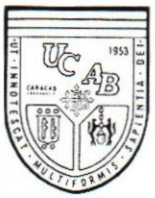


UNIVERSIDAD CATÓLICA ANDRÉS BELLO
FACULTAD DE INGENIERÍA
ESCUELA DE INGENIERÍA INFORMÁTICA

INGENIERÍA DE REQUISITOS

TRABAJO DE ASCENSO
Presentado ante la
UNIVERSIDAD CATÓLICA ANDRÉS BELLO

Realizado por:

Lic. Ing. Evelenir Barreto González

Caracas, Marzo 2004

TABLA DE CONTENIDO

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

INTRODUCCIÓN

Entender las necesidades y atender a los deseos de los clientes fue siempre visto como uno de los mayores desafíos de la Ingeniería del Software.

La Ingeniería de Requisitos es tradicionalmente entendida como una parte borrosa del ciclo de vida del software, en la que se obtiene una especificación formal de unas ideas informales. Desde mediados de los años 70 cobra una especial importancia en el desarrollo de sistemas; en los últimos años han sucedido grandes cambios en el campo de la Ingeniería de Requisitos.

Así, en primer lugar, ya se ha establecido un reconocimiento general de la importancia de la Ingeniería de Requisitos y de los riesgos en que se incurren si ésta se realiza de forma incorrecta o insuficiente. Actividades propias de esta área, como la especificación de requisitos o la gestión de requisitos del usuario, son algunas de las consideradas más críticas en el desarrollo y la producción del software. La preocupación por este tema se constata con la proliferación de libros, revistas, y conferencias que, recientemente, han aparecido y que están dedicadas especialmente a la Ingeniería de Requisitos.

En segundo lugar, y como consecuencia de toma de conciencia de esta problemática situación por parte de los implicados en la industria del software, se ha llegado a introducir explícitamente la Ingeniería de Requisitos en modelos de Mejora del Proceso Software. Y, por la misma razón, se han desarrollado guías e interpretaciones que permiten el estudio de la conformidad de procesos y productos de la Ingeniería de Requisitos a estándares internacionales de Calidad que buscan garantizar la satisfacción del cliente.

La posición de la Ingeniería de Requisitos es proveer al ingeniero del software, métodos, técnicas y herramientas que asistan al proceso de comprensión y captura de los requisitos que el software debe atender. Diferentemente de otras sub-áreas de la Ingeniería del Software, el área de requisitos tiene que ocuparse del conocimiento interdisciplinario, envolviendo muchas veces, aspectos de las ciencias sociales y las ciencias cognitivas.

Este trabajo de investigación trata la implicación entre aspectos de Calidad del producto final con las prácticas relacionadas con la Ingeniería de Requisitos. Para ello el contenido se ha organizado en cuatro principales capítulos:

- Establecimiento de los aspectos conceptuales en un entorno de la Ingeniería de Requisitos, independientemente de la metodología utilizada.

- Estudio del Proceso Unificado (Unified Process) involucrados con la Ingeniería de Requisitos.

- Estudio más amplio del Lenguaje de Modelación Unificado (Unified Modeling Language™, UML), además de aspectos importantes de la historia de UML, su estructura y objetivos, sus componentes, vistas, diagramas, elementos del modelo, mecanismos y extensiones.

- Discusión sobre técnicas, modelos y herramientas aplicables a la Ingeniería de Requisitos.

Capítulo I: INGENIERÍA DE REQUISITOS

I.1. INGENIERÍA DE REQUISITOS

La Ingeniería de Requisitos, es una sub-área de la ingeniería del software que se encarga de estudiar los procesos de definición de los requisitos que tendrá el software a desarrollar. El área surgió en 1993 con la realización del I Simposio Internacional de Requisitos de Ingeniería. El proceso de la definición de requisitos es una interfaz entre los deseos y las necesidades de los clientes para la posterior implementación de estos requisitos en forma de software.

Según Boehm [Bohem 1989], la Ingeniería de Requisitos es un proceso de establecimiento de los servicios que debe proporcionar un sistema, así como de las restricciones sobre las que deberá operar.

La IEEE (Institute of Electrical and Electronics Engineers) [IEEE 1994], define la Ingeniería de Requisitos como un proceso de estudio de las necesidades de los usuarios con el objeto de llegar a una definición del sistema hardware- software.

Según Loucopoulos [Loucopoulos 1990], la Ingeniería de Requisitos es un trabajo sistemático de desarrollo de requisitos, a través de un proceso iterativo y cooperativo de análisis del problema, documentando los resultados en una variedad de formatos y probando la exactitud del conocimiento adquirido.

Según Zave (2000) [Zave 2000], la Ingeniería de Requisitos es una rama de la Ingeniería del Software que trata con el establecimiento de los objetivos, funciones y restricciones de los sistemas de software. Asimismo, se ocupa de la relación entre estos factores con el objeto de establecer especificaciones precisas.

En una definición muy simple la Ingeniería de Requisitos busca identificar lo que espera un usuario/cliente de un sistema de software.

Luego de haber realizado el estudio de los materiales a los que se pudo acceder, en el presente trabajo se adopta una definición para Ingeniería de Requisitos que se apoya en definiciones de los autores Leite [Leite 1987], Loucopoulos [Loucopoulos 1989], Dorfman [Dorfman 1997] y de la IEEE [IEEE 1994] y en los problemas que aún siguen sin resolver. Por tanto se define a la Ingeniería de Requisitos como un proceso centrado en el cliente/usuario y sus necesidades, en donde las etapas de Elicitación, de Análisis, de Especificación y de Validación y Certificación de requisitos iteran hasta la obtención de documentos isomórficos representativos de las necesidades reales de clientes/usuarios, depuradas en base a procesos cooperativos que se llevan a cabo en distintas comunidades

del dominio de información y en donde el isomorfismo de los documentos finales permite salvar la brecha que existe entre el enfoque antropológico de la Ingeniería de Requisitos y las etapas siguientes de la Ingeniería de Software, permitiendo que las mismas continúen hacia la prosecución de un software sin fallos.

Diferentes autores descomponen al proceso de Ingeniería de Requisitos de diversas formas. Es así que, por ejemplo, Rzepka [Rzepka 1989], lo considera conformado por tres actividades:

- elicitar los requisitos de las diversas fuentes individuales;
- asegurar que las necesidades de todos los usuarios son consistentes y factibles; y
- validar que los requisitos que se derivaron son un reflejo exacto de las necesidades del usuario.

Oberg [Oberg 1998] plantea que, desde el momento en que los requisitos son necesidades que deben satisfacer los sistemas a ser construidos, y que la satisfacción de determinados conjuntos de requisitos define el éxito o fracaso de los proyectos, tiene sentido buscar lo que son los requisitos, escribirlos, organizarlos y seguirlos en el momento en que cambian. Dicho autor considera que la Administración de Requisitos -haciendo referencia a la Ingeniería de Requisitos es:

- un enfoque sistemático para elicitar, organizar y documentar los requisitos del sistema;
- un proceso que establece y mantiene un acuerdo entre el cliente, el usuario y el equipo del proyecto sobre los requisitos cambiantes del sistema.

La Rational Software Company, dice Oberg, asume que el término Administración hace una descripción más apropiada de todas las actividades involucradas, y enfatiza la importancia del seguimiento de los cambios para mantener los acuerdos entre la comunidad de usuarios y el equipo del proyecto.

Sin embargo, se considera que este término, en caso de incorporarse al proceso de definición de requisitos, hace referencia a una actividad de la Ingeniería de Requisitos que sirve para controlar y seguir los cambios de los requisitos. En el presente trabajo, se sostiene que el término "Ingeniería" es más apropiado ya que no solo se plantean etapas de definición, sino técnicas, métodos y herramientas involucradas en cada una, lo que lleva a definirla como una disciplina independiente.

Por su parte, Dorfman y Thayer [Dorfman 1997], plantean una definición similar considerando que la Ingeniería de Requisitos incluye tareas de *elicitación, análisis, especificación, validación y administración de requisitos de software*, siendo la "administración de requisitos de software" la planificación y control de todas esas actividades relacionadas.

Como puede verse, para sustentarse, la Ingeniería de Requisitos, sugiere la existencia de un eje troncal de etapas, dejando abierta la posibilidad de que cada uno de los estudiosos del

tema las refinen cuanto sea necesario. Por tanto, si bien existen diferentes enfoques, éstos tienen un común denominador, que puede resumirse en las siguientes etapas fundamentales: Elicitación, Análisis y Especificación, que son las que se adoptan en el presente trabajo:

Elicitación: Es la etapa de mayor interacción con el usuario. Es el momento en el que se recurre, por ejemplo, a la observación, lectura de documentos, entrevistas y relevamientos, entre otras técnicas; la instancia en que equipos multidisciplinarios trabajan conjuntamente con el cliente/usuario, para obtener los requisitos reales de la mejor manera.

Análisis: La etapa de análisis de requisitos permite al analista representar el dominio de la información (también conocido como Universo de Información - UdI) de la aplicación a desarrollar, a través del uso de un lenguaje más técnico, procurando reducir ambigüedades. Brinda al analista, la representación de la información y las funciones que facilitarán la definición del futuro diseño.

Especificación: No cabe ninguna duda de la importancia de esta etapa y de que la forma de especificar tiene mucho que ver con la calidad de la solución. Los analistas que se han esforzado en trabajar con especificaciones incompletas, inconsistentes o mal establecidas han experimentado la frustración y confusión que invariablemente se produce. Las consecuencias se padecen en la calidad, oportunidad e integridad del software resultante.

Como se puede apreciar existen grandes diferencias en la terminología usada por los autores de la bibliografía consultada, siendo éste un punto importante para destacar, a los efectos del entendimiento de la Ingeniería de Requisitos. Es así que hay autores que ven al Análisis de Requisitos como el proceso completo de definición de requisitos y no como una etapa metodológica de la Ingeniería de Requisitos. De igual manera, la Especificación de Requisitos tiene diferentes acepciones: algunos autores se refieren a ella como a una etapa en la que se describen los requisitos y otros como a la actividad completa, desde la Elicitación hasta la Especificación propiamente dicha.

De igual manera, se hace referencia a la Ingeniería de Requisitos con otros términos tales como "Etapa de Requisitos" (IEEE - Institute of Electric and Electronic Engineer) y "Administración de Requisitos" (Rational Software Corporation). Pero se debe destacar que tanto los autores Dorfman y Thayer, como Christel, hacen una correcta referencia a Ingeniería de Requisitos, diferenciando claramente sus actividades intermedias.

En cuanto a la adopción de las etapas de Elicitación, Análisis y Especificación como eje de la investigación, la decisión fue tomada por considerarse que facilitan el entendimiento de las tareas que en ellas se realizan.

I.2 Importancia de la Ingeniería de Requisitos

La Ingeniería de Requisitos es un campo muy activo dentro de la Informática, y en particular dentro de la Ingeniería del Software, y se dirige a unas actividades esenciales en el trabajo diario de las organizaciones de desarrollo de software. Se ha demostrado mediante varios estudios experimentales que la Ingeniería de Requisitos es crítica respecto del éxito o fracaso de numerosos proyectos informáticos y su mala gestión tiene una gran incidencia probada en relación con el desbordamiento de costos o el incumplimiento de plazos de finalización

Los grandes sistemas se construyen para mejorar el funcionamiento de un sistema complejo, que a veces tiene un nivel de automatización mínimo, lo que dificulta la previsión de las mejoras. Por otra parte, tienen un conjunto de usuarios diversos, en general, con diferentes requisitos y prioridades que pueden entrar en conflicto. El sistema final debe ser necesariamente una solución de compromiso. Trabajando en el diseño de grandes sistemas, existen problemas que por su complejidad sólo se entienden durante la fase de desarrollo, esto tiene la consecuencia que los requisitos no se puedan capturar desde su inicio.

Boehm [Bohem 1989], afirma que sólo entre un 9% y un 12% de la duración de un proyecto se invierte en la captura de requisitos. Estudios más reciente (1996), confirman los siguientes datos:

- Entre el 5% y el 15% de los costos de un proyecto se dedican a la captura de requisitos.
- El 15% de la duración del proyecto se emplea en la captura de requisitos.

Además afirma que los errores en esta fase son los más numerosos. El 10% de los errores se producen en la captura de requisitos y además la reparación de un error en la fase de codificación cuesta entre 5 y 10 veces más. Es por ello que el costo crece exponencialmente con el retraso al subsanar el requisito.

Estos porcentajes parecen sorprendentes si se piensa que los errores más numerosos, más costosos de reparar y que más tiempo consumen se deben a esta fase.

I.3. ¿Quién utiliza la Ingeniería de Requisitos?

Los profesionales de la informática y sus clientes cuando están implicados en un proceso de definición de requisitos.

I.4. Conceptos Básicos en la Ingeniería de Requisitos

Análisis del sistema: estudio detallado de los requisitos, previsto para probar su funcionabilidad como entrada a los sistemas diseñados.

Caso de uso: refiere al término acontecimiento conducido para denotar la parte de una actividad definida por el usuario en el contexto del producto.

Cliente: persona u organización para la cual se está construyendo el producto, generalmente responsable de pagar el desarrollo del producto.

Comprador: persona u organización que comprará el producto (note que la misma persona/organización puede jugar ambos papeles entre cliente, comprador y a veces el usuario).

Conductores del proyecto: son las fuerzas relacionadas con el negocio.

Contexto del producto: refiere a los límites del producto que se propone construir y la gente, las organizaciones, otros productos y las piezas de tecnología que tienen una interfaz directa con el producto.

Contexto del trabajo: refiere el tema, la gente y las organizaciones que pueden tener un impacto en los requisitos para el producto. El contexto del estudio identifica la intersección de todos los dominios del interés.

Criterio de adecuación: medida objetiva para definir el significado de un requisito, y probar eventualmente si una solución dada satisface el requisito original.

Desarrolladores: personas que especifican y construyen el producto.

Diseño o diseño de los sistemas: hacer una solución a mano que se ajuste a los requisitos.

Dominio de interés: un área del tema que tiene cierta importancia dentro del contexto de estudio.

Ediciones del proyecto: definen las condiciones bajo las cuales el proyecto será hecho. Se incluyen éstos en la especificación de requisitos, para presentar un cuadro coherente de todos los factores que contribuyen el éxito o fracaso del proyecto.

Evento: se utiliza el término evento del negocio para referirse a un suceso relacionado dentro del negocio de un sistema adyacente al trabajo que se esta estudiando. Este suceso origina el trabajo para producir una causa-efecto.

Producto: es lo que se procura entregar. Puede ser un software, la instalación de un paquete, un conjunto de procedimientos, un hardware, una maquinaria, una nueva organización, o cualquier cosa.

Requisito: La IEEE define requisito como: "condición o aptitud necesaria para resolver un problema o alcanzar un objetivo". "Condición o facilidad que debe proporcionar un sistema o algunos de sus subsistemas para satisfacer un contrato, norma, especificación o cualquier

otra condición impuesta formalmente a través de un documento". "Una representación documentada de una condición o facilidad".

Para establecer algún tipo de requisito básicamente se realiza la pregunta: ¿Qué hace el sistema? y no ¿Cómo lo hace?. Por ello se crean dos tipos de requisitos: funcionales y no funcionales.

Requisitos Funcionales: los requisitos funcionales son la materia fundamental del sistema y son cuantificados concretamente por los valores de los datos, decisiones lógicas y algoritmos. Básicamente describen los servicios o funciones del sistema.

Volere [Volere 2002], establece que los requisitos funcionales son el tema de fundamental del sistema y son medidos por medios concretos como valores de los datos, lógica de la toma de decisión y algoritmos.

Requisitos No Funcionales: los requisitos no funcionales refieren a las propiedades conductuales especificadas a las funciones que debe tener el sistema, tal como el funcionamiento, la usabilidad, etc. Estos requisitos pueden asignarse a una medida específica describiendo las restricciones del sistema o del proceso de desarrollo: Prestaciones, interfaz, atributos de calidad, etc. En general, los requisitos no funcionales son más difíciles de cuantificar y más adelante se mostrará cómo cuantificar estos tipos de requisitos.

Volere [27], establece que los requisitos no funcionales son las características del comportamiento que las funciones especificadas deben tener, por ejemplo el funcionamiento, utilidad, etc. Los requisitos no funcionales se le pueden asignar una medida específica.

Restricciones del proyecto: identifican cómo el producto eventualmente debe encajar en el mundo. Por ejemplo, el producto puede tener una interfaz con algún hardware existente, software o práctica del negocio, o puede que tenga que ajustarse a un presupuesto definido o estar listo para una fecha determinada.

Restricción global: restricciones que se aplican en conjunto al sistema.

Sistema: es el negocio al cual se estudian los requisitos.

Stakeholders: son todas las personas interesadas en el proyecto que pueden afectar el resultado/éxito del proyecto y/o pueden ser afectado por su resultado/éxito. Cada uno de los participantes.

Usuario final: alguien que tiene cierta clase de interfaz directa con el producto.

I.5. Etapas de la Ingeniería de Requisitos

La Ingeniería de Requisitos consta de las siguientes etapas:

- Elicitación de Requisitos
- Análisis de Requisitos
- Especificación de Requisitos
- Validación y Certificación de los Requisitos

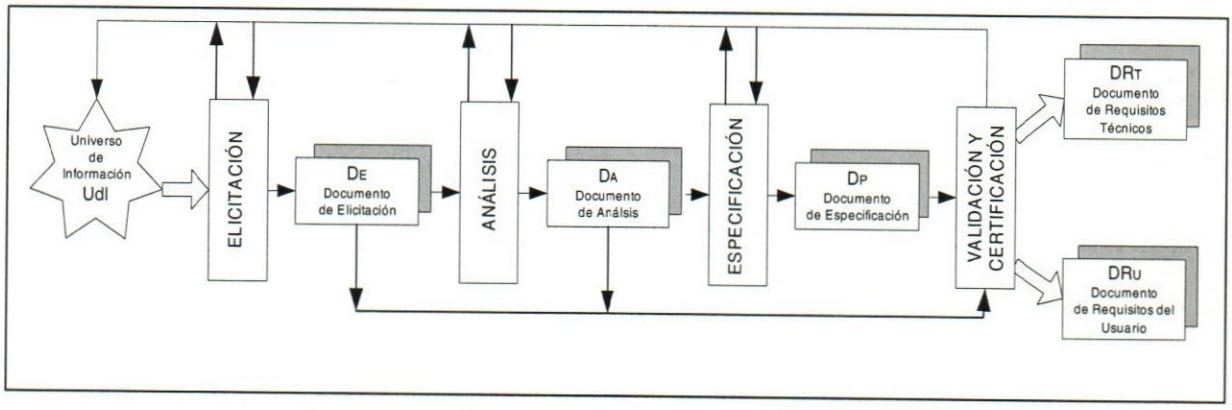


Figura I-1: Etapas de la Ingeniería de Requisitos

I.5.1 Elicitación de Requisitos

En esta etapa, en donde se adquiere el conocimiento del trabajo del cliente/usuario, se busca comprender sus necesidades y se detallan las restricciones medioambientales. Como resultado de las acciones realizadas se tiene el conjunto de los requisitos de todas las partes involucradas.

En cuanto al proceso de elicitación de requisitos, se considera apropiado los siguientes pasos:

Formar el equipo multidisciplinario

Considerando que la formación de la gente de sistemas, tratándose de problemas con alta incidencia del factor humano, no tiene la especialización necesaria como para diagnosticar el método de elicitación más apropiado para cada caso en particular, se aconseja que la recolección de requisitos sea efectuada con el asesoramiento de profesionales especializados. Este asesoramiento puede extenderse incluso a un liderazgo activo de las sesiones de elicitación por parte de especialistas en ciencias de la comunicación o en ciencias del conocimiento.

Buscar hechos

El primer paso en la elicitación de requisitos está involucrado con el problema a ser encarado, y quién necesita ser involucrado en esta toma de decisión, tanto como quién se

verá afectado por la formulación de los problemas y la eventual solución. Los resultados de esta actividad son: una declaración del contexto del problema, de los objetivos globales, límites e interfaces para el sistema original.

Este examen debe ser efectuado de manera tal que permita establecer, entre otros, cuál es el rol que desempeñará el sistema a desarrollar, sus objetivos y límites, las restricciones de arquitectura y la existencia o no de sistemas similares dentro de la organización.

Recolectar y clasificar requisitos

En esta etapa se obtienen: objetivos, necesidades y requisitos de clientes y usuarios. Estas necesidades y requisitos son verificadas comparándolas con los objetivos globales del sistema original expresados durante el hallazgo de hechos. Es importante recolectar tanta información como sea posible. Dependiendo de la manera en que el sistema se está desarrollando y los grupos que afectará, la etapa de recolección de requisitos es una combinación de los enfoques composición y descomposición. Es importante en este momento, destacar los términos que son propios del lenguaje del UdI.

Una vez recolectados los requisitos, se debe proceder a clasificar los mismos en funcionales y no funcionales.

Evaluar y racionalizar

Debe realizarse una valoración del riesgo, para encaminar las inquietudes técnicas, de costos y de tiempo. Debe examinarse la coherencia en la información reunida en subetapas previas, para determinar si los requisitos verdaderos están escondidos o expresados explícitamente. Se realizan abstracciones para responder preguntas del tipo ¿Por qué usted necesita X?, y si esta pregunta tiene una respuesta concreta, entonces es un requisito, si no es un falso requisito. Mediante el estudio comparativo de la información de requisitos se ponen en evidencia las inconsistencias que pueden surgir entre los requisitos extraídos. Cabe destacar que tanto en la presente subetapa como en la anterior, se dan instancias de evaluación de factibilidad, negociables entre el cliente/usuario y el analista.

Dar prioridad

En esta etapa, contando ya con requisitos consistentes, se da un orden de prioridades, de manera tal que las necesidades de alta prioridad pueden ser encaradas primero, lo que permite definirlas y reexaminar los posibles cambios de los requisitos, antes que los requisitos de baja prioridad (que también pueden cambiar) sean implementados.

Durante el desarrollo del sistema, esto permite una disminución de los costos y ahorro de tiempo en procesamiento de los inevitables cambios de los requisitos.

Los requisitos deben tener prioridades basándose en las necesidades del usuario. El costo y la dependencia.

Integrar y validar

Esta tarea se lleva a cabo de manera tal que sea posible obtener un conjunto de requisitos, expresados en el lenguaje del usuario, de los cuales se pueda validar la consistencia con respecto a las metas organizacionales obtenidas en la primera etapa. Las tareas de integración deben ser ejecutadas principalmente por el analista de sistemas, y los resultados del proceso de elicitación comunicarlos a las otras comunidades involucradas. Esta validación de los requisitos realizada por todas las partes afectadas, asegura que se alcanza lo deseado.

Documentar la etapa

Elaborar la lista final de los términos del lenguaje del UdI, y la de sentencias de los requisitos obtenidos (D_E).

Como es de esperar, a los efectos de obtener buenos requisitos, todos estos pasos deben iterar ante la menor inconsistencia detectada, aconsejándose que la iteración se realice recurriendo al cliente/usuario, tantas veces como sea necesario, para garantizar una correcta depuración del producto final de la etapa de elicitación.

Si se hiciera una representación gráfica para la etapa de elicitación de requisitos, la misma quedaría bosquejada de la manera que se puede apreciar en la figura I-2.

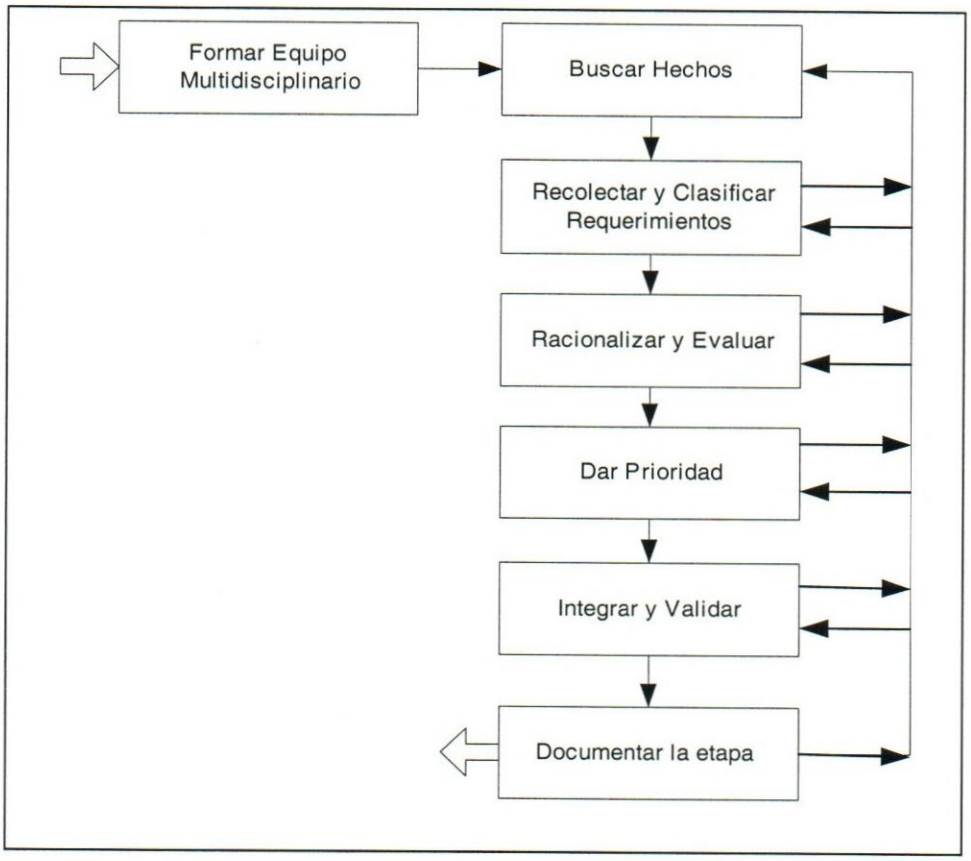


Figura I-2: Etapas de Elicitación de Requisitos

I.5.2. Análisis de Requisitos

En esta etapa se estudian los requisitos extraídos en la etapa previa a los efectos de poder detectar, entre otros, la presencia de áreas no especificadas, requisitos contradictorios y peticiones que aparecen como vagas e irrelevantes. El resultado de haber llevado a cabo las tareas que involucran estos términos puede, en más de una oportunidad, hacer que se deba regresar a la primera etapa, a los efectos de eliminar todas las inconsistencias y falencias que se han detectado. En esta etapa ya se realizan aproximaciones a un lenguaje técnico.

Los pasos a realizar durante esta etapa tienen como objetivo la obtención de buenos requisitos [Hooks 2000], para asegurar de esta manera que lo que se derivará a las etapas posteriores será de calidad tal que permita que disminuyan las fallas de los sistemas.

Las subetapas a contemplar durante esta etapa son:

Reducir ambigüedades en los requisitos
Los requisitos obtenidos como resultado final de la etapa de elicitación, deben ser tratados a los efectos de llevarlos a una notación que permita reducir la ambigüedad del lenguaje del usuario. Por consiguiente, en esta subetapa se realizan las tareas que permiten eliminar los términos que tienen más de una acepción, unificando el léxico empleado en el UdI.

Traducir a lenguaje técnico los requisitos
Los requisitos, ya con menos ambigüedades, deben ser tratados a los efectos de llevarlos a un lenguaje que se vaya aproximando al lenguaje técnico. Mediante esta traducción se busca aproximar los términos del usuario a los términos del sistema de software.

Plantear un modelo lógico
Partiendo del lenguaje obtenido en la etapa anterior, transformarlo en una estructura preliminar, es decir, en un primer modelo lógico. De esta manera, en la presente subetapa se debe construir un modelo del problema ya sea en términos de diagramas de flujo o cualquier otro tipo de representación que se considere conveniente para el modelado y que permita, además, establecer un vínculo con la Etapa de Especificación.

Documentar la etapa
Elaborar todo tipo de documento que se considere adecuado como soporte para la etapa siguiente. Este documento, dado el caso, puede resumirse a la colección de los modelos lógicos a que se ha arribado (D_A).

I.5.3. Especificación de Requisitos

Partiendo de lo elaborado en la etapa anterior tales como funciones, datos, requisitos no funcionales, objetivos, restricciones de diseño/implementación o costos, e independientemente de la forma en que se realice, esta etapa es un proceso de descripción del requisito. Si se presentan dificultades para especificar un requisito se debe volver a la etapa anterior que se crea conveniente.

Recién superadas las etapas anteriores debe comenzar a pensarse en la forma de describir los requisitos. Para ello, se deben seguir las subetapas planteadas a continuación:

Determinar el tipo de requisito
Considerando que existen diferentes tipos de requisitos, determinar unívocamente a cual de ellos pertenece el que se está tratando. Esto no significa que deba adoptarse la clasificación por la cual se han decidido los autoras de este estudio, sino que aquí también queda de manifiesto la flexibilidad de la metodología, ya que cada analista de requisitos puede utilizar la clasificación que considera como la más adecuada.

Elegir la herramienta de especificación acorde al tipo de requisito
Una vez definido el tipo de requisito, seleccionar la herramienta de representación acorde a dicho tipo y al tipo de especificación que se desea realizar. La única restricción al respecto es que la herramienta a seleccionar debe ser de índole formal o, a lo sumo, semiformal, ya que ellas son las únicas que permiten representar a los requisitos sin ambigüedades.

Especificar de acuerdo a la herramienta seleccionada
Representar el requisito sobre la base de la elección realizada en la etapa anterior. En caso de existir dificultades para su empleo, volver a la subetapa anterior para realizar una nueva selección o, incluso, a la primera ya que la dificultad de representación puede obedecer al intento de usar una herramienta para un requisito cuyo tipo ha sido mal definido, por lo cual se selecciona una inaplicable al caso.

Documentar la etapa
Confeccionar el documento representativo de la etapa tomando como base a los modelos formales o semiformales que se han elaborado al realizar la especificación de los requisitos. Incorporar al mismo toda extensión que se considere de utilidad para la etapa de Validación y Certificación de Requisitos (D_P).

I.5.4. Validación y Certificación de los Requisitos.

Esta etapa final se nutre de las anteriores y realiza la integración y validación final de lo obtenido en cada una de las etapas anteriores dando, como resultado final, el Documento de Requisitos. Este documento no es uno solo sino que, como mínimo, existen dos que son isomórficos entre sí: uno destinado al cliente/usuario a los efectos de la certificación de los Requisitos y el otro técnico, orientado a nutrir las restantes etapas de la Ingeniería de Software. Y, al igual que en el caso anterior, su resultado puede ser la necesidad de retornar a la especificación e incluso a la elicitación; iterando entre etapas y sin perder contacto con el cliente/usuario.

Todo el esfuerzo realizado durante las etapas anteriores puede darse por perdido si no es manifestado en forma correcta, ya que cualquier mala interpretación puede echar por tierra hasta el proceso de Ingeniería de Requisitos más consistente, desvirtuando la naturaleza de lo que fue, hasta el momento de su declaración, un buen requisito. Tratando de minimizar aún más la posibilidad de error, se proponen las siguientes subetapas para su elaboración:

Seleccionar las fuentes de información a partir de las cuales validar el documento de especificación.

En esta etapa se procede a validar el documento de especificación DP a partir de los documentos obtenidos de las etapas de elicitación (D_E) y análisis (D_A), seleccionando como fuente de información aquellos materiales que más aportan, por un lado a la claridad de su descripción y, por el otro, en cuanto a permitir la validación final entre los resultados de todas las etapas anteriores. El documento de especificación (D_P) validado se llamará, en adelante, documento de requisitos técnico (DR_T).

Elegir o diseñar el modelo de documento acorde al grado de detalle requerido y al lector final

Si bien muchos autores han propuesto modelos de documentación excelentes, es necesario decidirse por alguno de ellos. Dado el caso de que ninguno de los conocidos satisfaga las necesidades de documentación del analista de requisitos, se deberá proceder a diseñar aquel que mejor se ajuste a sus necesidades.

Elegir la herramienta de documentación que mejor se aplica al modelo seleccionado

Como no todos los modelos pueden ser plasmados con una misma herramienta, se debe seleccionar la que mejor se adecue al problema entre todas las alternativas posibles. Es así que en esta etapa se deberán tener en cuenta, no sólo a los procesadores de texto sino también a herramientas de recursos gráficos que permitan la incorporación de diagramas y figuras, si se considera que lo antedicho ayuda a la interpretación que hará el usuario del Documento de Requisitos.

Documentar respetando los estándares vigentes a la fecha de realización del documento de requisitos

Elaborar el documento de requisitos orientado al usuario (DR_U) a partir del documento de requisitos técnico (DR_T), realizando una traducción a un lenguaje entendible por aquél. Estos documentos deben ser elaborados respetando los estándares que existen a la fecha de su confección. Para ello, el personal de documentación debe estar al tanto de las normas IRAM e ISO y de las dictadas por instituciones como la IEEE. Como el DR_U tiene fines de certificación y contractuales, considerar como normas de redacción las disposiciones legales al momento de la confección.

Validar

Verificar la correspondencia entre los documentos obtenidos de la etapa anterior, controlando que solo difieran en lo sintáctico y no en lo semántico, es decir que su contenido difiera solamente en el lenguaje utilizado para su definición, alcanzando de esta manera el isomorfismo entre DR_T y DR_U.

Certificar

Proceder a la aprobación del DRU por medio del conforme del cliente, y de esta manera dar por aprobado el Documento de Requisitos Técnico DR_T, el que será utilizado por las restantes etapas de la Ingeniería de Software.

Capítulo II: PROCESO UNIFICADO

II.1. PROCESO UNIFICADO

En primer lugar el Proceso Unificado es un proceso de desarrollo de software. Un *proceso de desarrollo de software* es el conjunto de actividades necesarias para transformar los requisitos de un usuario en un sistemas software (véase figura II-1). Sin embargo, el Proceso Unificado es más que un simple proceso; es un marco de trabajo genérico que puede especializarse para una gran variedad de sistemas de software, para diferentes áreas de aplicación, diferentes tipos de organizaciones, diferentes niveles de aptitud y diferentes tamaños de proyecto.

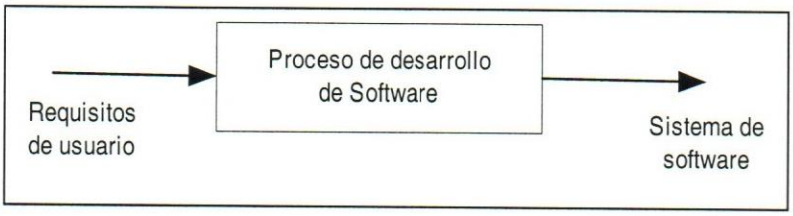


Figura II-1: Proceso de desarrollo de Software

El Proceso Unificado está basado en componentes, lo cual quiere decir que el sistema de software en construcción está formado por componentes software (una parte física y reemplazable de un sistema que se ajusta a, y proporciona la realización de, un conjunto de interfaces), interconectados a través de interfaces (una colección de operaciones que son utilizadas para especificar un servicio de una clase o de un componente) bien definidas.

El objetivo del Proceso Unificado es guiar a los desarrolladores en la implementación y distribución eficiente de sistemas que se ajusten a las necesidades de los clientes. La eficiencia se mide en términos de coste, calidad y tiempo de desarrollo.
El Proceso Unificado utiliza el Lenguaje Unificado de Modelado (Unified Modeling Language™, UML) para preparar todos esquemas de un sistema de software. De hecho, UML es una parte esencial del Proceso Unificado; sus desarrollos fueron paralelos.

Los verdaderos aspectos definitorios del Proceso Unificado se resumen en tres frases clave: dirigido por casos de uso, centrado en la arquitectura e iterativo e incremental. Esto es lo que hace único al Proceso Unificado.

El estar dirigido por los casos de uso significa que cada fase en el camino al producto final está relacionada con lo que los usuarios hacen realmente. Lleva a los desarrolladores a garantizar que el sistema se ajusta a las necesidades reales del usuario. El estar centrado en la arquitectura significa que le trabajo de desarrollo se centra en obtener el patrón de la arquitectura que dirigirá la construcción del s sistema en las primeras fases, garantizando un progreso continuo no sólo para la versión en curso del producto, sino para la visa entera del mismo y finalmente el estar centrado los pasos iterativos e incremental proporciona la estrategia para desarrollar un producto de software en pequeños pasos.

II.2. PROCESO UNIFICADO DIRIGIDO POR CASOS DE USO

Un caso de uso es una descripción de un conjunto de secuencia de acciones, incluyendo variaciones, que un sistema lleva a cabo y que conduce a un resultado observable de interés para un actor determinado. Los casos de uso representan los requisitos funcionales. Todos los casos de uso juntos constituyen el **modelo de casos de uso**; este modelo se describe fundamentalmente mediante lenguaje natural.

La figura II-2 muestra la serie de flujos de trabajo y modelos del Proceso Unificado. Los desarrolladores comienzan capturando los requisitos del cliente en la forma de casos de uso en el modelo de casos de uso. Después analizan y diseñan el sistema para cumplir los casos de uso, creando en primer lugar un modelo de análisis, después uno de diseño y después otro de implementación, el cual incluye todo el código, es decir, los componentes. Por último los desarrolladores preparan un modelo de prueba que les permite verificar que el sistema proporciona funcionalidad descrita en los casos de uso. Todos los modelos se relacionan con los otros mediante dependencias de traza.

Los casos de uso han sido adoptado casi universalmente para la captura de requisitos de sistemas de software en general, y de sistemas basados en componentes particular, pero los caos de uso son mucho más que una herramienta para capturar requisitos

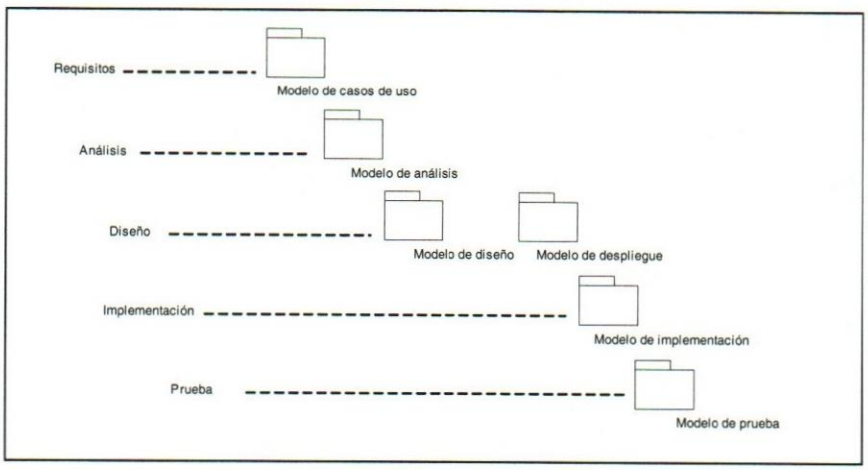


Figura II-2: Flujos de trabajo y modelos del Proceso Unificado

II.2.1. Requisitos

La captura de requisitos tiene dos objetivos: encontrar los verdaderos requisitos y representarlos de un modo adecuado para los usuarios, clientes y desarrolladores. Se entiende por "verdaderos requisitos" aquellos que cuando se implementen añadirán el valor esperado para los usuarios. Con "representarlos de un modo adecuado para los usuarios, clientes y desarrolladores" se quiere decir inconcreto que la descripción obtenida de los requisitos debe ser comprensible por usuarios y clientes. Este es uno de los retos principales del flujo de trabajo de los requisitos.

Normalmente, un sistema tiene muchos tipos de usuarios. Cada tipo de usuario se representa por un actor. Los actores utilizan el sistema interactuando con los casos de uso. Como se explico anteriormente, un caso de uso es una secuencia de acciones que el sistema lleva a cabo para ofrecer algún resultado de valor para un actor. El modelo casos de uso está compuesto por todos los actores y todos los casos de uso de un sistema.

Durante el análisis y el diseño, el modelo de casos de uso se transforma en un modelo de diseño a través de un modelo de análisis. En breve, tanto un modelo de análisis como un modelo de diseño son estructuras compuestas por **clasificadores** (son mecanismos que describen características estructurales y de comportamiento; incluyen interfaces, clases, tipos de datos, componentes y nodos) y por un conjunto de realizaciones de casos de uso que describen cómo esa estructura realiza los casos de uso. Los clasificadores son elementos parecidos a clase. Por ejemplo, tienen atributos y operaciones, se les puede describir con diagramas de estados, algunos de ellos pueden instanciarse, pueden ser participantes en colaboración, etc.

II.2.2. Análisis

El modelo de análisis es una especificación detallada de los requisitos y funciona como primera aproximación del modelo de diseño, aunque es un modelo con entidad propia. Los desarrolladores lo utilizan para comprender de manera más precisa los casos de uso descritos en el flujo de trabajo de los requisitos, refinándolos en forma de colaboraciones entre clasificadores conceptuales (diferentes de los clasificadores de diseño que serán objeto de implementación). El modelo de análisis también se utiliza para crear un sistema robusto y flexible (incluyendo una arquitectura) que emplea la reutilización de componentes de manera considerable. El modelo del análisis es diferente del de diseño en que es un modelo conceptual, en lugar de ser un esquema de la implementación. El modelo del análisis puede ser transitorio y sobrevivir sólo al primer par de iteraciones. Sin embargo, en algunos casos, especialmente para sistemas grandes y complejos, el modelo de análisis debe mantenerse durante toda la vida del sistema. En estos casos, existe una relación directa (mediante dependencias de traza) entre una realización de caso de uso en el modelo de análisis y la correspondiente realización de caso de uso en el modelo de diseño. Cada elemento del modelo de análisis es trazable a partir de elementos del modelo de diseño que lo realizan.

II.2.3. Diseño

El modelo de diseño posee las siguientes características:

- El modelo de diseño es jerárquico, pero también contiene relaciones que atraviesan la jerarquía. Las relaciones son las habituales en UML: asociaciones, generalizaciones y dependencias (Léase capitulo III).

- Las realizaciones de los casos de uso son estereotipos de colaboraciones. Una colaboración representa cómo los clasificadores participan y desempeñan papeles en hacer algo útil, como la realización de un caso de uso.

- El modelo de diseño es también un esquema de la implementación. Existe una correspondencia directa entre subsistemas del modelo de diseño y componentes del modelo de la implementación.

Los desarrolladores crean un modelo de análisis que utiliza el modelo casos de uso como entrada. Cada caso de uso en el modelo de casos de uso se traducirá en una realización de caso de uso en el modelo de análisis. La dualidad caso de uso/realización de caso de uso es la base de una trazabilidad directa entre los requisitos y el análisis. Tomando los casos de uso uno a uno, los desarrolladores pueden identificar las clases que participan en la realización de los casos de uso.

Luego del modelo de análisis y diseño, los desarrolladores diseñan las clases y las realizaciones de casos de uso para obtener mejor partido de los productos y tecnologías que se utilizarán para implementar el sistema. Las clases de diseño se agrupan en subsistemas, y pueden definirse interfaces entre ellos. Los desarrolladores también preparan el modelo despliegue, donde definen la organización física del sistema en términos de nodos de cómputo, y verifican que los casos de uso pueden implementarse como componentes que se ejecutan en esos nodos.

II.2.4. Implementación

A continuación, los desarrolladores implementan las clases diseñadas mediante un conjunto de ficheros (código fuente) en el modelo de implementación, a partir de los cuales se pueden producir (compilar y enlazar) ejecutables, como DLL's, JavaBeans y componentes ActiveX. Los casos de uso ayudan a los desarrolladores a determinar el orden de implementación e integración de los componentes.

II.2.5. Prueba

Por último, durante el flujo de trabajo de prueba los ingenieros de prueba verifican que el sistema implementa de verdad la funcionalidad descrita en los casos de uso y que satisface los requisitos del sistema. El modelo de prueba se compone de casos de prueba. Un caso de prueba define una colección de entradas, condiciones de ejecución y resultados. Muchos de

los casos de prueba se pueden obtener directamente de los casos de uso y por tanto tenemos una dependencia de traza entre el caso de prueba y el caso de uso correspondiente. Esto significa que los ingenieros de prueba verificarán que el sistema puede hacer lo que los usuarios quieren que haga, es decir, que ejecuta los casos de uso.

Lo novedoso y diferente del Proceso Unificado es que la prueba puede planificarse al principio del ciclo de desarrollo. Tan pronto como se hayan capturado los casos de uso, es posible especificar los casos de prueba (pruebas de caja negra) y determinar el orden en el cual realizarlos, integrarlos y probarlos. Más adelante, según se vayan realizando los casos de uso en el diseño, pueden detallarse las pruebas de los casos de uso (pruebas de caja blanca). Cada forma de ejecutar un caso de uso, es decir, cada camino a través de una realización de un caso de uso, es un caso de prueba candidato.

Hasta el momento, se ha presentado el Proceso Unificado como una secuencia de pasos, muy parecida al antiguo método en cascada. En la sección II.4 se especifica cómo estos pasos pueden desarrollarse de una forma mucho más interesante mediante una aproximación iterativa e incremental. Realmente, lo que se ha descrito hasta aquí es una sola iteración. Un proyecto de desarrollo completo será una serie de iteraciones, en la cual cada una de ellas (con la posible excepción de la primera) consiste en una pasada a través de los flujos de trabajo de requisitos, análisis diseño, implementación y prueba.

II.2.6. ¿Por qué casos de uso?

Existen diversos motivos por los cuales los casos de uso son buenos, se han hecho populares y se han adoptado universalmente. Las dos razones fundamentales son:

- Proporcionan un medio sistemático e intuitivo de capturar requisitos funcionales centrándose en el valor añadido para el usuario.

- Dirigen todo el proceso de desarrollo debido a que la mayoría de las actividades como el análisis, diseño y prueba se llevan a cabo partiendo de los casos de uso. El diseño y la prueba pueden también planificarse y coordinarse en términos de casos de uso. Esta característica es aún más evidente cuando la arquitectura se ha estabilizado en el proyecto, después del primer conjunto de iteraciones.

II.3. Proceso Unificado centrado en la arquitectura

Los casos de uso no son suficientes para producir un sistema de trabajo; se necesitan más cosas. Esas "cosas" son la arquitectura. Se puede pensar que la arquitectura de un sistema es la visión común en la que desarrolladores y otros usuarios deben estar de acuerdo, o como poco, deben aceptar. La arquitectura da una clara perspectiva del sistema completo, necesaria para controlar el desarrollo.

Se requiere de una arquitectura que describa los elementos del modelo que son más importantes. ¿Cómo se puede determinar qué elementos son importantes?. Su importancia reside en el hecho que guía en el trabajo con el sistema, tanto en este ciclo como a través del ciclo de vida completo. Estos elementos significativos, arquitectónicamente hablando, incluyen algunos subsistemas, dependencias, interfaces, colaboraciones, nodos y clases activas; describen los cimientos del sistema, que son necesarios como base para comprenderlo, desarrollarlo y producirlo económicamente.
Un sistema de software es una única entidad, pero al arquitecto del software y a los desarrolladores les resulta útil presentar el sistema desde diferentes perspectivas para comprender mejor el diseño. Estas perspectivas son vistas (proyecciones de un modelo, que es visto desde una perspectiva dada o un lugar estratégico, y que omite las entidades que no son relevantes para esta perspectiva) del modelo del sistema. Todas las vistas juntas representan la arquitectura.

II.3.1. ¿Qué es una arquitectura?

Una arquitectura es lo que especifica el arquitecto en la descripción de la arquitectura. La descripción de la arquitectura permite al arquitecto controlar el desarrollo del sistema desde la perspectiva técnica. La arquitectura de software se centra tanto en los elementos estructurales significativos del sistema, como subsistemas, clases, componentes y nodos, como en las colaboraciones que tienen lugar entre estos elementos a través de las interfaces.

Los casos de uso dirigen la arquitectura para hacer que el sistema proporcione la funcionalidad y uso deseados, alcanzando a la vez objetivos de rendimiento razonables. Una arquitectura debe ser completa, pero también debe ser suficientemente flexible como para incorporar nuevas funciones, y debe soportar la reutilización de software existente.

La arquitectura software abarca decisiones importantes sobre:

- La organización del sistema de software
- Los elementos estructurales que compondrán el sistema y sus interfaces, junto con sus comportamientos, tal y como se especifican en las colaboraciones entre estos elementos.
- La composición de los elementos estructurales y del comportamiento en subsistemas progresivamente más grandes.
- El estilo de la arquitectura que guía esta organización: los elementos y sus interfaces, sus colaboraciones y su composición.

Sin embargo, la arquitectura software está afectada no sólo por la estructura y el comportamiento, sino también por el uso, la funcionalidad, el rendimiento, la flexibilidad, la reutilización, la facilidad de comprensión, las restricciones y compromiso económicos y tecnológicos, y la estética.

II.3.2. Los pasos hacia una arquitectura

La arquitectura se desarrolla mediante iteraciones, principalmente durante la fase de elaboración. Cada iteración se desarrolla como se esbozó en la sección II.2, comenzando con los requisitos y siguiendo con el análisis, diseño, implementación y pruebas, pero centrándose en los casos de uso relevantes desde el punto de vista de la arquitectura y en otros requisitos. El resultado final de la fase de elaboración es una línea base de la arquitectura, un esqueleto del sistema con pocos músculos de software.

Los casos de uso arquitectónicamente relevantes son aquellos que ayudan a mitigar los riesgos más importantes, aquellos que son los más importantes para los usuarios del sistema, y aquellos que ayudan a cubrir todas las funcionalidades significativas, de forma que nada quede en penumbra. La implementación, integración y prueba de la línea base de la arquitectura proporciona seguridad al arquitecto y otros trabajadores de su equipo, por lo que comprender estos puntos es algo operativo. Esto es algo que no puede obtenerse mediante un análisis y diseño sobre el papel. La línea base de la arquitectura de operación proporciona una demostración que funciona para que los trabajadores puedan proporcionar sus retroalimentaciones.

La descripción de la arquitectura tiene cinco secciones, una para cada modelo. Tiene una vista del modelo de casos de uso, una vista del modelo de análisis (que no siempre se mantiene), una vista del modelo de diseño, una vista del modelo de despliegue, y una vista del modelo de implementación. No incluye una vista del modelo de prueba porque no desempeña ningún papel en la descripción de la arquitectura, y sólo se utiliza para verificar la línea base de la arquitectura.

II.4. Proceso Unificado iterativo e incremental

Un proceso de desarrollo de software debe tener una secuencia de puntos en el que han de tomarse decisiones de negocio claramente articulados para ser eficaz, que proporcionen a los directores y al resto del equipo del proyecto los criterios que necesitan para autorizar el paso de una fase a la siguiente dentro del ciclo del producto.

Dentro de cada fase el proceso pasa por una serie de iteraciones e incrementos que conducen a esos criterios.

En la fase de inicio el criterio esencial es la viabilidad, que se lleva a cabo mediante:

- La identificación y reducción de los riesgos críticos para la viabilidad del sistema
- La creación de una arquitectura candidata a partir del desarrollo de un subconjunto clave de los requisitos, pasando por el modelado de los casos de uso.
- La realización de una estimación inicial de coste, esfuerzo, calendario y calidad del producto con límites amplios.
- El inicio del análisis del negocio por el cual el proyecto parece que merece la pena económicamente, una vez más con límites amplios.

En la fase de elaboración, el criterio esencial es la capacidad de construir el sistema dentro de un marco de trabajo económico, que se lleva a cabo mediante:

- La identificación y reducción de los riesgos que afectan de manera significativa a la construcción del sistema.
- La especificación de la mayoría de los casos de uso que representan la funcionalidad que ha de desarrollarse.
- La extensión de la arquitectura candidata hasta las proporciones de una línea base.
- La preparación del plan de proyecto con suficiente detalle como para guiar la fase de construcción.
- La realización de una estimación con unos limites suficientemente ajustados como para justificar la inversión.
- La terminación del análisis del negocio.

En la fase de construcción, el criterio esencial es un sistema capaz de una operatividad inicial en el entorno de usuario, y se lleva a cabo mediante:

- Una serie de iteraciones, que llevan a incrementos y entregas periódicas, de forma que a lo largo de esta fase, la viabilidad del sistema siempre es evidente en la forma de ejecutables.

En la fase de transición, el criterio esencial es un sistema que alcanza una operatividad final, llevado a cabo mediante:

- La modificación del producto para subsanar problemas que no se identificaron en fases anteriores.
- La corrección de defectos (anomalía del sistema).

Uno de los objetivos del Proceso Unificado es hacer que los arquitectos, desarrolladores e interesados en general comprendan la importancia de las primeras fases.

II.4.1. Desarrollo en pequeños pasos

La tercera clave del Proceso Unificado, iterativo e incremental, proporciona la estrategia para desarrollar un producto software en pasos pequeños manejables:

- Planificar un poco
- Especificar, diseñar, e implementar un poco.
- Integrar, probar, y ejecutar un poco cada iteración.

Al estar contento con un paso se avanza al siguiente. Entre cada paso, se obtiene retroalimentación que permite ajustar los objetivos para el siguiente paso. Después se da el siguiente paso, y después otro. Cuando se han dado todos os pasos que se habían planificado, se tiene un producto desarrollado que se puede distribuir a los clientes y usuarios.

Las iteraciones en las primeras fases tratan en su mayor parte con la determinación del ámbito del proyecto, la eliminación de los riesgos críticos, y la creación de la línea base de la arquitectura. Después, a medida que se avanza a lo largo del proyecto y se va reduciendo gradualmente los riesgos restantes e implementando los componentes, la forma de las iteraciones cambia, dando incrementos como resultados.

Un proyecto de desarrollo de software transforma un cambio de los requisitos de usuario en un cambio del producto del software. Con el método de desarrollo iterativo incremental esta adaptación de los cambios se realiza poco a poco. Dicho de otra forma, se divide el proyecto en un número de miniproyectos, siendo cada uno de ellos una iteración. Cada iteración tiene todo lo que tiene un proyecto de desarrollo de software: planificación, desarrollo en una serie de flujos de trabajo (requisitos, análisis, diseño, implementación y prueba), y una preparación para la entrega.

Pero una iteración no es una entidad completamente independiente. Es una etapa dentro de un proyecto, y se ve fuertemente condicionada por ello. Se dice que un miniproyecto porque no es por sí mismo algo que los usuarios hayan solicitado. Además, cada uno de estos miniproyectos se parece al antiguo ciclo de vida en cascada debido a que se desarrolla a través de actividades en cascada. Se podría llamar una "minicascada" a casa iteración.

El ciclo de vida iterativo produce resultados tangibles en forma de versiones internas (aunque preliminares), y cada una de ellas aporta un incremento y demuestra la reducción de los riesgos con los que se relaciona. Estas versiones pueden presentarse a los clientes y usuarios, en cuyo caso proporcionan retroalimentación valiosa para la validación del trabajo.

II.4.2. ¿Por qué iterativo e incremental?

Existen diversos motivos por lo cual el desarrollo del proceso sea iterativo e incremental, en dos palabras: para obtener un software mejor. Dicho con unas cuantas palabras más: para cumplir los hitos principales y secundarios con lo cual se controla el desarrollo. Y dicho con algunas palabras más:

- Para tomar las riendas de los riesgos críticos y significativos desde el principio.
- Para poner en marcha una arquitectura que guíe el desarrollo del software.
- Para proporcionar un marco de trabajo que gestiones de mejor forma los inevitables cambios en los requisitos y en otros aspectos.
- Para construir el sistema a los largo del tiempo en lugar de hacerlo de una sola vez cerca del final, cuando el cambiar algo se vuelva costoso.
- Para proporcionar un proceso de desarrollo a través del cual el personal puede trabajar de manera más eficaz.

En resumen, las iteraciones ayudan a la dirección a planificar, organizar, hacer el seguimiento y a controlar el proyecto.

Capítulo III: Lenguaje de Modelación Unificado UML

El Lenguaje de Modelación Unificado, conocido por sus siglas en inglés UML (Unified Modeling Language), es un lenguaje para la especificación, visualización, construcción y documentación de los elementos que constituyen un sistema de software orientado a objetos u otros sistemas que no estén relacionados con la creación de software. UML, representa la unificación de los métodos de Booch [Booch 1994], OMT (Object Modeling Technique) [Rumbaugh 1995] y OOSE (Object-Oriented Software Engineering) [Jacobson 1992]. El lenguaje de Modelación Unificado es el estándar formal para construir modelos.

III.1. Definición y Antecedentes

El análisis y diseño estructurado fue, posiblemente, la primera familia de métodos de desarrollo de software que fue usada ampliamente, formalizado durante los inicios de los años 70`s. Hacia finales de los 80`s, los lenguajes y procesos se estaban moviendo al paradigma orientado a objetos. En general, las técnicas de orientación a objetos, resolvían los problemas de complejidad y eran más apropiados para un proceso iterativo.

La cantidad de métodos orientados a objetos se incrementó de menos de 10 a más de 50 durante el periodo 1989 y 1994. Muchos usuarios de estos métodos tenían problemas para seleccionar aquel que llenara sus necesidades completamente, creando así la llamada guerra de los métodos. Uno de los conceptos iniciales detrás de UML era ponerle fin a esa guerra de los métodos dentro de la comunidad orientada a objetos. A mediados de 1990, aparece una nueva generación de estos métodos. Booch, definió la noción de que un sistema es analizado en diferentes perspectivas (vistas), donde cada vista es descrita por una serie de Diagramas. Rumbaugh definió el modelo OMT, este es un modelo basado en la especificación de requerimientos, el sistema es descrito por una serie de modelos: el modelo de objetos, el modelo dinámico y el modelo funcional, los cuales se complementan entre sí, para dar una descripción del sistema.

En 1994, comienza el desarrollo de UML, cuando Rumbaugh se unió a Booch en Rational Software Corporation e inician un trabajo conjunto con el fin de crear un método unificado que uniría el método de Booch y el método OMT. La primera versión de este método fue liberada en Octubre de 1995. Alrededor de esta misma fecha Ivar Jacobson (el autor del método OOSE) se une a ellos y el alcance de UML fue expandido para incorporar OOSE.

Los autores estaban motivados a crear un lenguaje unificado ya que cada método evolucionaba independientemente de los otros, pero todos convergían a lo mismo, era lógico buscar una evolución conjunta, además al unificar la semántica y notación se podría traer cierta estabilidad al mercado orientado a objetos.

Los objetivos de UML, establecidos por los diseñadores son:

- Modelar sistemas (no sólo de software) utilizando conceptos orientados a objetos.
- Establecer un acoplamiento explícito entre los artefactos conceptuales y los ejecutables.
- Resolver problemas de menor escala en sistemas complejos de misión crítica.
- Crear un lenguaje de modelación utilizable por los humanos y las máquinas.

UML tiende a ser un lenguaje de modelación común, utilizado en la industria. Tiene un amplio rango de uso, está construido sobre técnicas bien establecidas y probadas para el modelaje de sistemas. Tiene el soporte para la industria, necesario para establecer un estándar en el mundo real. UML está bien documentado con metamodelos (un modelo de los elementos del modelo) del lenguaje y con una especificación formal de la semántica del lenguaje.

Con la unificación de los métodos predecesores, UML logra cumplir con una serie de objetivos planteados, entre los más importantes cabe resaltar:

- Proveer semántica y notación suficiente para dirigirse a una amplia variedad de problemas de modelación.
- Proveer semántica para dirigirse a futuros problemas de modelación (tecnológicos y distribuidos).
- Proveer semántica para facilitar el intercambio entre una gran variedad de herramientas del mercado orientado a objeto.
- Proveer semántica para especificar interfaces necesarias en el desarrollo del sistema.
- Proveer mecanismos de extensibilidad para proyectos individuales que puedan extender el metamodelo a través del uso de estereotipos, valores etiquetados y restricciones.

Todos los elementos y Diagramas de UML están basados en el paradigma orientado a objetos. UML es usado para modelar sistemas, cuyo rango es muy amplio (muchos tipos de sistemas pueden ser descritos). UML también puede ser usado en las diferentes fases de desarrollo de un sistema, desde la especificación de requerimientos hasta la prueba del sistema terminado.

Cuando se está construyendo sistemas con UML, hay distintos modelos en las diferentes fases del desarrollo y los propósitos de los modelos son también diferentes. En la fase de análisis, el propósito de los modelos es capturar los requerimientos del sistema y modelar las clases básicas del "mundo real" y las colaboraciones. En la fase de diseño, el propósito del modelo es expandir el modelo de análisis en una solución técnica de trabajo con consideración del ambiente de implementación. En la fase de implementación, el modelo es el código que es programado y compilado en los programas. Y, finalmente, en el modelo de

despliegue, una descripción explica la forma en que el sistema es desplegado en la arquitectura física. El control entre las fases y los modelos es mantenido a través de las propiedades y relaciones de refinamiento.

III.2. COMPONENTES DE UML

UML consiste de:

- Un metamodelo formal.
- Una notación gráfica.

Metamodelo

Es el modelo de un modelo. Un metamodelo describe los componentes de un modelo y sus relaciones. El metamodelo de UML se describe como una combinación de texto y Diagramas de Clase, usando el propio UML. El propósito del metamodelo es proveer una semántica y sintaxis común, simple y no ambigua de los elementos que conforman a UML.

Notación gráfica

Es la parte visible de UML y constituye una sintaxis gráfica que las personas y las herramientas usan, para modelar los sistemas.

UML tiene cuatro tipos de modelos: *el modelo funcional, el de comportamiento, el estructural y el de implementación*, un modelo es una descripción completa de un sistema desde una perspectiva particular. Ciertos conceptos están asociados a UML: vistas, diagramas, elementos del modelo y mecanismos generales.

Vistas

Muestran diferentes aspectos de los sistemas que son modelados. Una vista no es un gráfico, pero es una abstracción que consiste en una serie de diagramas. Solamente, definiendo una serie de vistas, cada una mostrando un aspecto particular del sistema, puede ser construida una imagen completa del sistema.

Diagramas

Son los gráficos que describen los contenidos de las vistas. UML tiene nueve tipos diferentes de diagramas, que son usados en combinación para proporcionar diferentes vistas de un sistema. En términos de las vistas de un modelo, UML define los siguientes diagramas:

- Diagramas de Secuencia.
- Diagramas de Colaboración.
- Diagramas de Estado.
- Diagrama de Actividad.
- Diagrama de Ejecución.
- Diagrama de Componentes.
- Diagrama de Objeto.

- Diagrama de Clases.
- Diagrama de Casos de Uso.

Los diagramas de secuencia, colaboración, estado y actividad, están asociados al modelo de comportamiento. Los diagramas de ejecución y componentes están asociados al modelo de implementación. Los diagramas de objetos y clases están asociados al modelo estructural y el diagrama de casos de uso está asociado al modelo funcional. La figura III-1 muestra los diagramas y Modelos de UML.

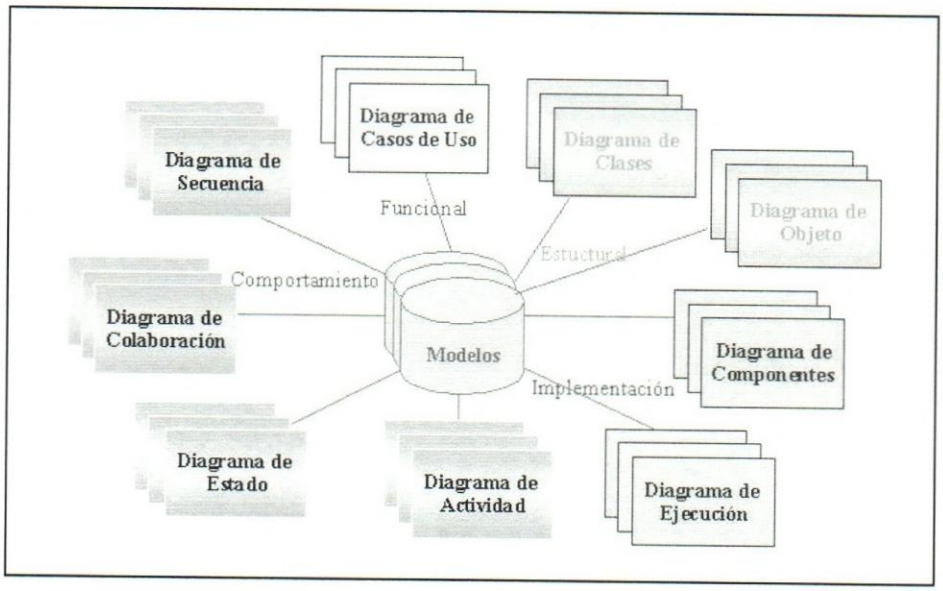


Figura III-1: Modelos y diagramas de UML

Elementos del modelo

Representan los conceptos utilizados en los diagramas, tales como clases, objetos, mensajes, relaciones de asociación, dependencia, generalización. Un elemento del modelo puede ser utilizado en diferentes tipos de diagramas, pero siempre tiene el mismo significado y símbolo.

Mecanismos generales

Proporcionan comentarios extras, información o semántica a cerca de un elemento del modelo, ellos proporcionan también mecanismos de extensión para adaptar o extender UML a un método, proceso, organización o usuario específico.

III.2.1. Vistas de UML

El modelaje de un sistema complejo es una tarea extensa. Idealmente, el sistema completo sería descrito con un solo diagrama gráfico que sea fácil de comunicar y entender. Sin embargo, en la práctica esto es, muchas veces, imposible. Un solo diagrama gráfico no puede capturar toda la información necesaria para describir sistemas complejos. Un sistema

puede analizarse de acuerdo a diferentes perspectivas: aspectos funcionales (su estructura estática e interacciones dinámicas) y aspectos no funcionales (requerimientos de tiempo, confiabilidad, despliegue, etc.). Es por ello que, un sistema es descrito mediante una serie de vistas, donde cada vista representa una proyección de la descripción completa, mostrando un aspecto particular del mismo.

Cada vista es descrita en una serie de diagramas, que contienen información, la cual enfatiza un aspecto particular del sistema. En ocasiones, un diagrama puede ser parte de una o más vistas. Analizando al sistema desde diferentes vistas, es posible concentrarse en un aspecto del sistema a la vez. Un diagrama en una vista particular deberá ser suficientemente simple para ser fácilmente comunicado, y ser coherente con los otros diagramas y vistas, de manera que una imagen completa del sistema es descrita por las vistas en conjunto (a través de sus respectivos diagramas). Un diagrama contiene símbolos gráficos que representan los elementos de modelo del sistema.
Las vistas que se definen en UML, son:

- *Vista de casos de uso*: es una vista que muestra las funcionalidades de un sistema.
- *Vista lógica*: Es una vista que muestra cómo es diseñada la funcionalidad del sistema, en términos de las estructuras estáticas del sistema y su comportamiento dinámico.
- *Vista de componentes*: es una vista que muestra la organización de los componentes de código.
- *Vista de procesos*: es una vista que muestra la concurrencia en el sistema, resolviendo problemas de comunicación y sincronización que están presentes en un sistema concurrente.
- *Vista de despliegue*: es una vista que muestra el despliegue de un sistema dentro de una arquitectura física con computadoras y dispositivos denominados nodos.

III.3. Diagramas de UML

UML tiene nueve diagramas, tal y como se observa en la figura III-1, cada uno de estos diagramas provee una visión o perspectiva del sistema, pueden ser construidos en el contexto de diferentes procesos de desarrollo y la importancia de cada uno de ellos para la comprensión de un sistema, depende del tipo de aplicación. Estos diagramas muestran diversas perspectivas del sistema. Un diagrama es una parte de una vista específica; es por ello, que cuando es dibujado, es usualmente adecuado para una vista. Algunos tipos de diagramas pueden ser parte de varias vistas, dependiendo de los contenidos del diagrama.

A continuación se dará una descripción de los conceptos básicos de cada diagrama.

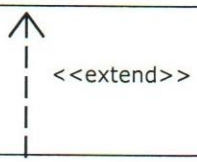
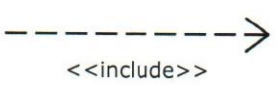
III.3.1. Diagrama de casos de uso

Un diagrama de casos de uso describe las funcionalidades del sistema, a partir de las interacciones del usuario. Un diagrama de casos de uso es una vista gráfica de algunos o

todos los actores, casos de uso y sus interacciones, identificados en un sistema. Cada sistema típicamente tiene un diagrama de caso de uso principal, el cual es la imagen de las fronteras del sistema (actores) y la funcionalidad principal proporcionada por el sistema (casos de uso). Otros diagramas de caso de uso pueden ser desarrollados o refinados en el momento que se requiera.

Los conceptos más importantes en el diagrama de casos de uso son: actor, caso de uso, asociación, extend, generalización e include, a continuación se explicará brevemente cada uno de ellos:

Tabla 1: Conceptos del diagrama de casos de uso

Nombre	Definición	Representación Gráfica
Actor	Es una entidad externa que interactúa con el sistema activando los casos de uso.	
Caso de uso	Es una secuencia de transacciones, iniciadas por un actor y que constituye una funcionalidad del sistema.	
Asociación	Relación que representa la participación de un actor en un caso de uso.	
Extend	Relación que define un curso alterno opcional (dependiendo de una condición) de otro caso de uso.	
Generalización	Relación que define un caso de uso, como una generalización de otro caso de uso.	
Include	Relación que define una instancia de un caso de uso como un curso obligatorio en otro caso de uso.	

La figura III-2 muestra un ejemplo de un diagrama de casos de uso, el cual modela las actividades que se realizan cuando se va al cine, se ve al actor interactuando con el caso de uso ir al cine, se ve una relación *include* con el caso de uso comprar entrada y una relación de *extend* con el caso de uso comprar cotufa.

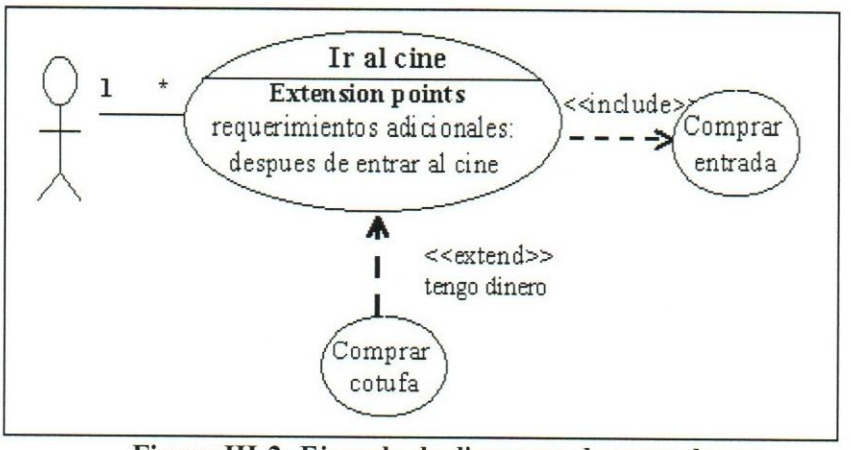


Figura III-2: Ejemplo de diagrama de casos de uso

III.3.2. Diagrama de clases

Un diagrama de clases es un modelo estático. Describe la vista estática del sistema. Aunque tiene similitudes con un modelo de datos (entidad-relación), las clases no sólo muestran la estructura de la información, sino que describen también el comportamiento.

Un diagrama de clases describe el sistema identificando sus clases, objetos e interrelaciones entre ellos. Un propósito de los diagramas de clases es definir una base para otros diagramas donde otros aspectos del sistema son mostrados (tales como los estados de los objetos o la colaboración entre ellos mostrados en los diagramas dinámicos). Una clase, en un diagrama de clase, puede ser directamente implementada en un lenguaje de programación orientado a objetos.

Los diagramas de clases son creados para proporcionar una imagen o vista de algunas o todas las clases en el modelo (véase figura III-3). El diagrama de clases principal en la vista lógica del modelo, es típicamente, una imagen de los paquetes del sistema (a veces, a este diagrama se le llama diagrama de paquetes). Cada paquete también tiene su diagrama de clases principal, que despliega las clases públicas del paquete. Otros diagramas se crean según sea necesario.

Otros usos de estos diagramas son:

- Vista de todas las clases de implementación en un paquete.
- Vista de la estructura y comportamiento de una o más clases.
- Vista de una jerarquía de herencia.

Los diagramas de clases también pueden ser creados a partir del análisis de la vista de casos de uso del modelo y contienen una vista de las clases que participan en el caso de uso.

Los conceptos más importantes en el diagrama de clases son: clase, objeto, relaciones de asociación, generalización y dependencia. Estos serán explicados en el transcurso del capítulo.

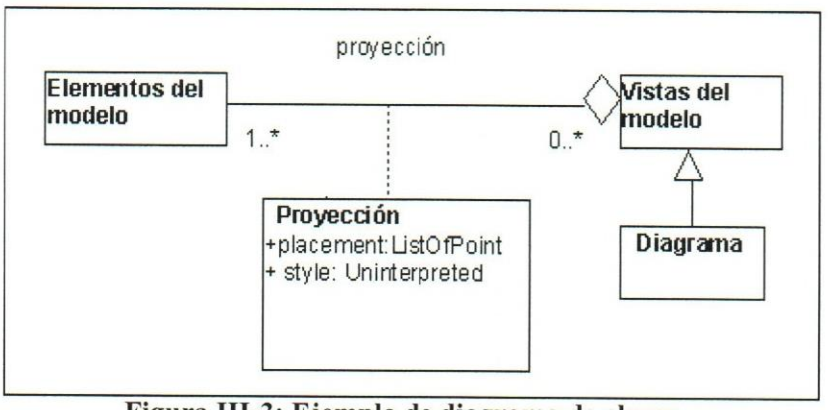


Figura III-3: Ejemplo de diagrama de clases

III.3.3. Diagrama de objetos

Un diagrama de objetos es una variante del diagrama de clases y utiliza casi la misma notación. La diferencia entre los dos es que el diagrama de objetos muestra una serie de objetos (instancias de las clases), en lugar de las clases. Un diagrama de objetos describe el sistema identificando sus objetos e interrelaciones. Un diagrama de objetos es, entonces, una instanciación de un diagrama de clases que muestra una posible imagen de la ejecución del sistema en algún punto dado del tiempo.
Los diagramas de objetos son también utilizados en los diagramas de colaboración, en los cuales es mostrada la colaboración dinámica entre los objetos.

Los conceptos más importantes del diagrama de objetos, al igual que para el diagrama de clases, serán explicados en el transcurso del capítulo.

III.3.4. Diagrama de estado

Un diagrama de estado describe la evolución de los objetos, esto es, los cambios de estado de cada objeto en su tiempo de vida. Un diagrama de estado es típicamente el complemento de la descripción obtenida en un diagrama de clases. En estos diagramas se muestran todos los estados posibles que los objetos de la clase puedan tener y qué eventos causan un cambio de estado. Un evento puede ser otro objeto que envía un mensaje. Un cambio de estado es llamado transición. Una transición puede tener también una acción conectada a él, en esta acción, se puede especificar que se realizará en la conexión con el estado de transición.

Los diagramas de estados no son dibujados para todas las clases, solamente para aquellas que tienen una serie de estados bien definidos y en donde el comportamiento de la clase es afectado y cambiado por otros diferentes estados. Los diagramas de estados pueden también ser dibujados para el sistema en su totalidad.

Los conceptos más importantes del diagrama de estado, son: estado, transición, evento y acción.

- *Estado*: situación durante el tiempo de vida de un objeto. Los estados se representan, gráficamente, con un rectángulo, cuyas puntas son redondeadas.
- *Transición de estado*: relación que indica un cambio de estado, se representan con una flecha.
- *Evento*: es el acto que genera una transición de estado, se representa con una cadena de caracteres.
- *Acción*: es el acto que se realiza sobre un estado y no genera ningún cambio sobre él.

La figura III-4 muestra un ejemplo de un diagrama de estado, que modela el comportamiento de un carro.

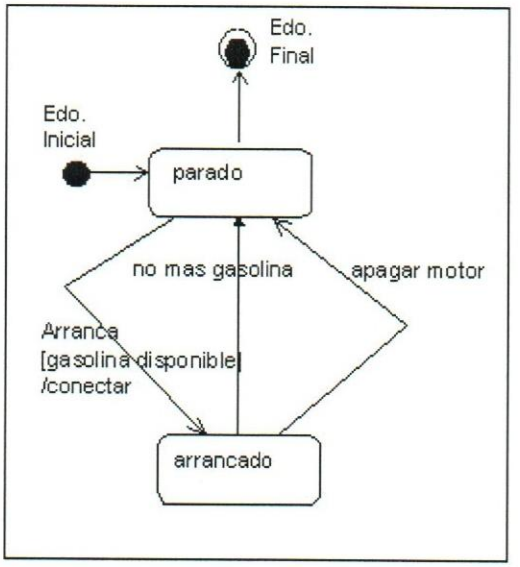


Figura III-4: Ejemplo de diagrama de estado

III.3.5. Diagrama de secuencia

Un diagrama de secuencia describe la interacción entre los objetos, ordenada en el tiempo. Muestra una colaboración dinámica entre una serie de objetos. El aspecto importante de este diagrama es mostrar la secuencia de mensajes enviados entre los objetos. También son mostradas las interacciones entre los objetos, algo que sucederá en un punto específico de la ejecución de un sistema, tal como se muestra en la figura III-5, un diagrama de secuencia consiste en una serie de objetos cuyos tiempos de activación son mostrados con líneas verticales. El tiempo pasa descendentemente en el diagrama y el diagrama muestra el intercambio de mensajes entre los objetos, a medida que pasa el tiempo en la secuencia o función. Los mensajes son mostrados como líneas, con flechas de mensajes entre las líneas verticales de los objetos. Las especificaciones de tiempo y otros comentarios son añadidos en una escritura en el margen del diagrama.

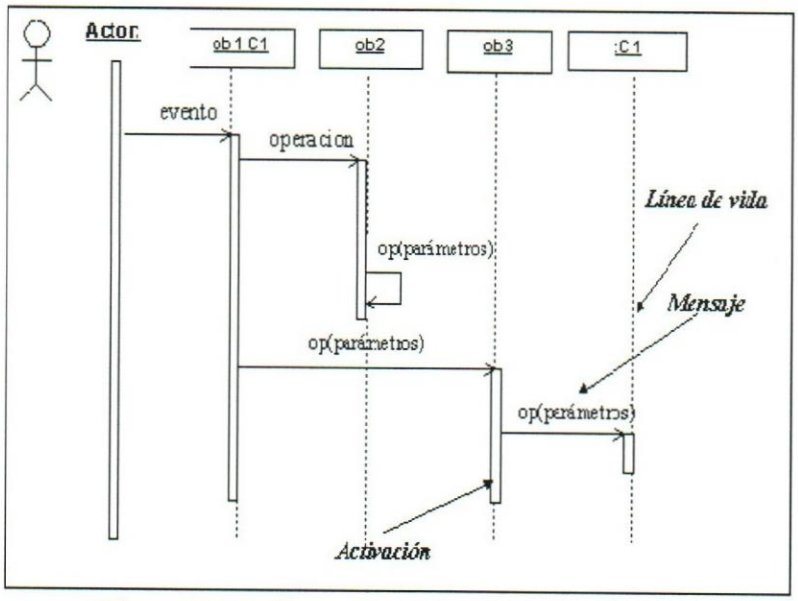


Figura III-5: Ejemplo de diagrama de secuencia

Los conceptos asociados al diagrama de secuencia son: actor (descrito anteriormente), objetos que se representan con líneas verticales como se ve en la figura III-5, eventos y las operaciones son de tipo mensaje de string (cadena de caracteres).

Un ejemplo de un diagrama de secuencia donde se modela las actividades relacionadas con la realización de una llamada telefónica, se puede observar en la figura III-6.

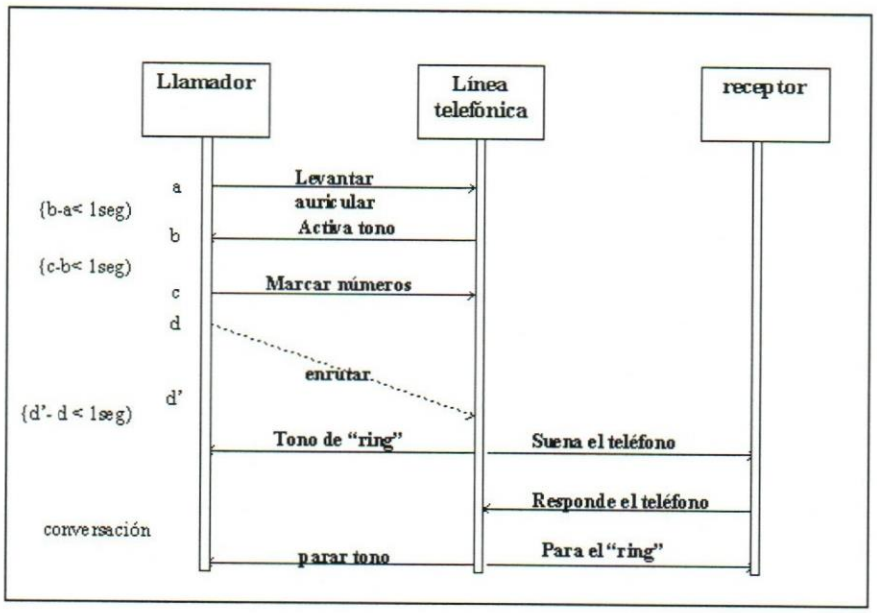


Figura III-6: Ejemplo de diagrama de secuencia de una llamada telefónica

III.3.6. Diagrama de colaboración

Un diagrama de colaboración describe la interacción entre los objetos, en un período de tiempo. Muestra una colaboración dinámica, tal como el diagrama de secuencia, lo que pasa es que el diagrama de secuencia ordena esas interacciones de los objetos, en el tiempo. Además de mostrar el intercambio de mensajes (interacción), el diagrama de colaboración muestra los objetos y sus relaciones (a veces referidos como el *contexto*). Una colaboración se puede mostrar con un diagrama de secuencia o con un diagrama de colaboración. A menudo, se puede decidir utilizar cualquiera de los dos: si el tiempo o la secuencia es el aspecto más importante a enfatizar, se escoge un diagrama de secuencia; si es importante enfatizar el contexto, se escoge un diagrama de colaboración. La interacción entre los objetos es mostrada en ambos diagramas.

El diagrama de colaboración es dibujado como un diagrama de objetos, donde una serie de objetos son mostrados junto con sus relaciones tal como se muestra en la figura III-7. Note, en esta figura, que las flechas de mensajes son dibujadas entre los objetos para mostrar el flujo de mensajes entre ellos. Se colocan etiquetas en los mensajes, lo cual, entre otras cosas, muestra el orden en el cual son enviados los mensajes. También pueden mostrarse las condiciones, iteraciones, valores de retorno y así sucesivamente. Cuando se está familiarizado con la sintaxis de etiquetas para los mensajes, el desarrollador puede leer la colaboración y seguir el flujo de ejecución y el intercambio de mensajes. Los diagramas de secuencia y de colaboración son diagramas de interacción, ambos, utilizan la misma información (objetos y las relaciones existentes entre ellos). La figura III-7 muestra un dibujo de un diagrama de colaboración, donde se observa un actor que interactúa con el objeto *ob1* y este objeto está relacionado con el objeto *ob2* y *ob3*, se observa también en la gráfica el uso de eventos como la acción que desencadena un cambio de estado y operaciones con parámetros y sin parámetros, que modifican el estado de los objetos.

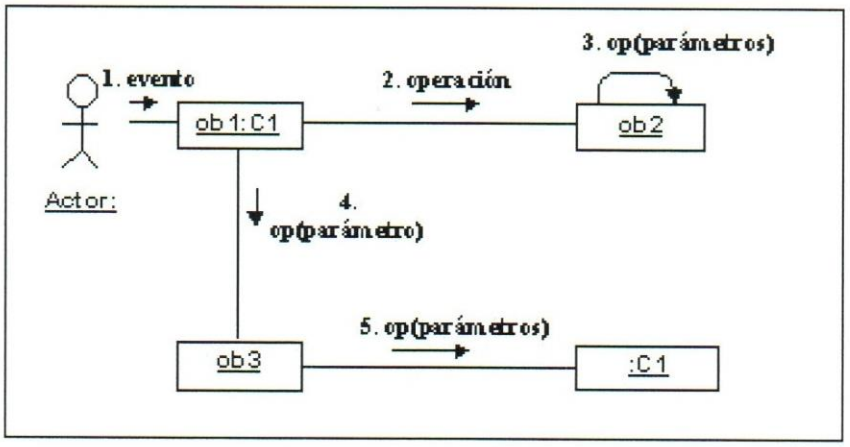


Figura III-7: Ejemplo de diagrama de colaboración

III.3.7. Diagrama de actividad

Un diagrama de actividad muestra el flujo secuencial de las actividades. El diagrama de actividad es utilizado típicamente para describir las actividades realizadas en una operación, aunque puede ser también utilizado para describir otros diagramas, tal como un caso de uso o de interacción. El diagrama de actividad consiste de estados, los cuales contienen una especificación de la actividad que va a ser realizada. Un estado termina cuando ha sido realizada la actividad. Por lo tanto, el control fluye entre los estados, que están conectados entre sí. Las decisiones y las condiciones, así como la ejecución en paralelo de los estados de acción, también pueden ser mostradas en el diagrama. El diagrama puede también tener especificaciones de los mensajes que han sido enviados o recibidos como parte de las acciones realizadas.

Los conceptos más importantes en un diagrama de actividad, son las actividades y las relaciones entre ellas. Las *actividades* son descripciones que se colocan dentro de estados, explicados anteriormente en el diagrama de estados y las relaciones se modelan con flechas. En el diagrama de actividad se agrega una barra horizontal gruesa que permite expresar la sincronización de condiciones; las condiciones, cuando existen, se colocan como un mensaje. La figura III-8 muestra un ejemplo de un diagrama de actividad que modela las actividades involucradas en el proceso de hacer café.

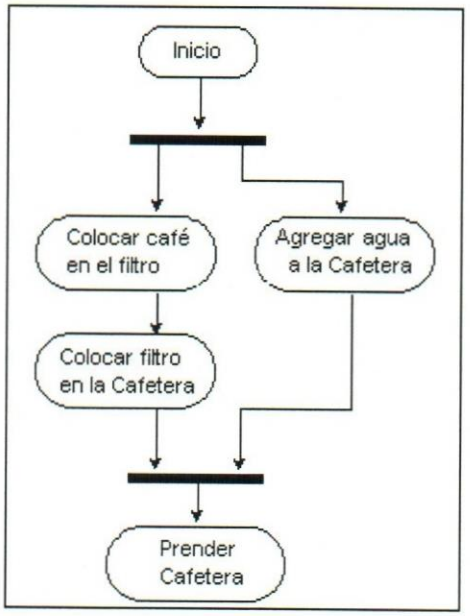


Figura III-8: Ejemplo de diagrama de actividad

III.3.8. Diagrama de componentes

Un diagrama de componentes describe la interacción entre componentes de software. Muestra la estructura del código en término de sus componentes. Un componente puede ser

un componente de código fuente, un componente binario, o un componente ejecutable. Un componente contiene información sobre la clase lógica o las clases que implementa, creando una correspondencia de la vista lógica a la vista de componentes. Las dependencias entre los componentes son mostradas, haciendo fácil de analizar como los otros componentes son afectados por un cambio en uno de los componentes. El diagrama de componentes es utilizado en trabajos prácticos de programación.

III.3.9. Diagrama de ejecución

El diagrama de ejecución muestra la arquitectura física del hardware y el software en el sistema. Se pueden mostrar las computadoras y los dispositivos (nodos), junto con las conexiones que tienen unos con otros; también se puede mostrar el tipo de conexión. Dentro de los nodos, los componentes ejecutables y objetos son localizados para mostrar que unidades de software son ejecutadas y en qué nodos. Además, se muestran las dependencias entre los componentes.

III.4. Elementos, Mecanismos Generales y Extensión del Modelo

Un elemento del modelo es definido con una semántica que precisa una definición formal del elemento o el significado exacto de lo que representa. Un elemento del modelo también tiene un elemento de vista correspondiente, el cual es una representación gráfica del elemento o el símbolo gráfico utilizado para representar al elemento en los diagramas. Un elemento puede existir en varios tipos diferentes de diagramas, pero hay reglas para las cuales los elementos pueden ser mostrados en un tipo de diagrama específico. Algunos ejemplos de elementos del modelo son:

- *Diagrama de clases*: clase, objeto, relaciones de asociación, generalización, especialización.
- *Diagrama de casos de usos*: actor, caso de uso, relaciones de asociación, extend, generalización, include.
- *Diagrama de estado*: estado, transición de estado.
- *Diagrama de actividad*: actividad.
- *Diagrama de implementación*: nodo, conexiones de nodos.
- *Administración del modelo*: paquete, subsistemas, modelo.
- *En general*: nota, interfaces.
- Otros elementos del modelo, además de los descritos incluyen mensajes, acciones y estereotipos.

UML utiliza algunos mecanismos generales en todos los diagramas, para añadir información adicional en los mismos, lo cual típicamente no puede ser representado utilizando las habilidades básicas de los elementos del modelo. Estos mecanismos son los adornos y las notas.

III.4.1. Adornos

Los adornos gráficos pueden ser incorporados a los elementos del modelo en los diagramas. Los adornos agregan semántica al elemento, es decir, representan símbolos o breves explicaciones que se deseen agregar a los elementos de los diagramas, para una mejor comprensión de los mismos.

Los adornos son descritos junto con la descripción del elemento que ellos afectan. Un ejemplo de adorno es la técnica utilizada para separar un tipo de una instancia. Cuando un elemento representa un tipo, su nombre es mostrado en negrillas. Cuando el mismo elemento representa una instancia del tipo, su nombre es subrayado y puede representar el nombre de la instancia, así como el nombre del tipo. Por ejemplo **Impresora,** representa un tipo y una instancia del tipo se representa así: Impresora HP 5M.

III.4.2. Notas

No todo puede ser definido en un lenguaje de modelación, no importa cuan extenso sea el lenguaje. Para permitir agregar información al modelo que de otra manera no puede ser representada, UML proporciona las notas. Una nota puede ser puesta en cualquier lugar del diagrama y puede contener cualquier tipo de información. Su tipo de información es una cadena que no puede ser interpretada por UML. Una nota contiene, a menudo, comentarios y preguntas del modelador como un recordatorio para resolver un dilema, más tarde. Las notas pueden tener también estereotipos que describen el tipo de nota. La nota es agregada a algún elemento en el diagrama con alguna línea punteada que especifica cuál elemento está siendo explicado o detallado, junto con la información en la nota. La figura III-9 muestra un ejemplo donde se utiliza una nota.

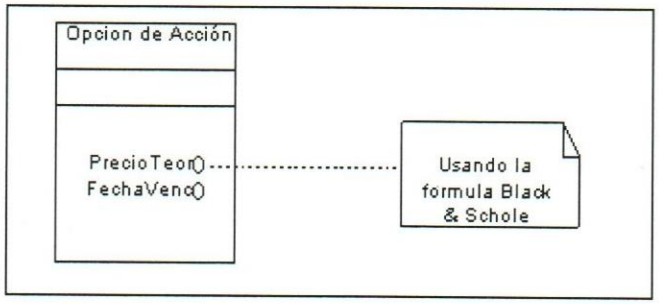


Figura III-9: Ejemplo de una nota

UML puede ser extendido o adaptado a un método específico, organización o usuario. Los estereotipos, valores agregados y las restricciones son mecanismos de extensión en UML.

III.4.3. Estereotipos

Los estereotipos son mecanismos de extensión que permiten definir un nuevo tipo de elemento del modelo, basado en un elemento de modelo existente. Un estereotipo "es justo

como" un elemento existente, es decir, un nombre, más una semántica extra que no está presente en el elemento existente. Un estereotipo de un elemento puede ser utilizado en las mismas situaciones en las cuales es utilizado el elemento original. Los estereotipos están basados en todos los tipos de elementos clases, nodos, componentes y notas; así como, relaciones tales como asociaciones, generalizaciones y dependencias.

Una serie de estereotipos son predefinido en UML, y son utilizados para ajustar un elemento del modelo existente en vez de definir uno nuevo, por ejemplo el estereotipo <<utility>> es una agrupación de variables y procedimientos globales en la forma de una declaración de clase (esta construcción corresponde a una convención de programación).

Un estereotipo es descrito poniendo su nombre como una cadena rodeada de los siguientes símbolos (<< >>). Un estereotipo puede también tener su propia representación gráfica, tal como un icono, conectado a él. Un elemento de un estereotipo específico puede ser mostrado en su representación normal con el nombre del estereotipo encima del nombre, como un icono gráfico representando al estereotipo ó una combinación de ambos.

Un estereotipo puede ser utilizado para agregar la semántica necesaria con el objetivo de definir elementos del modelo.

UML provee una lista de estereotipos predefinidos, pudiendo ser extendida por los usuarios. Algunos de los estereotipos que están definidos en UML, son: <<type>>, <<implementationClass>>, <<interface>>, <<utility>>. En la Figura III-10 se presenta un ejemplo de estereotipos, destacándose cómo estos permiten distinguir elementos de diferentes tipos (clase, tipo y paquete).

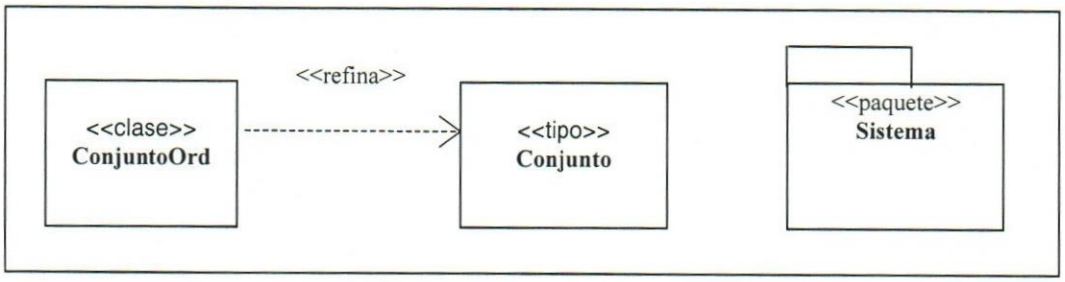


Figura III-10: Ejemplo de estereotipos

III.4.4. Valores agregados

Los elementos pueden tener propiedades que contienen pares de nombre-valor de la información sobre ellos. Estas propiedades son también conocidas como valores agregados. Una serie de propiedades son predefinidas en UML, pero las propiedades pueden también ser definidas por el usuario para guardar información adicional sobre los elementos. Cualquier tipo de información puede ser incorporada a los elementos: información de un método específico, información administrativa sobre el progreso del modelaje, información

utilizada por otras herramientas, tal como herramientas de generación de código o cualquier tipo de información que el usuario desee agregar a los elementos. En la figura III-11, se tienen las propiedades de la clase *instrumento*, donde a*bstrac* es una propiedad predefinida y *autor* y *estado* son valores agregados definidos por interés del diseñador.

Figura III-11: Ejemplo de una clase con valores agregados

III.4.5. Restricciones

Una restricción limita el uso de un elemento o la semántica del elemento. Una restricción es declarada en una herramienta y utilizada repetidamente en varios diagramas o es definida y aplicada en la medida que es necesitada en un Diagrama.

La figura III-12 muestra una asociación entre la clase *Grupo Ciudadanos Mayores* y la clase *Persona*, indicando que el grupo puede tener personas asociadas a él. Sin embargo, para indicar que solo las personas mayores de 60 años pueden incorporarse a él, se define una restricción que limita la participación a solamente las personas mayores de 60 años.

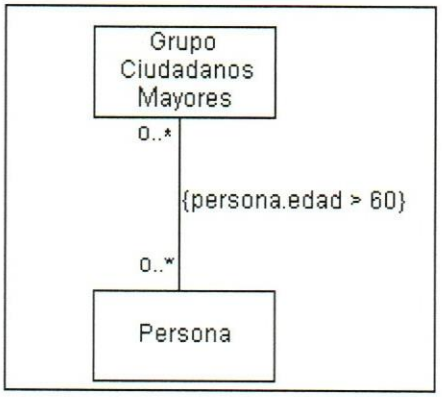


Figura III-12: Ejemplo de una restricción de los objetos personas

Capítulo IV: SINTAXIS Y SEMÁNTICA DE UML

Este capítulo presenta un resumen de la semántica y notación de UML, para lo cual se toman referencias de los documentos publicados por Rational Software, en especial los de resumen de UML [26], semántica de UML [25] y guía de notación de UML [24].

El metamodelo de UML define la semántica para representar modelos de objetos usando UML. Este se define usando un subconjunto de la notación y semántica de UML, para especificarlo, se utiliza el diagrama de clases. El metamodelo de UML forma parte de una arquitectura de cuatro capas. Debido a su complejidad, el metamodelo se descompone en paquetes lógicos.

La arquitectura de capas se estructura de la siguiente forma:

- Meta-metamodelo.
- Metamodelo.
- Modelo.
- Objetos de usuario.

Las funciones de estas capas son resumidas en la siguiente tabla:

Tabla 2: Arquitectura de capas del metamodelo

Capa	Descripción	Ejemplo
Meta-metamodelo	Es la infraestructura para una arquitectura de metamodelo. Define el lenguaje para especificar metamodelos.	*MetaClase, MetaAtributo, MetaOperación.*
Metamodelo	Es una instancia de un meta-metamodelo. Esta define el lenguaje para especificar un modelo.	*Clase, Atributo, Operación, Componente.*
Modelo	Es una instancia de un metamodelo, la cual define un lenguaje para describir un dominio de información.	*StockShare, askPrice, sellLimitOrder, StockQuoteServer.*
Objetos del usuario (user data)	Es una instancia de un modelo que define un dominio de una información específica.	*<Acme_Software_Share_98789>, 654.56, sell_limit_order, <Stock_Quote_Svr_32123>.*

La capa meta-metamodelo forma la base para la arquitectura del metamodelo. La principal función de esta capa es definir el lenguaje para especificar un metamodelo. Un meta-metamodelo define un modelo en un nivel de abstracción más alto que un metamodelo y, por lo general, es más compacto que el metamodelo que este describe. Un meta-metamodelo puede definir múltiples metamodelos y estos, a su vez, pueden tener múltiples meta-metamodelos asociados. Ejemplos de meta-metaobjetos en la capa de meta-metamodelo son: MetaClase, MetaAtributo y MetaOperación.

Un metamodelo es una instancia de un meta-metamodelo. La función primaria de la capa metamodelo es definir un lenguaje para especificar modelos. Los metamodelos son más elaborados que los meta-metamodelos que los describen, especialmente cuando ellos definen semántica dinámica. Ejemplos de metaobjetos en la capa de metamodelo son: *clase, atributo, operación y componente.*

Un modelo es una instancia de un metamodelo. La responsabilidad principal de la capa Modelo es definir un lenguaje que describa un dominio de información. Ejemplos de objetos en la capa de modelo son: *StockShare, askPrice, sellLimitOrder y StockQuoteServer.*

Objetos de usuario (user data) son instancias de un modelo. La función principal de la capa de Objetos de usuario es describir un dominio específico de información.

IV.1. Estructura en paquetes del metamodelo UML

El metamodelo de UML es complejo. Está compuesto de, aproximadamente, 90 metaclases y más de 100 meta-asociaciones e incluye, al menos, 50 estereotipos. Para disminuir la complejidad del metamodelo, éste se organiza en paquetes lógicos. Estos paquetes agrupan clases que muestran una fuerte cohesión entre ellas y reducen acoplamiento con las metaclases de otros paquetes, es decir se encuentran poco relacionadas. La descomposición del metamodelo de UML en paquetes es mostrada en la Figura IV-1.

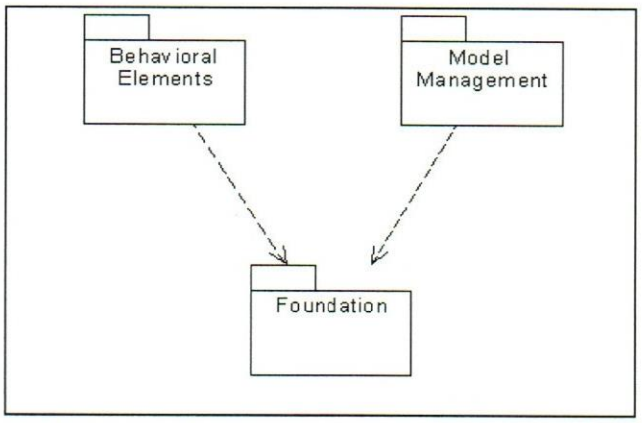


Figura IV-1: Descomposición en paquetes del metamodelo de UML.

La descomposición de los paquetes *Foundation* es mostrada en la Figuras IV-2.

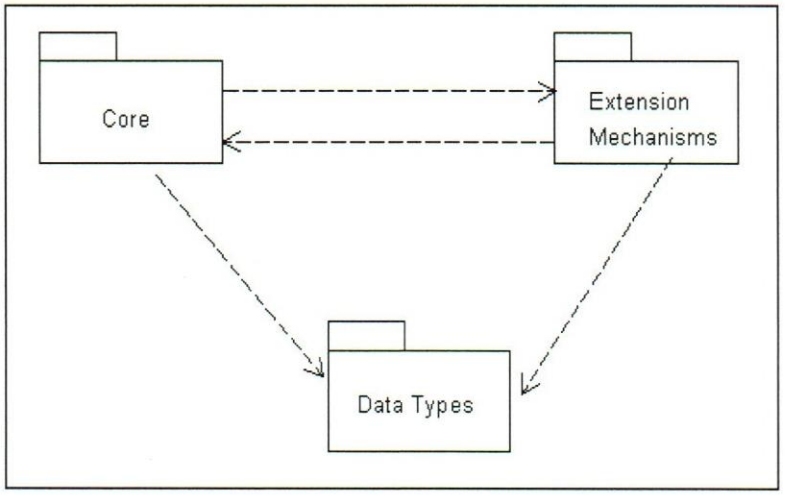


Figura IV-2: Paquete Foundation.

El estudio del subpaquete *Core* perteneciente al paquete *Foundation* es el más relevante, para el estudio de los Diagramas de Clases.

IV.2. PAQUETE FOUNDATION: CORE

El paquete *Core* es el más importante de los subpaquetes que componen el paquete *Foundation* de UML. Define las estructuras abstractas y concretas necesarias para el desarrollo de modelos de objeto. Las estructuras abstractas del metamodelo no son instanciables y, normalmente, se usan para concretar las estructuras importantes, partes de estructuras y organizar al modelo. Las estructuras concretas del metamodelo son instanciables y reflejan las estructuras de un modelo. Las estructuras abstractas definidas en el *Core* incluyen elementos del modelo, elementos generalizables y clasificadores. Las estructuras concretas especificadas en el *Core* incluyen clase, atributo, operación y asociación.

El paquete *Core* especifica las estructuras principales o centrales requeridas para un metamodelo básico y define un esquema principal para unir constructores adicionales al lenguaje tales como meta-clases, meta-asociaciones y meta-atributos. Este paquete es la base fundamental para el paquete *Foundation*, que a su vez sirve de infraestructura para el resto del lenguaje. En otros paquetes el *Core* es extendido agregando meta-clases al elemento principal que usa generalizaciones y asociaciones.

IV.3 Diagramas de Clases

Los diagramas de estructuras estáticas son los diagramas de clases y los diagramas de objetos. Los diagramas de clases muestran la estructura estática del modelo, en particular, aquellos elementos que lo conforman tales como clases, objetos, interfaces y tipos, su estructura interna y sus relaciones. Un diagrama de objetos es una instancia de un diagrama de clases, muestra una visión del sistema. Un diagrama de clases puede contener objetos, mientras que un diagrama de objetos sólo contiene objetos.

A continuación se describen las características de notación de los elementos de un Diagrama de Clases, así como las relaciones conceptuales entre ellos.

Clases
Son los elementos fundamentales del diagrama. Una clase describe un conjunto de objetos con estructura similar y el mismo comportamiento.

Una clase se representa por un rectángulo con tres compartimentos, donde el primero contiene el nombre de la clase y otras propiedades generales de la clase incluyendo estereotipos, el segundo contiene una lista de atributos y el tercero una lista de operaciones, como se muestra en la Figura IV-3. Los compartimentos de atributos y operaciones del rectángulo de clase se pueden omitir para reducir el nivel de detalle, cuando sea necesario.

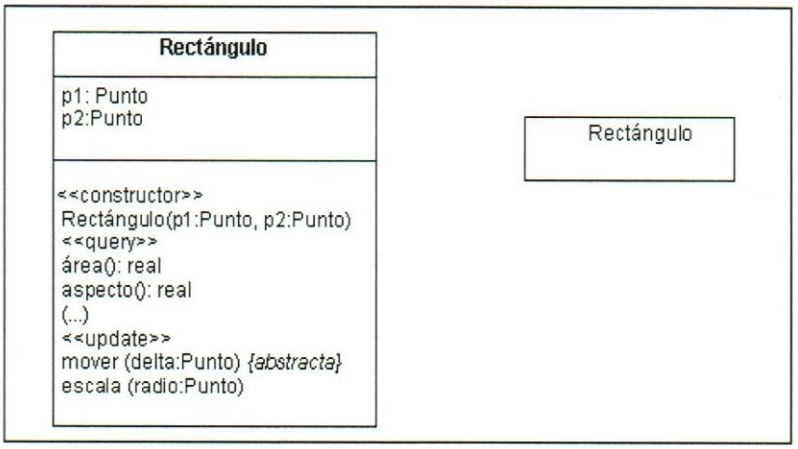


Figura IV-3: Notación UML para una clase Rectángulo

Componentes de una clase
Los componentes de una clase son sus atributos y operaciones.

Atributo
Permiten identificar las características propias de la clase. Su sintaxis está dada por:
<visibilidad nombre>:<expresión-de-tipo>=<valor-inicial{string-de-propiedades}>

La visibilidad, el valor inicial y el *string* de propiedades son opcionales.
Ejemplo: en la Figura IV-3, los atributos de la clase Rectángulo son:

p1: Punto
p2: Punto

Operación

Describe el comportamiento de los objetos de una clase, es un servicio que puede ser requerido por un objeto. La sintaxis de una operación es:
<visibilidadnombre(lista-de-parámetros)>:<expresión-de-tipo-retorno{string-de-propiedades}>

La visibilidad, la expresión de tipo de retorno y el *string* de propiedades son opcionales.
La lista de parámetros es una lista que contiene los parámetros formales de la operación separados por comas; por cada parámetro se tiene su nombre y la expresión de tipo asociada.

Ejemplo: en la Figura IV-3, las operaciones de la clase Rectángulo son:

Rectángulo(p1: Punto,p2: Punto)
area(): Real
aspecto(): Real
mover(delta: Punto)
escala(radio: Punto)

Variaciones de Clases

Las variaciones de una clase pueden ser clases parametrizadas, utilitarias o metaclases:

- Una clase parametrizada *(template)* define una familia de clases, en donde cada clase es instanciada especificando el enlace de los parámetros a los valores actuales. Usualmente los parámetros son nombres de clases, pero también pueden ser de tipos.
- Una clase utilitaria (*utility*) es una agrupación de variables y procedimientos globales en la forma de una declaración de clase (esta construcción corresponde a una convención de programación). Generalmente se identifica con el estereotipo *<<utility>>* ver Figura IV-4.
- Una metaclase es una clase cuyas instancias son clases, identificadas con el estereotipo *<<metaclase>>*.

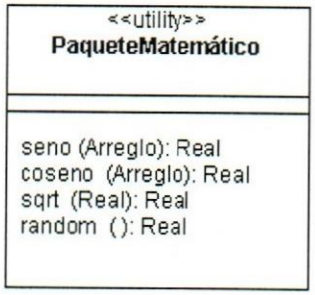


Figura IV-4: Notación UML para una clase *utility*.

Objetos

Un objeto es una entidad con identidad bien definida que encapsula estado y comportamiento. El estado es representado por atributos y relaciones, el comportamiento es representado por operaciones y métodos.

Un objeto es una instancia de una clase. Su representación es similar a la de la clase pero subrayando el string de identificación del objeto. En la Figura IV-5 se muestran diferentes notaciones para un objeto.

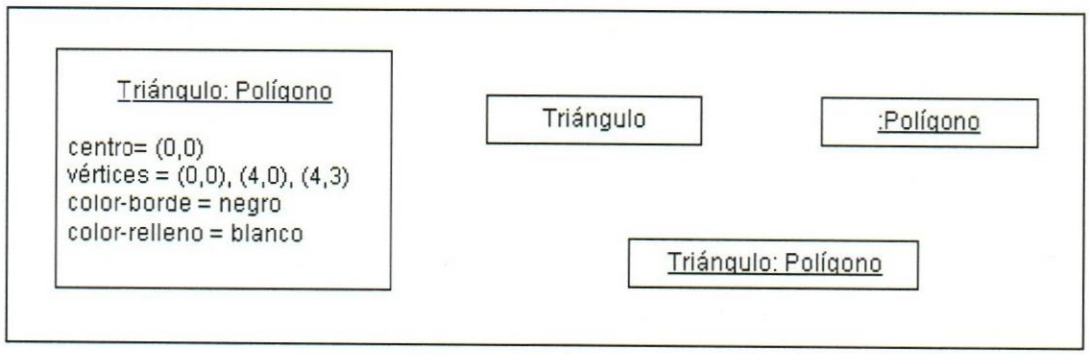


Figura IV-5: Objetos

Tipos e Intefaces

Un tipo es un descriptor para objetos con estados abstractos y especificación de operaciones, mientras que una clase es un descriptor para objetos con estados concretos e implementación de las operaciones (métodos). De esta manera, un tipo establece una especificación del comportamiento externo de una clase; una clase que soporta las operaciones definidas por un tipo, se dice que implementa al tipo. Esta relación puede ser considerada como una relación de refinamiento desde la clase al tipo que ésta implementa.

Decir que una clase implementa el comportamiento especificado por un tipo significa que la clase provee una representación concreta para el tipo (estructuras de datos) y la implementación procedural de las operaciones.
Un tipo puede contener atributos y operaciones. Los atributos en un tipo definen el estado abstracto del tipo. Las operaciones del tipo se definen mediante una especificación (nombre de la operación, parámetros y tipos asociados) y, opcionalmente, una descripción del efecto. UML posee una lista de tipos predefinidos (*integer*, *boolean*, etc.) pero evita especificar las reglas para definir las expresiones de tipo que aparecen en la declaración de atributos, variables y parámetros; supone que el usuario las define mediante *strings* de expresiones de tipo con una sintaxis orientada al lenguaje de programación.

Un tipo es, por lo tanto, una interfaz y, en general, se le usa como sinónimo. Una interfaz se puede definir como un tipo, que describe el comportamiento visible al exterior de diferentes entidades.

Una interfaz permite especificar las operaciones visibles externamente de una clase, componente o subsistema, sin indicar la estructura interna.

IV.4 RELACIONES ENTRE LOS ELEMENTOS DEL MODELO

Una relación es una conexión semántica entre elementos del modelo. En UML se definen relaciones de asociación, generalización y dependencia. Las relaciones de asociación, a su vez, se dividen en binarias, de agregación y de composición.

Una relación de asociación es una conexión semántica bidireccional entre elementos del modelo, esta relación es mostrada mediante enlaces representados por líneas sólidas que conectan los elementos y que son enriquecidas con una variedad de adornos que indican sus propiedades, estas relaciones de asociación pueden ser binarias, de agregación o de composición.

Relación Binaria

Si la relación se establece entre dos clases se llama asociación binaria y es representada por una línea que une las dos clases, tal y como se muestra en la Figura IV-6, donde cada extremo de la asociación determina un rol, el cual indica el comportamiento de una clase en la asociación (el rol es parte de la asociación no de la clase).

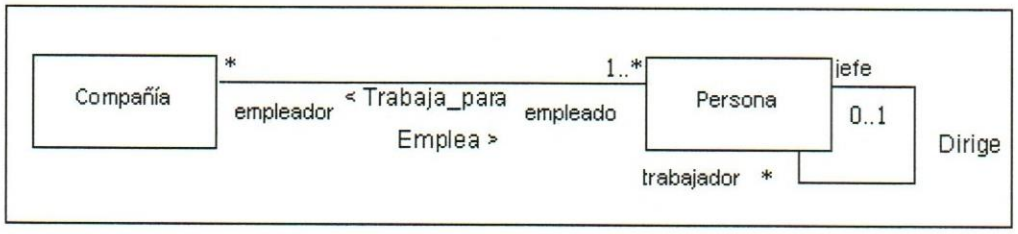


Figura IV-6: Ejemplo de asociación binaria

Una asociación puede tener los siguientes elementos, que permiten agregar información al modelo:

- El nombre de la asociación, con un símbolo de orientación ("<" ó ">") para facilidad de la lectura. En la figura IV-6, el nombre de la asociación es "Trabaja para" cuando la relación es de Persona a Compañía, luego una persona trabaja para una compañía. La otra relación es "Emplea", ésta ocurre cuando la relación es de Compañía a Persona, luego una compañía emplea a personas.
- Un estereotipo. En la figura IV-6 no hay ningún estereotipo, pero un ejemplo claro de estereotipo se puede encontrar en el capítulo III, sección III.4.3.

Un rol puede tener las siguientes propiedades:

- El nombre del rol, vinculado al extremo de la asociación. En la figura IV-6, los nombres de roles involucrados son del lado de la Compañía el rol es "empleador" y del lado de Persona es "empleado".
- Multiplicidad, la cual especifica el rango de la cardinalidad permitida (un intervalo con el formato: límite-inferior..límite-superior; un entero; el símbolo * que denota un número ilimitado de elementos, ...). En la figura IV-6 la multiplicidad es * en el lado de la Compañía y de 1..* del lado de la clase Persona, lo que significa que una persona en el tiempo trabaja en muchas compañías y que en una compañía trabajan 1 o más personas.
- Ordenamiento, si la multiplicidad es mayor que 1, entonces el conjunto de elementos relacionados puede estar ordenado en cuyo caso se especifican con la restricción "{ordenado}".
- Calificador, el cual se define como un atributo o tupla de atributos cuyos valores sirven para particionar el conjunto de objetos asociados con un objeto a través de una asociación. El calificador reduce la multiplicidad de una asociación. Un calificador se dibuja como un pequeño rectángulo en el final de la asociación, atado a una clase particular; de modo que es parte de la asociación y no de la clase.

Relación de Agregación

La relación de agregación es una forma especial de asociación que especifica una relación todo-parte entre el agregado (el todo) y sus componentes (las partes). Es una relación del tipo "es parte de".

La agregación es denotada con un diamante vacío vinculado a la clase que representa el agregado, (véase Figura IV-7). En la figura IV-7 se tiene la clase Polígono y la clase Punto, y la relación de agregación entre ellas, es "contiene", de aquí se deduce que un polígono es una agregación de puntos.

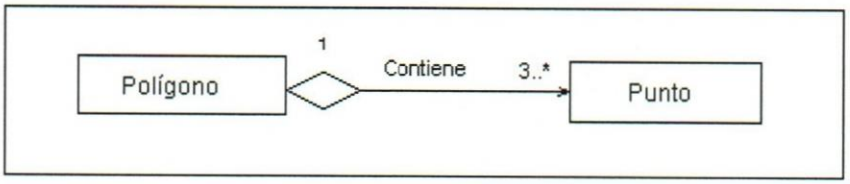


Figura IV-7: Relación de agregación

Relación de Composición

La composición es una relación más fuerte que la agregación en donde "el todo" y "las partes" coinciden en su tiempo de vida, lo que implica lo siguiente:

- Dependencia existencial. El elemento dependiente desaparece al destruirse el que lo contiene y, si la cardinalidad es 1, es creado al mismo tiempo.
- Hay una pertenencia fuerte. Se puede decir que el objeto contenido es parte constitutiva y vital del que lo contiene.
- Los objetos contenidos no son compartidos, esto es, no hacen parte del estado de otro objeto.

Se denota igual que la relación de agregación pero con un diamante relleno vinculado a la clase que representa el "todo", en lugar de uno vacío. Existen formas alternativas para mostrar la composición. UML provee una forma gráfica anidada, la cual puede ser conveniente para mostrar la composición en algunos casos, como puede observarse en la Figura IV-8, los atributos constituyen una relación de composición entre una clase (tipo) y las clases (tipos) de los atributos. En las relaciones de composición, el nexo es tan fuerte que es evidente que esa relación debe existir, en el ejemplo que se observa en la Figura IV-8, una ventana está compuesta de cursor, encabezado y panel, en este caso no se podría colocar una relación de agregación por que entonces, se estaría permitiendo que la clase ventana pueda prescindir de cursor, encabezado y panel.

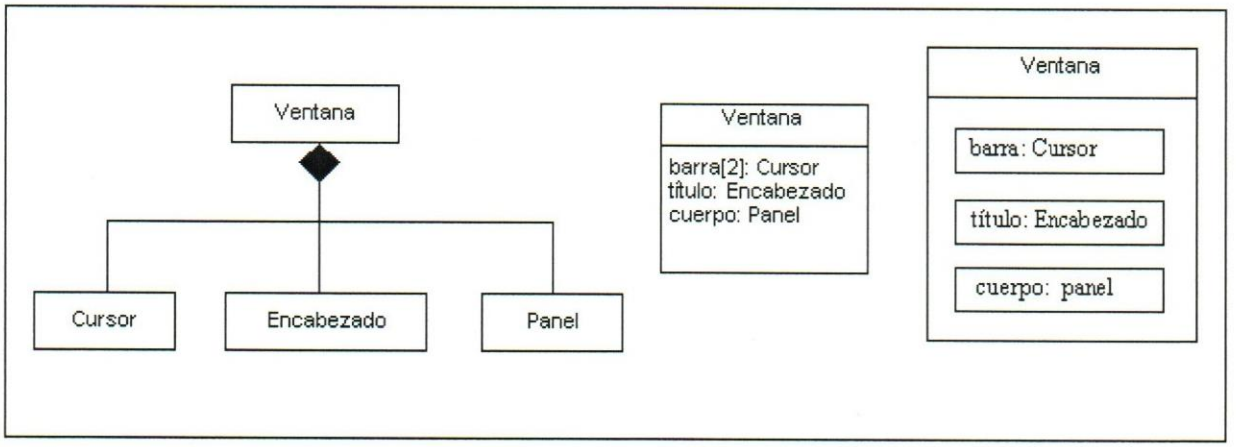


Figura IV-8: Diversas formas para denotar la relación de composición en UML

Relaciones de Generalización

La generalización es una relación de herencia entre clases. La subclase hereda todos los atributos y mensajes descritos en la superclase.

La generalización se denota como una línea sólida que va desde el elemento más específico (subclase) al elemento más general (superclase), con un triángulo no relleno en el extremo anexo al elemento más general.

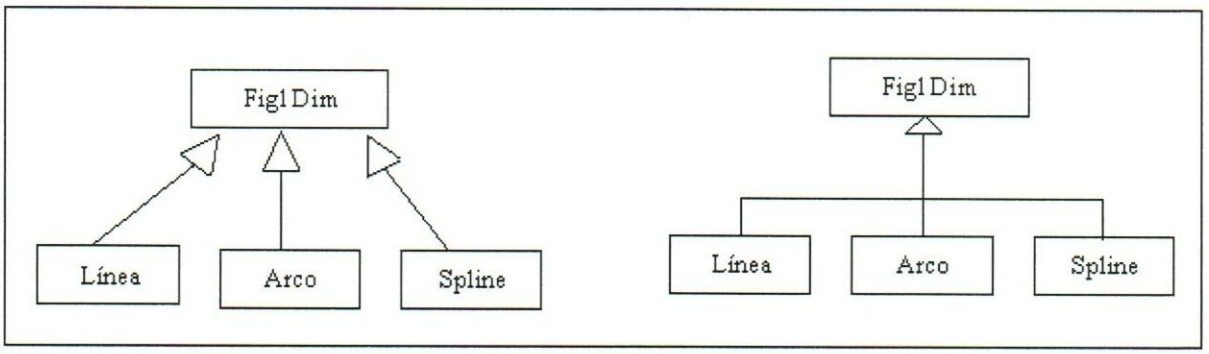


Figura IV-9: Diversas formas para denotar la relación de generalización en UML

La herencia es el mecanismo mediante el cual los elementos más específicos incorporan estructura y comportamientos desde el elemento más general. La generalización especifica una relación de herencia unidireccional y se aplica a clases, a tipos y a paquetes. Esta relación puede ser de los siguientes tipos:

- *Disjunta:* significa que cualquier clase descendiente de las subclases bajo la restricción, puede descender sólo de una de estas subclases.
- *Solapada:* significa que puede existir una clase que descienda de más de una de las subclases afectadas por la restricción.
- *Completa:* significa que todas las subclases han sido especificadas (hayan sido o no mostradas).
- *Incompleta:* significa que algunas subclases han sido especificadas y otras no.

Relaciones de Dependencia

Una dependencia indica una relación semántica entre dos (ó más) elementos del modelo, en el cual un cambio en el elemento destino requiere un cambio en el elemento fuente. Se denota con una flecha punteada que va del elemento fuente al elemento destino. Puede tener por medio de estereotipos una explicación del tipo de dependencia presentada, como se puede observar en la Figura IV-10.

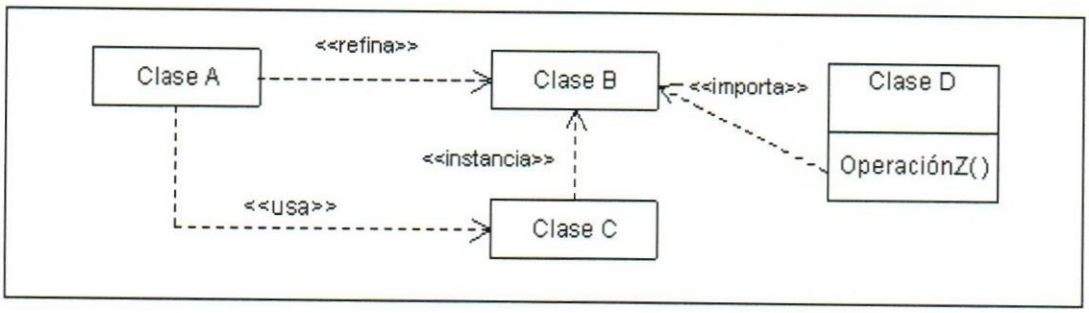


Figura IV-10: Ejemplo de la relación de dependencias entre clases

Una relación de dependencia particular es la relación de refinamiento ya que representa la especificación de algo que ya ha sido especificado en un mayor nivel de abstracción. Esta relación se establece para mostrar distintos niveles de abstracción de un mismo elemento; por ejemplo, la relación entre un tipo y la clase que lo implementa; entre una clase del análisis y una clase del diseño, etc. UML define los estereotipos *<<refina>>* e *<<implementa>>* para expresar tipos de refinamiento.

Capítulo V: METODOLOGÍA PARA LA ELICITACIÓN DE REQUISITOS DEL SISTEMA DE SOFTWARE

Como bien se estudio en el capitulo I, la Ingeniería de Requisitos, es una sub-área de la ingeniería del software que se encarga de estudiar los procesos de definición de los requisitos que tendrá el software a desarrollar; la misma consta de las siguientes etapas:

- Elicitación de Requisitos
- Análisis de Requisitos
- Especificación de Requisitos
- Validación y Certificación de los Requisitos

El objetivo de esta metodología [MERSS 2002] es la definición de las tareas a realizar, los productos a obtener y las técnicas a emplear durante la actividad de elicitación de requisitos de la fase de ingeniería de requisitos del desarrollo de software.

En esta metodología, versión 2.1, se distinguen dos tipos de productos o artefactos: los productos entregables y los productos no entregables o internos. Los productos entregables son aquellos que se entregan oficialmente al cliente como parte del desarrollo en fechas previamente acordadas, mientras que los no entregables son productos internos al desarrollo que no se entregan al cliente.

V.1. TAREAS RECOMENDADAS

Las tareas recomendadas para obtener los productos descritos en esta metodología son las siguientes:

Tarea 1: Obtener información sobre el dominio del problema y el sistema actual.
Tarea 2: Preparar y realizar las reuniones de elicitación/negociación.
Tarea 3: Identificar/revisar los objetivos del sistema.
Tarea 4: Identificar/revisar los requisitos de almacenamiento de información.
Tarea 5: Identificar/revisar los requisitos funcionales.
Tarea 6: Identificar/revisar los requisitos no funcionales.
Tarea 7: Priorizar objetivos y requisitos.

El orden recomendado de realización para estas tareas es: 1. . . 7, aunque las tareas 4, 5, y 6 pueden realizarse simultáneamente o en cualquier orden que se considere oportuno (ver figura IV-1). La tarea 1 es opcional y depende del conocimiento previo que tenga el equipo de desarrollo sobre el dominio del problema y el sistema actual.

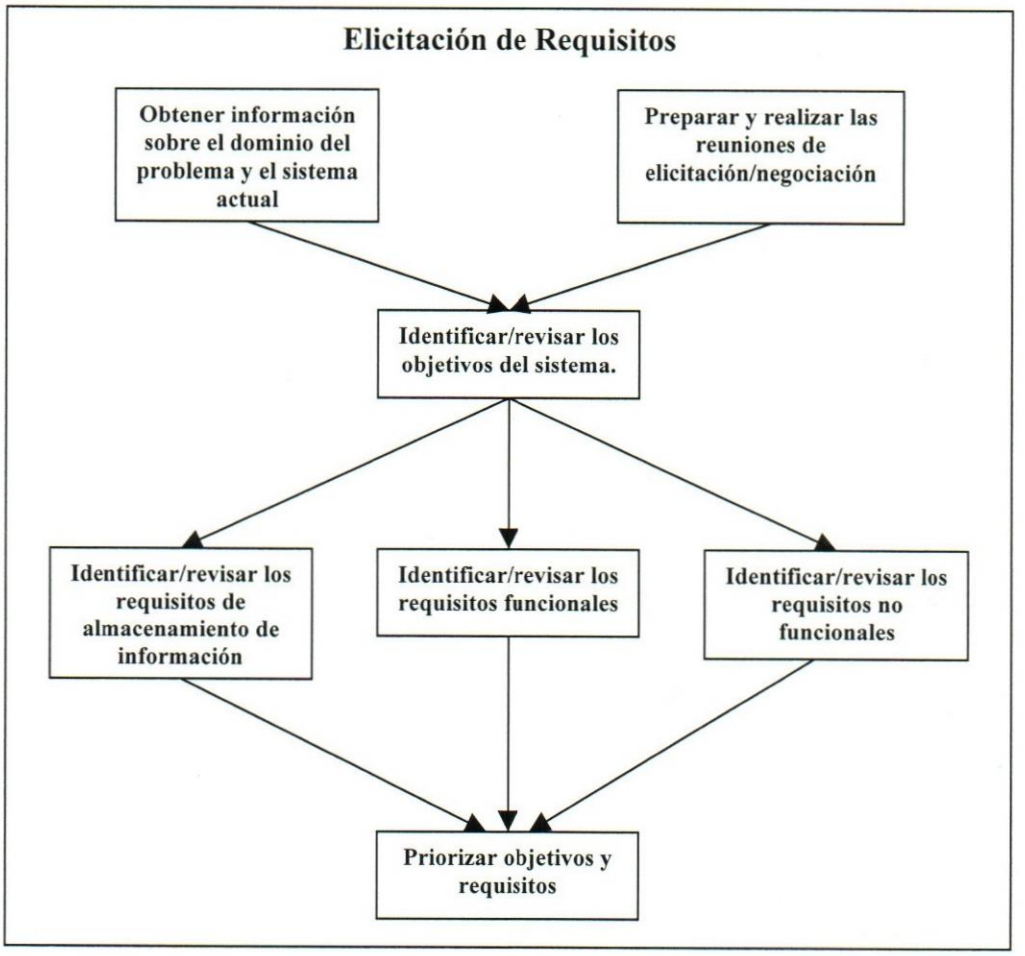


Figura V-1: Tareas de elicitación de requisitos

A continuación se describen cada una de las tareas mencionadas.

V.2 OBTENER INFORMACIÓN SOBRE EL DOMINIO DEL PROBLEMA Y EL SISTEMA ACTUAL

V.2.1 Objetivos

- Conocer el dominio del problema.
- Conocer la situación actual.

V.2.2 Descripción

Antes de mantener las reuniones con los clientes y usuarios e identificar los requisitos es fundamental conocer el dominio del problema y los contextos organizacional y operacional, es decir, la situación actual.

Enfrentarse a un desarrollo sin conocer las características principales ni el vocabulario propio de su dominio suele provocar que el producto final no sea el esperado por clientes ni usuarios. Por otro lado, mantener reuniones con clientes y usuarios sin conocer las características de su actividad hará que probablemente no se entiendan sus necesidades y que su confianza inicial hacia el desarrollo se vea deteriorada enormemente.

Esta tarea es opcional, ya que puede que no sea necesario realizarla si el equipo de desarrollo tiene experiencia en el dominio del problema y el sistema actual es conocido.

V.2.3 Productos internos

- Información recopilada: libros, artículos, folletos comerciales, desarrollos previos sobre el mismo dominio, etc.
- Modelos del sistema actual.

V.2.4 Productos entregables

Introducción, participantes en el proyecto, principalmente clientes y desarrolladores, y descripción del sistema actual como parte del ***D**ocumento de **R**equisitos del **S**istema* (DRS).

V.2.5 Técnicas recomendadas

- Obtener información de fuentes externas al negocio del cliente: folletos, informes sobre el sector, publicaciones, consultas con expertos, etc.
- En el caso de que se trate de un dominio muy específico puede ser necesario recurrir a fuentes internas al propio negocio del cliente, en cuyo caso pueden utilizarse las técnicas auxiliares de elicitación de requisitos como el estudio de documentación, observación *in situ*, cuestionarios, inmersión o aprendizaje, etc.
- Modelado del sistema actual (UML).

V.3 Preparar y realizar las reuniones de elicitación/negociación

V.3.1 Objetivos

- Identificar a los usuarios participantes.
- Conocer las necesidades de clientes y usuarios.
- Resolver posibles conflictos.

V.3.2 Descripción

Teniendo en cuenta la información recopilada en la tarea anterior, en esta tarea se deben preparar y realizar las reuniones con los clientes y usuarios participantes con objeto de obtener sus necesidades y resolver posibles conflictos que se hayan detectado en iteraciones previas del proceso.

Esta tarea es especialmente crítica y ha de realizarse con especial cuidado, ya que generalmente el equipo de desarrollo no conoce los detalles específicos de la organización para la que se va a desarrollar el sistema y, por otra parte, los clientes y posibles usuarios no saben qué necesita saber el equipo de desarrollo para llevar a cabo su labor.

V.3.3 Productos internos

- Notas tomadas durante las reuniones, transcripciones o actas de reuniones, formularios, grabaciones en cinta o vídeo de las reuniones o cualquier otra documentación que se considere oportuna.

V.3.4 Productos entregables

- Participantes en el proyecto, en concreto los usuarios participantes, como parte del DRS.
- Objetivos, requisitos o conflictos, que se hayan identificado claramente durante las sesiones de elicitación, como parte del DRS.

V.3.5 Técnicas recomendadas

- Técnicas de elicitación de requisitos (ver las plantillas de objetivos, requisitos y conflictos descritas más adelante, que pueden usarse directamente durante las sesiones de elicitación.
- Técnicas de negociación como WinWin [Boehm 1994].

V.4 IDENTIFICAR/REVISAR LOS OBJETIVOS DEL SISTEMA

V.4.1 Objetivos

- Identificar los objetivos que se esperan alcanzar mediante el sistema de software a desarrollar.
- Revisar, en el caso de que haya conflictos, los objetivos previamente identificados.

V.4.2 Descripción

A partir de la información obtenida en la tarea anterior, en esta tarea se deben identificar qué objetivos se esperan alcanzar una vez que el sistema de software a desarrollar se encuentre en explotación o revisarlos en función de los conflictos identificados. Puede que los objetivos hayan sido proporcionados antes de comenzar el desarrollo.

V.4.3 Productos internos

- No hay productos internos en esta tarea.

V.4.4 Productos entregables

- Objetivos del sistema como parte del DRS.

V.4.5 Técnicas recomendadas

- Análisis de factores críticos de éxito o alguna técnica similar de identificación de objetivos.
- Plantilla para especificar los objetivos del sistema.

V.5 IDENTIFICAR/REVISAR LOS REQUISITOS FUNCIONALES

V.5.1 Objetivos

- Identificar los actores del sistema del sistema de software a desarrollar.
- Identificar los requisitos funcionales (casos de uso) que deberá cumplir el sistema de software a desarrollar.
- Revisar, en el caso de que haya conflictos, los requisitos funcionales previamente identificados.

V.5.2 Descripción

A partir de la información obtenida en las tareas 1 y 2, y teniendo en cuenta los objetivos identificados en la tarea 3 y el resto de los requisitos, en esta tarea se debe identificar, o revisar si existen conflictos, qué debe hacer el sistema a desarrollar con la información identificada en la tarea anterior.

Inicialmente se identificarán los actores que interactuarán con el sistema, es decir aquellas personas u otros sistemas que serán los orígenes o destinos de la información que consumirá o producirá el sistema a desarrollar y que forman su entorno.
A continuación se identificarán los casos de uso asociados a los actores, los pasos de cada caso de uso y posteriormente se detallarán los casos de uso con las posibles excepciones hasta definir todas las situaciones posibles.

V.5.3 Productos internos

- No hay productos internos en esta tarea.

V.5.4 Productos entregables

- Requisitos funcionales como parte del DRS.

V.5.5 Técnicas recomendadas

- Casos de uso.
- Plantilla para actores.
- Plantilla para los requisitos funcionales.

V.6 IDENTIFICAR/REVISAR LOS REQUISITOS NO FUNCIONALES

V.6.1 Objetivos

- Identificar los requisitos no funcionales del sistema de software a desarrollar.

V.6.2 Descripción

A partir de la información obtenida en las tareas 1 y 2, y teniendo en cuenta los objetivos identificados en la tarea 3 y el resto de los requisitos, en esta tarea se deben identificar, o revisar si existen conflictos, los requisitos no funcionales, normalmente de carácter técnico o legal.

Algunos tipos de requisitos que se suelen incluir en esta sección son los siguientes:

Requisitos de comunicaciones del sistema
Son requisitos de carácter técnico relativos a las comunicaciones que deberá soportar el sistema de software a desarrollar. Por ejemplo: el sistema deberá utilizar el protocolo TCP/IP para las comunicaciones con otros sistemas.

Requisitos de interfaz de usuario
Este tipo de requisitos especifica las características que deberá tener el sistema en su comunicación con el usuario. Por ejemplo: la interfaz de usuario del sistema deberá ser consistente con los estándares definidos en IBM's Common User Access.
Se debe ser cuidadoso con este tipo de requisitos, ya que en esta fase de desarrollo todavía no se conocen bien las dificultades que pueden surgir a la hora de diseñar e implementar las interfaces, por esto no es conveniente entrar en detalles demasiado específicos.

Requisitos de fiabilidad
Los requisitos de fiabilidad deben establecer los factores que se requieren para la fiabilidad del software en tiempo de explotación. La fiabilidad mide la probabilidad del sistema de producir una respuesta satisfactoria a las demandas del usuario. Por ejemplo: la tasa de fallos del sistema no podrá ser superior a 2 fallos por semana.

Requisitos de entorno de desarrollo
Este tipo de requisitos especifican si el sistema debe desarrollarse con un producto específico. Por ejemplo: el sistema deberá desarrollarse con Oracle 7 como servidor y clientes Visual Basic 6.0.

Requisitos de portabilidad

Los requisitos de portabilidad definen qué características deberá tener el software para que sea fácil utilizarlo en otra máquina o bajo otro sistema operativo. Por ejemplo: el sistema deberá funcionar en los sistemas operativos Windows 95, Windows 98 y Windows NT 4.0, siendo además posible el acceso al sistema a través de Internet usando cualquier navegador compatible con HTML 3.0.

V.6.3 Productos internos

- No hay productos internos en esta tarea.

V.6.4 Productos entregables

- Requisitos no funcionales del sistema como parte del DRS.

V.6.5 Técnicas recomendadas

- Plantilla para requisitos no funcionales.

V.7 PRODUCTOS ENTREGABLES

El único producto entregable que se contempla en esta metodología es el ***Documento de Requisitos del Sistema*** (DRS). La estructura del DRS puede verse en la figura IV-2.

Portada
Lista de cambios
Índice
Lista de figuras
Lista de tablas
1 Introducción
2 Participantes en el proyecto
3 Descripción del sistema actual *[opcional]*
4 Objetivos del sistema
5 Catálogo de requisitos del sistema
5.1 Requisitos de almacenamiento de información
5.2 Requisitos funcionales
5.2.1 Diagramas de casos de uso
5.2.2 Definición de actores
5.2.3 Casos de uso del sistema
5.3 Requisitos no funcionales
6 Matriz de rastreabilidad objetivos/requisitos
7 Conflictos pendientes de resolución *[opcional, pueden ir en un documento aparte]*
8 Glosario de términos *[opcional]*
Apéndices *[opcionales]*

Figura V-2: Estructura del Documento de Requisitos del Sistema

En las siguientes secciones se describe con detalle cada sección del DRS.

V.7.1 Portada

La portada del DRS debe tener el formato que puede verse en la figura IV-3.
Los elementos que deben aparecer son los siguientes:
Nombre del proyecto: el nombre del proyecto al que pertenece el DRS.

Versión: la versión del DRS que se entrega al cliente. La versión se compone de dos números X e Y . El primero indica la versión, y se debe incrementar cada vez que se hace una nueva entrega formal al cliente. Cuando se incremente el primer número, el segundo debe volver a comenzar en cero. El segundo número indica cambios dentro de la misma versión aún no entregada, y se debe incrementar cada vez que se publica una versión con cambios respecto a la última que se publicó y que no se vaya a entregar formalmente todavía. Este tipo de versiones pueden ser internas al equipo de desarrollo o ser entregadas al cliente a título orientativo.

Fecha: fecha de la publicación de la versión.
Equipo de desarrollo: nombre de la empresa o equipo de desarrollo.
Cliente: nombre del cliente, normalmente otra empresa.

Proyecto *nombre del proyecto*

Documento de
Requisitos del Sistema

Versión *X:Y*
Fecha *fecha*

Realizado por *equipo de desarrollo*
Realizado para *cliente*

Figura V-3: Portada del Documento de Requisitos del Sistema

V.7.2 Lista de cambios

El documento debe incluir una lista de cambios en la que se especifiquen, para cada versión del documento, los cambios producidos en el mismo con un formato similar al que puede verse en la figura V-4. Para cada cambio realizado se debe incluir el número de orden, la fecha, una descripción y los autores.

Num	Fecha	Descripción	Autores
0	$fecha_0$	*Versión x.y*	$autor_0$
1	$fecha_1$	*Descripción* $cambio_1$	$autor_1$
...	...	...	...
n	$fecha_n$	*Descripción* $cambio_n$	$autor_n$

Figura V-4: Lista de cambios del Documento de Requisitos del Sistema

V.7.3 Índice

El índice del DRS debe indicar la página en la que comienza cada sección, subsección o apartado del documento. En la medida de lo posible, se sangrarán las entradas del índice para ayudar a comprender la estructura del documento.

V.7.4 Listas de figuras y tablas

El DRS deberá incluir listas de las figuras y tablas que aparezcan en el mismo. Dichas listas serán dos índices que indicarán el número, la descripción y la página en que aparece cada figura o tabla del DRS.

V.7.5 Introducción

Esta sección debe contener una descripción breve de las principales características del sistema de software que se va a desarrollar, la situación actual que genera la necesidad del nuevo desarrollo, la problemática que se acomete, y cualquier otra consideración que sitúe al posible lector en el contexto oportuno para comprender el resto del documento.

V.7.6 Participantes en el proyecto

Esta sección debe contener una lista con todos los participantes en el proyecto, tanto desarrolladores como clientes y usuarios. Para cada participante se deberá indicar su nombre, el papel que desempeña en el proyecto, la organización a la que pertenece y cualquier otra información adicional que se considere oportuna.

V.7.7 Descripción del sistema actual

Esta sección debe contener una descripción del sistema actual en el caso de que se haya acometido su estudio. Para describir el sistema actual puede utilizarse cualquier técnica que se considere oportuno, por ejemplo el diagrama de actividades en UML.

V.7.8 Objetivos del sistema

Esta sección debe contener una lista con los objetivos que se esperan alcanzar cuando el sistema de software a desarrollar esté en explotación, especificado mediante la plantilla para objetivos que se describe mas adelante.

V.7.9 Catálogo de requisitos del sistema

Esta sección se divide en las siguientes subsecciones en las que se describen los requisitos del sistema. Cada uno de los grandes grupos de requisitos, de almacenamiento de información, funcionales y no funcionales, podrá dividirse para ayudar a la legibilidad del documento, por ejemplo dividiendo cada subsección en requisitos asociados a un determinado objetivo, requisitos con características comunes, etc.

V.7.10 Requisitos de almacenamiento de información

Esta subsección debe contener la lista de requisitos de almacenamiento de información que se hayan identificado, utilizando para especificarlos la plantilla para requisitos de almacenamiento de información descrita mas adelante.

V.7.11 Requisitos funcionales

Esta subsección debe contener la lista de requisitos funcionales que se hayan identificado, dividiéndose en los siguientes apartados que se describen a continuación:

Diagrama de casos de uso: Este apartado debe contener los diagramas de casos de uso del sistema que se hayan realizado.

Definición de los actores: Este apartado debe contener una lista con los actores que se hayan identificado, especificados mediante la plantilla para actores de casos de uso.

Casos de uso del sistema: Este apartado debe contener los casos de uso que se hayan identificado, especificados mediante la plantilla para requisitos funcionales.

V.7.12 Requisitos no funcionales

Esta subsección debe contener la lista los requisitos no funcionales del sistema que se hayan identificado, especificados mediante la plantilla para requisitos no funcionales, que se explica en la sección V.9.4.

V.7.13 Matriz de rastreabilidad objetivos/requisitos

Esta sección debe contener una matriz objetivo–requisito, de forma que para cada objetivo se pueda conocer con qué requisitos está asociado. El formato de la matriz de rastreabilidad puede verse en la figura V-5.

	OBJ-01	OBJ-02	. . .	OBJ-n
RI–01	♦	♦		
RI–02		♦		
. . .				
RF01				
RF02	♦	♦		
. . .				
RNF-01				♦
RNF-02		♦		
. . .				

Figura V-5: Matriz de rastreabilidad del Documento de Requisitos del Sistema

V.7.14 Conflictos pendientes de resolución

Esta sección, que se incluirá en el caso de que no se opte por registrar los conflictos en un documento aparte, deberá contener los conflictos identificados durante el proceso y que aún están pendientes de resolución, descritos mediante la plantilla para conflictos.

V.7.15 Glosario de términos

Esta sección, que se incluirá si se considera oportuno, deberá contener una lista ordenada alfabéticamente de los términos específicos del dominio del problema, acrónimos y

abreviaturas que aparezcan en el documento y que se considere que su significado deba ser aclarado. Cada término deberá acompañarse de su significado.

V.7.16 Apéndices

Los apéndices se usarán para proporcionar información adicional a la documentación obligatoria del documento. Sólo deben aparecer si se consideran oportunos y se identificarán con letras ordenadas alfabéticamente: A, B, C, etc.

V.8 Técnicas

A continuación, se describen algunas de las técnicas que se proponen en esta metodología para obtener los productos de las tareas que se han descrito.

Las técnicas más habituales en la elicitación de requisitos son las entrevistas, el *Joint Application Development* (JAD) o Desarrollo Conjunto de Aplicaciones, el *brainstorming* o tormenta de ideas y la utilización de escenarios, más conocidos como casos de uso [Jacobson 1993, Booch 1999].

A estas técnicas, que se describen en los siguientes apartados, se las suele apoyar con otras técnicas complementarias como la observación in situ, el estudio de documentación, los cuestionarios, la inmersión en el negocio del cliente [Goguen y Linde 1993] o haciendo que los ingenieros de requisitos sean aprendices del cliente [Beyer y Holtzblatt 1995].

V.8.1 Entrevistas

Las entrevistas son la técnica de elicitación más utilizada, y de hecho son prácticamente inevitables en cualquier desarrollo ya que son una de las formas de comunicación más naturales entre personas.

En las entrevistas se pueden identificar tres fases: preparación, realización y análisis [Piattini et al. 1996]. En la primera fase refiere a que las entrevistas no deben improvisarse, por lo que conviene estudiar el dominio del problema, seleccionar las personas que se va a entrevistar y determinar el objetivo y contenido de las entrevistas, es decir en pocas palabras se debe realizar una buena planificación para realizar la entrevista. En la segunda fase, durante la realización de la entrevista, se debe exponer una apertura de la entrevista, el desarrollo, en el cual se deben hacer preguntas abiertas, utilizar palabras apropiadas, mostrar interés en todo momento y la terminación. En la tercera, y última fase, es necesario leer las notas tomadas y pasarlas a limpio, reorganizar la información y contrastarlas con las otras entrevistas para determinar los aspectos a evaluar.

V.8.2 Joint Application Development

La técnica denominada JAD (Joint Application Development, Desarrollo Conjunto de Aplicaciones), desarrollada por IBM en 1977, es una alternativa a las entrevistas individuales que se desarrolla a lo largo de un conjunto de reuniones en grupo durante un

periodo de 2 a 4 días. En estas reuniones se ayuda a los clientes y usuarios a formular problemas y explorar posibles soluciones, involucrándolos y haciéndolos sentirse partícipes del desarrollo.

Esta técnica se basa en cuatro principios [Raghavan et al. 1994]: dinámica de grupo, el uso de ayudas visuales para mejorar la comunicación (diagramas, transparencias, multimedia, herramientas CASE, etc.), mantener un proceso organizado y racional y una filosofía de documentación WYSIWYG (What You See Is What You Get, lo que se ve es lo que se obtiene), por la que durante las reuniones se trabaja directamente sobre los documentos a generar.

El JAD tiene dos grandes pasos, el JAD/Plan cuyo objetivo es elicitar y especificar requisitos, y el JAD/Design, en el que se aborda el diseño del software. En este documento sólo se verá con detalle el primero de ellos.

Debido a las necesidades de organización que requiere y a que no suele adaptarse bien a los horarios de trabajo de los clientes y usuarios, esta técnica no suele emplearse con frecuencia, aunque cuando se aplica suele tener buenos resultados, especialmente para elicitar requisitos en el campo de los sistemas de información [Raghavan 1994].

En comparación con las entrevistas individuales, presenta las siguientes ventajas:

- Ahorra tiempo al evitar que las opiniones de los clientes se contrasten por separado.
- Todo el grupo, incluyendo los clientes y los futuros usuarios, revisa la documentación generada, no sólo los ingenieros de requisitos.
- Implica más a los clientes y usuarios en el desarrollo.

V.8.3 Brainstorming

El brainstorming o tormenta de ideas es una técnica de reuniones en grupo cuyo objetivo es la generación de ideas en un ambiente libre de críticas o juicios [Gause y Weinberg 1989, Raghavan 1994]. Las sesiones de brainstorming suelen estar formadas por un número de cuatro a diez participantes, uno de los cuales es el jefe de la sesión, encargado más de comenzar la sesión que de controlarla.

Como técnica de elicitación de requisitos, el brainstorming puede ayudar a generar una gran variedad de vistas del problema y a formularlo de diferentes formas, sobre todo al comienzo del proceso de elicitación, cuando los requisitos son todavía muy difusos.

Frente al JAD, el brainstorming tiene la ventaja de que es muy fácil de aprender y requiere poca organización, de hecho, hay propuestas de realización de brainstorming por vídeo–conferencia a través de Internet [Raghavan1994]. Por otro lado, al ser un proceso poco estructurado, puede no producir resultados con la misma calidad o nivel de detalle que otras técnicas.

V.9 PLANTILLAS Y PATRONES LINGÜÍSTICOS PARA ELICITACIÓN DE REQUISITOS

Las plantillas y patrones lingüísticos que se presentan en los siguientes apartados están pensados para utilizarse tanto durante las reuniones de elicitación con clientes y usuarios como para registrar y gestionar los requisitos.

Su objetivo es doble: por un lado intentar paliar la falta de propuestas concretas sobre la expresión de requisitos. Por otro lado, también pueden usarse como elementos de elicitación y negociación durante las reuniones con clientes y usuarios de forma similar a las conocidas tarjetas CRC (Clase, Responsabilidad, Colaboración) [Wirfs-Brock et al. 1990] (véase figura V-6).

De esta forma se consigue que durante las sesiones de elicitación se trabaje con una filosofía WYSIWYG, tal como se propone en las técnicas de JAD o brainstorming, ya que los participantes manejan directamente la documentación final, favoreciéndose así su implicación en el proceso.

Como fruto de la experiencia de su utilización, para algunos campos de las plantillas se han identificado frases "estándar" que son habituales en las especificaciones de requisitos y que se han parametrizado. Estas frases, a las que hemos denominado patrones lingüísticos, o abreviadamente patrones–L, pueden usarse para rellenar los campos de las plantillas dándole valores a los parámetros con la información oportuna.

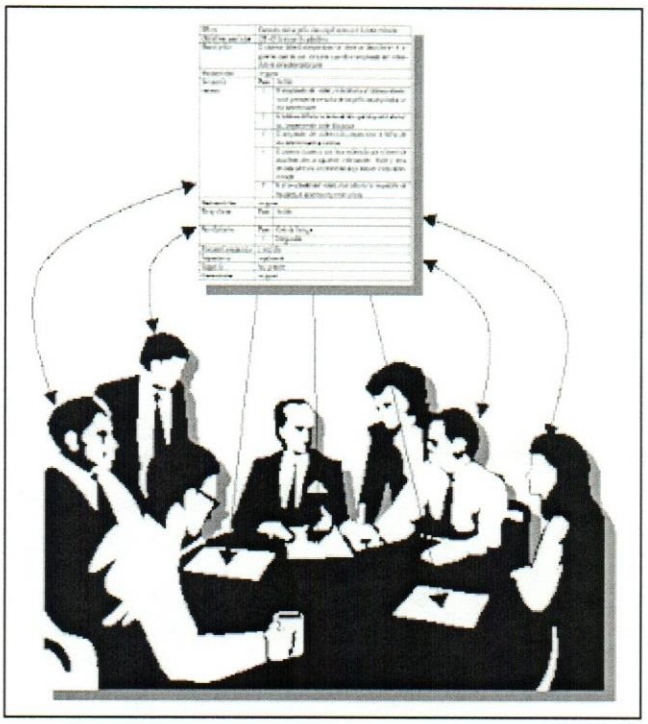


Figura V-6: La plantilla como elemento de elicitación y negociación

V.9.1 Plantilla para los objetivos del sistema

Los objetivos del sistema pueden considerarse como requisitos de alto nivel [Sawyer y Kontoya 1999], de forma que los requisitos propiamente dichos serían la forma de alcanzar los objetivos. La plantilla propuesta para los objetivos puede verse en la figura V-7.

OBJ–<*id*>	***<nombre descriptivo>***
Versión	*<nº de la versión actual>* (*<fecha de la versión actual>*)
Autores	♦ *<autor de la versión actual>* (*<organización del autor>*) . . .
Fuentes	*<fuente de la versión actual>* (*<organización de la fuente>*) . . .
Descripción	El sistema deberá *<objetivo a cumplir por el sistema>*
Subobjetivos	♦ *OBJ–x <nombre del subobjetivo>* . . .
Importancia	*<importancia del objetivo>*
Urgencia	*<urgencia del objetivo>*
Estado	*<estado del objetivo>*
Estabilidad	*<estabilidad del objetivo>*
Comentarios	*<comentarios adicionales sobre el objetivo>*

Figura V-7: Plantilla y patrones–L para objetivos

El significado de los campos que la componen, cuya mayoría está presente también en las plantillas para los requisitos, es el siguiente:

Identificador y nombre descriptivo: siguiendo la propuesta, entre otros, de [Sawyer et al. 1997], cada objetivo debe identificarse por un código único y un nombre descriptivo. Con objeto de conseguir una rápida identificación, los identificadores de los objetivos comienzan con OBJ.

Versión: para poder gestionar distintas versiones, este campo contiene el número y la fecha de la versión actual del objetivo.

Autores, Fuentes: estos campos contienen el nombre y la organización de los autores (normalmente desarrolladores) y de las fuentes (clientes o usuarios), de la versión actual del objetivo, de forma que la rastreabilidad pueda llegar hasta las personas que propusieron la necesidad del requisito.

Descripción: este campo contiene un patrón–L que se debe completar con la descripción del objetivo.

Subobjetivos: en este campo pueden indicarse los subobjetivos que dependen del objetivo que se está describiendo. En sistemas complejos puede ser necesario establecer una

jerarquía de objetivos previa a la identificación de los requisitos. En caso de que ésto no sea necesario, puede ignorarse este campo.

Importancia: este campo indica la importancia del cumplimiento del objetivo para los clientes y usuarios. Se puede asignar un valor numérico o alguna expresión enumerada como vital, importante o quedaría bien, tal como se propone en [IBM OOTC 1997]. En el caso de que no se haya establecido aún la importancia, se puede indicar que está por determinar (PD), equivalente al TBD (To Be Determined) empleado en las especificaciones escritas en inglés.

Urgencia: este campo indica la urgencia del cumplimiento del objetivo para los clientes y usuarios en el supuesto caso de un desarrollo incremental. Como en el caso anterior, se puede asignar un valor numérico o una expresión enumerada como inmediatamente, hay presión o puede esperar [IBM OOTC 1997], o PD en el caso de que aún no se haya determinado.

Estado: este campo indica el estado del objetivo desde el punto de vista de su desarrollo. El objetivo puede estar en construcción si se está elaborando, pendiente de negociación si tiene algún conflicto asociado pendiente de solución, pendiente de validación si no tiene ningún conflicto pendiente y está a la espera de validación o, por último, puede estar validado si ha sido validado por clientes y usuarios.

Estabilidad: este campo indica la estabilidad del objetivo, es decir una estimación de la probabilidad de que pueda sufrir cambios en el futuro. Esta estabilidad puede indicarse mediante un valor numérico o mediante una expresión enumerada como alta, media o baja o PD en el caso de que aún no se haya determinado.
La información sobre la estabilidad, bien a nivel de objetivos como en este caso, bien a nivel de requisitos, ayuda a los diseñadores a diseñar software que prevea de antemano la necesidad de posibles cambios futuros en aquellos aspectos relacionados con los elementos identificados como inestables durante la fase de ingeniería de requisitos, favoreciendo así el mantenimiento y la evolución del software [Brackett 1990].

Comentarios: cualquier otra información sobre el objetivo que no encaje en los campos anteriores puede recogerse en este apartado.

V.9.2 Plantilla para requisitos de almacenamiento de información

Lo más importante en los sistemas de información es precisamente la información que gestionan. La plantilla para requisitos de almacenamiento de información, que puede verse en la figura V-8, ayuda a los clientes y usuarios a responder a la pregunta "¿qué información, relevante para los objetivos de su negocio, debe ser almacenada por el sistema?".

RI–<*id*>	***<nombre descriptivo>***
Versión	<*no de la versión actual*> (<*fecha de la versión actual*>)
Autores	♦ <*autor de la versión actual*> (<*organización del autor*>) . . .
Fuentes	<*fuente de la versión actual*> (<*organización de la fuente*>) . . .
Objetivos asociados	♦ OBJ–*x* <*nombre del objetivo*> . . .
Requisitos asociados	♦ Rx–y <*nombre del requisito*> . . .
Descripción	El sistema deberá almacenar la información correspondiente a <*concepto relevante*>. En concreto:
Datos específicos	♦ <*datos específicos sobre el concepto relevante*> . . .
Intervalo temporal	{ pasado y presente, sólo presente }
Importancia	<*importancia del requisito*>
Urgencia	<*urgencia del requisito*>
Estado	<*estado del requisito*>
Estabilidad	<*estabilidad del requisito*>
Comentarios	<*comentarios adicionales sobre el objetivo*>

Figura V-8: Plantilla y patrones–L para requisitos de almacenamiento de información

El significado de los campos de la plantilla es el siguiente:

Identificador y nombre descriptivo: siguiendo las recomendaciones, entre otros, de [IEEE 1993] y [Sawyer et al. 1997], cada requisito se debe identificar por un código único y un nombre descriptivo. Con objeto de conseguir una rápida identificación, los identificadores de los requisitos de almacenamiento de información comienzan con RI.

Versión, Autores, Fuentes: estos campos tienen el mismo significado que en la plantilla para objetivos aunque referidos al requisito.

Objetivos asociados: este campo debe contener una lista con los objetivos a los que está asociado el requisito. Esto permite conocer qué requisitos harán que el sistema a desarrollar alcance los objetivos propuestos y justifican de esta forma la existencia o propósito del requisito.

Descripción: para los requisitos de almacenamiento de información este campo usa un patrón–L que se debe completar con el concepto relevante sobre el que se debe almacenar información.

Datos específicos: este campo contiene una lista de los datos específicos asociados al concepto relevante, de los que pueden indicarse todos aquellos aspectos que se considere oportunos (descripción, restricciones, ejemplos, etc.).

Intervalo temporal: este campo indica durante cuánto tiempo es relevante la información para el sistema. Puede tomar los valores pasado y presente, si la información es siempre relevante, o sólo presente si la información tiene un periodo de validez concreto. Por ejemplo, si el concepto es empleados, y el intervalo de tiempo es pasado y presente, quiere decir que los ex–empleados son relevantes para el sistema, mientras que un periodo de tiempo de sólo presente indicaría que los ex–empleados no se deben considerar.

Requisitos asociados: en este campo se indican otros requisitos que estén asociados por algún motivo con el requisito que se está describiendo, permitiendo así tener una rastreabilidad horizontal, similar a las relaciones entre *assets* del mismo nivel descritas en [García 2000].

Importancia, Urgencia, Estado, Estabilidad, Comentarios: estos campos tienen el mismo significado que en la plantilla para objetivos aunque referidos al requisito.

V.9.3 Plantilla para actores

Aunque, estrictamente hablando, los actores de los casos de uso no son requisitos, por homogeneidad con el estilo de definición del resto de los elementos que componen el catálogo de requisitos se ha descrito la plantilla para definirlos que puede verse en la figura V-9.

El único campo específico de esta plantilla es la descripción, en la que se usa un patrón–L que debe completarse con la descripción del rol o papel que representa el actor respecto al sistema. El significado del resto de los campos es el mismo que para las plantillas anteriores.

ACT–<*id*>	***<nombre descriptivo>***
Versión	*<nº de la versión actual>* (*<fecha de la versión actual>*)
Autores	♦ *<autor de la versión actual>* (*<organización del autor>*) . . .
Fuentes	*<fuente de la versión actual>* (*<organización de la fuente>*) . . .
Descripción	Este actor representa a <rol que representa el actor>
Comentarios	*<comentarios adicionales sobre el actor>*

Figura V-9: Plantilla y patrones–L para actores

V.9.4 Plantilla para requisitos funcionales

Los sistemas de información no sólo almacenan información, también deben proporcionar servicios usando la información que almacenan. La plantilla de requisitos funcionales, que puede verse en la figura V-10, describe casos de uso y ayuda a los clientes y usuarios a responder a la pregunta *"¿qué debe hacer el sistema con la información almacenada para alcanzar los objetivos de su negocio?"*.

RF–<*id*>	**<*nombre descriptivo*>**	
Versión	<*no de la versión actual*> (<*fecha de la versión actual*>)	
Autores	♦ <*autor de la versión actual*> (<*organización del autor*>) . . .	
Fuentes	<*fuente de la versión actual*> (<*organización de la fuente*>) . . .	
Objetivos asociados	♦ OBJ–*x* <*nombre del objetivo*> . . .	
Requisitos asociados	♦ Rx–y <*nombre del requisito*> . . .	
Descripción	El sistema deberá comportarse tal como se describe en el siguiente caso de uso { durante la realización de los casos de uso <*lista de casos de uso*>, cuando <*evento de activación*> }	
Datos específicos	♦ <*datos específicos sobre el concepto relevante*> . . .	
Secuencia Normal	**Paso**	**Acción**
	p1	{El actor <*actor*>, El sistema} <*acción/es realizada/s por actor/sistema*>
	p2	Se realiza el caso de uso <*caso de uso (RF–x)*>
	p3	Si <*condición*>, {el actor <*actor*>, el sistema} <*acción/es realizada/s por actor/sistema*>
	p4	Si <*condición*>, se realiza el caso de uso <*caso de uso (RF–x)*>
	. . .	. . .
Postcondición	<*postcondición del caso de uso*>	
Excepciones	**Paso**	**Acción**
	pi	Si <*condición de excepción*>, {el actor <*actor*>, el sistema} <*acción/es realizada/s por actor/sistema*>, a continuación este caso de uso {continúa, termina}
	pj	Si <*condición de excepción*>, se realiza el caso de uso <*caso de uso (RF–x)*>, a continuación este caso de uso {continúa, termina}
	. . .	. . .
Rendimiento	**Paso**	**Cota de tiempo**
	q	*m* <*unidad de tiempo*>
	. . .	. . .
Frecuencia esperada	<*no de veces*> veces / <*unidad de tiempo*>	
Importancia	<*importancia del requisito*>	
Urgencia	<*urgencia del requisito*>	
Estado	<*estado del requisito*>	
Estabilidad	<*estabilidad del requisito*>	
Comentarios	<*comentarios adicionales sobre el requisito*>	

Figura V-10: Plantilla y patrones–L para requisitos funcionales (casos de uso)

El significado de los campos específicos de esta plantilla es el siguiente (los campos comunes con la plantilla para requisitos de almacenamiento de información tienen el mismo significado):

Identificador y nombre descriptivo: igual que en la plantilla anterior, excepto que los identificadores de los requisitos funcionales empiezan con RF y que el nombre descriptivo suele coincidir con el objetivo que los actores esperan alcanzar al realizar el caso de uso.
No se debe confundir este objetivo con los objetivos del sistema. El objetivo que los actores esperan alcanzar al realizar un caso de uso es de más bajo nivel, por ejemplo registrar un nuevo socio o consultar los pedidos pendientes.

Descripción: para los requisitos funcionales, este campo contiene un patrón–L que debe completarse de forma distinta en función de que el caso de uso sea abstracto o concreto (léase capítulo III, diagramas de caso de uso).

Si el caso de uso es abstracto, deben indicarse los casos de uso en los que se debe realizar, es decir, aquellos desde los que es incluido o a los que extiende. Si, por el contrario, se trata de un caso de uso concreto, se debe indicar el evento de activación que provoca su realización.

En versiones anteriores de este patrón–L, aparecían las expresiones caso de uso abstracto y caso de uso concreto. La experiencia durante la utilización de estas plantillas en proyectos reales nos ha llevado a eliminar dichas expresiones, que resultaban difíciles de entender por los participantes en el proceso de elicitación.

Precondición: en este campo se expresan en lenguaje natural las condiciones necesarias para que se pueda realizar el caso de uso.

Secuencia normal: este campo contiene la secuencia normal de interacciones del caso de uso. En cada paso, un actor o el sistema realiza una o más acciones, o se realiza (se incluye) otro caso de uso. Un paso puede tener una condición de realización, en cuyo caso si se realizara otro caso de uso se tendría una relación de extensión. Se asume que, después de realizar el último paso, el caso de uso termina.

Otras propuestas similares, por ejemplo [Coleman 1998], proponen utilizar estructuras similares al pseudocódigo para expresar las interacciones de los casos de uso. En nuestra opinión, esto puede llevar a que dichas descripciones sean excesivamente complejas de entender para los participantes sin conocimientos de programación y se corre el peligro de especificar los casos de uso con un estilo cercano a la programación.

Postcondición: en este campo se expresan en lenguaje natural las condiciones que se deben cumplir después de la terminación normal del caso de uso.

Excepciones: este campo especifica el comportamiento del sistema en el caso de que se produzca alguna situación excepcional durante la realización de un paso determinado.

Después de realizar las acciones o el caso de uso asociados a la excepción (una extensión), el caso de uso puede continuar la secuencia normal o terminar, lo que suele ir acompañado por una cancelación de todas las acciones realizadas en el caso de uso dejando al sistema en el mismo estado que antes de comenzar el caso de uso, asumiendo una semántica transaccional.

Inicialmente, la expresión utilizada para indicar una terminación anormal del caso de uso como resultado de una excepción era "este caso de uso aborta". La experiencia durante su aplicación nos llevó a la conclusión de que el termino abortar resultaba emocionalmente molesto para algunos participantes [Goleman 1996], por lo que se cambió por "este caso de uso termina" con el significado comentado anteriormente.

Rendimiento: en este campo puede especificarse las restricciones de tipo tecnológicas, para que el sistema realice una funcionalidad. Por ejemplo, el número de terminales, el número de usuarios simultáneos que debe soportar el sistema, volumen y tipo de información, etc.

Frecuencia esperada: en este campo se indica la frecuencia esperada de realización del caso de uso, que aunque no es realmente un requisito, es una información interesante para los desarrolladores.

V.9.5 Plantilla para requisitos no funcionales

Los requisitos no funcionales del sistema se pueden expresar usando la plantilla que puede verse en la figura V-11. El único campo específico de esta plantilla es la descripción, en la que se usa un patrón–L que debe completarse con la capacidad que deberá presentar el sistema, el significado del resto de los campos es el mismo que para las plantillas anteriores.

RNF–<*id*>	***<nombre descriptivo>***
Versión	*<nº de la versión actual>* (*<fecha de la versión actual>*)
Autores	♦ *<autor de la versión actual>* (*<organización del autor>*) . . .
Fuentes	*<fuente de la versión actual>* (*<organización de la fuente>*) . . .
Objetivos asociados	♦ OBJ–*x* *<nombre del objetivo>* . . .
Requisitos asociados	♦ Rx–y *<nombre del requisito>* . . .
Descripción	El sistema deberá *<capacidad del sistema>*.
Importancia	*<importancia del requisito>*
Urgencia	*<urgencia del requisito>*
Estado	*<estado del requisito>*
Estabilidad	*<estabilidad del requisito>*
Comentarios	*<comentarios adicionales sobre el objetivo>*

Figura V-11: Plantilla y patrones–L para requisitos no funcionales

V.9.6 Plantilla para conflictos

Como ya se ha comentado, durante las sesiones de elicitación puede ser necesario resolver mediante algún tipo de negociación posibles conflictos en los requisitos–C elicitados en iteraciones previas del proceso. Para documentar dichos conflictos, y las soluciones adoptadas, se propone la plantilla que puede verse en la figura V-12.

CFL–<*id*>	***<nombre descriptivo>***
Versión	*<nº de la versión actual> (<fecha de la versión actual>)*
Autores	♦ *<autor de la versión actual>* (*<organización del autor>*) . . .
Fuentes	*<fuente de la versión actual>* (*<organización de la fuente>*) . . .
Objs./Reqs. en conflicto	♦ OBJ/Ryy-x *<nombre del objetivo o requisito en conflicto>* . . .
Descripción	*<descripción del conflicto>*.
Alternativas	♦ *<descripción alternativa de solución>* (*<autores alternativa>*) . . .
Solución	*<descripción de la solución adoptada (si se ha acordado)>*
Importancia	*<importancia de la resolución del conflicto>*
Urgencia	*<urgencia de la resolución del conflicto>*
Estado	*<estado del resolución del conflicto>*
Estabilidad	*<estabilidad del requisito>*
Comentarios	*<comentarios adicionales sobre el conflicto>*

Figura V-12: Plantilla para conflictos

El significado de los campos de la plantilla es el siguiente:

Identificador y nombre descriptivo: al igual que el resto de la información correspondiente a los requisitos–C, cada conflicto debe poderse identificar de forma única y tener un nombre descriptivo. El prefijo propuesto para lograr una rápida identificación es CFL.

Versión, Autores, Fuentes: estos campos tienen el mismo significado que en las plantillas para objetivos y requisitos, aunque referidos al conflicto. En este caso especial, las fuentes son los participantes que deben participar en las posibles negociaciones necesarias para su resolución.

Objetivos y requisitos en conflicto: este campo debe contener una lista con los objetivos y/o requisitos afectados por el conflicto.

Descripción: este campo debe contener la descripción del conflicto.

Alternativas: este campo debe contener una lista con las posibles alternativas de solución que se hayan identificado para solucionar el conflicto así como los autores de dicha alternativas.

Solución: este campo debe contener la descripción de la solución negociada del conflicto, una vez que se haya acordado.

Importancia, Urgencia: estos campos indican respectivamente la importancia y la urgencia de la resolución del conflicto.

Estado: este campo indica el estado de resolución del conflicto, que podrá estar en negociación o bien resuelto.

Comentarios: este campo tiene el mismo significado que en las plantillas descritas previamente.

Capítulo VI: CONCLUSIONES

La evolución de los estudios encarados por la Ingeniería de Requisitos se fue dando paulatinamente. Sin embargo, a partir de los años 90, los esfuerzos se concentraron en la búsqueda de técnicas, métodos y herramientas que pudieran ser aplicados durante el proceso de definición de requisitos para arribar a una etapa de diseño exitosa, dejando de lado la obtención de una metodología capaz de adaptarse a cualquier tipo de sistema y paradigma, brindando un marco de trabajo referencial, independiente del método a aplicar.

La Ingeniería de Requisitos resulta ser un campo muy activo dentro de la Informática, y en particular dentro de la Ingeniería del Software, y se dirige a unas actividades esenciales en el trabajo diario de las organizaciones del desarrollo de software. Se ha demostrado mediante varios estudios experimentales que la Ingeniería de Requisitos es crítica respecto al éxito o fracaso de numerosos proyectos informáticos, y su mala gestión tiene una gran incidencia probada en relación con el desbordamiento de costos o el incumplimiento de plazos de finalización.

Actualmente, en esta área de la informática, se están desarrollando gran cantidad de métodos, técnicas, herramientas, y estándares que en muchas ocasiones no son conocidos por parte de los profesionales del sector.

De la metodología para la elicitación de requisitos se puede destacar que:

- Contribuye al entendimiento de la Ingeniería de Requisitos detallando etapas bien definidas, con lo que disminuyen los problemas existentes tanto en lo que hace a la terminología como a las actividades por ella involucradas.

- Ofrece libertad de acción para realizar la selección e integración de las herramientas a emplear en cada una de las etapas.

- Contempla aspectos metodológicos de documentación tendientes a conseguir un Documento de Requisitos de clara interpretación.

- Por ser una metodología flexible, puede ser aplicada a diferentes paradigmas con el uso de métodos, técnicas y herramientas que se crean convenientes para cada etapa.

- Cubre las falencias en cuanto a la documentación orientada al usuario, al proponer documentos isomórficos (DR_T y DR_U), destacando además la importancia de la validación y certificación de los documentos de requisitos.

Capítulo VII: REFERENCIAS BIBLIOGRÁFICAS

[Beyer y Holtzblatt 1995] H. R. Beyer y K. Holtzblatt (1995). Apprenticing with the Customer. *Communications of the ACM*, 38(5).

[Bohem 1989] Bohem, B. and Ross, R. (1989). Theory-*W Software Project Management: Principles and Examples. IEEE Transactions on Software Engineering*.Vol. 15 N° 7.

[Booch 1994] Booch, G. (1994). *Object-Oriented Analysis and Design with Application* Benjamin/Cummings Company.

[Dorfman 1997] Dorfman M. y Thayer, R. (1997). *Software Engineering*, IEEE Computer Society Press, Los Alamitos, CA.

[García 2000] J. García, M. J. Ortín, B. Moros, y J. Nicolás (2000). *Modelado de Casos de Uso y Conceptual a partir del Modelado del Negocio. En Actas de las V Jornadas de Trabajo Menhir*, Granada.

[Gause y Weinberg 1989] D. C. Gause y G. M. (1989). Weinberg. *Exploring Requirements: Quality Before Design*. Dorset House.

[Goleman 1996] D. Goleman. *La Inteligencia Emocional*. Kairós, 1996.

[Goguen y Linde 1993] J. A. Goguen y C. Linde (1993). Techniques for Requirements Elicitation. En *Proceedings of the First International Symposium on Requirements Engineering*.

[Hooks 2000] Hooks, I. (2000). *Writing Good Requirements*. Fourth INCOSE Symposium.

[IBM OOTC 1997] IBM OOTC (1997). *Developing Object–Oriented Software*. IBM Object–Oriented Technology Center. Prentice–Hall.

[IEEE 1993] IEEE Recommended Practice for Software Requirements Specifications. IEEE/ANSI Standard 830–1993, Institute of Electrical and Electronics Engineers.

[IEEE 1994] Institute of Electrical and Electronics Engineers (1994). *IEEE Standard Glossary of Software Engineering Terminology. IEEE Standard 610*. 12-1990 (revision and redesignation of IEEE Std. 729-1983). New York.

[Jacobson 2000] Jacobson I. , Booch G. y Rumbaugh J. (2000). *El Proceso Unificado de Desarrollo de Software*. Editorial Addison Wesley.

[Jacobson 1993] I. Jacobson, M. Christerson, P. Jonsson, y G. Övergaard (1993). *Object–Oriented Software Engineering: A Use Case Driven Approach*. Addison–Wesley, 4a edición.

[Kontoya 1998] Kotonya, G. y Sommerville, I (1998). *Requirements Engineering. Processes and Techniques*, John Wiley & Sons.

[Leite 1987] Leite (1987). *A Survey on Requirements Analysis, Advancing Software Engineering Project Technical Report* RTP-071, University of California at Irvine, Department of Information and Computer Science.

[Loucopoulos 1989] Loucopoulos, P. y Champion, R.E.M (1989). *Knowledge-Based Support for Requirements Engineering, Information and Software Technology*. Vol.31, Num.3.

[Loucopoulos 1990] Loucopoulos, P. and Champion R.E.M.(1990). *Concept Acquisition and Analysis in Requirements Specifications, Software Engineering Journal*. Vol 5, No 2, pp. 116-124.

[MERSS 2002] Durán A. y Bernárdez B.(2002). Metodología para la elicitación de requisitos de sistema de software. Universidad de España.

[Oberg 1998] Oberg, R., Probassco, L., Ericsson, M. (1998). *Applying Requirements Management with Use Cases*, Rational Software Corporation, Technical Paper TP505.

[Piattini 1996] M. G. Piattini, J. A. Calvo-Manzano, J. Cervera, y L. Fernández. (1996). *Análisis y Diseño Detallado de Aplicaciones Informáticas de Gestión*. ra–ma.

[Robertson 1999] Robertson Suzanne and James (1999). VOLERE. *Mastering the Requirements Process*. Addison-Wesley, London.

[Rumbaugh 1995] Rumbaugh J. (1995). *OMT: The Development Process*. Journal of Object-Oriented Programming (JOOP). Vol. 8, Nro. 1.

[Rzepka 1989] Rzepka, W. E.(1989). *A Requirements Engineering Tested: Concept, Status, and First Results*. Bruce D. Shriver (Ed.) Proceedings of the 22 nd Annual Hawai International Conference of Systems Sciences, IEEE Computer Society.

[Wieringa 1996] Wieringa, J. (1996) *Requirements Engineering: A Framework for understanding*. John Wiley & Sons.

[Wirfs-Brock 1990] R. Wirfs-Brock, B. Wilkerson, y L (1990). Wiener. *Designing Object–Oriented Software*. Prentice–Hall.

[Zave 2000] Zave, P. y otros (2000). *A Reference Model for Requirements and Specifications*. IEEE Software 17.

[Brackett 1990] J. W. Brackett. Software Requirements. Curriculum Module SEI–CM–19–1.2, Software Engineering Institute, Carnegie Mellon University, 1990. Disponible en http://www.sei.cmu.edu

[Brasilia 2002] Universidade de Brasília. Departamento de Ciẽncia Da Computacao. Engenharia de Requisitos [En línea].Disponible en: < http://www.er.les.inf.puc-rio.br/er_portugues.htm > [2002, 1 de Noviembre]

[Coleman 1998] D. Coleman (1998). A Use Case Template: Draft for Discussion. *Fusion Newsletter*. Disponible en http://www.hpl.hp.com/fusion/md_newletters.html

[Electronic 2002] Requirements Engineering Journal. Electronic Simple Copy. Coupling Use Cases and Class Models as a Means for Validation and Verification of Requirements Specifications [En línea]. Disponible en:
< http://link.springer.de/link/service/journals/00766/index.htm > [2002, 1 de Noviembre]

[Murcia 2002] Universidad de Murcia. Análisis y Gestión de Requisitos del Software.[En línea]. Disponible en: < http://www.um.es/estructura/escuelas/ept/cir.html > [2002, 5 de Noviembre]

[Raghavan 1994] S. Raghavan, G. Zelesnik, y G. Ford. Lecture Notes on Requirements Elicitation. Educational Materials CMU/SEI–94–EM–10, Software Engineering Institute, Carnegie Mellon University, 1994. Disponible en: http://www.sei.cmu.edu

[Rational 2001] Rational Software Corporation. UML "Notation Guide". Versión 1.4 Septiembre 2001 [En línea]. Disponible en: < http://www.rational.com/uml > [2002, 1 de Diciembre]

[Rational 2002] Rational Software Corporation. "UML Semantics". Versión 1.4. Septiembre 2001 [En línea]. Disponible en: < http://www.rational.com/uml > [2002, 1 de Diciembre].

[Volere 2002] The Atlantic Systems Guild Inc. copyright © 1995 – 2002. "Volere Requirements Specification Template". Edition 8 James & Suzanne Robertson. [En línea]. Disponible en:
< http://www.systemsguild.com/GuildSite/Robs/Template.html > [2002, 15 de Julio]

[Vigo 2002] Universidad de Vigo. Grupo de Redes e Ingenería del Software. Area de Ingeniería Telemática. Escuela Técnica Superior de Ingenieros de Telecomunicación. Ingeniería de Requisitos.[En línea]. Disponible en: < http://www-gris.det.uvigo.es/~jose/doctorado/re/> [2002, 1 de Noviembre].

[Sawyer 1997] P. Sawyer, I. Sommerville, y S. Viller (1997). Requirements Process Improvement through The Phased Introduction of Good Practice. *Software Process – Improvement and Practice*, 3(1). Disponible en : http://www.comp.lancs.ac.uk/computing/research/cseg/reaims/publications.html

[Sawyer y Kontoya 1999] P. Sawyer y G. Kontoya. SWEBOK: Software Requirements Engineering Knowledge Area Description. Informe Técnico Versión 0.5, SWEBOK Project, 1999. Disponible en: http://www.swebok.org