que pueda cumplirse con los requerimientos del proyecto.

La gestión de proyectos se lleva a cabo mediante el uso de procesos tales como: iniciación, planificación, ejecución, control y término. El equipo del proyecto gestiona el trabajo de los proyectos, trabajo que comúnmente implica:

- Distintas demandas de: alcance, tiempo, costo, riesgo y calidad.
- Clientes con diferentes necesidades y expectativas.
- Requerimientos identificados.

Es importante hacer notar que muchos de los procesos contenidos dentro de la gestión de proyectos son iterativos por naturaleza. Esto se debe, en parte, a la existencia de y a la necesidad de una elaboración progresiva de un proyecto durante todo su ciclo de vida.

La gestión de proyectos de software tiene por finalidad:

- Estimar que sucederá en un proyecto nuevo
- Analizar que sucedió en un proyecto ya finalizado

En todos los casos, se tratara de dar respuesta cuantitativas a:

- ¿Cuál será el plazo de entrega?
- ¿Cuántas personas necesito?
- ¿Cuánto costará el proyecto?

La gestión de proyectos de software es una rama especializada de la Ingeniería de Software que:

- Emplea metodologías bien definidas
- Realiza medidas repetibles y confiables
- Estima costos y tiempos
- Da elementos para la gestión de proyectos
- Replantea resultados para ajustar la información disponible

4. Tipos de Proyectos

En proyectos de software es importante diferenciar entre tres tipos de proyectos desde el punto de vista de su gestión, pues siempre debe tenerse en cuenta que los tres casos presentan situaciones diferentes y no pueden ser tratados por igual.
Tenemos los siguientes tipos de proyectos:

- **Proyectos nuevos**: Se busca analizar costos, tiempos y cantidad de personas. Es el caso más difícil de todos

- **Replanteo de proyectos viejos:** Se busaca afinar las metodologías de estimación. Es la principal fuente de información

- **Extensiones:** O ampliaciones de un proyecto existente. Es un caso intermedio donde se desea tener buena precisión de plazos y costos.

5. Tamaño de los Proyectos:

Los proyectos de software son diferentes por la sola razón de su tamaño. Existen tres categorías diferenciadas de proyectos, con problemas diferentes cada una:

- **Proyectos pequeños:** Consisten solamente en implementación. No tienen costos indirectos importantes.
  Los proyectos pequeños poseen menos de un año de tiempo de desarrollo, menos de 25 meses – persona de esfuerzo total, menos de tres personas en el equipo de trabajo.

- **Proyectos grandes**: Poseen además de implementación, gerencia de proyecto, control de calidad, capacitación de personal, hay un plan de mantenimiento, hay documentación importante para uso externo e interno. Se genera información para mercadeo.
  Los proyectos grandes poseen más de tres años de tiempo de desarrollo, menos de 100 meses – persona de esfuerzo total, más de diez personas en el equipo de trabajo.

- **Proyectos medianos**: Es un caso intermedio entre los otros dos.

Un error clásico en la historia de la gestión de proyectos fue no advertir la existencia de categorías diferentes de proyectos de software. Pensar que la información o la experiencia adquirida en proyectos pequeños pueden servir para proyectos grandes es uno de los orígenes de resultados catastróficos en la gestión de proyectos.

Un equipo de trabajo de menos de tres personas puede trabajar con una documentación informal, un equipo de mas de 10 personas, durante varios años, no puede confiar ni en los contactos personales ni en la memoria de la gente. Es mas, puede ocurrir que muchos de los que comienzan el proyecto sean reemplazados por otros. Estas diferencias en el manejo interno de la información hacen una gran distinción entre los dos extremos de tamaños de proyectos.

Es muy importante observar que la barrera de tres años es una barrera dramática para la informática. En tres años hay cambios importantes en las plataformas. De acuerdo con la Ley de Moore, la capacidad de almacenamiento y la velocidad de las plataformas se multiplicaran en tres años. Tenemos un orden de magnitud de crecimiento. Esto implica, para un proyecto grande, que:

- La plataforma para la cual diseñamos será obsoleta en el momento de entrega
- La disponibilidad de memoria, la velocidad de procesamiento y el tamaño del almacenamiento, que al comenzar el proyecto puede ser una limitación del diseño, puede dejar de serlo al final del proyecto.
- Es posible que aparezcan nuevas tecnologías y nuevas herramientas de trabajo con los cuales no se contaba al comenzar el proyecto
- Es posible que aparezcan nuevas exigencias con las cuales no se contaban al comenzar el proyecto

Estas consideraciones muestran que hay diferencias profundas en el armado y realización de un proyecto grande.

6. Inicio de un proyecto.

Las empresas emprenden proyectos por una o más de las siguientes razones:

a) Mejorar la capacidad de la empresa:

Las actividades de la empresa están influenciadas por la capacidad de esta para procesar transacciones con rapidez y eficiencia. Los sistemas de información mejoran esta capacidad en tres formas estas son:

- Aumento de la velocidad de procesamiento.
- Permiten el manejo de un volumen creciente de transacciones.
- Recuperan con rapidez la información.

b) Integrar las funciones individuales

La falta de comunicación es una fuente común de dificultades que afectan a todos los que laboran en una empresa. Sin embargo, los sistemas de información bien desarrollados tratan de ampliar la comunicación y facilitan la integración de funciones individuales.

c) Control de los costos de Operación en la empresa

Muchas empresas han desaparecido y muchas otras se encuentran imposibilitadas para alcanzar el éxito debido al poco control sobre los costos o por el total desconocimiento para el control de estos. Los sistemas de información juegan un papel importante tanto con el control como en la reducción de los costos de operación.

d) Lograr ventaja competitiva:

Los sistemas de información son un arma estratégica que puede cambiar la forma en como compite la empresa en el mercado. Los sistemas de información mejoran la organización y ayudan a la empresa a ser más competitiva.
Una empresa puede ganar ventaja competitiva a través de su sistema de información en cuatro formas diferentes que garantizan la competitividad en el mercado estos son: clientes, competidores, proveedores y servicios.

7. Formulación Básica del equipo de trabajo.

Todo proyecto de sistemas de información debe ser desarrollado bajo las actividades de un grupo de trabajo que se haga responsable del inicio y culminación del sistema de información.

El grupo de trabajo va a depender del tamaño del proyecto que va a desarrollarse. Sin embargo, todo grupo de trabajo debe contar por lo menos con un líder de proyecto, un analista de sistemas y un programador o programadores.

Director o Líder de proyecto.

Un líder de proyecto es la persona encargada de aprobar la propuesta o solicitudes de proyectos a llevarse a cabo, el líder del proyecto se encarga de entregar la propuesta una vez aprobada por el, a los directivos con el fin de obtener los recursos ya sea económicos y materiales para el inicio del proyecto. El líder de proyecto se compromete a entregar el sistema de información en la fecha pactada, el líder de proyecto administra las actividades del grupo de trabajo, verifica y controla las actividades que han sido calendarizadas por él y el analista de sistemas para llevar a cabo una buena administración de proyectos.

Analista de sistemas.

Es el encargado de hacer todo lo concerniente al análisis del problema apoyándose en la aplicación de cuestionarios, entrevistas y observación directa para determinar las causas concretas del problema y poder proponer soluciones adecuadas a este.

El analista de sistemas realiza el bosquejo de la base de datos, el desarrollo de procedimientos y algoritmos, la interfaz del usuario, etc para después ofrecer toda esta información al programador para su posterior codificación.

Programador.

Es la persona encargada de la codificación de procedimientos y algoritmos que fueron entregados por el analista de sistemas. Cabe mencionar que el programador mantiene una estrecha comunicación con el analista de sistemas, ya que el analista de sistemas apoya y asesora al programador en la codificación de los módulos del sistema de información.

El programador además de codificar también tiene a su cargo la prueba de los módulos codificados con el fin de encontrar errores lógicos y físicos para su resolución. Una vez realizada las pruebas a los módulos ya codificados y resuelto los errores tanto lógicos como físicos y sabiendo de antemano que todo funciona correctamente; el programador tiene a su cargo la elaboración del manual del sistema con la asesoría del analista de sistemas; el cual contendrá toda la información de como fue desarrollado el sistema. Además de esto debe elaborarse otro manual que guiará al usuario acerca del manejo del software que se implantará.

8. Funciones básicas del Director de Proyectos

El director de proyectos no es simplemente un analista experimentado que se haga cargo del proyecto, sino más bien debe aplicar un conjunto de técnicas y conocimientos diferentes de los que aplica un analista. Las funciones básicas de un director o un jefe de proyectos han sido analizadas y diseccionadas por teóricos de la gestión durante muchos años. Entre estas funciones, se incluyen la planificación, la selección de personal, la organización, la definición de calendarios, la dirección y el control.

Planificación de las tareas de proyecto y selección del equipo de proyectos

Un buen director siempre tiene un plan. El director evalúa las necesidades de recursos y formula un plan para llegar al sistema objeto. Ello se basa en el conocimiento que tiene el director de los requisitos del sistema objeto en cada momento del desarrollo. Un plan básico para el desarrollo de un sistema de información es el suministrado por el ciclo de vida del desarrollo de sistemas. Muchas empresas tienen su propio ciclo de vida estándar, y algunas de ellas tienen también normas sobre métodos y herramientas que han de usarse.
Así ha de planificarse cada una de las tareas requeridas para completar el proyecto:

- ¿Cuánto tiempo se requerirá?
- ¿Cuántas personas serán necesarias?
- ¿Cuánto costara la tarea?
- ¿Qué tareas deben terminarse antes de empezar otras?
- ¿Pueden solaparse algunas de ellas?

Los directores de los proyectos son, frecuentemente, los encargados de seleccionar a los analistas y los programadores de un equipo de proyecto. El director de proyectos debería tener muy en cuenta los conocimientos técnicos y de empresa que pueden ser necesarios para terminar un proyecto con éxito. La clave de esta misión es saber elegir adecuadamente a las personas que habrían de desarrollar las tareas requeridas e identificada como parte de la planificación de proyecto.

Organización y definición de calendarios para el proyecto

Dados el plan y el equipo de proyecto, el director de proyecto de proyecto es el responsable de la organización y la definición del calendario del mismo. Los miembros del equipo de proyecto deberán conocer su cometido y sus responsabilidades concretas, así como su relación de dependencia con el respecto al director de proyecto.
El calendario de proyecto debería desarrollarse con un conocimiento preciso de los requisitos de tiempo, las asignaciones de personal y las dependencias de unas tareas con otras. Muchos proyectos tienen un límite a la fecha de entrega solicitada. El director del proyecto debe determinar si puede elaborarse un calendario factible basado en dicha fecha. Si no fuera así, debería retrasarse el límite o reajustarse el ámbito del proyecto.

Dirección y control del proyecto

Una vez iniciado el proyecto, el director del proyecto se convierte en su máximo responsable. Como tal, dirige las actividades del equipo y hace evaluaciones del

avance del proyecto. Por consiguiente, todo director de proyectos debe demostrar ante su equipo cualidades de dirección, como son saber motivar, recompensar, asesorar, coordinar, delegar funciones y reconocer el trabajo de los miembros de su equipo. Además, el director debe informar frecuentemente del avance del proyecto ante sus superiores.

Tal vez, la función más difícil e importante del director sea controlar el proyecto. Pocos planes hay que puedan llevarse a la practica sin problemas y retrasos. La labor del director de proyectos es hacer un seguimiento de las tareas, los plazos, los costes y las expectativas, con el fin de controlar todos estos elementos. Si el ámbito del proyecto tiende a crecer, el director del mismo debe tomar una decisión: ¿habría que reducir el ámbito del proyecto para respetar el presupuesto y los plazos, o revisar dicho presupuesto y dichos plazos? El director del proyecto debería ser capaz de presentar alternativas, y sus implicaciones, a los plazos y presupuestos para saber responder a las expectativas.

9. Causas de proyectos fallidos por la gestión de proyectos.

Dentro de las principales causas por las que puede fallar un proyecto, se encuentra el hecho de que los analistas no respetan o no conocen bien las herramientas y las técnicas del análisis y diseño de sistemas, además de esto puede haber una mala gestión y dirección del proyecto. Además existen una serie de factores que pueden hacer que el sistema sea mal evaluado, entre estas están:

- Necesidades no satisfechas o no identificadas
- Cambio no controlado del ámbito del proyecto
- Exceso de costo
- Retrasos en la entrega

Aunque estos factores pueden influir de manera muy trascendente en la falla de un proyecto, generalmente están acompañados de otro tipo de falencias.

Pero ¿Cuáles de estos errores de gestión de proyectos ocasionan que no se cumplan los requisitos, que se sobrepase los tiempos de entrega o se aumenten repetidas veces los costos?

La respuesta a esta pregunta puede ser hallada en dos fuentes principalmente:

- Deficiencias en las herramientas y las técnicas de análisis y diseño de sistemas
- Mala gestión de los proyectos.

En el caso de las necesidades no satisfechas o no identificadas, el error puede aparecer debido a que se omiten datos durante el desarrollo del proyecto, es por esto que es muy importante no saltar ninguna etapa del ciclo de vida del desarrollo de sistemas.

Otra causa de insatisfacción de necesidades es la mala definición de las expectativas de un proyecto en sus orígenes, ya que si no están bien definidos los requerimientos máximos y mínimos que el proyecto debe satisfacer desde el comienzo, los desarrolladores verán afectados su trabajo por el síndrome de las necesidades que crecen el cual les dejara hacer cambios en el proyecto en cualquier momento sin detenerse a pensar si esos cambios serán buenos para el proyecto como un todo; por supuesto todos estas modificaciones acarrearan alteraciones en los costos y en los tiempos de entrega

El costo que un proyecto involucra puede aumentar durante el desarrollo de este debido a que para comenzar un proyecto generalmente se exige una estudio de viabilidad en el cual no se incluyen datos completamente precisos de la cantidad de recursos que cada tarea consumirá, y es en base a este estudio que se hacen estimaciones de los recursos totales que el proyecto va a necesitar.

Además el costo puede aumentar por el uso de criterios de estimación poco eficientes por parte de los analistas.

Otro factor que puede aumentar los costos es el aumento en los tiempos de entrega que generalmente se debe a que los directores del proyecto no son buenos gestionando los tiempos de entrega de cada una de las diferentes tareas que el proyecto involucra, es así que cuando tienen un retraso no son capaces de alterar los plazos de entrega finales creyendo que

podrán recuperar el tiempo perdido, en general esta es una muy mala política de trabajo porque no siempre es posible acelerar otras tareas para ahorrar tiempo en la entrega final.

10. Herramientas y técnicas de gestión de proyectos

Todas las fases de desarrollo de sistemas de información involucran muchos tipos de actividades diferentes que juntos forman un proyecto. El líder del proyecto debe administrar el proyecto cuidadosamente para que llegue a ser un proyecto exitoso. La administración de proyectos involucra todas las tareas generales de planeación y control.

La planeación incluye la selección de un equipo y la asignación de los miembros del equipo a las actividades adecuados, la estimación del tiempo requerido para completar cada tarea y la calendarización del proyecto para que las actividades sean terminadas en forma ordenada.

Gráficos PERT

PERT, que significa Project Evaluation and Rewiev Technique (técnica de evaluación y revisión de proyectos), fue desarrollado a finales de la década de los '50 para planear y controlar los grandes proyectos de desarrollo armamentístico del ejercito estadounidense. Fue desarrollado para evidenciar la interdependencia de las tareas de los proyectos cuando se realiza la planificación de los mismos. En esencia, PERT es una técnica de modelos gráficos interrelacionados.

Estimación de los requisitos de tiempo del proyecto y elaboración de un PERT

Antes de dibujar un graficar un grafico PERT, debe hacerse una estimación del tiempo requerido por cada tarea del proyecto. El grafico PERT puede utilizarse para indicar los tiempos máximos y mínimos para la finalización de las tareas. Aunque estos tiempos se expresan a menudo en forma de personas-día, no es recomendable este planteamiento. No existe ninguna prueba de que exista dependencia lineal entre el tiempo de terminación de un proyecto y el número de personas asignadas al equipo del proyecto. Muchos proyectos de sistemas que se entregaron con retraso aumentaron más su desviación en los plazos cuando se añadieron mas personas al equipo de proyecto. Por el hecho de que dos personas hagan un trabajo en cuatro días no puede suponerse que cuatro personas lo hagan en dos días. Por esto es mejor que se exprese este tiempo en días de calendario para un número dado de personas asignadas por tarea.
Los requisitos de tiempo de los proyectos deben ser calculados por estimación. Un buen director de proyectos analista de sistemas se basa en sus datos y su experiencia en otros proyectos anteriores. Existen productos CASE, como SPQR/20 que pueden ayudar a los directores de proyectos a realizar mejores estimaciones de tiempo.

Otras organizaciones han puesto en práctica normas internas para calcular las estimaciones de tiempo de los proyectos de una forma mas estructurada. Estas normas pueden suponer tener que analizar las tareas en función de su dificultad, de los conocimientos y técnicas necesarios y de otros factores identificables. Alternativamente, podría hacerse una estimación optimista y después ajustarse usando factores de peso a diversos criterios, como el tamaño del equipo, el numero de usuarios finales con los que se tiene que trabajar, la disponibilidad de dichos usuarios finales, y así sucesivamente. Cada factor de peso puede tanto aumentar como reducir el valor de la estimación.

Uso de PERT para planificación y control

El uso y las ventajas principales del gráfico PERT se derivan de su capacidad para asistir al director de proyectos en la planificación y el control de los mismos. En la planificación, el gráfico PERT sirve de ayuda para determinar el tiempo estimado requerido para completar un proyecto dado, obteniendo fechas reales para el proyecto y asignando los recursos necesarios.

Como herramienta de control, el gráfico PERT ayuda al director a identificar los problemas actuales y potenciales. Debe ponerse especial atención en el camino crítico de un proyecto. Cuado un director de un proyecto detecta que una tarea crítica va con retraso, deberán plantearse diversas alternativas de acción. Podrán entonces tomarse medidas correctivas, como la redistribución de recursos humanos. Estos recursos probablemente se obtendrán de tareas no críticas que en la actualidad marchen correctamente. Estas tareas no críticas ofrecen al proyecto un cierto tiempo muerto disponible.

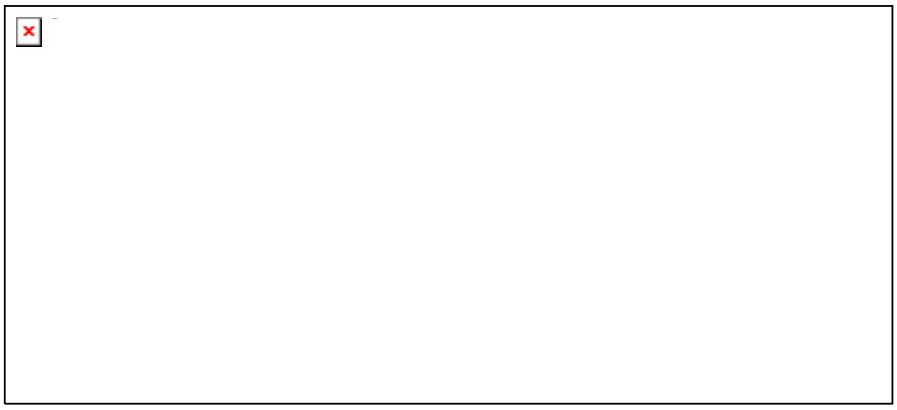


Figura 4.27: *Grafico PERT. Los círculos denotan actividades y las flechas relaciones entre ellas*

Graficos De Gantt

Este fue desarrollado por Henry L.Gantt en 1917 y es una sencilla herramienta de gráficos de tiempos, ya que son fáciles de aprender, leer y escribir. Estos resultan bastante eficaces para la planificación y la evaluación del avance de los proyectos.
Los gráficos Gantt se basan en un enfoque gráfico. Un grafico de Gantt es un sencillo gráfico de barras. Cada barra simboliza una tarea del proyecto. En donde el eje horizontal representa el tiempo. Como estos gráficos se emplean para encadenar tareas entre sí, el eje horizontal debería incluir fechas. Verticalmente, y en la columna izquierda, se ofrece una relación de las tareas.
Una ventaja importante de los gráficos Gantt es que ilustran claramente el solapamiento entre tareas planificadas. A diferencia con los gráficos PERT los gráficos Gantt no muestran bien la dependencia que existe entre tareas diferentes.

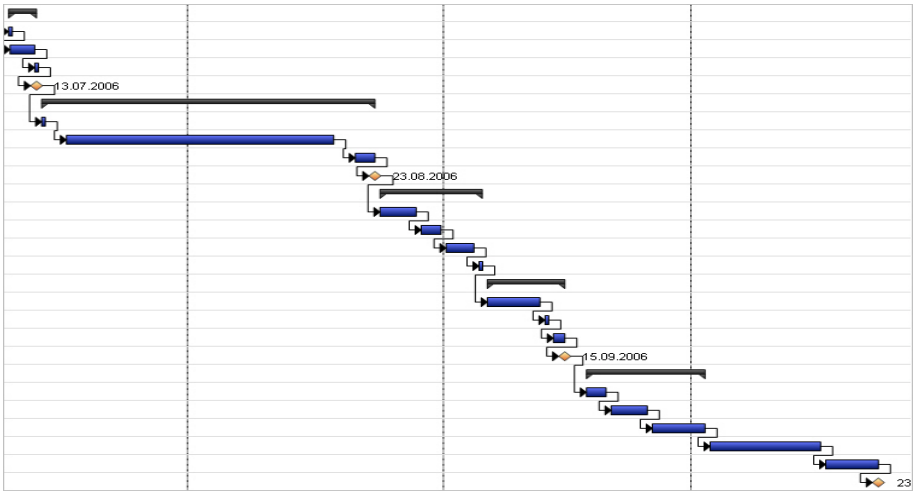
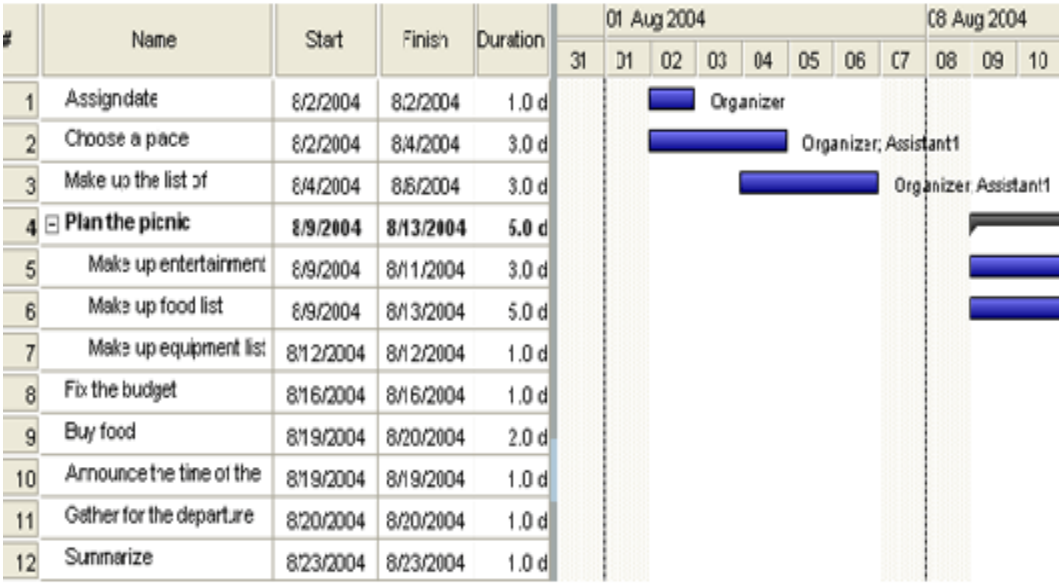


Figura 5.2: *Diagramas de Gantt*



#	Name	Start	Finish	Duration
1	Assign date	8/2/2004	8/2/2004	1.0 d
2	Choose a pace	8/2/2004	8/4/2004	3.0 d
3	Make up the list of	8/4/2004	8/6/2004	3.0 d
4	**Plan the picnic**	**8/9/2004**	**8/13/2004**	**5.0 d**
5	Make up entertainment	8/9/2004	8/11/2004	3.0 d
6	Make up food list	8/9/2004	8/13/2004	5.0 d
7	Make up equipment list	8/12/2004	8/12/2004	1.0 d
8	Fix the budget	8/16/2004	8/16/2004	1.0 d
9	Buy food	8/19/2004	8/20/2004	2.0 d
10	Announce the time of the	8/19/2004	8/19/2004	1.0 d
11	Gather for the departure	8/20/2004	8/20/2004	1.0 d
12	Summarize	8/23/2004	8/23/2004	1.0 d

Cómo usar un gráfico de Gantt para planificación:

Para generar un calendario de proyecto utilizando gráficos Gantt, primero se tiene que identificar las tareas que deben planificarse. A continuación, se determinara la duración de cada tarea a través de técnicas y formulas para la estimación apropiada de tiempos. Si ya se ha preparado un grafico PERT ya se habrían identificado las tareas y deberían al menos determinarse las dependencias mutuas entre tareas, ya que los gráficos Gantt no muestran claramente estas dependencias, pero es imperativo que el calendario de planificación las reconozca.

Entonces estamos preparados para planificar tareas.
Primero, se escribe la lista de actividades en la columna de la izquierda del gráfico Gantt. Las fechas correspondientes a la duración del proyecto se anotan en el eje horizontal del gráfico. Habrán de determinarse fechas de inicio y fin de cada tarea, fijándose bien en las dependencias parciales o totales de entre tareas.

Uso de gráficos de Gantt para evaluar el avance de proyecto:

Una de las responsabilidades más habituales del director de proyectos es informar sobre el avance del proyecto a sus superiores. Los gráficos Gantt suelen utilizarse para mostrar el avance de los proyectos, en virtud de que pueden comparar de forma conveniente la planificación original con el desarrollo real. Para informar del avance del proyecto se tiene que ampliar las convecciones propias del gráfico de Gantt. Si una tarea ha sido completada, su barra correspondiente aparecerá más oscura. Si ha sido completada solo parcialmente, la parte proporcional de la barra estará más oscura. El porcentaje de barra oscurecida debería corresponder al porcentaje de tarea completa. Las barras más claras simbolizan tareas que no han sido empezadas. A continuación, se trazara una línea vertical perpendicular al eje horizontal y que cortará a éste en la fecha del día. Entonces, se puede evaluar el avance del proyecto.

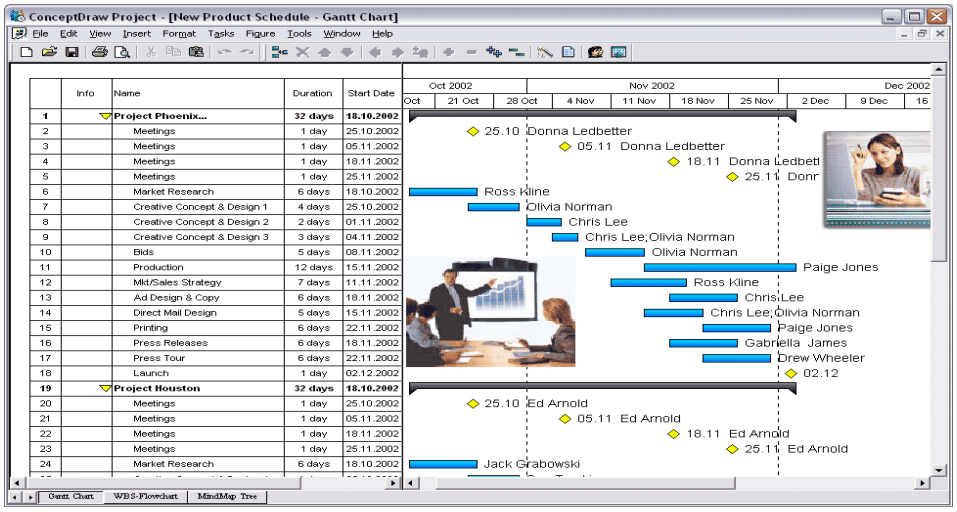


Figura 5.3: *La barra negra muestra el avance de un proyecto haciendo uso de graficas Gantt*

Software De Gestión De Proyectos

Ejemplos de paquetes de este tipo son Project de Microsoft, y Project Manager Workbench, de Applied Business Technology. Estos paquetes simplifican enormemente la preparación de gráficos PERT y Gantt, permitiendo la transformación automática de ambos tipos de gráficos. El software permite también a los directores de proyectos asignar recursos humanos y económicos a las tareas, informar sobre la evolución del proyecto y hacer ensayos del tipo "si-entonces" cuando se intente modificar el plan del proyecto como consecuencia de desviaciones en el calendario.

Algunos paquetes ofrecen también software de recuento de tiempos para llevar a cabo un seguimiento del tiempo real invertido en los diferentes tipos de actividades. Esta información puede ser de utilidad para comparaciones de rendimiento y cuentas de clientes.

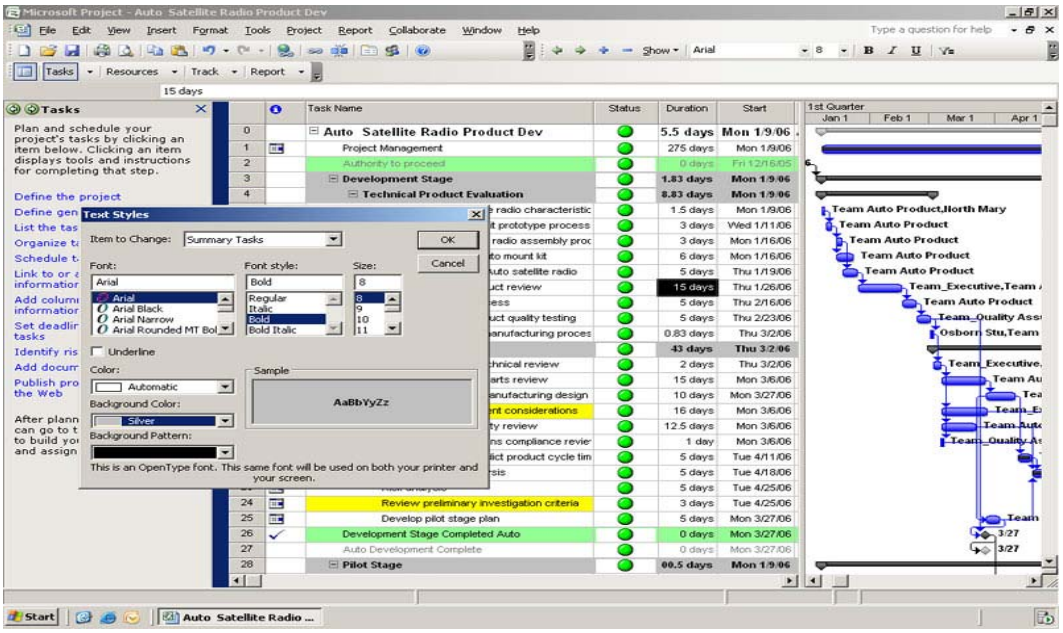


Figura 5.5: *Microsoft Project*

1. Asuma que tiene un proyecto en manos (puede tomar como referencia el caso del hotel detallado en la autoevaluación de la unidad anterior) Reúna un equipo de trabajo entre compañeros de clase y definan roles y asignen responsabilidades para cada participante del proyecto.

2. Tomando como referencia el proyecto que ha elegido llevar a cabo, utilice la herramienta Microsoft Project para definir actividades, asignar responsabilidades y lapsos de tiempo para cada actividad

La gestión de proyectos es una de las áreas clave dentro del proceso de desarrollo de sistemas software. Existen en la gestión de proyectos múltiples tareas a realizar. Por ejemplo, gestión de recursos humanos, estimación, planificación de tareas, seguimiento de proyectos, gestión de riesgos, etc

Una mala gestión de proyectos desemboca a menudo en la no definición de necesidades de usuario final, en excesos de costos y en retrasos en la entrega de los proyectos. Las causas de estos problemas pueden ser omisiones realizadas durante el desarrollo de sistemas, definición imprecisa de objetivos, estimaciones de costos prematuras, deficientes técnicas de estimación, mala gestión de tiempo y falta de liderazgo. Es responsabilidad del director del proyecto evitar estos errores y llevar a buen término el proyecto tanto en tiempo como en presupuesto. Entre las funciones básicas de la dirección de proyectos se incluyen la planificación de las tareas de proyecto , la elección del equipo de proyecto , la organización y la planificación de los esfuerzos del proyecto, la dirección del equipo y el control de la evaluación del proyecto.

[1] PRESSMAN, Roger S. **Ingeniería del Software. Un enfoque práctico.** Mc Graw Hill, Ed. Interamericana de España S.A.U, 2006

[2] ROYCE, W., **Guía de los Fundamentos de la Dirección de Proyectos**, Newtown Square. Pennsylvania, 2004.

[3] SOMMERVILLE, I., **Ingeniería de Software**, Ed. Pearson Educación, 2002

[4] International Organization for Standardization/International Electrotechnical Commission (ISO/IEC), **Guide 2**, ISO Press, 1996

[5] Turner, J. Rodney. **The Handbook of Project-Based Management**, McGraw-Hill,1992.

[6] MILLS, H., O´NEILL, D., **The Management of Software Engineering**, IBM Systems, 1980.

En la siguiente unidad temática detallaremos aspectos relacionados con la calidad y planificación de proyectos de software y el uso de métricas para tal fin

INGENIERIA DEL SOFTWARE
UNIDAD ACADÉMICA Nº 5

NOMBRE:_____

APELLIDOS:_____FECHA; ____/_____/_____

CIUDAD:_____SEMESTRE:_____

1. ¿Qué entiende por gestión de proyectos de software?

   _____
   _____
   _____
   _____

2. ¿Cuál es el rol que debe cumplir el director del proyecto?

   _____
   _____
   _____
   _____

3. ¿Cuales son las principales actividades de la gestión de proyectos de software?

   _____
   _____
   _____
   _____

4. Enumere las causas por las cuales un proyecto de software puede no cumplir con los objetivos deseados

   _____
   _____
   _____
   _____

5. ¿Cuál es el apoyo que brinda una herramienta CASE en la gestión de proyectos de software?

   _____
   _____
   _____
   _____

Unidad Académica VI

METRICAS, CALIDAD Y PLANIFICACIÒN DE PROYECTOS DE SOFTWARE

Cuando se planifica un proyecto se tiene que obtener estimaciones del costo y esfuerzo humano requerido por medio de las mediciones de software que se utilizan para recolectar los datos cualitativos acerca del software y sus procesos para aumentar su calidad.
Estas mediciones, conocidas como métricas del software pueden ayudar a planificar l proyecto de software así como a medir su calidad.
En la presente unidad conoceremos algunas métricas y su clasificaron y como intervienen en la planificación y determinación de la calidad del software

Al finalizar el estudio de la presente unidad temática el estudiante:

- Conceptualiza una métrica
- Clasifica las métricas
- Entiende el papel que desempeñan las métricas en la planificación de proyectos y en la determinación de la calidad del software

1. **Métricas**

Una **métrica** es una medida efectuada sobre algún aspecto del sistema en desarrollo o del proceso empleado que permite, previa comparación con unos valores (medidas) de referencia, obtener conclusiones sobre el aspecto medido con el fin de adoptar las decisiones necesarias. Con esta definición, la definición y aplicación de una métrica no es un objetivo en sí mismo sino un medio para controlar el desarrollo de un sistema de software e intentar aumentar su calidad.

El principio, podría parecer que la necesidad de la medición es algo evidente. Después de todo es lo que nos permite cuantificar y por consiguiente gestionar de forma más efectiva. Pero la realidad puede ser muy diferente. Frecuentemente la medición conlleva una gran controversia y discusión.

- ¿Cuáles son las métricas apropiadas para el proceso y para el producto?
- ¿Cómo se deben utilizar los datos que se recopilan?
- ¿Es bueno usar medidas para comparar gente, procesos o productos?

Estas preguntas y otras tantas docenas de ellas siempre surgen cuando se intenta medir algo que no se ha medido en el pasado.

La medición es muy común en el mundo de la ingeniería. Medimos potencia de consumo, pesos, dimensiones físicas, temperaturas, voltajes, señales de ruidos por mencionar algunos aspectos. Desgraciadamente la medición se aleja de lo común en el mundo de la ingeniería del software. Encontramos dificultades en ponernos de acuerdo sobre que medir y como va evaluar las medidas.

Hay varias razones para medir un producto.

- Para indicar la calidad del producto.
- Para evaluar la productividad de la gente que desarrolla el producto.
- Par evaluar los beneficios en términos de productividad y de calidad, derivados del uso de nuevos métodos y herramientas de la ingeniería de software.
- Para establecer una línea de base para la estimación
- Para ayudar a justificar el uso de nuevas herramientas o de formación adicional.

Las mediciones del mundo físico pueden englobarse en dos categorías: medidas directas y medidas indirectas.

Medidas Directas. En el proceso de ingeniería se encuentran el costo y el esfuerzo aplicado, las líneas de código producidas, velocidad de ejecución, el tamaño de memoria y los defectos observados en un determinado periodo de tiempo.

Medidas Indirectas. Se encuentra la funcionalidad, calidad, complejidad, eficiencia, fiabilidad, facilidad de mantenimiento, etc.

2. Métricas del software.

Son las que están relacionadas con el desarrollo del software como funcionalidad, complejidad, eficiencia.

a) **Métricas Técnicas:** Se centran en las características del software, por ejemplo: la complejidad lógica, el grado de modularidad. Mide la estructura del sistema, el cómo esta hecho.

b) **Métricas de Calidad**: proporcionan una indicación de cómo se ajusta el software a los requisitos implícitos y explícitos del cliente. Es decir cómo voy a medir para que mi sistema se adapte a los requisitos que me pide el cliente.

c) **Métricas de productividad**. Se centran en el rendimiento del proceso de la ingeniería del software. Es decir que tan productivo va a ser el software que voy a diseñar.

d) **Métricas orientadas a la persona**. Proporcionan medidas e información sobre la forma que la gente desarrolla el software y sobre todo el punto de vista humano de la efectividad de las herramientas y métodos. Son las medidas que se hacen acerca del personal que desarrollará el sistema.

e) **Métricas orientadas al tamaño.** Es para saber en que tiempo se va a terminar el software y cuantas personas se van a necesitar. Son medidas directas al software y el proceso por el cual se desarrolla. Si una organización de software mantiene registros sencillos, se puede crear una tabla de datos orientados al tamaño como se muestra en la siguiente figura:

PROYECTO	ESFUERZO	$	KLDC	PAGS. DOC	ERRORES	GENTE
999-01	24	168	12.1	365	29	3
CCC-04	62	440	27.2	1124	86	5
FFF-03	43	314	20.2	1050	64	6
.	.	.	.	.	.	.
.	.	.	.	.	.	.

***Figura 6.1**: Tabla de datos orientados al tamaño*

La tabla lista cada proyecto de desarrollo de software de los últimos años y los datos orientados al tamaño de cada uno de ellos.
Refiriéndonos a la entrada de la tabla del proyecto 999-01 se desarrollaron 12.1 KLDC (miles de líneas de código) con un esfuerzo de 24 personas mes y un costo de 168 mil dólares. Debe

tenerse en cuenta que el esfuerzo y el costo registrados en la tabla incluyen todas las actividades de la ingeniería de software como son análisis, diseño, codificación y prueba.
Otra información del proyecto 222-01 indica que se desarrollaron 365 paginas mientras que se encontraron 29 errores tras entregárselo al cliente, dentro del primer año de utilización. También sabemos que trabajaron 3 personas en el desarrollo del proyecto.

En los rendimientos del sistema y los rudimentarios datos contenidos en la tabla se puede desarrollar, para cada proyecto un conjunto de métricas sencillas de productividad y calidad orientadas al tamaño. Se obtienen las siguientes formulas:

Productividad = KLDC/persona-mes[25]

Calidad = errores/KLDC

Documentación = pags. Doc/ KLDC

Costo = $/KLDC

f) **Métricas orientadas a la Función.** Son medidas indirectas del software y del proceso por el cual se desarrolla. En lugar de calcular las LDC, las métricas orientadas a la función se centran en la funcionalidad o utilidad del programa.
Las métricas orientadas a la función fueron el principio propuestas por Allan Alberche, de IBM en 1979, quien sugirió un acercamiento a la medida de la productividad denominado método del punto de función. Los puntos de función se obtienen utilizando una función empírica basado en medidas cuantitativas del dominio de información del software y valoraciones subjetivas de la complejidad del software.
Los puntos de función se calculan rellenando la tabla como se muestra en la siguiente figura:

		FACTOR DE PONDERACIÓN			
Parámetro de medición	**Cuenta**	**Simple**	**Medio**	**Complejo**	
Numero de entradas de usuario	☐ x	3	4	6	= ☐
Numero de salidas de usuario	☐ x	4	5	7	= ☐
Numero de peticiones de usuario	☐ x	3	4	6	= ☐
Numero de archivos	☐ x	7	10	15	= ☐
Numero de interfaces externas	☐ x	5	7	10	= ☐
Cuenta = Total	→				☐

***Figura 6.2**: Calculo de métricas de punto de función.*

Se determinan 5 características del ámbito de la información y los cálculos aparecen en la posición apropiada de la tabla. Los valores del ámbito de información están definidos de la siguiente manera:

a) **Números de entrada de usuario:** se cuenta cada entrada del usuario que proporcione al software diferentes datos orientados a la aplicación. Las entradas deben ser distinguidas de las peticiones que se contabilizan por separado.

b) **Numero de salida del usuario:** se encuentra cada salida que proporciona la usuario información orientada ala aplicación. En este contexto las salidas se refieren a informes, pantalla, mensajes de error. Los elementos de datos individuales dentro de un informe se encuentran por separado.

[25] *persona-mes* representa el esfuerzo

c) **Números de peticiones al usuario***:* una petición esta definida como una entrada interactiva que resulta de la generación de algún tipo de respuesta en forma de salida interactiva. Se cuenta cada petición por separado.

d) **Numero de archivos:** se cuenta cada archivo maestro lógico, o sea una agrupación lógica de datos que puede ser una parte en una gran base de datos o un archivo independiente.

e) **Numero de interfaces externas:** se cuentan todas las interfaces legibles por la maquina por ejemplo: archivos de datos, en cinta o discos que son utilizados para transmitir información a otro sistema.

Cuando han sido recogidos los datos anteriores se asocian el valor de complejidad a cada cuenta. Las organizaciones que utilizan métodos de puntos de función desarrollan criterios para determinar si una entrada es denominada simple, media o compleja. No obstante la determinación de la complejidad es algo subjetivo.

Para calcular los puntos de función se utiliza la siguiente relación.

$$PF = cuenta\_total * [0.65 + 0.01 * sum(fi)]$$

Donde:

cuenta_total es la suma de todas las entradas de PF obtenidas de la tabla anterior.

Fi donde ***i*** puede ser de uno hasta 14 los valores de ajuste de complejidad basados en las respuestas a las cuestiones señaladas de la siguiente tabla (Evaluar cada factor en escala 0 a 5):

.

0	1	2	3	4	5
Sin influencia	Incidental	Moderado	Medio	Significativo	Esencial

Fi:

1. ¿Requiere el sistema copias de seguridad y recuperación fiables?
2. ¿Se requiere comunicación de datos?
3. ¿Existen funciones de procesamiento distribuido?
4. ¿Es critico el rendimiento?
5. ¿Será ejecutado el sistema en un entorno operativo existente y frecuentemente utilizado?
6. ¿Requiere el sistema entrada de datos interactivo?
7. ¿Requiere la entrada de datos interactivo que las transiciones de entrada se lleven a cabo sobre múltiples o variadas operaciones?
8. ¿Se actualizan los archivos maestros en forma interactiva?
9. ¿Son complejas las entradas, las salidas, los archivos o peticiones?
10. ¿Es complejo el procesamiento interno?
11. ¿Se ha diseñado el código para ser reutilizable?
12. ¿Están incluidos en el diseño la conversión y la instalación?
13. ¿Se ha diseñado el sistema para soportar múltiples instalaciones en diferentes organizaciones?
14. ¿Se ha diseñado la aplicación para facilitar los cambios y para ser fácilmente utilizada por el usuario?

Los valores constantes de la ecuación anterior y los factores de peso aplicados en las encuestas de los ámbitos de información han sido determinados empíricamente.

Una vez calculado los puntos de función se usan de forma analógica a las LDC como medida de la productividad, calidad y otros productos del software.

Productividad = PF / persona-mes

Calidad = Errores / PF

Costo = Dólares / PF

Documentación = Pags. Doc / PF

La medida de puntos de función se diseño originalmente para ser utilizada en aplicación de sistemas de información de gestión. Sin embargo, en algunas aplicaciones se les denomina puntos de característica. La medida del punto de característica da cabida a aplicaciones cuya complejidad algorítmica es alta.
Las aplicaciones de software de tiempo real de control de procesos tienden a tener una complejidad algorítmica alta y por tanto son fácilmente tratables por puntos de características.
Para calcular los puntos de características, nuevamente se cuentan y ponderan los valores del ámbito de información, como se describió anteriormente. Además, las métricas de punto de característica tienen en cuenta otra característica del software, los algoritmos.
Un algoritmo se define como un problema de complejidad computacional limitada que se incluye dentro de un determinado programa de computadora. La inversión de una matriz, la decodificación de una cadena de bits o el manejo de una interrupción son todo ellos ejemplos de algoritmos.

Para calcular los puntos de característica, se utiliza la siguiente tabla.

Puntos de característica

	FACTOR DE PONDERACIÓN			
Parámetro de medición	**Cuenta**		**Peso**	
Numero de entradas de usuario		x	4	=
Numero de salidas de usuario		x	5	=
Numero de peticiones de usuario		x	4	=
Numero de archivos		x	7	=
Numero de interfaces externas		x	7	=
Algoritmos		x	3	=
Cuenta = Total				

Se usa único valor de peso para cada uno de los parámetros de medida y se calcula el valor del punto característica global mediante la ecuación.

$$PF = cuenta\_total * [0.65 + 0.01 * sum(fi)]$$

Debe tenerse en cuenta que los puntos de característica y los puntos de función representan lo mismo. "funcionalidad o utilidad" en forma de software.

3. Calidad del software

Todas las metodologías y herramientas tienen un único fin: **producir software de gran calidad.**

a) Definiciones de calidad del software:

"Concordancia con los requisitos funcionales y de rendimiento explícitamente establecidos con los estándares de desarrollo explícitamente documentados y con las características implícitas que se espera de todo software desarrollado profesionalmente" R. S. Pressman (1992).

"El conjunto de características de una entidad que le confieren su aptitud para satisfacer las necesidades expresadas y las implícitas" ISO 8402 (UNE 66-001-92).

b) Aseguramiento de calidad del software (Software Quality Assurance)

El aseguramiento de calidad del software es el conjunto de actividades planificadas y sistemáticas necesarias para aportar la confianza en que el producto (software) satisfará los requisitos dados de calidad.

El aseguramiento de calidad del software se diseña para cada aplicación antes de comenzar a desarrollarla y no después. El aseguramiento de calidad del software está presente en:

- Métodos y herramientas de análisis, diseño, programación y prueba
- Inspecciones técnicas formales en todos los pasos del proceso de desarrollo del software
- Estrategias de prueba multiescala
- Control de la documentación del software y de los cambios realizados
- Procedimientos para ajustarse a los estándares
- Mecanismos de medida (métricas)
- Registro de auditorias y realización de informes

Actividades para el aseguramiento de calidad del software

- Métricas de software para el control del proyecto
- Verificación y validación del software a lo largo del ciclo de vida (Incluye las pruebas los procesos de revisión e inspección)
- La gestión de la configuración del software

c) Gestión de la calidad del software (Software Quality Management)

Gestión de la calidad (ISO 9000)

Conjunto de actividades de la función general de la dirección que determina la calidad, los objetivos y las responsabilidades y se implanta por medios tales como la planificación de la calidad, el control de la calidad, el aseguramiento (garantía) de la calidad y la mejora de la calidad, en el marco del sistema de calidad.
La gestión de la calidad se aplica normalmente a nivel de Empresa. También puede haber una gestión de calidad dentro de la gestión de cada proyecto

Política de calidad (ISO 9000)

Directrices y objetivos generales de una organización, relativos a la calidad, tal como se expresan formalmente por la alta dirección.

d) Control de la calidad del software (Software Quality Control)

Son las técnicas y actividades de carácter operativo, utilizadas para satisfacer los requisitos relativos a la calidad, centradas en dos objetivos fundamentales:

- Mantener bajo control un proceso
- Eliminar las causas de los defectos en las diferentes fases del ciclo de vida

En general son las actividades para evaluar la calidad de los productos desarrollados

e) Sistema de calidad

Estructura organizativa, procedimientos, procesos y recursos necesarios para implantar la gestión de calidad
El sistema de calidad se debe adecuar a los objetivos de calidad de la empresa
La dirección de la empresa es la responsable de fijar la política de calidad y las decisiones relativas a iniciar, desarrollar, implantar y actualizar el sistema de calidad.
Un sistema de calidad consta de varias partes:

- Documentación: Manual de calidad (documento principal para establecer e implantar un sistema de calidad. Puede haber manuales a nivel de empresa, departamento, producto, específicos como compras, proyectos,…)
- Parte física: Locales, herramientas ordenadores, etc.
- Aspectos humanos: Formación de personal, creación y coordinación de equipos de trabajo

Normativas

- ISO:
    - ISO 9000: Gestión y aseguramiento de calidad (conceptos y directrices generales)
    - Recomendaciones externas para aseguramiento de la calidad (ISO 9001, ISO 9002, ISO 9003)
    - Recomendaciones internas para aseguramiento de la calidad (ISO 9004)

- MALCOM BALDRIGE NATIONAL QUALITY AWARD
- Software Engineering Institute (SEI) Capability Maturity Model (CMM) for software

f) Certificación de la calidad (Quality certification)

Un sistema de certificación de calidad permite una valoración independiente que debe demostrar que la organización es capaz de desarrollar productos y servicios de calidad
Los pilares básicos de la certificación de calidad son tres [1]:

- Una metodología adecuada
- Un medio de valoración de la metodología
- La metodología utilizada y el medio de valoración de la metodología deben estar reconocidos ampliamente por la industria

g) Factores que determinan la calidad del software

Se clasifican en tres grupos:

a) **Operaciones del producto**: características operativas
    - **Corrección** (¿Hace lo que se le pide?)
      El grado en que una aplicación satisface sus especificaciones y consigue los objetivos encomendados por el cliente
    - **Fiabilidad** (¿Lo hace de forma fiable todo el tiempo?)
      El grado que se puede esperar de una aplicación lleve a cabo las operaciones especificadas y con la precisión requerida
    - **Eficiencia** (¿Qué recursos hardware y software necesito?)
      La cantidad de recursos hardware y software que necesita una aplicación para realizar las operaciones con los tiempos de respuesta adecuados
    - **Integridad** (¿Puedo controlar su uso?)
      El grado con que puede controlarse el acceso al software o a los datos a personal no autorizado
    - **Facilidad de uso** (¿Es fácil y cómodo de manejar?)

El esfuerzo requerido para aprender el manejo de una aplicación, trabajar con ella, introducir datos y conseguir resultados

b) **Revisión del producto**: capacidad para soportar cambios
   - **Facilidad de mantenimiento** (¿Puedo localizar los fallos?)
     El esfuerzo requerido para localizar y reparar errores
   - **Flexibilidad** (¿Puedo añadir nuevas opciones?)
     El esfuerzo requerido para modificar una aplicación en funcionamiento
   - **Facilidad de prueba** (¿Puedo probar todas las opciones?)
     El esfuerzo requerido para probar una aplicación de forma que cumpla con lo especificado en los requisitos

c) **Transición del producto**: adaptabilidad a nuevos entornos
   - **Portabilidad** (¿Podré usarlo en otra máquina?)
     El esfuerzo requerido para transferir la aplicación a otro hardware o sistema operativo
   - **Reusabilidad** (¿Podré utilizar alguna parte del software en otra aplicación?)
     Grado en que partes de una aplicación pueden utilizarse en otras aplicaciones
   - **Interoperabilidad** (¿Podrá comunicarse con otras aplicaciones o sistemas informáticos?
     El esfuerzo necesario para comunicar la aplicación con otras aplicaciones o sistemas informáticos

h) Métricas de la calidad del sofware

Es difícil, y en algunos casos imposible, desarrollar medidas directas de los factores de calidad del software.
Cada factor de calidad Fc se puede obtener como combinación de una o varias métricas:

$$Fc= c1 * m1 + c2 * m2 + \ldots + cn * mn$$

Donde:

- ci factor de ponderación de la métrica i, que dependerá de cada aplicación específica
- m_i métrica i
- Habitualmente se puntúan de 0 a 10 en las métricas y en los factores de calidad

Métricas para determinar los factores de calidad

- Facilidad de auditoria
- Exactitud
- Normalización de las comunicaciones
- Completitud
- Concisión
- Consistencia
- Estandarización de los datos
- Tolerancia de errores
- Eficiencia de la ejecución
- Facilidad de expansión
- Generalidad
- Independencia del hardware
- Instrumentación
- Modularidad
- Facilidad de operación
- Seguridad
- Autodocuemntación
- Simplicidad
- Independencia del sistema software
- Facilidad de traza
- Formación

4. Estimación

Es una pequeña planeación sobre el proyecto. Una de las actividades cruciales del proceso de gestión del proyecto del software es la planificación. Cuando se planifica un proyecto de software se tiene que obtener estimaciones de esfuerzo humano requerido, de la duración cronológica del esfuerzo humano requerido, de la duración cronológica del proyecto y del costo. Pero en muchos de los casos las estimaciones se hacen valiéndose de la experiencia pasada como única guía. Si un proyecto es bastante similar en tamaño y funciona un proyecto pasado es probable que el nuevo proyecto requiera aproximadamente la misma cantidad de esfuerzo, que dure aproximadamente lo mismo que el trabajo anterior. Pero que pasa si el proyecto es totalmente distinto entonces puede que la experiencia obtenida no sea lo suficiente.
Se han desarrollado varias técnicas de estimación para el desarrollo de software, aunque cada una tiene sus puntos fuertes y sus puntos débiles, todas tienen en común los siguientes atributos.

- Se han de establecer de antemano el ámbito del proyecto.
- Como bases para la realización de estimaciones se usan métricas del software de proyectos pasados.
- El proyecto se desglosa en partes más pequeñas que se estiman individualmente.

5. Planeación Del Proyecto

La planeación efectiva de un proyecto de software depende de la planeación detallada de su avance, anticipado problemas que puedan surgir y preparando con anticipación soluciones tentativas a ellos. Se supondrá que el administrador del proyecto es responsable de la planeación desde la definición de requisitos hasta la entrega del sistema terminado. No se analizara la planeación que implica a la estimación de la necesidad de un sistema de software y la habilidad de producir tal sistema, la asignación de prioridad al proceso de su producción.
Los puntos analizados posteriormente generalmente son requeridos por grandes sistemas de programación, sin embargo estos puntos son validos también para sistemas pequeños.

Panorama. Hace una descripción general del proyecto detalle de la organización del plan y resume el resto del documento.
Plan de fases. Se analiza el ciclo de desarrollo del proyecto como es: análisis de requisitos, fase de diseño de alto nivel, fase de diseño de bajo nivel, etc. Asociada con cada fase debe de haber una fecha que especifique cuando se debe terminar estas fases y una indicación de como se pueden solapar las distintas fases del proyecto.
Plan de organización. Se definen las responsabilidades específicas de los grupos que intervienen en el proyecto.
Plan de pruebas. Se hace un esbozo general de las pruebas y de las herramientas, procedimientos y responsabilidades para realizar las pruebas del sistema.
Plan de control de modificaciones. Se establece un mecanismo para aplicar las modificaciones que se requieran a medida que se desarrolle el sistema.
Plan de documentación. Su función es definir y controlar la documentación asociada con el proyecto.
Plan de capacitación. Se describe la preparación de los programadores que participan en el proyecto y las instrucciones a los usuarios para la utilización del sistema que se les entregue.
Plan de revisión e informes. Se analiza como se informa del estado del proyecto y se definen las revisiones formales asociadas con el avance de proyecto.
Plan de instalación y operación. Se describe el procedimiento para instalar el sistema en la localidad del usuario.
Plan de recursos y entregas. Se resume los detalles críticos del proyecto como fechas programadas, marcas de logros y todos los artículos que deben entrar bajo contrato.
Plan de mantenimiento. Se establece un bosquejo de los posibles tipos de mantenimiento que se tienen que dar para futuras versiones del sistema.

1. Utilice las métricas estudiadas anteriormente: Métricas orientadas al tamaño y métricas orientadas a la función y aplíquelas al proyecto de software que esta desarrollando

2. Elabore un plan para medir y garantizar la calidad del software del proyecto que esta desarrollando

El concepto de métrica es el término que describe muchos y muy variados casos de medición. Siendo una métrica una medida estadística no necesariamente cuantitativa se aplica a muchos aspectos en la gestión de software, los cuales pueden ser medidos desde diferentes puntos de vista y pueden ayudarnos en aspectos como la planificación del proyecto y la calidad del software, entre otros.

[1] PRESSMAN, Roger S. **Ingeniería del Software. Un enfoque práctico.** Mc Graw Hill, Ed. Interamericana de España S.A.U, 2006

[2] SOMMERVILLE, I., **Ingeniería de Software**, Ed. Pearson Educación, 2002

[3] PIATTINI y GARCIA. **Calidad en el desarrollo y mantenimiento del Software.** Editorial RA-MA, Madrid, 2002.

[4] FENTON y PFLEEGER.:**Software Metrics** . PWS Publishing CO. 2002.

[5] Londeix B, **Cost Estimation for Software Development**, Addison-Wesley Publishers Company. 1997

[6] MILLS, H., O´NEILL, D., **The Management of Software Engineering**, IBM Systems, 1980.

Bibliografía electrónica:

- Fundamentos de Ingeniería del Software
  http://dis.um.es/~jnicolas/09BK_FIS.html

En la siguiente unidad temática detallaremos aspectos relacionados a las pruebas del software.

AUTOEVALUACIÓN FORMATIVA

INGENIERIA DEL SOFTWARE
UNIDAD ACADÉMICA Nº 6

NOMBRE:_____

APELLIDOS:_____FECHA; ____/_____/_____

CIUDAD:_____SEMESTRE:_____

PARTE I: Marque la alternativa correcta:

1. Los pilares básicos de la certificación de calidad del software son:

   a) Una metodología adecuada
   b) Un medio de valoración de la metodología
   c) Un reconocimiento de la industria de la metodología utilizada y del medio de valorar la metodología
   d) Todas las afirmaciones anteriores son correctas
   e) Ninguna respuesta anterior es correcta.

2. La calidad del software implica

   a) La concordancia entre el software diseñado y los requisitos
   b) Seguir un estándar o metodología en el proceso de desarrollo de software
   c) Tener en cuenta los requisitos implícitos (no expresados por los usuarios
   d) Todas las afirmaciones anteriores son correctas
   e) Ninguna respuesta anterior es correcta

PARTE II: Desarrolle los siguientes ejercicios

3. Métricas orientadas al tamaño

Proyecto	Esfuerzo	$	KLDC	Pag. doc	Errores	Gente
Farmacia	30	168,500	12,100	378	29	5
Hospital	60	578,300	39,443	921	540	20

Calcular:

a) Productividad = KLDC/esfuerzo
   - Hopital = ?
   - Farmacia = ?

b) Calidad = Errores/KLDC
   - Hopital = ?
   - Farmacia = ?

c) Costo = $/KLDC
   - Hopital = ?
   - Farmacia = ?

d) Documentación = Pags. doc/KLDC
   - Hopital = ?
   - Farmacia = ?

4. Métricas orientadas a la función

Se tiene un sistema el cual cuenta con 3 entradas de catalogo: productos, proveedores y clientes; una pantalla de la elaboración de facturas, 4 tipos de reportes proporcionados tanto en pantalla como en papel (factura, lista de inventario, estado de cuenta de los clientes y estado de cuenta con los proveedores). Además la entrada de factura tiene alrededor de 30 peticiones, el sistema genera alrededor de 30 archivos además de estar conectado a un lector óptico y una impresora. Calcular los puntos de función:

Parámetro de medida	Cuenta		Factor de peso medio		
Numero de entradas al usuario	4	*	4	=	16
Numero de salidas al usuario	8	*	5	=	40
Numero de peticiones al usuario	30	*	4	=	120
Numero de archivos	30	*	10	=	300
Numero de interfaces externas	2	*	7	=	14
			Cuenta total	=	490

Repuesta a las preguntas basada en la complejidad media:
1=0; 2=5; 3=3; 4=5; 5=5; 6=5; 7=1; 8=5; 9=2; 10=2; 11=4; 12=0; 13=0; 14=4

Hallar:
- Fi =?
- PF = ?
- Productividad = ?
- Calidad = ?
- Costo = ?
- Documentación = ?

Unidad Académica VII

PRUEBAS Y MANTENIMIENTO DE SOFTWARE

Las pruebas de software son un conjunto de técnicas que permiten determinar la calidad de un producto software. Las pruebas de software se integran dentro de las diferentes fases del Ciclo del software dentro de la Ingeniería de software. Así se ejecuta un programa y mediante técnicas experimentales se trata de descubrir que errores tiene.

La calidad de un sistema software es algo subjetivo que depende del contexto y del objeto que se pretenda conseguir. Para determinar dicho nivel de calidad se deben efectuar unas medidas o pruebas que permitan comprobar el grado de cumplimiento respecto de las especificaciones iniciales del sistema.

En la presente unidad se pretende que el alumno tome conciencia de la necesidad de la prueba del software y de lo importante que es realizarla desde el principio. Así mismo, el alumno deberá convencerse de lo importante que es la actividad de mantenimiento del software.

Al finalizar el estudio de la presente unidad temática el estudiante:

- Entiende la necesidad de llevar a cabo pruebas de software durante el ciclo de vida del software
- Elabora pruebas de software utilizando diversos criterios
- Entiende la importancia del mantenimiento de software

1. Pruebas del software

Una de las características típicas del desarrollo de software basado en el ciclo de vida es la realización de controles periódicos. Estos controles pretenden una evaluación de la calidad de los productos generados (especificación de requisitos, documentos de diseño, etc.) para poder detectar posibles defectos cuanto antes. Sin embargo, todo sistema o aplicación, independientemente de estas revisiones, debe ser probado mediante su ejecución controlada antes de ser entregado al cliente. Estas ejecuciones o ensayos de funcionamiento, posteriores a la terminación del código del software, se denominan habitualmente pruebas.

Las pruebas constituyen un método más para poder verificar y validar el software. Se puede definir la verificación como [IEEE, 1990] “el proceso de evaluación de un sistema o de uno de sus componentes para determinar si los productos de una fase dada satisfacen las condiciones impuestas al comienzo de dicha fase”. Por ejemplo, verificar el código de un módulo significa comprobar si cumple lo marcado en la especificación de diseño donde se describe. Por otra parte, la validación es “el proceso de evaluación de un sistema o de uno de sus componentes durante o al final del proceso de desarrollo para determinar si satisface los requisitos especificados”. Así, validar una aplicación implica comprobar si satisface los requisitos marcados por el usuario. Podemos recurrir a la clásica explicación informal de **Boehm** de estos conceptos:

- Verificación: ¿estamos construyendo correctamente el producto?
- Validación: ¿estamos construyendo el producto correcto?

Como hemos dicho, las pruebas permiten verificar y validar el software cuando ya está en forma de código ejecutable. A continuación, expondremos algunas definiciones de conceptos relacionados con las pruebas.

2. Definiciones

Las siguientes definiciones son algunas de las recogidas en el diccionario IEEE en relación a las pruebas [IEEE, 1990], que complementamos con otras más informales:

- **Pruebas** (test): "una actividad en la cual un sistema o uno de sus componentes se ejecuta en circunstancias previamente especificadas, los resultados se observan y registran y se realiza una evaluación de algún aspecto". Para Myers [MYERS, 1979], probar (o la prueba) es el "proceso de ejecutar un programa con el fin de encontrar errores". El nombre "prueba", además de la actividad de probar, se puede utilizar para designar "un conjunto de casos y procedimientos de prueba" [IEEE, 1990].

- **Caso de prueba** (test case): "un conjunto de entradas, condiciones de ejecución y resultados esperados desarrollados para un objetivo particular como, por ejemplo, ejercitar un camino concreto de un programa o verificar el cumplimiento de un determinado requisito". También se puede referir a la documentación en la que se describen las entradas, condiciones y salidas de un caso de prueba.

- **Defecto** (defect, fault, "bug"): "un defecto en el software como, por ejemplo, un proceso, una definición de datos o un paso de procesamiento incorrectos en un programa".

- **Fallo** (failure): «La incapacidad de un sistema o de alguno de sus componentes para realizar las funciones requeridas dentro de los requisitos de rendimiento especificados».

- **Error** (error): tiene varias acepciones:

    1. La diferencia entre un valor calculado, observado o medido y el valor verdadero, especificado o teóricamente correcto. Por ejemplo, una diferencia de dos centímetros entre el valor calculado y el real.

    2. Una acción humana que conduce a un resultado incorrecto (una metedura de pata). Por ejemplo, que el operador o el programador pulse una tecla equivocada.

    La relación entre error, fallo y defecto queda más clara en la Figura 7.1.

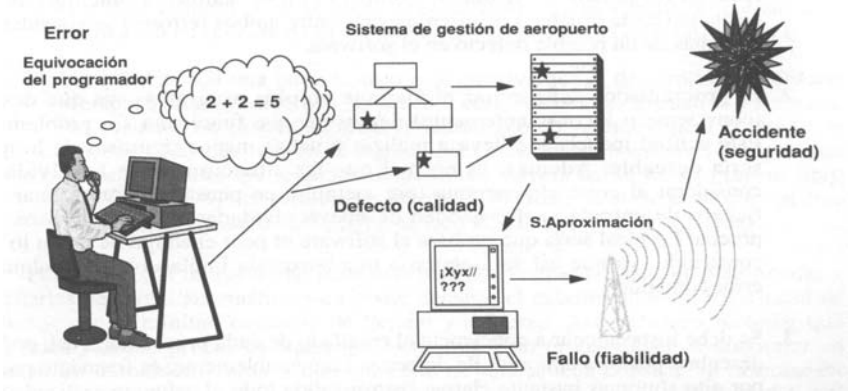


Figura 7.1. *Relación entre error, defecto y fallo.*

3. El proceso de prueba

El proceso de prueba comienza con la generación de un plan de pruebas en base a la documentación sobre el proyecto y la documentación sobre el software a probar. A partir de dicho plan, se entra en detalle diseñando pruebas específicas basándose en la documentación del software a probar. Una vez detalladas las pruebas (especificaciones de casos y de procedimientos), se toma la configuración del software (revisada, para confirmar que se trata de la versión apropiada del programa) que se va a probar para ejecutar sobre ella los casos. En algunas situaciones, se puede tratar de reejecuciones de pruebas, por lo que es conveniente tener constancia de los defectos ya detectados aunque aún no corregidos. A partir de los resultados de salida, se pasa a su evaluación mediante comparación con la salida esperada. A partir de ésta, se pueden realizar dos actividades:

- La depuración (localización y corrección de defectos).
- El análisis de la estadística de errores.

La depuración puede corregir o no los defectos. Si no consigue localizarlos, puede ser necesario realizar pruebas adicionales para obtener más información. Si se corrige un defecto, se debe volver a probar el software para comprobar que el problema está resuelto.

Por su parte, el análisis de errores puede servir para realizar predicciones de la fiabilidad del software y para detectar las causas más habituales de error y mejorar los procesos de desarrollo.

4. Técnicas de diseño de casos de prueba

El diseño de casos de prueba está totalmente mediatizado por la imposibilidad de probar exhaustivamente el software. Pensemos en un programa muy sencillo que sólo suma dos números enteros de dos cifras (del 0 al 99). Si quisiéramos probar, simplemente, todos los valores válidos que se pueden sumar, deberíamos probar 10.000 combinaciones distintas de cien posibles números del primer sumando y cien del segundo. Pensemos que aún quedarían por probar todas las posibilidades de error al introducir los datos (por ejemplo, teclear una letra en vez de un número). La conclusión es que, si para un programa tan elemental debemos realizar tantas pruebas, pensemos en lo que supondría un programa medio.

En consecuencia, las técnicas de diseño de casos de prueba tienen como objetivo conseguir una confianza aceptable en que se detectarán los defectos existentes (ya que la seguridad total sólo puede obtenerse de la prueba exhaustiva, que no es practicable) sin necesidad de consumir una cantidad excesiva de recursos (por ejemplo, tiempo para probar o tiempo de ejecución). Toda la disciplina de pruebas debe moverse, por lo tanto, en un equilibrio entre la disponibilidad de recursos y la confianza que aportan los casos para descubrir los defectos existentes. Este equilibrio no es sencillo, lo que convierte a las pruebas en una disciplina difícil que está lejos de parecerse a la imagen de actividad rutinaria que suele sugerir.

Ya que no se pueden probar todas las posibilidades de funcionamiento del software, la idea fundamental para el diseño de casos de prueba consiste en elegir algunas de ellas que, por sus características, se consideren representativas del resto. Así, se asume que, si no se detectan defectos en el software al ejecutar dichos casos, podemos tener un cierto nivel de confianza (que depende de la elección de los casos) en que el programa no tiene defectos. La dificultad de esta idea es saber elegir los casos que se deben ejecutar. De hecho, una elección puramente aleatoria no proporciona demasiada confianza en que se puedan detectar todos los defectos existentes. Por ejemplo, en el caso del programa de suma de dos números, elegir cuatro pares de sumandos al azar no aporta mucha seguridad a la prueba (una probabilidad de 4/10.000 de cobertura de posibilidades). Por eso es necesario recurrir a ciertos criterios de elección que veremos a continuación.

Existen tres enfoques principales para el diseño de casos:

1. El enfoque **estructural** o de **caja blanca** (véase la figura 7.2). Consiste en centrarse en la estructura interna (implementación) del programa para elegir los casos de prueba. En este caso, la prueba ideal (exhaustiva) del software consistiría en probar todos los posibles caminos de ejecución, a través de las instrucciones del código, que puedan trazarse.

2. El enfoque **funcional** o de **caja negra** (véase la figura 7.2). Consiste en estudiar la especificación de las funciones, la entrada y la salida para derivar los casos. Aquí, la prueba ideal del software consistiría en probar todas las posibles entradas y salidas del programa.

3. El enfoque **aleatorio** consiste en utilizar modelos (en muchas ocasiones estadísticos) que representen las posibles entradas al programa para crear a partir de ellos los casos de prueba. La prueba exhaustiva consistiría en probar todas las posibles entradas al programa.

Estos enfoques no son excluyentes entre sí, ya que se pueden combinar para conseguir una detección de defectos más eficaz. Veremos a continuación una presentación de cada uno de ellos.

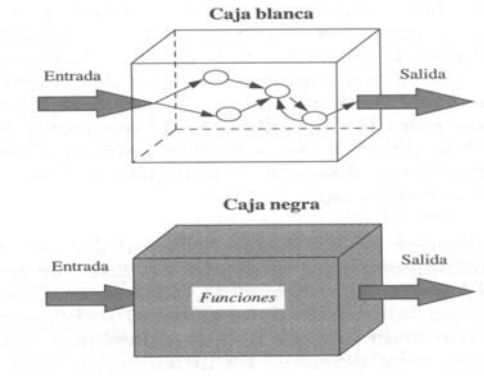


***Figura 7.2:**. Los enfoques de diseño de pruebas de caja blanca y de caja negra.*

a) Pruebas estructurales. (Prueba de la caja blanca)

Como se ha mencionado, las pruebas exhaustivas son impracticables. Podemos recurrir al clásico ejemplo de Myers de un programa de 50 líneas con 25 sentencias IF en serie, en el que el número total de caminos contiene 33,5 millones de secuencias potenciales (contando dos posibles salidas para cada IF tenemos 2^{25} posibles caminos). El diseño de casos tiene que basarse en la elección de caminos importantes que ofrezcan una seguridad aceptable en descubrir defecto, y para ello se utilizan los llamados criterios de cobertura lógica. Antes de pasar a examinarlos, conviene señalar que estas técnicas no requieren el uso de ninguna representación gráfica específica del software, aunque es habitual tomar como base los llamados diagramas de flujo de control (flowgraph charts o flowcharts). Como ejemplo de diagrama de flujo junto al código correspondiente, véase la Figura 7.3.

En el recuadro adjunto pueden consultarse algunas recomendaciones para dibujar los grafos de flujo de los programas para poder generar los casos de prueba correspondientes. También existen herramientas que dibujan el grafo de flujo de un programa sólo con facilitar el código fuente del mismo como entrada.

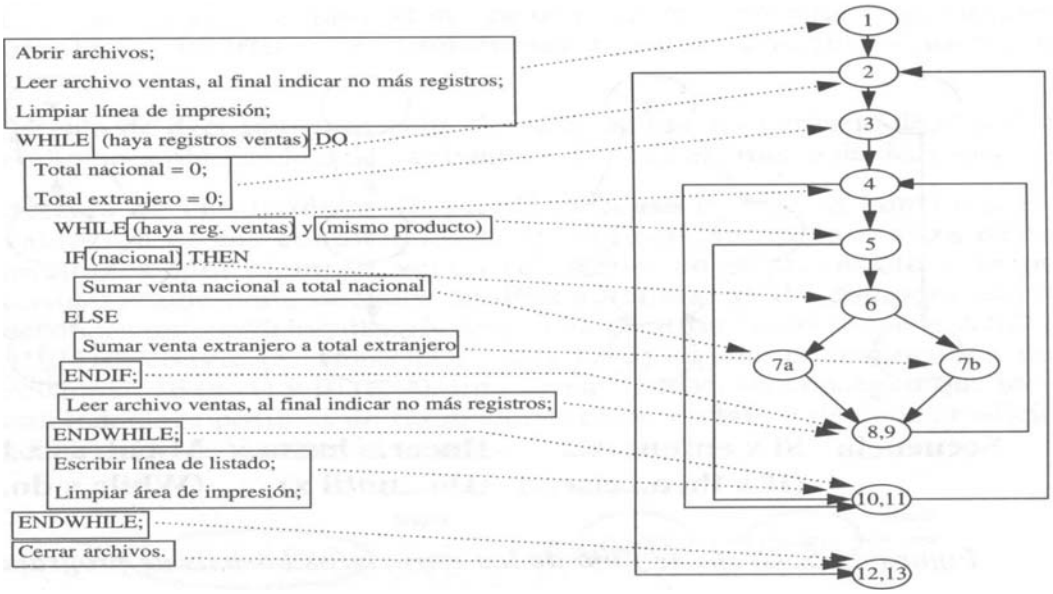


***Figura 7.3**: Grafo de flujo de un programa (pseudocódigo).*

Una posible clasificación de criterios de cobertura lógica es la que se ofrece abajo. Hay que destacar que los criterios de cobertura que se ofrecen están en orden de exigencia y, por lo tanto, de coste económico. Es decir, el criterio de cobertura de sentencias es el que ofrece una menor seguridad de detección de defectos, pero es el que cuesta menos en número de ejecuciones del programa.

1. Cobertura de sentencias. Se trata de generar los casos de prueba necesarios para que cada sentencia o instrucción del programa se ejecute al menos una vez.

2. Cobertura de decisiones. Consiste en escribir casos suficientes para que cada decisión tenga, por lo menos una vez, un resultado verdadero y, al menos una vez, uno falso. En general, una ejecución de pruebas que cumple la cobertura de decisiones cumple también la cobertura de sentencias.

3. Cobertura de condiciones. Se trata de diseñar tantos casos como sea necesario para que cada condición de cada decisión adopte el valor verdadero al menos una vez y el falso al menos una vez. No podemos asegurar que si se cumple la cobertura de condiciones se cumple necesariamente la de decisiones.

4. Criterio de decisión/condición. Consiste en exigir el criterio de cobertura de condiciones obligando a que se cumpla también el criterio de decisiones.

5. Criterio de condición múltiple. En el caso de que se considere que la evaluación de las condiciones de cada decisión no se realiza de forma simultánea (por ejemplo, según se ejecuta en el procesador), se podría considerar que cada decisión multicondicional se descompone en varias decisiones unicondicionales. Es decir, una decisión como **IF((a=1)AND(c=4)) THEN** se convierte en una concatenación de dos decisiones: **IF(a=1)** y **IF(c=4)**. En este caso, debemos conseguir que todas las combinaciones posibles de resultados (verdadero/falso) de cada condición en cada decisión se ejecuten al menos una vez.

La cobertura de caminos (secuencias de sentencias) es el criterio más elevado: cada uno de los posibles caminos del programa se debe ejecutar al menos una vez. Se define **camino** como la secuencia de sentencias encadenadas desde la sentencia inicial del programa hasta su sentencia final. Como hemos visto, el número de caminos en un programa pequeño puede ser

impracticable para las pruebas. Para reducir el número de caminos a probar, se habla del concepto de **camino de prueba** (test path): un camino del programa que atraviesa, como máximo, una vez el interior de cada bucle que encuentra. La idea en la que se basa consiste en que ejecutar un bucle más de una vez no supone una mayor seguridad de detectar defectos en él. Sin embargo, otros especialistas recomiendan que se pruebe cada bucle tres veces: una sin entrar en su interior, otra ejecutándolo una vez y otra más ejecutándolo dos veces. Esto último es interesante para comprobar cómo se comporta a partir de los valores de datos procedentes, no del exterior del bucle (como en la primera iteración), sino de las operaciones de su interior.

Si trabajamos con los caminos de prueba, existe la posibilidad de cuantificar el número total de caminos utilizando algoritmos basados en matrices que representan el grafo de flujo del programa. Así, tienen diversos métodos basados en ecuaciones, expresiones regulares y matrices que permiten tanto calcular el número de caminos como enumerar dichos caminos expresados como series de arcos del grafo de flujo (Vea el ANEXO 2 para guiarse en la construcción de un grafo de flujo). Saber cuál es el número de caminos del grafo de un programa ayuda a planificar las pruebas y a asignar recursos a las mismas, ya que indica el número de ejecuciones necesarias. También sirve de comprobación a la hora de enumerar los caminos.

Los bucles constituyen el elemento de los programas que genera un mayor número de problemas para la cobertura de caminos. Su tratamiento no es sencillo ni siquiera adoptando el concepto de camino de prueba. Pensemos, por ejemplo, en el caso de varios bucles anidados o bucles que fijan valores mínimo y máximo de repeticiones.

Utilización de la complejidad ciclomática de McCabe.

La utilización de la métrica de McCabe ha sido muy popular en el diseño de pruebas desde su creación. Esta métrica es un indicador del número de caminos independientes que existen en un grafo. El propio McCabe definió como un buen criterio de prueba la consecución de la ejecución de un conjunto de caminos independientes, lo que implica probar un número de caminos igual al de la métrica. Se propone este criterio como equivalente a una cobertura de decisiones, aunque se han propuesto contraejemplos que invalidan esta suposición.

La complejidad de McCabe V (G) se puede calcular de las tres maneras siguientes a partir de un grafo de flujo G:

1. V (G) = *a* - *n* + *2*, siendo *a* el número de arcos o aristas del grafo y *n* el número de nodos.
2. V (G) = *r*, siendo *r* el número de regiones cerradas del grafo.
3. V (G) = *c* + *1*, siendo *c* el número de nodos de condición

Veamos cómo se aplican estas fórmulas sobre el grafo de flujo de la figura 7.5:

1. V (G) = 14 - 11 + 2 = 5. Los arcos han sido identificados con las marcas desde a1 hasta a14. Los nodos están numerados del 1 al 11.

2. V (G) = 5. Las regiones o áreas cerradas (limitadas por aristas) del grafo son cinco. Como puede verse, se ha marcado un área (región 5) añadiendo un arco ficticio desde el nodo 11 al nodo 1. Esto se debe a que las fórmulas de McCabe sólo son aplicables a grafos fuertemente conexos, es decir, aquellos para los cuales existe un camino entre cualesquiera dos nodos que se elijan. Los programas, con un nodo de inicio y otro de final, no cumplen esta condición. Por eso, debemos marcar dicho arco o, como alternativa, contabilizar la región externa al grafo como una más.

3. V (G) = 4 + 1. Los nodos de condición son el 2, el 4, el 5 y el 6. Todos ellos son nodos de decisión binaria, es decir, surgen dos aristas de ellos. En el caso de que de un nodo de condición (por ejemplo, una sentencia Case-of) partiera n arcos (n>2), debería

contabilizarse como n-1 para la fórmula (que equivale al número de bifurcaciones binarias necesarias para simular dicha bifurcación “n-aria”).

Una vez calculado el valor V (G) podemos afirmar que el número máximo de caminos independientes de dicho grafo es cinco. El criterio de prueba de McCabe consiste en elegir cinco caminos que sean independientes entre sí y crear casos de prueba cuya ejecución siga dichos caminos. Para ayudar a la elección de dichos caminos, McCabe creó un procedimiento llamado “**método del camino básico**”, consistente en realizar variaciones sobre la elección de un primer camino de prueba típico denominado camino básico.

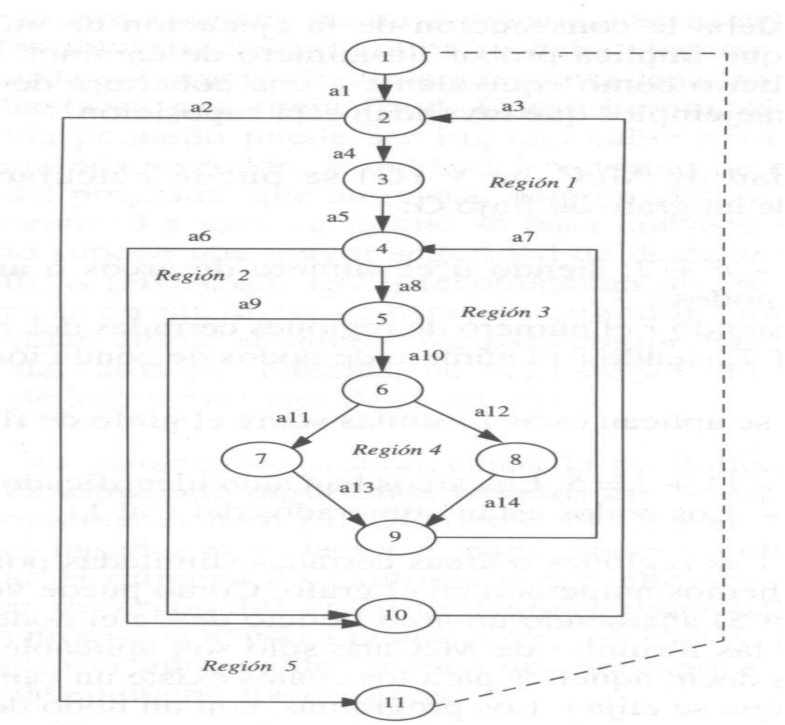


***Figura 7.5**. Cálculo de la complejidad ciclomática sobre un grafo*.

Para el caso, un posible conjunto de caminos (descritos como secuencias de nodos visitados) podría ser el siguiente:

- 1-2-11
- 1-2-3-4-10-2
- 1-2-3-4-5-10-2
- 1-2-3-4-5-6-7-9-4-10-2-11
- 1-2-3-4-5-6-8-9-4-10-2-11

Se han subrayado los elementos de cada camino que lo hacen independiente de los demás. Conviene aclarar que algunos de los caminos quizás no se puedan ejecutar solos y requieran una concatenación con algún otro. A partir de estos caminos, el diseñador de las pruebas debe analizar el código para saber los datos de entrada necesarios para forzar la ejecución de cada uno de ellos. Una vez determinados los datos de entrada hay que consultar la especificación para averiguar cuál es la salida teóricamente correcta para cada caso.

Puede ocurrir también que las condiciones necesarias para que la ejecución pase por un determinado camino no se puedan satisfacer de ninguna manera. Nos encontraríamos entonces ante un “camino imposible”. En ese caso, debemos sustituir dicho camino por otro

posible que permita satisfacer igualmente el criterio de prueba de McCabe, es decir, que ejecute la misma arista o flecha que diferencia al imposible de los demás caminos independientes.

b) Prueba funcional. (Prueba de la caja negra)

La prueba funcional o de caja negra se centra en el estudio de la especificación del software, del análisis de las funciones que debe realizar, de las entradas y de las salidas. Lamentablemente, la prueba exhaustiva de caja negra también es generalmente impracticable: pensemos en el ejemplo de la suma (suma dos números enteros de dos cifras del 0 al 99). De nuevo, ya que no podemos ejecutar todas las posibilidades de funcionamiento y todas las combinaciones de entradas y de salidas, debemos buscar criterios que permitan elegir un subconjunto de casos cuya ejecución aporte una cierta confianza en detectar los posibles defectos del software. Para fijar estas pautas de diseño de pruebas, nos apoyaremos en las siguientes dos definiciones de Myers que definen qué es un caso de prueba bien elegido:

- El que reduce el número de otros casos necesarios para que la prueba sea razonable. Esto implica que el caso ejecute el máximo número de posibilidades de entrada diferentes para así reducir el total de casos.

- Cubre un conjunto extenso de otros casos posibles, es decir, nos indica algo acerca de la ausencia o la presencia de defectos en el conjunto específico de entradas que prueba, así como de otros conjuntos similares.

Veremos a continuación distintas técnicas de diseño de casos de caja negra.

Particiones o clases de equivalencia

Esta técnica utiliza las cualidades que definen un buen caso de prueba de la siguiente manera:

- Cada caso debe cubrir el máximo número de entradas.

- Debe tratarse el dominio de valores de entrada dividido en un número finito de clases de equivalencia que cumplan la siguiente propiedad: la prueba de un valor representativo de una clase permite suponer “razonablemente” que el resultado obtenido (existan defectos o no) será el mismo que el obtenido probando cualquier otro valor de la clase.

El método de diseño de casos consiste entonces en:

- Identificación de clases de equivalencia.
- Creación de los casos de prueba correspondientes.

Para identificar las posibles clases de equivalencia de un programa a partir de su especificación se deben seguir los siguientes pasos:

1. Identificación de las condiciones de las entradas del programa, es decir, restricciones de formato o contenido de los datos de entrada.

2. A partir de ellas, se identifican clases de equivalencia que pueden ser:

   - De datos válidos.
   - De datos no válidos o erróneos.

   La identificación de las clases se realiza basándose en el principio de igualdad de tratamiento: todos los valores de la clase deben ser tratados de la misma manera por el programa.

3. Existen algunas reglas que ayudan a identificar clases:

   - Si se especifica un rango de valores para los datos de entrada (por ejemplo, “el número estará comprendido entre 1 y 49”), se creará una clase válida (1 ≤ número ≤ 49) y dos clases no válidas (número < 1 y número > 49).

   - Si se especifica un número de valores (por ejemplo, “se pueden registrar de uno a tres propietarios de un piso”), se creará una clase válida (1 ≤ propietarios ≤ 3) y dos no válidas (propietarios < 1 y propietarios > 3).

   - Si se especifica una situación del tipo “debe ser” o booleana (por ejemplo, “el primer carácter debe ser una letra”), se identifican una clase válida (“es una letra”) y una no válida (“no es una letra”).

   - Si se especifica un conjunto de valores admitidos (por ejemplo, “pueden registrarse tres tipos de inmuebles: pisos, chalés y locales comerciales”) y se sabe que el programa trata de forma diferente cada uno de ellos, se identifica una clase válida por cada valor (en este caso son tres: piso, chalé y local) y una no válida (cualquier otro caso: por ejemplo, plaza de garaje).

   - En cualquier caso, si se sospecha que ciertos elementos de una clase no se tratan igual que el resto de la misma, deben dividirse en clases menores.

La aplicación de estas reglas para la derivación de clases de equivalencia permite desarrollar los casos de prueba para cada elemento de datos del dominio de entrada. La división en clases deberían realizarla personas independientes al proceso de desarrollo del programa, ya que, si lo hace la persona que preparó la especificación o diseñó el software, la existencia de algunas clases (en concreto, las no consideradas en el tratamiento) no será, probablemente, reconocida.

El último paso del método es el uso de las clases de equivalencia para identificar los casos de prueba correspondientes. Este proceso consta de las siguientes fases:

1. Asignación de un número único a cada clase de equivalencia.

2. Hasta que todas las clases de equivalencia hayan sido cubiertas por (incorporadas a) casos de prueba, se tratará de escribir un caso que cubra tantas clases válidas no incorporadas como sea posible.

3. Hasta que todas las clases de equivalencia no válidas hayan sido cubiertas por casos de prueba, escribir un caso para una única clase no válida sin cubrir.

La razón de cubrir con casos individuales las clases no válidas es que ciertos controles de entrada pueden enmascarar o invalidar otros controles similares. Por ejemplo, en un programa donde hay que “introducir cantidad (1-99) y letra inicial (A-Z)” ante el caso “105 &” (dos errores), se puede indicar sólo el mensaje “105 fuera de rango de cantidad” y dejar sin examinar el resto de la entrada (el error de introducir “&” en vez de una letra).

Veamos un ejemplo de aplicación de la técnica. Se trata de una aplicación bancaria en la que el operador deberá proporcionar un código, un nombre para que el usuario identifique la operación (por ejemplo, “nómina”) y una orden que disparará una serie de funciones bancarias.

Especificación

- Código área: número de 3 dígitos que no empieza por 0 ni por 1.
- Nombre de identificación: 6 caracteres.
- Órdenes posibles: “cheque”, “depósito”, “pago factura”, “retirada de fondos”.

Aplicación de las reglas

- Código
  - número, regla 3, booleana:
    - clase válida (número)
    - clase no válida (no es número)
  - regla 5, la clase número debe subdividirse; por la regla 1, rango, obtenemos:
    - subclase válida (200 código 999)
    - dos subclases no válidas (código < 200; código > 999)

- Nombre de id., número específico de valores, regla 2:
  - clase válida (6 caracteres)
  - dos clases no válidas (más de 6; menos de 6 caracteres)

- Orden, conjunto de valores, regla 4:
  - una clase válida para cada orden ("cheque", "depósito" ...); 4 en total.
  - una clase no válida ("divisas", por ejemplo).

En la tabla 7.1 se han enumerado las clases identificadas y la generación de casos (presuponiendo que el orden de entrada es código-nombre-orden) se ofrece a continuación.

Condición de entrada	Clases válidas	Clases inválidas
Código área	(1) 200 ≤ código ≤ 999	(2) código <200 (3) código >999 (4) no es número
Nombre para identificar la operación	(5) seis caracteres	(6) menos de 6 caracteres (7) más de 6 caracteres
Orden	(8) «cheque» (9) «depósito» (10) «pago factura» (11) «retirada fondos»	(12) ninguna orden válida

Tabla 7.1. Tabla de clases de equivalencia del ejemplo.

- Casos válidos:

  - 300 Nómina Depósito (1) (5) (9)
  - 400 Viajes Cheque (1) (5) (8)
  - 500 Coches Pago-factura (1) (5) (10)
  - 600 Comida Retirada-fondos (1) (5) (11)

- Casos no válidos:

  - 180 Viajes Pago-factura (2) (5) (10)
  - 1032 Nómina Depósito (3) (5) (9)
  - XY Compra Retirada-fondos (4) (5) (11)
  - 350 A Depósito (1) (6) (9)
  - 450 Regalos Cheque (1) (7) (8)
  - 550 Casa &%4 (1) (5) (12)

Análisis de valores límite (AVL).

Mediante la experiencia (e incluso a través de demostraciones) se ha podido constatar que los casos de prueba que exploran las condiciones límite de un programa producen un mejor

resultado para la detección de defectos, es decir, es más probable que los defectos del software se acumulen en estas condiciones. Podemos definir las condiciones límite como las situaciones que se hallan directamente arriba, abajo y en los márgenes de las clases de equivalencia.

El análisis de valores límite es un técnica de diseño de casos que complementa a la de particiones de equivalencia. Las diferencias entre ambas son las siguientes:

1. Más que elegir "cualquier" elemento como representativo de una clase de equivalencia, se requiere la selección de uno o más elementos tal que los márgenes se sometan a prueba.

2. Más que concentrarse únicamente en el dominio de entrada (condiciones de entrada), los casos de prueba se generan considerando también el espacio de salida.

El proceso de selección de casos es también heurístico, aunque existen ciertas reglas orientativas. Aunque parezca que el AVL es simple de usar (a la vista de las reglas), su aplicación tiene múltiples matices que requieren un gran cuidado a la hora de diseñar las pruebas. Las reglas para identificar clases son las siguientes:

1. Si una condición de entrada especifica un rango de valores ("-l.0 valor +1.0") se deben generar casos para los extremos del rango (-1.0 y +1.0) y casos no válidos para situaciones justo más allá de los extremos (- 1.001 y + 1.001, en el caso en que se admitan 3 decimales).

2. Si la condición de entrada especifica un número de valores ("el fichero de entrada tendrá de 1 a 255 registros"), hay que escribir casos para los números máximo, mínimo, uno más del máximo y uno menos del mínimo de valores (0, 1, 255 y 256 registros).

3. Usar la regla 1 para la condición de salida ("el descuento máximo aplicable en compra al contado será el 50%, el mínimo será el 6%"). Se escribirán casos para intentar obtener descuentos de 5,99%, 6%, 50% y 50,01 %.

4. Usar la regla 2 para cada condición de salida ("el programa puede mostrar de 1 a 4 listados"). Se escriben casos para intentar generar 0, 1, 4 y 5 listados.

   En esta regla, como en la 3, debe recordarse que:
   - Los valores límite de entrada no generan necesariamente los valores límite de salida (recuérdese la función seno, por ejemplo).
   - No siempre se pueden generar resultados fuera del rango de salida (pero es interesante considerarlo).

5. Si la entrada o la salida de un programa es un conjunto ordenado (por ejemplo, una tabla, un archivo secuencial, etc.), los casos se deben concentrar en el primero y en el último elemento.

Es recomendable utilizar el ingenio para considerar todos los aspectos y matices, a veces sutiles, en la aplicación del AVL.

Conjetura de errores.

La idea básica de esta técnica consiste en enumerar una lista de equivocaciones que pueden cometer los desarrolladores y de las situaciones propensas a ciertos errores. Después se generan casos de prueba en base a dicha lista (se suelen corresponder con defectos que aparecen comúnmente y no con aspectos funcionales). Esta técnica también se ha denominado generación de casos (o valores) especiales, ya que no se obtienen en base a otros métodos sino mediante la intuición o la experiencia.

No existen directrices eficaces que puedan ayudar a generar este tipo de casos, ya que lo único que se puede hacer es presentar algunos ejemplos típicos que reflejan esta técnica. Algunos valores a tener en cuenta para los casos especiales son los siguientes:

- El valor cero es una situación propensa a error tanto en la salida como en la entrada.
- En situaciones en las que se introduce un número variable de valores (por ejemplo, una lista), conviene centrarse en el caso de no introducir ningún valor y en el de un solo valor. También puede ser interesante una lista que tiene todos los valores iguales.
- Es recomendable imaginar que el programador pudiera haber interpretado algo mal en la especificación.
- También interesa imaginar lo que el usuario puede introducir como entrada a un programa. Se dice que se debe preveer toda clase de acciones de un usuario como si fuera "completamente tonto" o, incluso, como si quisiera sabotear el programa.

c) Pruebas aleatorias.

En las pruebas aleatorias simulamos la entrada habitual del programa creando datos de entrada en la secuencia y con la frecuencia con las que podrían aparecer en la práctica, de forma continua sin parar., Esto implica usar una herramienta denominada un generador de pruebas, a las que se alimenta con una descripción de las entradas y las secuencias de entrada posibles y su probabilidad de ocurrir en la práctica. Este enfoque de prueba es muy común en la prueba de compiladores en la que se generan aleatoriamente códigos de programas que sirven de casos de prueba para la compilación.
Si el proceso de generación se ha realizado correctamente, se crearán eventualmente todas las posibles entradas del programa en todas las posibles combinaciones y permutaciones. También se puede conseguir, indicando la distribución estadística que siguen, que la frecuencia de las entradas sea la apropiada para orientar correctamente nuestras pruebas hacia lo que es probable que suceda en la práctica. No obstante, esta forma de diseñar casos de prueba es menos utilizada que las técnicas de caja blanca y de caja negra.

Enfoque práctico recomendado para el diseño de casos.

Los dos enfoques estudiados, caja blanca y caja negra, representan aproximaciones diferentes para las pruebas. El enfoque práctico recomendado para el uso de las técnicas de diseño de casos pretende mostrar el uso más apropiado de cada técnica para la obtención de un conjunto de casos útiles sin perjuicio de las estrategias de niveles de prueba:

1. Si la especificación contiene combinaciones de condiciones de entrada, comenzar formando sus grafos de causa-efecto (ayudan a la comprensión de dichas combinaciones).
2. En todos los casos, usar el análisis de valores-límites para añadir casos de prueba: elegir límites para dar valores a las causas en los casos generados asumiendo que cada causa es una clase de equivalencia.
3. Identificar las clases válidas y no válidas de equivalencia para la entrada y la salida, y añadir los casos no incluidos anteriormente (¿cada causa es una única clase de equivalencia? ¿Deben dividirse en clases?).
4. Utilizar la técnica de conjetura de errores para añadir nuevos casos, referidos a valores especiales.
5. Ejecutar los casos generados hasta el momento (de caja negra) y analizar la cobertura obtenida.

6. Examinar la lógica del programa para añadir los casos precisos (de caja blanca) para cumplir el criterio de cobertura elegido si los resultados de la ejecución del punto 5 indican que no se ha satisfecho el criterio de cobertura elegido (que figura en el plan de pruebas).

Aunque éste es el enfoque integrado para una prueba "razonable", en la práctica la aplicación de las distintas técnicas está bastante discriminada según la etapa de la estrategia de prueba.

Otra cuestión importante en relación a los dos tipos de diseño de pruebas es: ¿por qué incluir técnicas de caja blanca para explorar detalles lógicos y no centrarnos mejor en probar los requisitos funcionales con técnicas de caja negra? (si el programa realiza correctamente las funciones ¿por qué hay que intentar probar un determinado número de caminos?).

- Los errores lógicos y las suposiciones incorrectas son inversamente proporcionales a la probabilidad de que se ejecute un camino del programa (a menor probabilidad de ejecutarse un camino, mayor número de errores).
- Se suele creer que un determinado camino lógico tiene pocas posibilidades de ejecutarse cuando, de hecho, se puede ejecutar regularmente.
- Los errores tipográficos son aleatorios; pueden aparecer en cualquier parte del programa (sea muy usada o no).
- La probabilidad y la importancia de un trozo de código suele ser calculada de forma muy subjetiva.

Debemos recordar que tanto la prueba exhaustiva de caja blanca como de caja negra son impracticables. ¿Bastaría, no obstante, una prueba exhaustiva de caja blanca solamente? Véase el siguiente programa:

```
if ((x+y+z)/13 = x)
      printf("X, Y y Z son iguales");
else
      printf("X, Y y Z no son iguales");
```

En este programa, una prueba exhaustiva de caja blanca (que pase por todos los caminos) no asegura necesariamente la detección de los defectos de su diseño. Véase, por ejemplo, cómo los dos casos siguientes no detectan ningún problema en el programa:

- x=5, y=5, z=5
- x=2, y=3, z=7[26]

Estrategias de prueba del software. Niveles de prueba

- **Prueba de unidad:** es la prueba de cada módulo, que normalmente realiza el propio personal de desarrollo en su entorno

- **Prueba de integración:** con el esquema del diseño del software, los módulos probados se integran para comprobar sus interfaces en el trabajo conjunto

- **Prueba de validación:** el software totalmente ensamblado se prueba como un todo para comprobar si cumple los requisitos funcionales y de rendimiento, facilidad de mantenimiento, recuperación de errores, etc.

- **Prueba del sistema:** el software ya validado se integra con el resto del sistema (rendimiento, seguridad, recuperación y resistencia)

- **Prueba de aceptación:** el usuario comprueba en su propio entorno de explotación si lo acepta como está o no

[26] El caso x=2, y=3, z=1 sí permitiría detectar el defecto de diseño de la decisión.

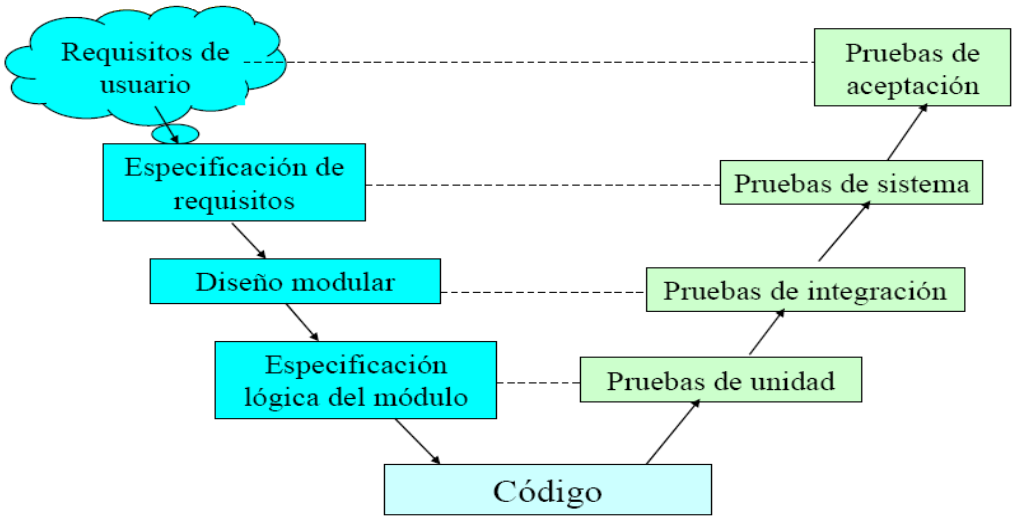


***Figura 7.6**: Relación entre productos de desarrollo y niveles de prueba*

5. Mantenimiento de Software:

El mantenimiento de software es una de las actividades más comunes en la Ingeniería de Software y es el proceso de mejora y optimización del software desplegado (es decir; revisión del programa), así como también corrección de los defectos

La fase de mantenimiento de software involucra cambios al software en orden de corregir defectos y dependencias encontradas durante su uso tanto como la adición de nueva funcionalidad para mejorar la usabilidad y aplicabilidad del software.

El mantenimiento del software involucra varias técnicas específicas. Una técnica es el rebanamiento estático, la cual es usada para identificar todo el código de programa que puede modificar alguna variable. Es generalmente útil en la refabricación del código del programa y fue específicamente útil en asegurar conformidad para el problema del Año 2000.

En un ambiente formal de desarrollo de software, la organización o equipo de desarrollo tendrán algún mecanismo para documentar y rastrear defectos y deficiencias. El Software tan igual como la mayoría de otros productos, es típicamente lanzado con un conjunto conocido de defectos y deficiencias. El software es lanzado con esos defectos conocidos porque la organización de desarrollo decide que la utilidad y el valor del software en un determinado nivel de calidad compensa el impacto de los defectos y deficiencias conocidas.

Las deficiencias conocidas son normalmente documentadas en una carta de consideraciones operacionales o notas de lanzamiento (release notes) es así que los usuarios del software serán capaces trabajar evitando las deficiencias conocidas y conocerán cuando el uso del software sería inadecuado para tareas específicas.

Con el lanzamiento del software (software release), otros defectos y deficiencias no documentados serán descubiertas por los usuarios del software. Tan pronto como estos defectos sean reportados a la organización de desarrollo, serán ingresados en el sistema de rastreo de defectos.

Las personas involucradas en la fase de mantenimiento de software esperan trabajar en estos defectos conocidos, ubicarlos y preparar un nuevo lanzamiento del software, conocido como un lanzamiento de mantenimiento, el cual resolverá los temas pendientes.

Tipos de mantenimiento

A continuación se señalan los tipos de mantenimientos existentes, y entre paréntesis el porcentaje aproximado respecto al total de operaciones de mantenimiento:

- **Perfectivo** (60%): mejora del software (rendimiento, flexibilidad, reusabilidad..) o implementación de nuevos requisitos. También se conoce como mantenimiento evolutivo.
- **Adaptativo** (18%): adaptación del software a cambios en su entorno tecnológico (nuevo hardware, otro sistema de gestión de bases de datos, otro sistema operativo...)
- **Correctivo** (17%): corrección de fallos detectados durante la explotación.

- **Preventivo** (5%): facilitar el mantenimiento futuro del sistema (verificar precondiciones, mejorar legibilidad...).

1. Elabore un plan de pruebas para el proyecto que viene desarrollando
2. Compare las
3. Elabore un plan de mantenimiento para el proyecto que viene desarrollando
4. Elabore un cuadro resumen para los tres enfoques principales del diseño de pruebas (estructural, funcional y aleatorio) y liste los casos en que usaría cada tipo de prueba
5. Elija un sistema operativo que Ud. conozca e investigue acerca de los tipos de prueba que deben soportar antes de salir al mercado

Las pruebas permiten verificar y validar el software cuando ya está en forma de código ejecutable.

El mantenimiento permite la mejora y optimización del software desplegado, así como también la corrección de los defectos

[1] PRESSMAN, Roger S. **Ingeniería del Software. Un enfoque práctico.** Mc Graw Hill, Ed. Interamericana de España S.A.U, 2006

[2] SOMMERVILLE, Ian. **Software engineering.** Addison-Wesley, 2004

[3] DE MILLO, R. **Software Testing and Evaluation**, Benjamin/Cummings Pub. 1987SOMMERVILLE, I., **Ingeniería de Software**, Ed. Pearson Educación, 2002

En la siguiente unidad temática tocaremos extensiones de los temas desarrollados anteriormente

INGENIERIA DEL SOFTWARE
UNIDAD ACADÉMICA Nº 7

NOMBRE:_____

APELLIDOS:_____FECHA; ____/_____/_____

CIUDAD:_____SEMESTRE:_____

1. ¿Qué tipo de pruebas deben realizarse cuando se modifica un módulo de un programa?

   _____
   _____
   _____
   _____

2. Si al realizar pruebas de caja negra se encuentran errores ¿Qué acciones deben llevarse a cabo?

   _____
   _____
   _____
   _____

3. Para que la persona encargada de las pruebas pueda elaborar los casos de prueba usando la caja negra, el analista debe proporcionarle:

   _____
   _____
   _____
   _____

4. Para que la persona encargada de las pruebas pueda elaborar los casos de prueba usando la caja blanca, el analista debe proporcionarle:

   _____
   _____
   _____
   _____

5. Diseñe un conjunto de casos de prueba de caja blanca para el siguiente procedimiento Pascal, de manera que se asegure la cobertura de sentencias.

```
procedure búsqueda_binaria( var p: indice; x : integer; L : array [1..MAX]
of elemento; n : integer);

{Busca la llave "x" en el array ordenado "L", y devuelve en "p" la posición
de la llave si se encuentra, de otro modo p=0; "n" indica el número de
elementos válidos en "L"}

{"elemento" es un registro que contiene, entre otros, un campo "llave" con
el que se compara el elemento a buscar}

var
        tope, base,
        medio : integer; {"medio" será el índice de x cuando se encuentre
en L}
begin
        tope := n;
        base := 1;
        repeat
                medio := (tope+base) div 2;
                if x < L[medio].llave then
                        tope := medio – 1
                else
                        base := medio +1;
        until (x=L[medio].llave) or (tope < base);
        if x = L[medio].llave then
                p := medio
        else
                p := 0;
end;
```

6. Diseñe una Interfaz y sugiera algunos casos de prueba de Caja Negra

Unidad Académica VIII

TOPICOS AVANZADOS DE INGENIERIA DEL SOFTWARE

La ingeniería de Software se ha relacionado con temas actuales como son BMP (Bussiness Process Modeling), CMMI, desarrollo de software basado en componentes, etc; temas en las que las empresas están incursionado como una posible alternativa de solución a sus problemas de software.

En esta unidad hablaremos acerca de estos temas para tener una visión panorámica de la dirección que va tomando la Ingeniería de software en estos tiempos.

Al finalizar el estudio de la presente unidad temática el estudiante:

- Conoce temas de vanguardia en Ingeniería de Software.
- Genera una visión de la dirección que va tomando la Ingeniería del software en nuestros días

1. Arquitectura de software

En Los inicios de la informática, la programación se consideraba un arte, debido a la dificultad que entrañaba para la mayoría de los mortales, pero con el tiempo se han ido desarrollando metodologías para conseguir nuestros propósitos. Y a todas estas técnicas se les ha dado en llamar Arquitectura Software.

Una Arquitectura Software, también denominada Arquitectura lógica, consiste en un conjunto de patrones y abstracciones coherentes que proporcionan el marco de referencia necesario para guiar la construcción del software para un sistema de información.

La arquitectura software establece los fundamentos para que analistas, diseñadores, programadores, etc. trabajen en una línea común que permita alcanzar los objetivos y necesidades del sistema de información.

Una arquitectura software se selecciona y diseña con base en unos objetivos y restricciones. Los objetivos son aquellos prefijados para el sistema de información, pero no solamente los de tipo funcional, también otros objetivos como la mantenibilidad, auditabilidad, flexibilidad e interacción con otros sistemas de información. Las restricciones son aquellas limitaciones derivadas de las tecnologías disponibles para implementar sistemas de información. Unas arquitecturas son más recomendables de implementar con ciertas tecnologías mientras que otras tecnologías no son aptas para determinadas arquitecturas.

Por ejemplo, no es viable emplear una arquitectura software de tres capas para implementar sistemas en tiempo real.

La arquitectura software define, de manera abstracta, los componentes que llevan a cabo alguna tarea de computación, sus interfaces y la comunicación ente ellos. Toda arquitectura

software debe ser implementable en una arquitectura física, que consiste simplemente en determinar qué computadora tendrá asignada cada tarea de computación.

La arquitectura de software, tiene que ver con el diseño y la implementación de estructuras de software de alto nivel. Es el resultado de ensamblar un cierto número de elementos arquitectónicos de forma adecuada para satisfacer la mayor funcionalidad y requerimientos de desempeño de un sistema, así como requerimientos no funcionales, como la confiabilidad, escalabilidad, portabilidad, y disponibilidad.[27]

La definición oficial de Arquitectura del Software es la IEEE Std 1471-2000 que reza así: "La Arquitectura del Software es la organización fundamental de un sistema formada por sus componentes, las relaciones entre ellos y el contexto en el que se implantarán, y los principios que orientan su diseño y evolución".

Breve reseña histórica

En los años 1960 ya se acariciaba el concepto de arquitectura software en los círculos de investigación (por ejemplo, por Edsger Dijkstra). No obstante, toma popularidad en los años 1990 tras reconocerse la denominada crisis del software y como tema de interés de la incipiente disciplina de la ingeniería del software.

Modelos o vistas

Toda arquitectura software debe describir diversos aspectos del software. Generalmente, cada uno de estos aspectos se describe de una manera más comprensible si se utilizan distintos modelos o vistas. Es importante destacar que cada uno de ellos constituye una descripción parcial de una misma arquitectura y es deseable que exista cierto solapamiento entre ellos. Esto es así porque todas las vistas deben ser coherentes entre sí, evidente dado que describen la misma cosa.

Cada paradigma de desarrollo exige diferente número y tipo de vistas o modelos para describir una arquitectura. No obstante, existen al menos tres vistas absolutamente fundamentales en cualquier arquitectura:

- La visión **estática**: describe qué componentes tiene la arquitectura.
- La visión **funcional**: describe qué hace cada componente.
- La visión **dinámica**: describe cómo se comportan los componentes a lo largo del tiempo y como interactúan entre sí.

Las vistas o modelos de una arquitectura pueden expresarse mediante uno o varios lenguajes. El más obvio es el lenguaje natural, pero existen otros lenguajes tales como los diagramas de estado, los diagramas de flujo de datos, etc. Estos lenguajes son apropiados únicamente para un modelo o vista. Afortunadamente existe cierto consenso en adoptar UML (Unified Modeling Language, lenguaje unificado de modelado) como lenguaje único para todos los modelos o vistas. Sin embargo, un lenguaje generalista corre el peligro de no ser capaz de describir determinadas restricciones de un sistema de información (o expresarlas de manera incomprensible).

Arquitecturas más comunes

Generalmente, no es necesario inventar una nueva arquitectura software para cada sistema de información. Lo habitual es adoptar una arquitectura conocida en función de sus ventajas e inconvenientes para cada caso en concreto. Así, las arquitecturas más universales son:

- **Monolítica.** Donde el software se estructura en grupos funcionales muy acoplados.

[27] Kruchten, Philippe, 2000

- **Cliente-servidor**. Donde el software reparte su carga de cómputo en dos partes independientes pero sin reparto claro de funciones.

- **Arquitectura de tres niveles.** Especialización de la arquitectura cliente-servidor donde la carga se divide en tres partes con un reparto claro de funciones: una capa para la presentación, otra para el cálculo y otra para el almacenamiento. Una capa solamente tiene relación con la siguiente.

Ejemplos de Arquitectura del Software: J2EE y MVC

Para ilustrar un poco lo que se ha explicado hasta ahora, a continuación se muestran dos diagramas de arquitectura en un entorno J2EE. Ambos diagramas están disponibles en Designing Enterprise Applications with the J2EE Platform, Second Edition
El primer diagrama consiste en una vista lógica que muestra los componentes y servicios típicos de un entorno J2EE.

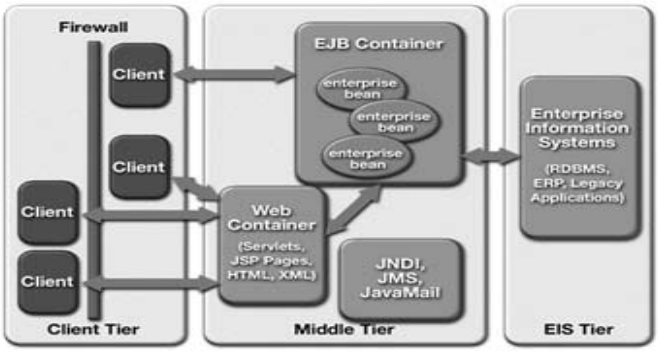


***Figura 8.1**: Entorno J2EE*

El segundo diagrama es una vista de proceso que muestra las relaciones entre las capas model, view y controller de la arquitectura MVC bajo J2EE.

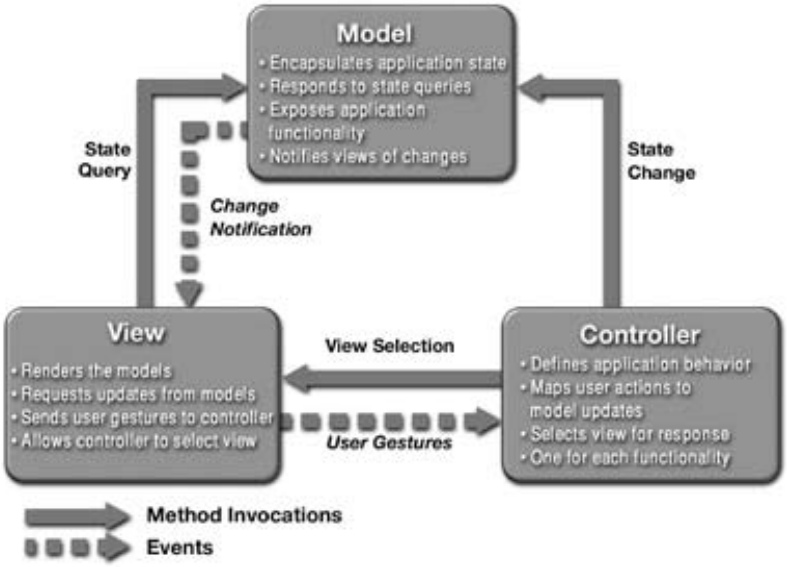


***Figura 8.2**:Arquitectura Modelo, Vista, Controlador*

2. Ingeniería del Software Basado en Componentes

La complejidad de los sistemas computacionales actuales nos ha llevado a buscar la reutilización del software existente. El desarrollo de software basado en componentes permite reutilizar piezas de código preelaborado que permiten realizar diversas tareas, conllevando a diversos beneficios como las mejoras a la calidad, la reducción del ciclo de desarrollo y el mayor retorno sobre la inversión. Al comparar la evolución del ambiente de IT con el crecimiento de las metrópolis actuales, podemos entender el origen de muchos problemas que se han presentado históricamente en la construcción de software y vislumbrar las posibles y probables soluciones que nos llevarán hacia la industrialización del software moderno.

Este proceso de industrialización ha dado ya sus inicios con implementaciones como la plataforma .net o J2EE, las cuales impulsan la idea de industrializar el software utilizando tecnologías de componentes. Los avances y mejoras presentados en estas plataformas convierten a sus componentes en verdaderas piezas de ensamblaje. Asimismo, los nuevos paradigmas como las Fábricas de Software proveen de los medios para hacer la transición desde el 'hacer a mano' hacia la fabricación o manufactura de software.

Si hay algo que ha aprendido el ser humano desde tiempos muy antiguos es a reutilizar el conocimiento existente para sus cada vez más ambiciosas empresas. En efecto, al reutilizar trozos de experiencias, ideas y artefactos, no solo nos aseguramos de no cometer los mismos errores del pasado, sino que logramos construir cosas cada vez más grandes y maravillosas, con bases firmes y calidad incomparable. Este concepto de la reutilización, uno de los primeros que se nos enseñan a quienes entramos al mundo del desarrollo de software, habremos de utilizarlo desde el mismo instante en que escribamos nuestra primera línea de código.

Los sistemas de hoy en día son cada vez más complejos, deben ser construidos en tiempo récord y deben cumplir con los estándares más altos de calidad. Para hacer frente a esto, se concibió y perfeccionó lo que hoy conocemos como Ingeniería de Software Basada en Componentes (ISBC), la cual se centra en el diseño y construcción de sistemas computacionales que utilizan componentes de software reutilizables. Esta ciencia trabaja bajo la filosofía de "comprar, no construir", una idea que ya es común en casi todas las industrias existentes, pero relativamente nueva en lo que a la construcción de software se refiere.

Para este momento, ya muchos conocen las ventajas de este enfoque de desarrollo y, de la misma forma, muchos se preguntan día a día el por qué son tan pocos los que realmente alcanzan el éxito siguiendo esta filosofía. En realidad, hasta ahora solo hemos tanteado un poco con las posibilidades del software basado en componentes, y es justo hora, en la presente década, que la industria del software despegará y se perfeccionará para estar a la par de cualquier otra industria del medio. Las analogías que nos han llevado a estudiar a los sistemas comparándolos con las complejas metrópolis de la actualidad, así como las iniciativas más innovadoras como las Fábricas de Software de Microsoft, son la clara representación de que estamos a punto de presenciar un nuevo gran cambio en la forma como pensamos en software.

a) Beneficios del Desarrollo de Software basado en Componentes

En esencia, un componente es una pieza de código preelaborado que encapsula alguna funcionalidad expuesta a través de interfaces estándar**.** Los componentes son los "ingredientes de las aplicaciones", que se juntan y combinan para llevar a cabo una tarea**.** Es algo muy similar a lo que podemos observar en el equipo de música que tenemos en nuestra sala. Cada componente de aquel aparato ha sido diseñado para acoplarse perfectamente con sus pares, las conexiones son estándar y el protocolo de comunicación está ya preestablecido. Al unirse las partes, obtenemos música para nuestros oídos.
El paradigma de ensamblar componentes y escribir código para hacer que estos componentes funcionen se conoce como Desarrollo de Software Basado en Componentes. El uso de este paradigma posee algunas ventajas:

1. **Reutilización del software.** Nos lleva a alcanzar un mayor nivel de reutilización de software.

2. **Simplifica las pruebas.** Permite que las pruebas sean ejecutadas probando cada uno de los componentes antes de probar el conjunto completo de componentes ensamblados.

3. **Simplifica el mantenimiento del sistema.** Cuando existe un débil acoplamiento entre componentes, el desarrollador es libre de actualizar y/o agregar componentes según sea necesario, sin afectar otras partes del sistema.

4. **Mayor calidad.** Dado que un componente puede ser construido y luego mejorado continuamente por un experto u organización, la calidad de una aplicación basada en componentes mejorará con el paso del tiempo.

De la misma manera, el optar por comprar componentes de terceros en lugar de desarrollarlos, posee algunas ventajas:

1. **Ciclos de desarrollo más cortos.** La adición de una pieza dada de funcionalidad tomará días en lugar de meses ó años.

2. **Mejor ROI.** Usando correctamente esta estrategia, el retorno sobre la inversión puede ser más favorable que desarrollando los componentes uno mismo.

3. **Funcionalidad mejorada.** Para usar un componente que contenga una pieza de funcionalidad, solo se necesita entender su naturaleza, más no sus detalles internos. Así, una funcionalidad que sería impráctica de implementar en la empresa, se vuelve ahora completamente asequible.

3. CMMI (Capability Maturity Model Integration)

El CMMI es un modelo de referencia para la calidad en los procesos de desarrollo y mantenimiento de software, que incrementa la satisfacción de las necesidades de los usuarios internos del sistema (trabajadores), permitiendo la elaboración de productos de calidad, dentro del tiempo y costos previstos

El nacimiento de CMM - CMMI

El departamento de defensa de los estados unidos tenía muchos problemas con el software que encargaba desarrollar a otras empresas, los presupuestos se disparaban, las fechas alargaban más y más. ¿Quién no se ha encontrado con este tipo de problemas si ha trabajado con una empresa de software?

Como esta situación les parecía intolerable convocó un comité de expertos para que solucionase estos problemas, en el año 1983 dicho comité concluyó "Tienen que crear un instituto de la ingeniería del software, dedicado exclusivamente a los problemas del software, y a ayudar al Departamento de Defensa".

Convocaron un concurso público en el que dijeron: "Cualquiera que quiera enviar una solicitud tiene que explicar como van a resolver estos 4 problemas", se presentaron diversos estamentos y la Universidad Carnegie Mellon ganó el concurso en 1985, creando el SEI.

El SEI (Software Engineering Institute) es el instituto que creó y mantiene el modelo de calidad CMM - CMMI

¿Qué es el CMM - CMMI?

El CMM - CMMI es un modelo de calidad del software que clasifica las empresas en niveles de madurez. Estos niveles sirven para conocer la madurez de los procesos que se realizan para producir software.

Niveles CMM - CMMI

Los niveles CMM - CMMI son 5:

- **Inicial o Nivel 1 CMM - CMMI**. Este es el nivel en donde están todas las empresas que no tienen procesos. Los presupuestos se disparan, no es posible entregar el proyecto en fechas, te tienes que quedar durante noches y fines de semana para terminar un proyecto. No hay control sobre el estado del proyecto, **el desarrollo del proyecto es completamente opaco**, no sabes lo que pasa en él.

Es el típico proyecto en el que se da la siguiente situación:

- *¿Cómo va el proyecto?*
  - Bien, bien.

- Dos semanas después…
  - ¿Cómo va el proyecto?
  - Bien, bien.

- Tres semanas después…
  - El lunes hay que entregar el proyecto.
  - El lunes!!?. Todavía falta mucho!!
  - ¿Cómo? Me dijiste que el proyecto iba bien!! Arréglatelas como quieras, pero el proyecto tiene que estar terminado para el lunes.

Si no sabes el tamaño del proyecto y no sabes cuanto llevas hecho, nunca sabrás cuando vas a terminar.

- **Repetible o Nivel 2 CMM - CMMI**. Quiere decir que el éxito de los resultados obtenidos se pueden repetir. La principal diferencia entre este nivel y el anterior es que el proyecto es gestionado y controlado durante el desarrollo del mismo. El desarrollo no es opaco y se puede saber el estado del proyecto en todo momento.
  Los procesos que hay que implantar para alcanzar este nivel son:
  - Gestión de requisitos
  - Planificación de proyectos
  - Seguimiento y control de proyectos
  - Gestión de proveedores
  - Aseguramiento de la calidad
  - Gestión de la configuración
- **Definido o Nivel 3 CMM - CMMI**. Este nivel significa que la forma de desarrollar proyectos (gestión e ingeniería) esta definida, por definida quiere decir que esta establecida, documentada y que existen métricas (obtención de datos objetivos) para la consecución de objetivos concretos.

  Los procesos que hay que implantar para alcanzar este nivel son:
  - Desarrollo de requisitos
  - Solución Técnica
  - Integración del producto
  - Verificación
  - Validación
  - Desarrollo y mejora de los procesos de la organización
  - Definición de los procesos de la organización
  - Planificación de la formación
  - Gestión de riesgos
  - Análisis y resolución de toma de decisiones

  La mayoría de las empresas que llegan al nivel 3 paran aquí, ya que es un nivel que proporciona muchos beneficios y no ven la necesidad de ir más allá porque tienen cubiertas la mayoría de sus necesidades.

- **Cuantitativamente Gestionado o Nivel 4 CMM - CMMI**. Los proyectos usan objetivos medibles para alcanzar las necesidades de los clientes y la organización. Se usan métricas para gestionar la organización.

  Los procesos que hay que implantar para alcanzar este nivel son:
  - Gestión cuantitativa de proyectos
  - Mejora de los procesos de la organización

- **Optimizado o Nivel 5 CMM - CMMI**. Los procesos de los proyectos y de la organización están orientados a la mejora de las actividades. Mejoras incrementales e

innovadoras de los procesos que mediante métricas son identificadas, evaluadas y puestas en práctica.

Los procesos que hay que implantar para alcanzar este nivel son:

- Innovación organizacional
- Análisis y resolución de las causas

Normalmente las empresas que intentan alcanzar los niveles 4 y 5 lo realizan simultáneamente ya que están muy relacionados.

4. Business Process Modeling (BPM)

Qué és Business Process Modeling (BPM)?

Para comprender el Business Process Modeling es necesario explicar el Business Process Management (gestión de procesos empresariales). Este último es un conjunto de actividades que las empresas pueden desarrollar para optimizar o adaptar los procesos de la empresa al mercado o a la organización. Las empresas han estado realizando Business Process Management desde hace tiempo. El proceso habitual de diseño de una aplicación de software para una empresa se basa actualmente en traducir la lógica de procesos de la empresa a una lógica de programación con las tecnologías requeridas para su ejecución.
Últimamente, las empresas proporcionan herramientas de diseño y modelado de procesos empresariales (llamadas aplicaciones BPMS Business Process Modeling Systems) para facilitar las tareas de Business Process Management.

Situación Actual

La última evolución de las tecnologías de programación es la tecnología de WebServices(WS). Esta tecnología permite la ejecución remota de código (RPC) utilizando la red indiferentemente de la tecnología de programación implementada para ejecutar los procesos mediante el sistema de mensajes SOAP (Simple Object Access Protocol). El UDDI (Universal Description Discovery Interaction) permite la búsqueda de WebServices y WSDL (WebServices Description Language) describe los contenidos.
Todos estos lenguajes y protocolos se describen usando el formato XML.
Esta última evolución ha significado un salto importante en la interoperabilidad entre programas, empresas e instituciones. En empresas e instituciones permite la automatización de procesos mediante estas definiciones estándares y la reducción de costes correspondiente.

Evolución del BPM

El primer estudio sobre el futuro de las aplicaciones BPM se publicó en 2002[28],. Este previó el uso de aplicaciones BPMS, que permitirían el modelado gráfico de procesos, la simulación y ejecución.

Se formó entonces una iniciativa, BPMI (Business Process Management Iniciative), para estandarizar la notación gráfica del flujo de procesos. Esta comisión redactó el estándar gráfico BPMN (Business Process Modeling Notation) pero no describió ningún lenguaje escrito para describirlo. Por otro lado, la OMG (Object Management Group) ya tenía estandarizado un diagrama de flujo de procesos en su estándar UML (Unified Modeling Language). Este estándar no permite ni la ejecución ni simulación de flujos de procesos.

La organización WfMC(Workflow Management Coalition), organización para la realización de estándares de workflow entre aplicaciones, ya tenía un bosquejo de XPDL(XML Process Definition Language) a finales de 1999 y publicó su estándar a finales de 2002. Este estándar permite tanto la descripción gráfico de flujo de procesos como la ejecución y simulación, tanto la interacción entre máquinas como la interacción con los usuarios del flujo. En 2005 se ha

[28] Business Process Management – The Third Wave, Smith and Fingar

redactado un bosquejo sobre la versión 2.0 de XPDL que define íntegramente las definiciones BPMN.

A finales de 2002 se realizó un bosquejo de BPML (Business Process Management Language) una capa descriptiva de procesos síncronos. Se amplio entonces la descripción BPEL (Business Process Execution Language) para que pudiera incorporar la descripción de BPML y la ejecución de WebServices en un bosquejo de BPEL4WS. Pero este bosquejo, todavía no permite describir los conceptos descritos en BPMN. Esta definición describe solamente el lenguaje entre máquinas y flujos abiertos (flujos no cíclicos). Se están realizando otra serie de formatos para complementar esta definición (BPXL (Business Process eXtension Language) y BPQL (Business Process Query Language))
Desde muchos sectores se critica la sobre-estandarización de estos estándares y el marketing que se realiza en determinadas soluciones.

5. Reingeniería

No necesitamos inventar la rueda, solo debemos rehusarla creativamente..."

Las modernas organizaciones deben estar preparadas para enfrentar nuevos y feroces competidores, y un cambiante y desafiante mercado en el que hay que conocer de antemano los gustos y las necesidades de los clientes, las estrategias de la competencia y cualquier otra influencia del entorno para no quedar estancados y correr riesgos excesivos.

Para lograr esto, las empresas deben tener sus sistemas de información en línea, y preparados para responder al medio. Existen muchas herramientas que pueden ayudar al profesional en esta empresa de mantener con vida y en crecimiento a la organización; las mas vigentes, de mejores resultados y las que se valen de todas las otras para lograr un sistema eficiente son sin duda las herramientas de Mejora Continua, y Reingeniería.
La reingeniería es nada menos que una visión totalmente nueva de cómo se deben organizar y administrar los procesos, negocios y recursos humanos de una organización para que tengan éxito en la actualidad y en el nuevo milenio que se acerca.
Una breve definición de la reingeniería podría ser que significa "empezar de nuevo". En sí la reingeniería, formalmente es la revisión fundamental y el rediseño radical de procesos para alcanzar mejoras espectaculares en medidas críticas y contemporáneas de rendimiento, tales como costos, calidad, servicio y rapidez.
Cuatro palabras claves se incluyen en esta definición:

a. Fundamental

La reingeniería empieza sin ningún preconcepto, sin dar nada por sentado; no significa emprender la reingeniería de los supuestos, que la mayoría de procesos ya han arraigado en ellas. La reingeniería determina primero qué debe hacer la organización; luego, cómo debe hacerlo. No da nada por sentado; se olvida por completo de lo que es y se concentra en lo que debe ser y lo que se desea obtener.

b. Radical

Se tiene que rediseñar todo desde la "raíz", es decir no hacer cambios superficiales o tratar de arreglar lo que ya está hecho, sino rediseñar todas las estructuras y procedimientos existentes. Es reinventar la organización. El realizar cambios parciales o progresivos no es Reingeniería, puede ser la aplicación de Mejoramiento Continuo, cuyo tratado será motivo de otra Cultura Informática, en la que también se trate temas como Calidad Total.

c. Espectacular

La reingeniería no es cuestión de hacer mejoras marginales o incrementales sino de dar saltos gigantescos en rendimiento. La mejora marginal requiere afinación cuidadosa; la mejora espectacular exige volar lo viejo y cambiar por algo nuevo, que sea útil y no esté de moda necesariamente.

d. Procesos

Considerada la más importante de las cuatro. Muchas personas no están orientadas a los procesos; están enfocadas en tareas, oficios, personas, estructuras, pero no en procesos.
Se define un proceso como un conjunto de actividades que recibe uno o más "insumos" y crea un producto de valor, de utilidad para el cliente, para el ciudadano**.** Es necesario en nuestros

tiempos tomar conciencia de esta nueva forma de trabajo, que es lo mejor para todos, para la organización y para cada uno de sus integrantes, y por supuesto para el país.
El trabajar por objetivos y productos en plazos determinados, dentro de un planeamiento previamente elaborado, a nivel de la organización, e inclusive a nivel personal, es lo más beneficioso hoy en día. Debe pasar a la historia, aquellas frases de que "se trabaja de 8.00 am. a 4.30pm", y ser reemplazadas por frases similares como "se produce 4000 piezas metálicas de 8.00am a 4.30pm" ó "se atendió a cien clientes diariamente entre "8.30 am a 5.00 pm".
Existen tres clases de organizaciones que emprenden la reingeniería:

1. Las primeras son **las que se encuentran en graves dificultades**. Estas no tienen más remedio. Organizaciones que están en una desventaja bastante "abismal" con respecto a sus competidores, o que no pueden lidiar con sus costos que son muy altos.

2. La segunda categoría es de aquellas **organizaciones que todavía no se encuentran en dificultades,** pero que cuya administración tiene la previsión de detectar que se avecinan problemas. Estas organizaciones tienen la visión de rediseñar antes de caer en la adversidad.

3. El tercer tipo de organizaciones que emprenden la reingeniería lo constituyen **las que están en óptimas condiciones**. No tienen dificultades visibles ni ahora ni en el horizonte, pero su administración tiene aspiraciones y energía. Ven a la reingeniería como una oportunidad de ampliar su ventaja sobre los competidores.

Para desarrollar bien la reingeniería hay que:

- Orientarse al proceso.
- Tener ambición y voluntad.
- Infringir las reglas, crear nuevas, pero prácticas y útiles.
- Ser un creativo de la informática.
- Tener una paciencia inicial, para comprender y entender las diversas tecnologías y como adaptarlas y emplearlas en la organización y en todo nivel.

Reingeniería no es lo mismo que automatización. Automatizar los procesos existentes con la informática, simplemente ofrece maneras más eficientes de hacer lo que no se debe hacer, y en este punto hay que tener cuidado, no basta comprar computadoras y usarlas acelerando procesos existentes, primero se debe optimizar o cambiar los procesos por otros más eficientes, y luego seguir optimizando con el uso de computadoras.

Rediseñar una organización no es lo mismo que reorganizarla, reducir el número de niveles o hacerlas más plana, aunque la reingeniería puede producir una reorganización más plana. El problema que enfrentan las organizaciones no proviene de la estructura de sus procesos.

La reingeniería tampoco es lo mismo que la mejora de la calidad, ni gestión de la calidad total, ni ninguna otra manifestación del movimiento contemporáneo de la calidad. La reingeniería busca avances decisivos, no mejorando los procesos existentes sino descartándolos por completo y cambiándolos por otros sistemas más nuevos. La reingeniería implica, igualmente, un enfoque de gestión diferente del que necesitan los programas de calidad.
La reingeniería es volver a empezar a escribir en una hoja de papel en blanco. Es rechazar las creencias o dogmatismos. Es inventar nuevos enfoques de la estructura del proceso que tienen poca o ninguna semejanza con los de épocas anteriores. La reingeniería es un nuevo comienzo.

1. Investigue si CMMI y los estándares de calidad ISO son excluyentes. Fundamente su respuesta

2. Investigue acerca de la relación que existe entre UML y BMP(Business Process Modeling)

3. Investigue acerca de la relación que existe entre la arquitectura de software y el desarrollo de software basado en componentes

En ésta unidad se ha tratado acerca de temas actuales relacionados con la Ingeniería de software tales como el desarrollo de software basado en componentes, arquitecturas de software, CMMI, BPM y reingeniería

[1] PRESSMAN, Roger S. **Ingeniería del Software. Un enfoque práctico.** Mc Graw Hill, Ed. Interamericana de España S.A.U, 2006

[2] Mary B. CHRISSIS, Mike KONRAD, Sandy SHRUM: **CMMI. Guidelines for Process Integration and Product Improvement.** Addison-Wesley, 2006,

[3] SOMMERVILLE, I., **Ingeniería de Software**, Ed. Pearson Educación, 2002

[4] KRUCHTEN, Philippe. **Architectural Blueprints--The 4+1 View Model of Software Architecture**. IEEE Software, Institute of Electrical and Electronics Engineers, 1995

[5] HAMMER, Michael y James CHAMPY. **Reingeniería.** Editorial Norma.1994

Bibliografía electrónica:

- Fundamentos de Ingeniería del Software
  http://dis.um.es/~jnicolas/09BK_FIS.html

- Desarrollo basado en componentes
  http://webcabcomponents.com/componentization.shtml

- Qué son componentes?
  http://www.componentsource.com/Services/WhatAreComponents.asp?bhcp=1

AUTOEVALUACIÓN FORMATIVA

INGENIERIA DEL SOFTWARE
UNIDAD ACADÉMICA Nº 8

NOMBRE:_____

APELLIDOS:_____FECHA; ____/_____/_____

CIUDAD:_____SEMESTRE:_____

1. Una arquitectura de software consiste en:
   a) Un conjunto de patrones y abstracciones coherentes que proporcionan el marco de referencia necesario para guiar la construcción del software para un sistema de información.
   b) El diseño y la implementación de estructuras de software denominados componentes
   c) Un conjunto de programas que forman parte del sistema integral
   d) La definición de componentes hardware donde se va a desplegar el sistema
   e) N.A.

2. J2EE es una arquitectura de software de tres capas. Esta afirmación es:
   a) Verdadera
   b) Falsa

3. La ingeniería de software basada en componentes busca:
   a) Generar componentes fuertemente acoplados
   b) La reutilización del software existente
   c) Evitar la reutilización de software
   d) Eliminar las arquitecturas de software de tres capas
   e) N.A.

4. Son beneficios del desarrollo de software basado en componentes:
   a) Reutilización, simplificación de pruebas y mayor complejidad
   b) Simplificación de las pruebas, menor calidad y mayor costo
   c) Reutilización, simplificación de pruebas y mayor calidad
   d) Mayor calidad y poca reutilización de codigo.
   e) Todas las anteriores

5. BPM es un modelo de calidad del software que clasifica las empresas en niveles de madurez. Estos niveles sirven para conocer la madurez de los procesos que se realizan para producir software. Esta afirmación es
   a) Verdadera
   b) Falsa

6. Reingeniería significa:
   a) Mejora de la calidad,
   b) Gestión de la calidad total
   c) Inventar nuevos enfoques de la estructura del proceso
   d) Automatizar los procesos empresariales
   e) Todas las anteriores

7. Business Process Management (BPM) se enfoca en la administración de los procesos del negocio. Esta afirmación es
   c) Verdadera
   d) Falsa

ANEXOS

ANEXO 1

ESPECIFICACIÓN DE CASOS DE USOS

REGISTRANDO PACIENTE

1. BREVE DESCRIPCIÓN

El Cliente – Paciente, solicita servicio, como es primera vez, se solicita sus datos.

2. EVENTO INICIAL

El CLIENTE – PACIENTE solicita consulta

3. PRECONDICIÓN

Ninguna

4. FLUJO DE EVENTOS

4.1 FLUJO BÁSICO

PASO	Acción
1	Este caso de uso empieza cuando el cliente Solicita un servicio a recepción.
2	La recepcionista le pregunta, sus datos y verifica si es Cliente de la Clínica.
3	Si es Cliente Nuevo, Registra datos del Cliente –Paciente Si el Cliente ya esta registrado se verifican sus datos

4.2 FLUJOS ALTERNATIVOS

5. EXCEPCIONES

ANEXO 2

CÓMO DIBUJAR UN GRAFO DE FLUJO.

Para dibujar el grafo de flujo de un programa es recomendable seguir los siguientes pasos:

1. Señalar sobre el código cada condición de cada decisión (por ejemplo, «mismo producto» en la figura 3 es una condición de la decisión «haya registros y mismo producto») tanto en sentencias If-then y Case-of como en los bucles While o Until.

2. Agrupar el resto de las sentencias en secuencias situadas entre cada dos condiciones según los esquemas de representación de las estructuras básicas que mostramos a continuación.

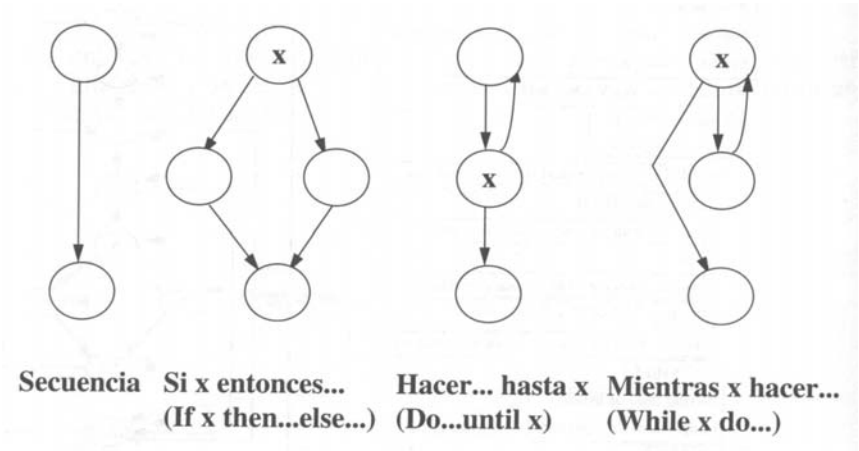


Figura 7. 4*. Grafo de flujo de las estructuras básicas de programa*

Numerar tanto las condiciones como los grupos de sentencias, de manera que se les asigne un identificador único. Es recomendable alterar el orden en el que aparecen las condiciones en una decisión multicondicional, situándolas en orden decreciente de restricción (primero, las más restrictivas). El objetivo de esta alteración es facilitar la derivación de casos de prueba una vez obtenido el grafo. Es conveniente identificar los nodos que representan condiciones asignándoles una letra y señalar cuál es el resultado que provoca la ejecución de cada una de las aristas que surgen de ellos (por ejemplo, si la condición x es verdadera o falsa)