22

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

UML, LENGUAJE UNIFICADO DE MODELADO DE SISTEMAS ORIENTADOS A OBJETOS.

T E S I N A

QUE PARA OBTENER EL TITULO DE:

LICENCIADO EN MATEMATICAS APLICADAS Y COMPUTACION

P R E S E N T A :

MAURICIO JAVIER HURTADO FIGUEROA

ASESOR: ING. RUBEN ROMERO RUIZ

UNAM
CAMPUS ACATLÁN

NAUCALPAN, EDO. DE MEXICO SEPTIEMBRE 2001

Gracias.

A Dios, por haberme dado tanto.

A mis padres, por todo su apoyo, sin ustedes esto no hubiera sido posible, este es sólo uno más de sus muchos logros, gracias por su ejemplo de amor, trabajo y tenacidad.

A mis hermanos, por todo su cariño, por ser un apoyo cuando los he necesitado, por haberme retado en momentos de flaqueza y por ser un aliento para seguir superándome.

A mis abuelitas, tíos, tías y primos, por permitirme ser parte de la gran familia que somos, estructura fundamental en el desarrollo del ser humano.

A mis amigas y amigos en mis diferentes etapas escolares, por los momentos inolvidables que vivimos, de los cuales he aprendido tanto.

A mi asesor, por su orientación y comprensión.

A Marisol, por su participación en este trabajo.

A la Universidad Nacional Autónoma de México, mi alma mater, por haberme enseñado a pensar libremente, espero poder ser el líder que ustedes demandan de mi y para lo cual me han preparado tan bien.

Al Banco de México, por todos los recursos que puso a mi alcance para la consecución de este trabajo.

Índice

Índice de Figuras.

INTRODUCCIÓN

UML (Unified Modeling Language) es un lenguaje de modelado de sistemas que se ha convertido en un estándar de facto para especificación, visualización, construcción y documentación de sistemas de software orientados a objetos, cada vez más es utilizado *en todo el mundo para cualquier desarrollo de software, en una industria en donde los* sistemas son cada vez más complejos y en donde los tiempos de espera y márgenes de error son cada vez más pequeños De ahí la importancia para alumnos de la carrera de Matemáticas Aplicadas y Computación y carreras afines, de conocer y saber utilizar UML, ya que para este momento UML es *un lenguaje ampliamente extendido y materia de* conocimiento fundamental para cualquier ingeniero de software que realmente se precie de serlo

Debido a lo anterior, fue que decidí hacer una tesina al respecto, que sirviera de libro de texto para alumnos de estas carreras y como una fuente de consulta fácilmente entendible, con un tratamiento teórico pero principalmente práctico del tema.

El trabajo trata de explicar claramente que es UML, su origen y las diferentes etapas por las que ha pasado hasta convertirse en lo que es actualmente; su relación con las metodologías de orientación a objetos y con cada una de las fases del desarrollo de un sistema, su relación con las herramientas CASE (Específicamente Rational Rose) Trata de ser una guía práctica y teórica de cómo utilizar UML, ofreciendo una visión detallada *de los elementos que conforman el lenguaje y de cómo utilizarlos y finalmente* ejemplifica el uso de UML mediante un caso de estudio el cual explica como utilizar el lenguaje en cada una de las fases del desarrollo de un sistema

Para esto, la tesina se encuentra organizada de la siguiente manera:

Capitulo 1. ¿Qué es UML?

En este capítulo primeramente se describe qué es UML, cuales son sus principales ventajas y él porque de su nacimiento e importancia. A continuación se da una visión del entorno que rodea al lenguaje y como es que se da su nacimiento.

Una parte muy importante de este capítulo es dar una visión general del lenguaje al explicar brevemente los diferentes conceptos que se manejan en UML, como lo son: las vistas, los diagramas, los modelos, los elementos de modelado y los artificios de extensión del lenguaje. En este capitulo se describen muy brevemente estos elementos para dar una visión general del lenguaje, ya que en los capítulos posteriores se explica a profundidad el uso de estos elementos en cada una de las etapas de desarrollo de un sistema.

Capitulo 2. El Modelo de Casos de Uso.

Los Casos de Uso, son un artificio relativamente nuevo para expresar los requerimientos de un sistema, son la base sobre la cual se construyen los demás elementos de análisis y diseño de un sistema, ya que cada uno de los diferentes modelos que se elaboran están diseñados para cumplir con los requerimientos expresados en los Casos de Uso. En este capitulo se explica como realizar un diagrama de casos de uso, se resalta su importancia en la especificación de requerimientos y se explica su utilización como medio de comunicación entre el usuario y el desarrollador, donde el usuario puede entender y verificar de manera fácil y gráfica que toda la funcionalidad requerida para el sistema, ha sido expresada en el modelo. En los casos de uso intervienen tres elementos primordiales: los actores, los casos de uso y las relaciones. Estos tres elementos son explicados dentro de este capitulo y se brindan algunos ejemplos de cómo especificar requerimientos mediante este artificio.

Capitulo 3. El Modelo Estático.

Los diagramas utilizados en las metodologías orientadas a objetos, tanto en la fase de análisis como en la de diseño son de dos tipos básicamente. Diagramas que en su conjunto constituyen el modelo estático del sistema y diagramas cuyo conjunto constituye el modelo dinámico del sistema. El modelo estático refleja la estructura del sistema, que clases, objetos y relaciones intervienen en el sistema. En este capítulo se describe los elementos del modelo estático, se explican los diferentes diagramas que UML tiene para retratar la estructura del sistema, explicando las características de los mismos dentro de cada fase del desarrollo en la que intervienen (el análisis y el diseño). Y brinda además una explicación de cómo crear este modelo, dando algunos ejemplos que facilitan su entendimiento.

Capitulo 4. El Modelo Dinámico.

El modelo dinámico refleja el comportamiento activo del sistema. Este modelo es utilizado para expresar la interacción entre los diferentes elementos de la estructura estática del sistema, como colaboran unos con otros, que tipos de mensajes son enviados y recibidos

entre ellos, la secuencia de dichos mensajes y los diferentes estados por los que atraviesan los objetos de acuerdo a la etapa del proceso en la que se encuentren. En este capitulo se explican los diferentes diagramas que intervienen en este modelo y como utilizarlos, brindando ejemplos que facilitan la explicación de los mismos.

Capitulo 5. Arquitectura.

La arquitectura física del sistema es la estructura de todas los componentes de software y hardware que en su conjunto definen al sistema: su estructura, interfases y mecanismos que utilizan para comunicarse. Definir una arquitectura apropiada, hace más fácil navegar a través de los componentes del sistema, lo que colabora en la localización de funciones o conceptos específicos, y permite localizar donde agregar nueva funcionalidad o conceptos que cumplan con la arquitectura global. La arquitectura debe servir como un mapa para los desarrolladores que indique como está construido el sistema y donde está localizada cierta funcionalidad o concepto. Este capitulo explica como crear la arquitectura *de un sistema, explicando los diferentes elementos y diagramas UML que intervienen en* la creación de la arquitectura.

Capitulo 6. Caso de Estudio.

En este capitulo se pretenden aterrizar los conceptos presentados, mediante un ejemplo práctico que le permita al lector de esta tesina observar como se utilizan los distintos modelos en un ejemplo de la vida real. Pasando desde el estudio de *requerimientos hasta* la codificación en un lenguaje orientado a objetos como lo es Java o C++. El ejemplo es un problema muy sencillo el cual no se pretende resolver en su totalidad, ya que muy probablemente ese sería tema para otra tesina.

A través de la tesina para explicar los conceptos y ejemplificar los tipos de diagramas que se pueden crear con UML, se utilizan distintos ejemplos de la vida real, de diferentes aplicaciones y de diferentes tipos de industria o negocio, algunos de estos ejemplos fueron sacados de mi experiencia laboral y algunos otros *simplemente extraídos* de situaciones de la vida diaria. En algunos trabajos de este tipo y en algunos de los libros de la bibliografía de esta tesina, se acostumbra plantear un problema de la vida real y a través de él ir explicando los diferentes conceptos y diagramas de UML, siguiendo el ejemplo a través de los diferentes capítulos, sin embargo, este estilo de presentar las cosas obliga al lector a seguir de principio a fin el trabajo para poderlo entender, por lo que yo decidí no plantearlo de esta manera, sino plantear en cada situación un ejemplo diferente, de diferente tipo de problemática y negocio, con lo que pretendo que el lector de esta tesina pueda consultar cualquier *capitulo sin tener que conocer los capitulos* anteriores. En todos los casos los ejemplos y diagramas son una simplificación de problemas de la vida cotidiana, ya que en ninguno de ellos se pretende atacar un problema con todas las variables y complejidad del mismo. El objetivo de plantear ejemplos de diferentes tipos de negocio o industria, es con el fin de que el lector pueda darse cuenta de la diversidad de campos de aplicación para UML, y que tal vez si trabaja en alguno de ellos o se le presenta una situación similar, se sienta motivado a utilizar UML debido a que ya se dio cuenta de que es aplicable para su tipo de problema.

Al final de la tesina se presenta un ejemplo completo de cómo se modelaría un sistema de automatización de una biblioteca. Se escogió este ejemplo porque cualquier lector de esta tesina, conoce y puede visualizar un problema de este tipo sin tener ninguna experiencia

previa. Sin embargo el ejemplo es también una simplificación de este tipo de problemas, ya que lo que se pretende es ejemplificar el uso de los conceptos y diagramas de UML, sin pretender atacar todas las variables de un sistema de este tipo, ya que de hacerlo, muy bien podría ser el objeto de otra tesina.

Esta tesina puede ser leída de distintas maneras, ha sido a propósito estructurada de esta manera, de acuerdo a mi experiencia laboral y como estudiante. Hay veces que uno necesita conocer a grandes rasgos de que se trata un lenguaje de este tipo, para después realizar un estudiarlo más a fondo, para este propósito es el primer capítulo, el cual explica de manera global que es UML, dando algunos elementos de cómo surgió y explicando a grandes rasgos en que consiste. La segunda manera de leer este trabajo es leerlo de principio a fin, con lo cual uno puede conocer de manera razonablemente detallada todos los elementos de UML y como utilizarlos. La tercer manera es para cuando uno ya tiene el conocimiento teórico del lenguaje y tiene un proyecto en puerta, pero necesita saber de manera práctica como empezar a utilizar el lenguaje, este es el objetivo del último capítulo, brindar un ejemplo de cómo realizar el análisis de un sistema de principio a fin pasando por las diferentes etapas o ciclos del desarrollo de un sistema.

La bibliografía al respecto de UML y la información que se puede encontrar en Internet es amplia, en nuestro país podemos ver cada vez más libros al respecto, sin embargo, la bibliografía que utilicé para esta tesina fue en su totalidad en inglés, debido a que cuando la seleccioné, no se contaba con una versión en español de los libros utilizados, aunque esta tendencia ha ido cambiando y cada vez podemos ver más libros en español al respecto, hago esta aclaración porque todas las referencias a las que se hace alusión como citas textuales en está tesina, son más bien una traducción hecha por mi de las mismas.

CAPÍTULO 1

1. ¿QUÉ ES UML?

El *Lenguaje Unificado de Modelado* **UML** (por sus siglas en inglés Unified Modeling Language), "es un lenguaje visual de modelado de propósito general que es utilizado para especificar, visualizar, construir y documentar sistemas de software"[1] Es un conjunto de símbolos, diagramas y reglas utilizado para representar o modelar sistemas del mundo real. Es un lenguaje estándar en la industria del software para el modelado de sistemas basados en alguna metodología orientada a objetos, el cual es utilizado en el análisis, diseño e implementación de los mismos.

1.1 Importancia de los modelos

¿Qué es un modelo?. "Un modelo es una abstracción de algo con el propósito de entenderlo antes de construirlo, debido a que un modelo omite detalles no esenciales, es más fácil de manipular que la entidad original"[2].

La importancia de los modelos ha sido evidente en todas las disciplinas de la ingeniería desde hace ya un largo tiempo. Cuando se construye algo, se hacen dibujos que

---

[1] Rumbaugh, James [et al.] The Unified Modeling Language Reference Manual p. 3

[2] Rumbaugh, James. [et. al.] Object-Oriented Modeling and Design. p 15.

describen la apariencia y comportamiento del objeto en cuestión. El objeto bajo construcción puede ser una casa, una máquina, o un nuevo departamento en una compañía. Los dibujos funcionan como una especificación de cómo queremos que se vea el producto final. Dichos dibujos son usados como contratos entre el diseñador y el cliente en los cuales se especifica como será el objeto en construcción. Planes de costos, estimaciones de tiempo, distribución de trabajos y asignación de recursos están también basados en la información que estos dibujos contienen.

Así, estos dibujos son modelos de algo, son descripciones de ese algo que puede existir, estar en desarrollo o simplemente ser una propuesta. Durante el trabajo de hacer un modelo (modelado), los diseñadores deben investigar los requerimientos del producto final, los entre los que se incluyen: requerimientos funcionales, de apariencia, de desempeño y de confiabilidad. Los diseñadores deben pues crear un modelo que describa todos los aspectos del producto, dicho modelo es dividido frecuentemente en diferentes vistas, las cuales describen un aspecto especifico del producto o sistema bajo construcción. El modelo puede ir a través de diferentes fases donde en cada fase se agregan detalles al modelo.

La creación de modelos es un trabajo altamente creativo, no existe una solución final, no hay una respuesta correcta, los diseñadores mediante una serie de iteraciones aseguran que sus modelos cumplen las metas y requerimientos del proyecto bajo construcción. Normalmente un modelo nunca es un modelo final, típicamente es cambiado y actualizado durante la construcción del proyecto. Mediante iteraciones de diferentes posibilidades, los diseñadores alcanzan un mejor entendimiento del sistema y pueden finalmente crear modelos de sistemas que reflejen las metas y requerimientos del sistema y de sus usuarios.

Los modelos (sobre todo en ingeniería) son usualmente descritos mediante un lenguaje visual, lo que significa que mucha de la información en los modelos es expresada por símbolos gráficos y conexiones entre ellos. La vieja frase de que una imagen dice más que mil palabras es muy representativa en el modelado de sistemas. Usando descripciones visuales se facilita la descripción de relaciones complejas y hacen el trabajo de modelado más sencillo. Sin embargo, en algunas ocasiones no se puede expresar todo en un dibujo y se debe recurrir a una descripción en texto de estos casos.

Un modelo debe de cumplir con las siguientes características.

- *Ser exacto:* Debe describir correctamente el sistema a construir.
- *Consistente:* Las diferentes vistas no deben expresar conflictos entre otras vistas.
- *Fácil de comunicar.*
- *Fácil de corregir.*
- *Entendible:* Tan simple como sea posible (pero no simplista).

1.2 Origen de UML

Durante un ya largo tiempo, ha habido discusiones en la industria del software acerca de algo que se le ha denominado la crisis del software. Estas discusiones están basadas en el hecho de que muchos proyectos de sistemas fallan al producir soluciones que no

cumplen realmente con los requerimientos de los usuarios, que los tiempos de terminación exceden las agendas propuestas y que por consiguiente los costos de producción del software se elevan. Nuevas técnicas y herramientas como la programación orientada a objetos, lenguajes de programación visuales y avanzados ambientes de desarrollo (RAD) han sido producidos para ayudar a incrementar la productividad en el desarrollo de sistemas. Sin embargo, en muchos casos estos esfuerzos están dirigidos al nivel más bajo en el desarrollo de un sistema "la programación". Uno de los principales problemas en el desarrollo actual de software es que muchos proyectos empiezan demasiado rápido con la programación y concentran demasiado esfuerzo en la codificación. Esto es en parte porque los administradores carecen del entendimiento del proceso de desarrollo de un sistema y se sienten impacientes y ansiosos cuando su equipo de desarrollo no esta produciendo código. Esto es también debido a que los programadores se sienten más cómodos y seguros cuando se encuentran programando – una tarea con la que están muy familiarizados- que cuando se encuentran construyendo modelos abstractos del sistema que están creando.

En la actualidad han surgido una gran variedad de métodos de desarrollo de sistemas que intentan dar mayor importancia a las labores fundamentales en el desarrollo de un sistema que son el "análisis y el diseño", estos métodos implementan los ya no novedosos conceptos de orientación a objetos que además de brindar grandes ventajas en el momento de la codificación, brindan también grandes ventajas en las etapas anteriores a esta, que como ya se dijo son las más importantes. Esta variedad de métodos, todos ellos con su propia y única notación y herramientas han dejado a muchos desarrollados confundidos, ya que muchos de ellos deben enfrentarse no solo a la pronunciada curva de aprendizaje que representa entender y aplicar los conceptos de orientación a objetos, sino a una multitud de notaciones utilizadas para cada uno de estos métodos de desarrollo. La falta de una bien establecida notación que pueda ser aplicada entre los diferentes métodos y herramientas han hecho más difícil aprender como usar un buen método.

La tendencia de la industria del software se centra en la necesidad de crear modelos de los sistemas a construir, ya que los sistemas tienden cada vez a ser más grandes y complejos, distribuidos en muy diversos tipos de arquitecturas, construidos a través de muy diversas y sofisticadas herramientas de desarrollo en su mayoría visuales, etc Se hace de vital importancia la creación de modelos que especifiquen todos estos factores y sirvan como una especie de contrato entre el usuario y el desarrollador en el que ambas partes están de acuerdo en que el modelo representa el sistema a construir.

UML surge entonces como un intento de resolver los problemas arriba descritos. UML tiene entonces la potencialidad de convertirse en el estándar formal y de facto de la creación de modelos.

1.2.1 La guerra de métodos

Uno de los propósitos iniciales detrás de UML fue poner fin a "la guerra de métodos" dentro de la comunidad de orientación a objetos. La Orientación a Objetos fue expandida inicialmente a raíz del surgimiento del lenguaje de programación Simula, pero no fue tan popular sino hasta la década de los 80's con el surgimiento de lenguajes como C++ y Smaltalk. Cuando la programación orientada a objetos se convirtió en un suceso surgió la necesidad de crear métodos de desarrollo que soportaran esta nueva manera de conceptualizar los sistemas. Algunos de los métodos más populares que surgieron a principio de los 90's son:

- *"Booch:* El método de orientación a objetos creado por Grady Booch en diferentes versiones. Booch definió la noción de que un sistema es analizado como un conjunto de vistas, donde cada vista es descrita por un conjunto de diagramas de modelado. El método de notación de Booch fue ampliamente extendido aún cuando algunos usuarios encontraban algunos de los símbolos difíciles de dibujar (las infames nubes). El método también contiene un proceso mediante el cual el sistema es analizado desde un macro y micro punto de vista, y esta basado en un proceso altamente iterativo e incremental.
- *OMT:* La Técnica de Modelado de Objetos (por sus siglas en ingles Object Modeling Technique) es un método desarrollado en General Electric por James Rumbaugh. Esta basado en un franco proceso de pruebas, basadas en la especificación de requerimientos. El sistema es descrito mediante un conjunto de modelos: el modelo de objetos, el modelo dinámico, el modelo funcional, y el modelo de casos de uso, los cuales se complementan ente sí para dar una descripción completa del sistema. El método OMT toma también en cuenta la concurrencia y acceso a bases de datos relacionales.
- *OOSE/Objectory:* El método (Object Oriented Software Enginiere) y el Objectory está construidos desde el mismo punto de vista de su creador Ivar Jacobson. Ambos métodos se encuentran basados en los casos de usos, en mi opinión, la contribución más grande de Jacobson a la ingeniería de Software. Los casos de uso definen los requerimientos del sistema desde el punto de vista de los actores externos. En los dos métodos los casos de uso son implementados en todas las fases del desarrollo. El método Objectory ha sido adaptado también a la ingeniería de negocios, donde las ideas son usadas para modelar y mejorar los procesos de negocios.
- *Fusión:* Este método fue desarrollado en Hewlett-Packard. Es llamado la segunda generación de los métodos, porque está basado en las experiencias de muchos de los métodos iniciales. Esté método ha mejorado muchas ideas de los métodos anteriores, incluyendo técnicas para la especificación de operaciones e interacción entre objetos.
- *Coad/Yoardon:* Conocido también como OOA/OOD (Object Oriented Analysis and Design) fue uno de los primeros métodos usados para el análisis y diseño orientados a objetos. Era muy fácil y simple de aprender, tanto, que funcionaba bien para introducir a los novatos en las ideas y terminologías de la tecnología orientada a objetos. Sin embargo, la notación y el método no podían escalarse a sistemas de mayor tamaño. Consecuentemente, es raramente usado en la actualidad."[3]

Cada uno de estos métodos tenia su propia notación, procesos y herramientas (las herramientas CASE que soportaban la notación y el proceso). Esto hacía la elección del método una decisión muy importante y generalmente se entraba en discusión y debate acerca de que método era el mejor, más avanzado y el método correcto a usarse en un proyecto especifico. Como en toda discusión de este tipo, ninguno era el correcto, ya que todos tenían sus ventajas y desventajas. Desarrolladores experimentados usualmente tomaban un método como la base y utilizaban algunas ideas que le fueran útiles de otros métodos. En la practica, las diferencias entre métodos eran realmente insignificantes. Esto fue reconocido por algunos de los más calificados gurus en la creación de métodos, quienes empezaron a buscar caminos para cooperar entre sí.

[3] Eriksson, Martin con Penker, Magnus. UML Toolkit. p.3-4.

1.2.2 El nacimiento de UML

"Grady Booch y James Rumbaugh en Rational Software Corporation empezaron el trabajo de UML en 1994. Su objetivo era crear un nuevo método, el "Unified Method" que unificara el método de Booch y el método del cual Rumbaugh era el líder de proyecto, el OMT-2. En 1995, el creador de los métodos OOSE y el Objectory, Ivar Jacobson también fue contratado por Rational para unirse al equipo de desarrollo. Rational también compró la compañía Suiza Objective Systems creadora del Objectory. En esté punto los desarrolladores del lenguaje se dieron cuenta que su tarea era crear un lenguaje de modelado mas que un método unificado, ya que el establecer un lenguaje estándar de modelado es mucho mas fácil que el establecer un método estándar de desarrollo, ya que los diferentes métodos diferían en algunos casos substancialmente entre un autor a otro. Es difícil si no es que imposible, crear un proceso estándar que pueda ser usado por cualquier persona u organización"[4].

Aún cuando UML esta basado en los métodos de Booch, OMT, y OOSE, los diseñadores también incluyeron conceptos de otros métodos. Por ejemplo, el trabajo de David Harel en los diagramas de estado ha sido adoptado en los diagramas de estado de UML.

UML está destinado a ser el estándar de facto de los lenguajes de modelado de sistemas utilizados por la industria. Tiene un amplio rango de uso, esta basado en una bien establecida y probada tecnología de modelado de sistemas y tiene el respaldo de la industria para establecerse como un estándar en el mundo real.

1.2.3 La aceptación de UML

Para establecer a UML como un estándar, Rational y sus diseñadores se dieron cuenta de que era necesario que estuviera disponible a todo mundo, por eso, el lenguaje es "no propietario" y está abierto para cualquiera que quiera trabajar en él. Las compañías pueden usarlo con sus propios métodos, los vendedores de herramientas CASE pueden crear herramientas CASE que lo contengan como lenguaje de modelado, y los autores tienen la posibilidad de escribir libros acerca de él.

"Durante 1996 diferentes compañías se unieron a Rational como socios de negocios dándole su reconocimiento como un estándar en el modelado de sistemas. Entre las compañías que firmaron el acuerdo están Digital Equipment Corporation, HP, I-Logix, Intellicorp, IBM, ICON Computing, Mci Systemhouse, Microsoft, Oracle, Texas Instruments, Unisys y por supuesto Rational

Rational también firmó un acuerdo con Microsoft de cooperación para el desarrollo de herramientas CASE que soportaran el lenguaje"[5].

---

[4] Eriksson, Martin con Penker, Magnus UML Toolkit p 5

[5] Idem

1.3 Métodos y lenguajes de modelado

Existen diferencias importantes entre un método y un lenguaje de modelado. Un método es una manera explícita de estructurar nuestros pensamientos y acciones. Un método le dice al usuario que hacer, como hacerlo, cuando hacerlo y porque hacerlo, es decir, el propósito de cada actividad. Los métodos contienen modelos y estos modelos son usados para describir algo y para comunicar los resultados de usar un método. La principal diferencia entre un método y un lenguaje de modelado es que el lenguaje de modelado carece de un proceso o de instrucciones de qué hacer, cómo hacerlo, cuándo hacerlo y por qué hacerlo.

Cuando construimos modelos, estamos estructurando nuestras ideas. Un modelo es siempre un modelo de algo y tiene un propósito. Si el modelo no tiene ningún propósito explícito, causará problemas porque nadie sabrá cómo usarlo y por qué usarlo. Un modelo es expresado en un lenguaje de modelado. Un lenguaje de modelado consiste de una notación –los símbolos gráficos usados en un modelo- y de un conjunto de reglas relacionadas en como usarlo. Estas reglas son: sintácticas, semánticas y pragmáticas.

- *Las sintácticas:* Nos dicen como se deben de ver los símbolos y cómo se deben de combinar entre sí.
- *Las semánticas:* Nos dicen que significa cada símbolo y cómo debe de ser interpretado en el contexto en el que esté siendo usado.
- *Las pragmáticas:* Definen la intención de los símbolos a través del propósito que cumplen dentro del modelo convirtiendo en entendible para otros.

Naturalmente no es garantía el aprender el lenguaje para producir buenos modelos. Haciendo una analogía con el lenguaje cotidiano. No basta que el autor aprenda el lenguaje, que será solo una herramienta para la escritura de un libro, es tarea del autor escribir un buen libro.

1.4 Elementos del lenguaje, una visión general de UML

El propósito de esta sección es dar una visión general de lo que es UML, demostrar el ámbito en el que puede ser usado y su estructura, además de explicar brevemente sin entrar en mayor detalle los elementos que lo conforman. En este punto de la tesina es posible que no se entienda completamente los diagramas mostrados, pero no hay que preocuparse ya que lo que se intenta es dar una perspectiva del lenguaje y de sus capacidades y más adelante dentro del desarrollo del trabajo se demostrará el uso del los diagramas asociándolos a la metodología.

El Lenguaje Unificado de Modelado UML tiene un amplio campo de aplicación, puede ser usado para el modelado de negocios, el modelado de software en todas sus fases de desarrollo y en todos los tipos de sistemas, y en general para el modelado de cualquier construcción que tenga tanto una estructura estática como un comportamiento dinámico. Para permitir toda esta amplia gama de posibilidades, el lenguaje es definido para ser lo suficientemente extensivo y genérico para el modelado de muy diversos sistemas, evitando la especialización y la complejidad.

A continuación se describen las diferentes partes que conforman a UML.

- *Vistas (Views):* Las vistas muestran diferentes aspectos del sistema que esta siendo modelado. Una vista no es una gráfica, es más bien, una abstracción consistente en un número de diagramas. Solo definiendo un número de vistas, las cuales muestran cada una un aspecto particular del sistema, se puede tener una representación del sistema que se esta construyendo. Las vistas también ligan el lenguaje de modelado con el método de desarrollo escogido.
- *Diagramas:* Los diagramas son las gráficas que describen el contenido en una vista. UML tiene nueve diferentes tipos de diagramas que son usados en combinación para proveer todas las vistas del sistema.
- *Elementos de modelado:* Los conceptos usados en los diagramas son elementos de modelado que representan conceptos comunes de orientación a objetos como clases, objetos y mensajes, y la relación entre estos conceptos incluyendo asociaciones, dependencia y generalización. Un elemento de modelado es usado en diferentes diagramas, pero siempre tiene el mismo significado y símbolo.
- *Mecanismos generales:* Los mecanismos generales proveen comentarios extra, información o semántica acerca de un elemento de modelado, también proveen un mecanismo de extensión para adaptar o extender el UML a un método especifico, organización o usuario.

1.4.1 Vistas

El modelado de sistemas complejos representa una tarea extensiva. Idealmente, el sistema entero es descrito en una sola gráfica que define al sistema sin ambigüedades, y que es fácil de comunicar y entender. Sin embargo, esto es usualmente imposible. Una sola gráfica no puede capturar toda la información necesaria para describir un sistema. Un sistema es descrito mediante diferentes aspectos: el funcional (su estructura estática e interacción dinámica), el no funcional (requerimientos en cuanto al tiempo, disponibilidad, confiabilidad, de distribución, etc.) y aspectos organizacionales (organización del trabajo, la distribución de los módulos, etc.) Es por esto que un sistema es descrito mediante un conjunto de vistas, donde cada vista representa una proyección de la descripción completa del sistema, mostrando un aspecto particular del sistema.

"Una vista es simplemente un subconjunto de construcciones de modelado de UML que representan un aspecto del sistema"[6].

Cada vista es descrita mediante un conjunto de diagramas que contienen información que enfatiza un aspecto particular del sistema. Hay una ligera coincidencia, así es que un diagrama puede formar parte de más de una vista. Al mirar al sistema de desde diferentes vistas, es posible concentrarse en solo un aspecto del sistema a la vez. Un diagrama en una vista particular debe ser lo suficientemente simple para ser fácilmente comunicado, y aún coherente con los otros diagramas y vistas ya que la representación completa del sistema es descrita al juntar todas las vistas. Un diagrama contiene símbolos gráficos que representan elementos de modelado del sistema. La Figura 1-1 muestra las vistas de UML. Las cuales son:

[6] Rumbaugh, James [et al] The Unified Modeling Language Reference Manual. p 23

- *Vista de casos de uso (Use-case view):* Una vista que muestra la funcionalidad del sistema tal y como es percibida por los actores externos.
- *Vista Lógica (Logical view):* Una vista que muestra como es diseñada la funcionalidad dentro del sistema, en términos de las estructura dinámica del sistema y su comportamiento dinámico.
- *Vista de Componentes (Component view):* Una vista que muestra la organización de los componentes de software.
- *Vista de Concurrencia (Concurrency view):* Una vista que muestra la concurrencia en el sistema mostrando los problemas de comunicación y sincronización que están presentes en un sistema concurrente.
- *Vista de Distribución (Deployment view):* Una vista que muestra la distribución del sistema en la arquitectura física, con computadoras y dispositivos llamados nodos.

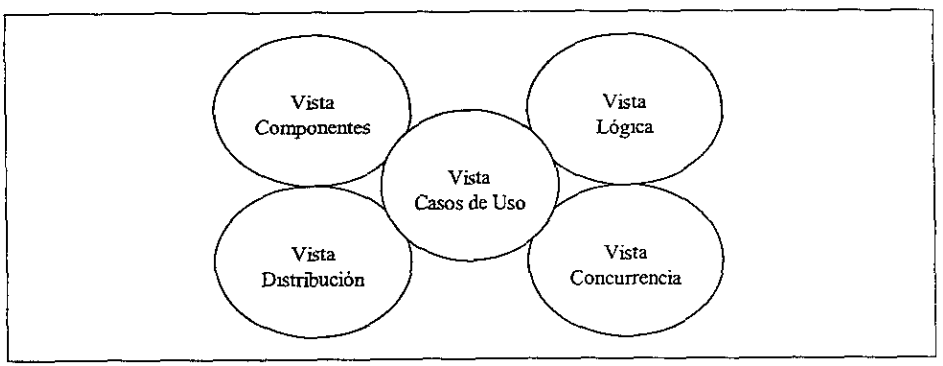

Figura 1-1. *Las Vistas en UML.*[7]

1.4.1.1 VISTA DE CASOS DE USO (USE-CASE VIEW)

La vista de casos de uso describe la funcionalidad que el sistema deberá desempeñar desde el punto de vista de los actores del sistema. Un actor puede ser un usuario del sistema u otro sistema. Esta vista es para los clientes, diseñadores, desarrolladores y el personal encargado de realizar las pruebas. El sistema completo es descrito por un conjunto de casos de uso, que son una descripción de alguna función que el sistema deberá realizar, mostrando que actores intervienen en el requerimiento, cuáles son las entradas que proporcionan, los pasos que se siguen en el desarrollo de la función y cuáles son las salidas que se desean obtener de este caso de uso.

La vista de casos de uso es central en el desarrollo del sistema, ya que es la que guía el desarrollo de otras vistas. La meta final del sistema es proveer la funcionalidad descrita en esta vista –junto con algunos otros requerimientos no funcionales -.

[7] Eriksson, Martin con Penker, Magnus. UML Toolkit. p. 15.

Es muy importante destacar que vista servirá al final del desarrollo para comprobar que los requerimientos expresados han sido cubiertos, mediante la pregunta: ¿El sistema produce los resultados esperados de acuerdo a esta vista?.

1.4.1.2 Vista Lógica (Logical view)

Esta vista describe como es realizada la funcionalidad interna del sistema. Es útil para el diseñador y el desarrollador. En contraste con la vista de casos de uso, la vista lógica explica lo que pasa dentro del sistema. Describe tanto la estructura estática (clases, objetos y relaciones) como la colaboración dinámica que ocurre cuando los objetos mandan mensajes entre sí para proveer una funcionalidad dada Propiedades como persistencia y concurrencia son también descritas en esta vista así como las interfases y estructura interna de las clases.

La estructura estática es descrita en diagramas de clases y objetos. Mientras que el comportamiento dinámico es descrito mediante diagramas de estado, secuencia, colaboración y actividad.

1.4.1.3 Vista de Componentes (Component view)

Esta vista es una descripción de los módulos de implementación y sus dependencias. Es principalmente para los desarrolladores, y consiste en el diagrama de componentes. Los componentes son diferentes tipos de módulos de código que se muestran con su estructura y sus dependencias.

1.4.1.4 Vista de Concurrencia (Concurrency view)

Modela la división del sistema mediante la división del mismo en procesos y procesadores. Este aspecto, el cual es una propiedad no funcional del sistema, posibilita el eficiente uso de recursos, la ejecución en paralelo, y el manejo de eventos asíncronos Además de dividir el sistema en threads de ejecución de control, la vista también deberá modelar la comunicación y sincronización de estos threads

Esta vista es especialmente útil para los desarrolladores e integradores del sistema y consiste en diagramas dinámicos (de estado, de secuencia, de colaboración, de actividad) y diagramas de implementación (de componentes y de distribución).

1.4.1.5 Vista de Distribución (Deployment view)

Finalmente, la vista de distribución muestra la distribución física del sistema, como computadoras y otros dispositivos (nodos) y como se conectan entre sí. Esta vista es para los desarrolladores, integradores y probadores y es representada mediante el diagrama de distribución. Esta vista también incluye una descripción que muestra como están distribuidos los componentes en la arquitectura física. Por ejemplo, que programas o objetos se están ejecutando en cada maquina.

1.4.2 Diagramas

Los diagramas son las gráficas que muestran símbolos de elementos de modelado organizados para ilustrar una parte particular o aspecto del sistema. Un modelo de sistema típicamente tiene varios diagramas de cada tipo. Un diagrama es parte de una vista en específico, y cuando es dibujado, es guardado en la vista. Algunos tipos de diagramas pueden ser parte de más de una vista, dependiendo de los contenidos del diagrama.

En esta sección describo solamente los conceptos básicos de cada diagrama, más adelante dentro del desarrollo de la tesina se mostraran los detalles de la mayoría de los diagramas con ejemplos de la vida cotidiana.

1.4.2.1 Diagrama de Casos de Uso (Use-Case Diagram)

El diagrama de casos de uso muestra a los diferentes actores que intervienen en un sistema y su relación con los casos de uso que provee dicho sistema. (Ver Figura 1-2). Un ***caso de uso*** es la descripción de una funcionalidad (un uso específico del sistema) que el sistema provee. La descripción es usualmente escrita en texto, pero existe la posibilidad de hacerla como un diagrama de actividad. "El caso de uso es descrito desde el punto de vista del actor, el comportamiento como el usuario lo percibe, y no describe como es implementada la funcionalidad dentro del sistema."[8] Los casos de uso definen los requerimientos de funcionamiento del sistema.

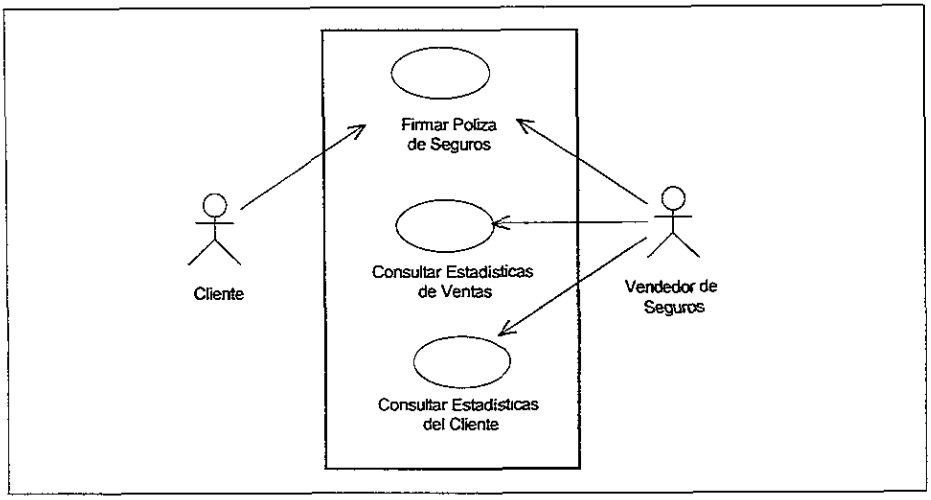

Figura 1-2 *Diagrama de casos de uso de un sistema para una aseguradora.*

[8] Jacobson, Ivar. [et. al.] Object-Oriented Software Engineering. p. 157-159.

1.4.2.2 Diagrama de Clases

El diagrama de clases muestra la estructura estática de las clases del sistema (Ver Figura 1-3). Una **clase** describe un grupo de objetos con propiedades similares (atributos), comportamiento en común (operaciones) y relaciones en común con otros objetos. Las clases pueden estar relacionadas unas con otras de diferentes maneras: asociadas (conectadas unas con otras), dependientes (una clase depende/usa otra clase), especializadas (una clase es una especialización de otra), o empaquetadas (reunidas como una unidad) *Todas estas relaciones son mostradas en un diagrama de clases junto* con la estructura interna de la clase en términos de atributos y operaciones. Este diagrama es considerado como estático ya que la estructura descrita es siempre valida en cualquier punto de la vida del sistema.

Un sistema cuenta típicamente con varios diagramas de clase --no todas las clases son insertadas en un solo diagrama de clases-- y una clase puede participar en diferentes diagramas.

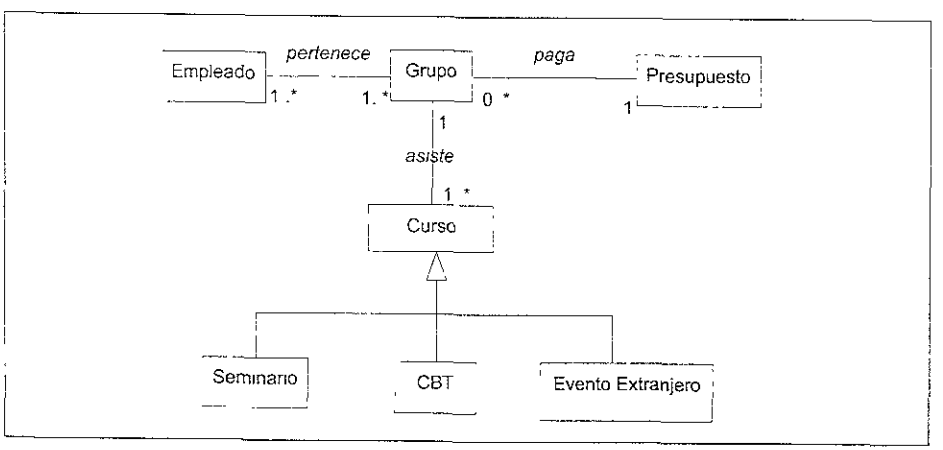

Figura 1-3 *Diagrama de clases para un sistema de control de capacitación.*

1.4.2.3 Diagrama de Objetos

Un diagrama de objetos es una variante de un diagrama de clases y usa casi la misma notación. La diferencia entre los dos es que un diagrama de objetos muestra un conjunto de instancias de una clase en lugar de las clases en sí Un diagrama de objetos es entonces un ejemplo de un diagrama de clases que muestra una posible fotografía del sistema cuando se encuentra en ejecución –como podría verse el sistema en un punto específico--. La misma notación de los diagramas de clases es usada con dos excepciones: los nombres de los objetos son subrayados y todas las instancias en la relación son mostradas. (Ver Figura 1-4).

Los diagramas de objetos no son tan útiles como los diagramas de clase, si embargo, pueden servir para ejemplificar una relación compleja, mostrando como podrían verse las instancias y las relaciones entre ellas. Los diagramas de objeto son también usados como

parte de los diagramas de colaboración, donde se observa la colaboración dinámica de los objetos.

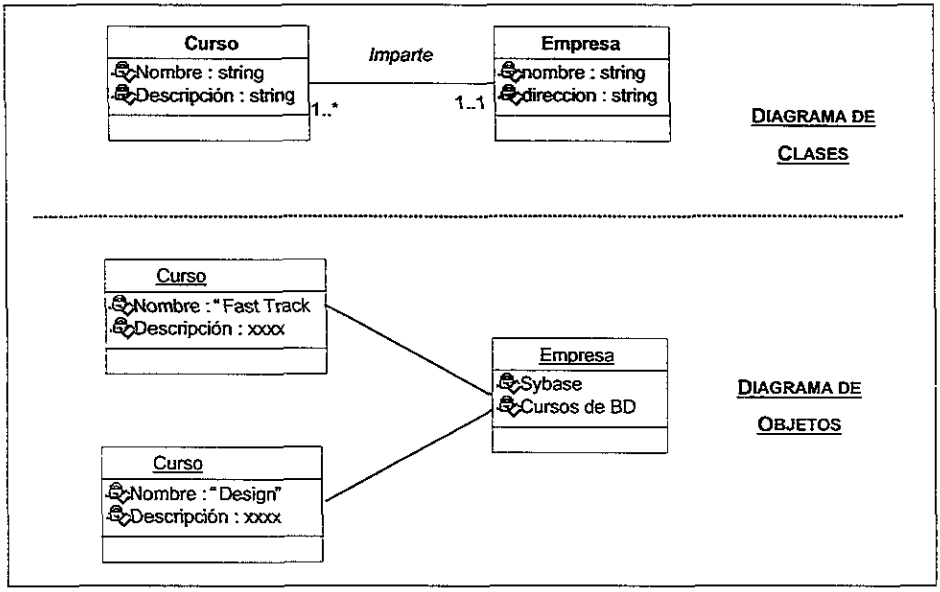

Figura 1-4 *Diagrama de clases y diagrama de objetos mostrando instancias de las clases.*

1.4.2.4 DIAGRAMA DE ESTADOS

Un diagrama de estados es típicamente un complemento de la descripción de una clase. Muestra todos los posibles estados que los objetos de una clase pueden tener y que eventos causan el cambio de estado. (Ver Figura 1-5). Un evento puede ser otro objeto que manda un mensaje al objeto en cuestión. Un cambio de estado es llamado una *transición.* Una transición puede también tener una acción asociada a ella que especifique que se debe hacer en relación con la transición de estado.

Los diagramas de estado no son dibujados para todas las clases, solo para aquellas que tienen un conjunto bien definido de estados y donde el comportamiento de la clase es afectado y cambiado por los diferentes estados. También pueden ser dibujados diagramas de estado para el sistema entero.

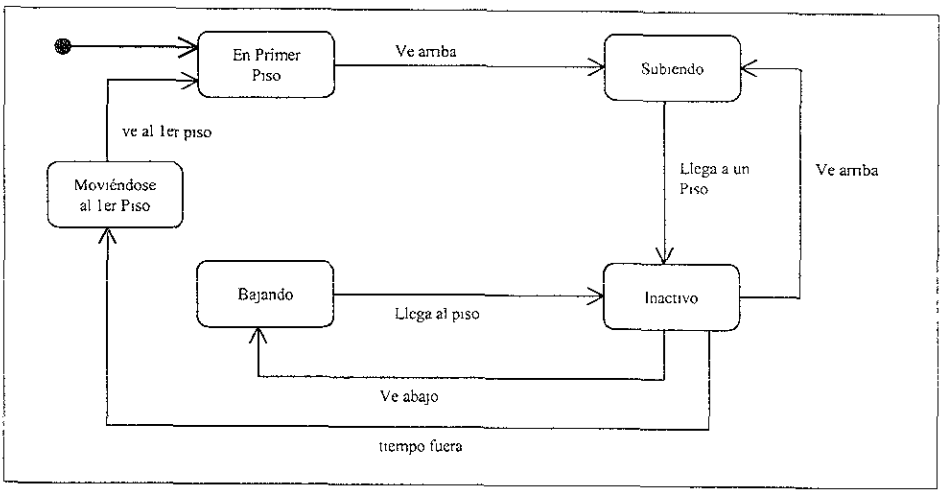

Figura 1-5 *Diagrama de estados de un elevador.*

1.4.2.5 DIAGRAMA DE SECUENCIA

Un diagrama de secuencia muestra la colaboración dinámica entre un conjunto de objetos (Ver Figura 1-6). La importancia de este diagrama radica en que muestra la secuencia de mensajes enviados entre los objetos. También muestra la interacción entre los objetos, algo que sucederá en algún punto específico en la ejecución del sistema. El diagrama consiste en un número de objetos mostrados con líneas verticales El tiempo pasa a lo largo del diagrama y se muestra el intercambio de mensajes entre los objetos. Los mensajes son representados con líneas con dirección entre las líneas verticales que representan a los objetos. Especificaciones de tiempo y otros comentarios son agregados en un letrero en el margen del diagrama.

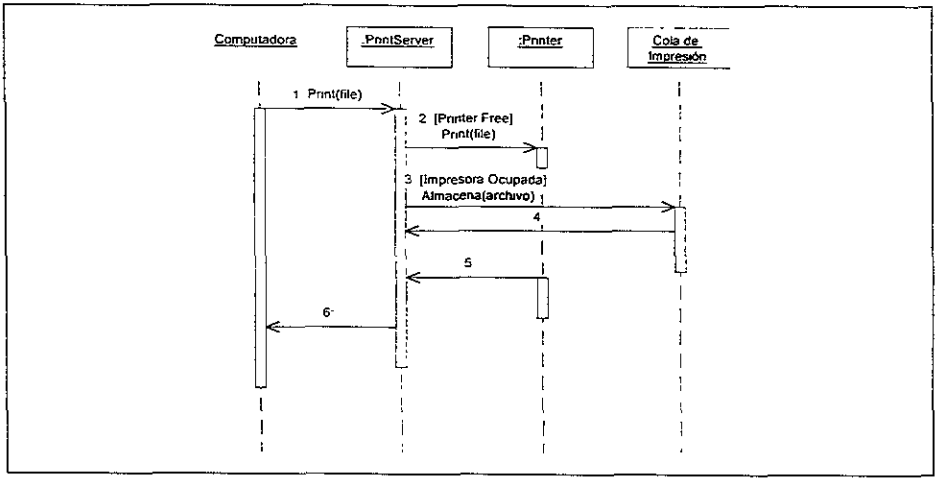

Figura 1-6 *Diagrama de secuencia para un servidor de impresión.*

1.4.2.6 DIAGRAMA DE COLABORACIÓN

Este tipo de diagrama muestra la colaboración dinámica entre un conjunto de objetos, justo como los diagramas de secuencia. Usualmente hay que realizar una elección entre que diagrama usar. Además de mostrar el intercambio de mensajes (llamado interacción) , los diagramas de colaboración muestran los objetos y sus relaciones (algunas veces referido como contexto). Decidir entre usar un diagrama de secuencia o uno de colaboración puede ser decidido por: Si el tiempo o la secuencia es el aspecto mas importante a enfatizar, se deberá seleccionar un diagrama de secuencia; si el contexto es el aspecto a enfatizar, seleccionar los diagramas de colaboración. La interacción entre objetos se muestra en ambos diagramas.

Los diagramas de colaboración son dibujados utilizando la notación de los diagramas de clase/objeto. Las flechas de los mensajes son dibujadas entre los objetos para mostrar el flujo de mensajes entre ellos. En las flechas de los mensajes son puestas etiquetas que muestran el orden en que los mensajes son enviados. También pueden mostrar condicionales, iteraciones, valores de retorno, etc. Cuando uno se familiariza con la sintaxis de las etiquetas de los mensajes, se puede leer la colaboración entre los objetos y el intercambio de mensajes. (Ver Figura 1-7)

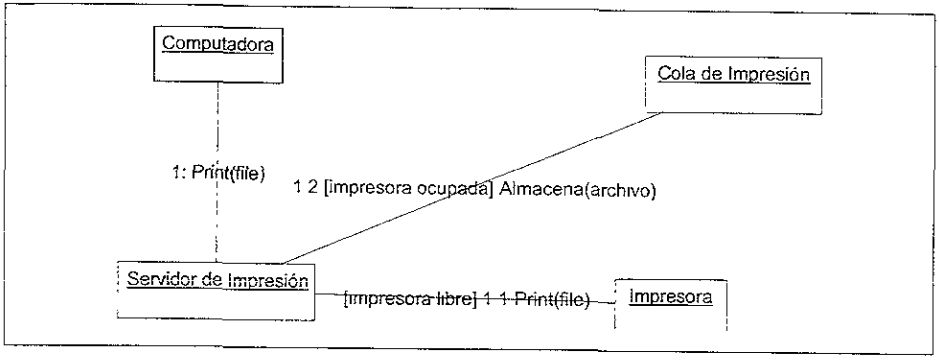

Figura 1-7 *Diagrama de colaboración para un servidor de impresión.*

1.4.2.7 Diagrama de Componentes

Un diagrama de componentes muestra la estructura física del sistema en términos de componentes de código. Un componente puede ser algún código fuente, un componente binario, o un ejecutable. Un componente contiene información acerca de la clase lógica o clases que implementa, esto creando un mapeo de la vista lógica a la vista de componentes. Las dependencias entre diferentes componentes son también mostradas, haciendo fácil analizar como otro componente es afectado por los cambios en algún otro componente. Los componentes pueden ser agrupados en paquetes y son usados en el trabajo práctico de programación (Ver Figura 1-8)

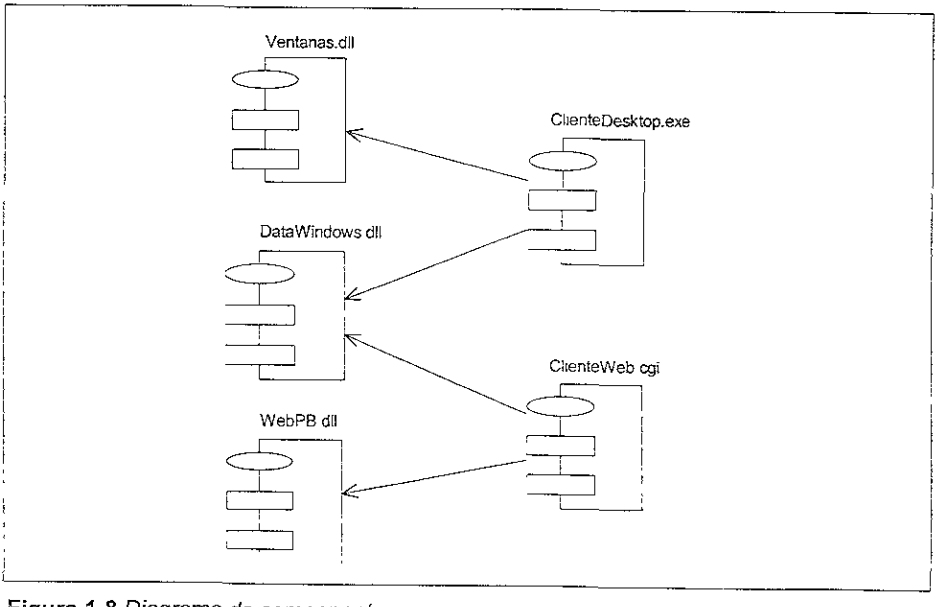

Figura 1-8 *Diagrama de componentes.*

1.4.2.8 DIAGRAMAS DE DISTRIBUCIÓN

Este diagrama muestra la arquitectura física del hardware y software en el sistema. Se puede observar las computadoras y periféricos (nodos) utilizados, junto con las conexiones entre ellos. También se pueden mostrar los tipos de conexiones. Dentro de los nodos, componentes ejecutables y objetos son puestos para mostrar que unidades de software son ejecutadas en que nodos. También se pueden mostrar las dependencias entre los componentes.

Al mostrar la arquitectura física del sistema se está muy lejos de la descripción funcional del sistema hecha en los casos de uso, sin embargo, si se define un buen modelo, es posible navegar desde algún nodo en la arquitectura física del sistema hacia los componentes que esa clase implementa hacia la interacción de los objetos y las clases que participan y finalmente llegar hasta los casos de uso. De esta manera diferentes vistas del sistema son usadas para dar una descripción coherente del sistema entero. (Ver Figura 1-9)

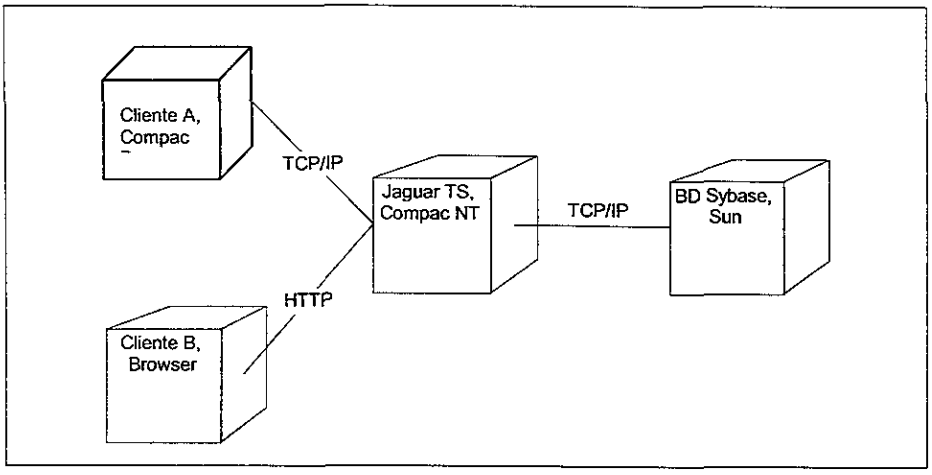

Figura 1-9 *Diagrama de distribución mostrando la arquitectura física de un sistema.*

1.4.3 Elementos de Modelado

Los conceptos usados en los diagramas son llamados elementos de modelado. Un elemento de modelado es definido por ciertas reglas semánticas, una definición formal del elemento o el significado exacto de lo que representa en oraciones no ambiguas. Un elemento de modelado tiene también un elemento visual, que es la representación gráfica del elemento usado para representarlo en los diagramas. Un elemento puede existir en diferentes tipos de diagramas, pero hay reglas que indican que elementos pueden ser mostrados en cada tipo de diagrama. Algunos ejemplos de elementos de modelado son: clases, objetos, estados, nodos, paquetes y componentes, como se muestra en la Figura 1-10.

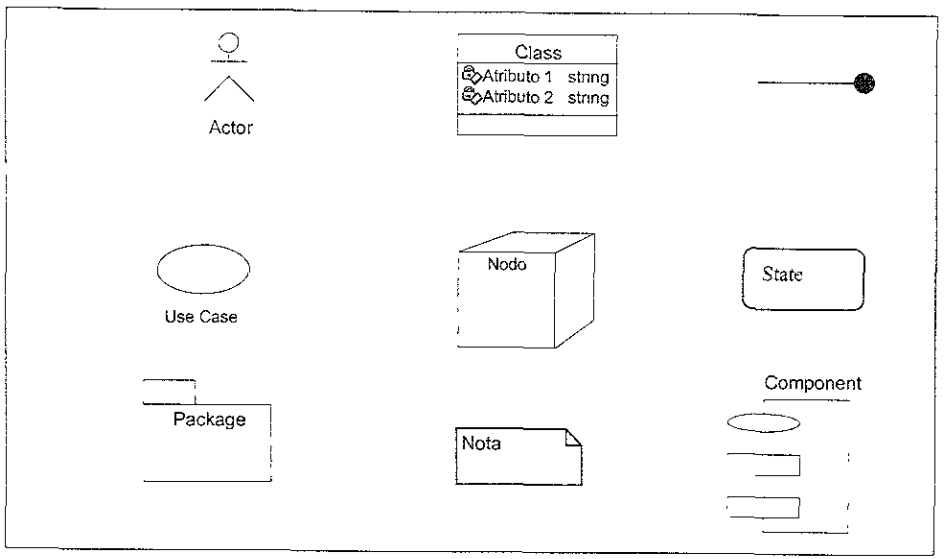

Figura 1-10 *Algunos elementos de modelado comunes.*

Además de estos elementos, también las Relaciones son otros ejemplos de elementos de modelado, y son usadas para conectar otros elementos de modelado. Algunos ejemplos son:

- *Asociación*: Conecta elementos y liga instancias.
- *Generalización*: También llamada herencia, significa que un elemento puede ser una especialización de algún otro.
- *Dependencia*: Indica que un elemento depende de alguna manera de algún otro.
- *Agregación*: Una forma de asociación en que un elemento contiene a otro.

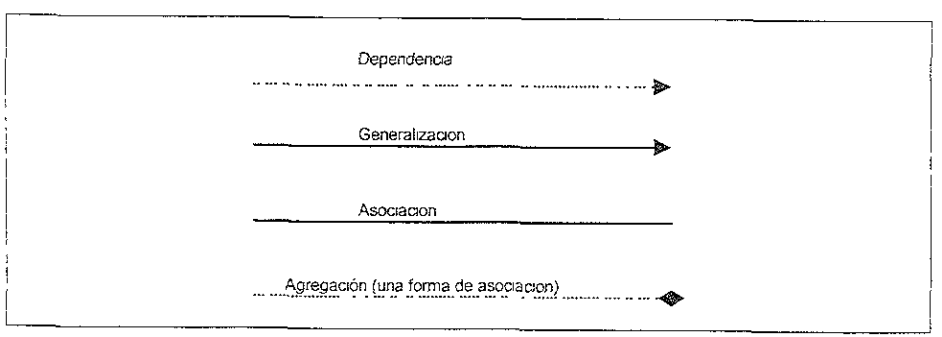

Figura 1-11 *Ejemplos de algunas relaciones.*

Otros elementos de modelado además de los descritos podrían ser: mensajes, acciones y estereotipos. Todos los elementos de modelado son descritos más a detalle en los

siguientes capítulos, incluyendo una descripción de su semántica la cual es ejemplificada con ejemplos prácticos.

1.4.4 Mecanismos Generales

UML, utiliza algunos mecanismos generales en todos los diagramas, para agregar información en los mismos, típicamente esta información, es la información que no puede ser expresada utilizando los atributos básicos de los elementos de modelado.

1.4.4.1 Formatos (Adornments)

Algunos formatos adicionales pueden ser agregados a los elementos de modelado en los diagramas, dichos formatos agregan cierta semántica al elemento, un ejemplo de un formato es la técnica usada para distinguir una Clase de una Instancia. Cuando un elemento representa una Clase, su nombre es mostrado en negritas. Cuando el mismo elemento representa una Instancia, su nombre es subrayado y puede especificar tanto el nombre de la Clase como el nombre de la Instancia. Otros formatos son la especificación de multiplicidad de una relación, donde la multiplicidad es un número o rango que indica cuantas instancias de los tipos conectados pueden estar envueltas en una relación.

Los formatos son escritos cerca de los elementos a los cuales agregan información. Todos los formatos son descritos junto con los elementos de modelado a los que afectan en los capítulos posteriores.

Figura 1-12 *Ejemplo del uso de formatos (adornments) para distinguir entre una clase y un objeto.*

1.4.4.2 Notas

No se puede definir todo en un lenguaje de modelado, no importa que tan extenso sea el lenguaje. Para permitir agregar información a un modelo que de otra manera no podría ser representada, UML brinda la capacidad de agregar Notas. Una nota puede ser puesta en cualquier parte de un diagrama y puede contener cualquier tipo de información. Su tipo de información es una cadena que no es interpretada por UML. La nota es típicamente agregada a algún elemento del diagrama mediante una línea punteada que especifica que elemento está siendo explicado o detallado.

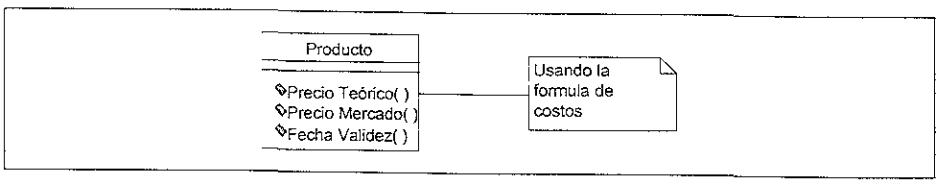

Figura 1-13 *Ejemplo del uso de notas, las cuales agregan información que pueden ser simples comentarios*

1.4.4.3 ESPECIFICACIONES

Los elementos de modelado tienen propiedades que almacenan los valores acerca del documento. Una propiedad es definida junto con un nombre y un valor llamado "tagged value", el cual es de un tipo específico como un entero o una cadena. Existe un número predefinido de propiedades como lo podrían ser: Documentación, Responsabilidad, Persistencia y concurrencia.

Las propiedades son usadas para agregar especificaciones adicionales acerca de las instancias del elemento que no son normalmente mostradas en los diagramas. Típicamente una clase es descrita con algún texto que informalmente describe las responsabilidades y capacidades de sus instancias. Este tipo de especificación no es normalmente mostrada en los diagramas, pero en una herramienta CASE son normalmente desplegadas cuando se da doble click sobre un elemento.

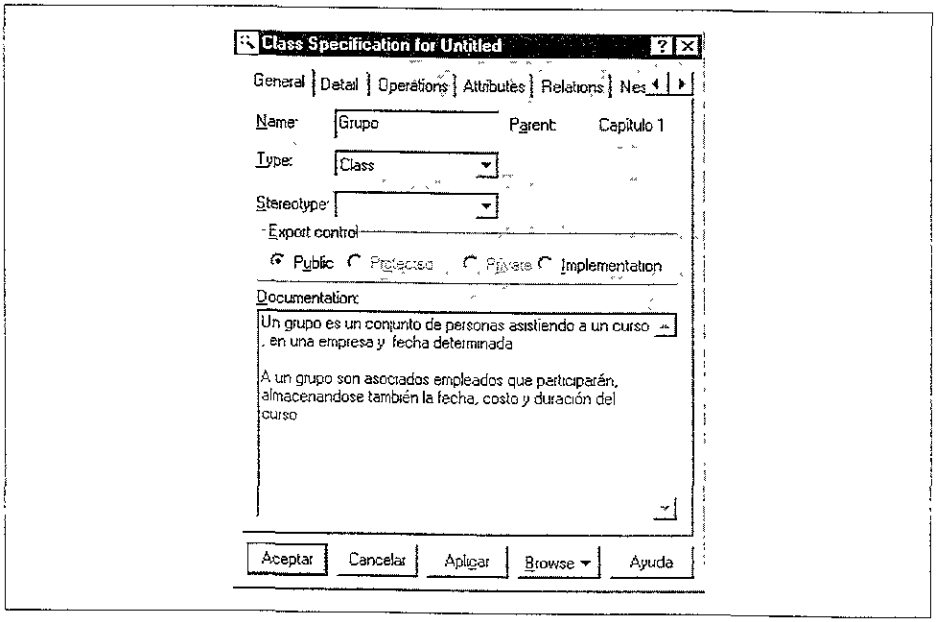

Figura 1-14 *Especificación de una clase en una herramienta CASE.*

1.4.5 ¿Cómo extender el lenguaje?

UML puede ser extendido o adaptado a un método específico, organización, o usuario. A continuación se describen tres mecanismos para hacerlo.

1.4.5.1 ESTEREOTIPOS

Un estereotipo es un mecanismo de extensión que define un nuevo tipo de elemento de modelado basándose en uno existente. Por lo tanto, un Estereotipo es "casi" como un elemento existente, más algunas características semánticas extras. Un estereotipo de un elemento puede ser usado en las mismas situaciones que el elemento original. Un estereotipo puede estar basado en cualquier tipo de elemento como puede ser una clase, un nodo, un componente o una nota, así como una relación que puede ser una Asociación, Generalización o Dependencia. Un conjunto de estereotipos están predefinidos en UML; y pueden ser usados para ajustar un elemento de modelado existente en lugar de definir uno nuevo, lo que mantiene simple al lenguaje UML básico.

Un estereotipo es descrito al poner su nombre como una cadena (por ejemplo <<Actor>>) encerrado por paréntesis triangulares llamados "guillemets". Un estereotipo puede también tener su propio símbolo gráfico, como un icono conectado a el. Un elemento específico de un estereotipo puede ser mostrado mediante su representación normal, con el nombre del estereotipo, mediante su icono o mediante una combinación de ambos. Cuando un elemento tenga un nombre de estereotipo o icono conectado a él, se lee como un tipo de elemento del estereotipo especificado. Por ejemplo, una clase con el estereotipo <<Windows>>se le como "una clase del estereotipo Window". Las características de la clase Window deben ser definidas cuando el estereotipo es definido.

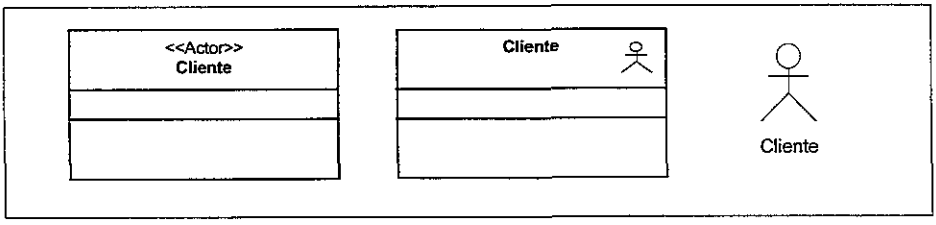

Figura 1-15 *Ejemplo del uso de estereotipos en una clase.*

1.4.5.2 VALORES TAGGED

Como se explico antes, los elementos pueden tener propiedades que contienen valores relativos al elemento. Estas propiedades son también llamadas valores "tagged". Un número de propiedades son definidas en UML, pero las propiedades pueden ser también definidas por el usuario para almacenar información adicional acerca de los elementos. Cualquier tipo de información puede ser asociada a un elemento: información específica a un método, información administrativa acerca del progreso de un modelo, información usada por otras herramientas, como aquellas que generan código o cualquier tipo de información que el usuario quiera asociar a un elemento.

1.4.5.3 CONSTRAINTS

Un constraint es una restricción a un elemento que limita su uso o semántica (significado). Un constraint es declarado en las herramientas o repetidamente usado en diferentes diagramas o es definido y aplicado cuando es necesario en un diagrama.

Un ejemplo de constraint se presenta a continuación:

En este caso, el constraint es definido para impedir que personas mayores de 50 años puedan ser o formar parte del Grupo de Gerentes.

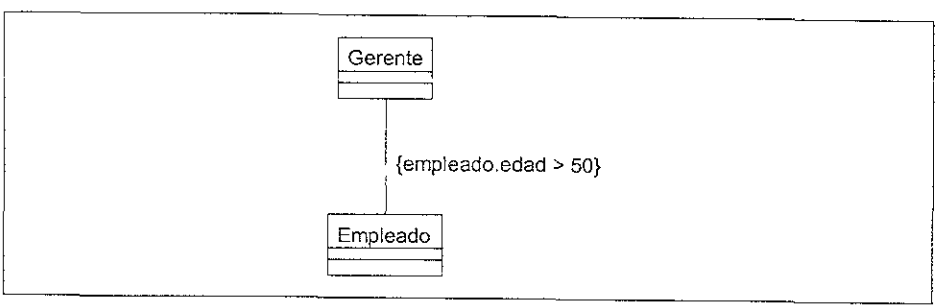

Figura 1-16 *Ejemplo de constrain que limita la relación a empleados mayores de 50 años.*

1.5 Herramientas CASE

Usar un lenguaje de modelado tan complejo y extenso como UML requiere del soporte de herramientas computarizadas que faciliten su uso. De acuerdo a algunos autores es "humanamente imposible" manejar un proyecto de mediano tamaño sin el uso de una herramienta que te asista en las diferentes etapas del proyecto. Aún cuando los primeros modelos son hechos en borradores a mano, el trabajo de mantenerlos, sincronizarlos y mantener la consistencia en los diferentes diagramas, resulta casi imposible sin una herramienta.

El *mercado de herramientas de modelado* o herramientas CASE (Computer Asisted Software Engenere) permanece sorprendentemente inmaduro desde la liberación de las primeras versiones de programas usados para producir programas. Muchas herramientas son poco más que herramientas de dibujo, con algunos chequeos de consistencia o conocimiento de el método a lenguaje de modelado presentado. Estas herramientas también están limitadas por el hecho que todas ellas tienen sus propios lenguajes de modelado o al menos su propia definición del lenguaje. Con el lanzamiento de UML, los vendedores de herramientas puede ahora invertir mas tiempo en mejorar sus productos y menos tiempo en definir nuevos métodos y lenguajes.

Una herramienta CASE moderna debe contar con las siguientes características:

- Dibujo de diagramas
- Repositorio central

- Soporte de navegación
- Soporte multiusuario
- Generación de código
- Ingeniería en reversa
- Integración con otras herramientas
- Intercambio de modelos

1.5.1 Dibujo de diagramas

La herramienta debe soportar un fácil dibujo de los diagramas, debe hacerle al usuario fácil, poder insertar elementos de modelado en el diagrama, conectarlos, organizar los elementos de modelado, etc. Además de esto para que pueda considerarse una herramienta CASE, debe conocer y advertir las reglas semánticas de los elementos y sí algún elemento no cumple con estas reglas o se esta cometiendo una inconsistencia, la herramienta lo debe de advertir.

1.5.2 Repositorio central

La herramienta debe soportar un repositorio común donde se almacene toda la información relativa a un modelo en un solo lugar. Si un cambio es hecho en un elemento en un diagrama, este cambio se debe de reflejar en los demás diagramas donde este siendo utilizado ese elemento.

La herramienta CASE debe mantener un repositorio del modelo que brinde una base de datos de toda la información acerca de los elementos utilizados en el modelo, independientemente de los diagramas de donde provenga la información. La Figura 1-17 nos muestra esta situación.

Algunas de las tareas que la herramienta puede realizar con la ayuda de un repositorio son:

- *Chequeo de inconsistencias.* Si un elemento es usado inconsistentemente en diferentes diagramas, la herramienta debe de advertir o prohibirlo. Si el diseñador trata de borrar un elemento que está siendo usado en otro diagrama, se le debe advertir acerca de esta situación. Si el diseñador insiste en borrarlo, se debe de borrar de todos los diagramas donde se este utilizando.
- *Criticar.* Usando la información del repositorio, una herramienta puede criticar el modelo, señalando las partes que no han sido especificadas o mostrando posibles errores o soluciones inapropiadas.
- *Reportar.* La herramienta debe automáticamente generar la documentación de todos los elementos del modelo.
- *Reutilizar elementos o diagramas.* Un repositorio puede soportar el reutilizar los elementos y sacar toda la ventaja posible de una de los conceptos de orientación a objetos, la reutilización.

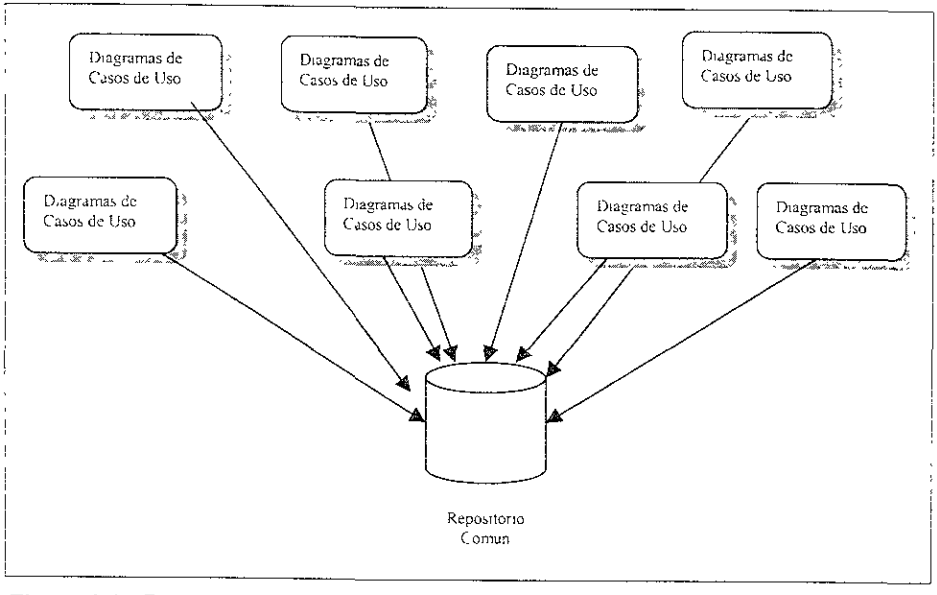

Figura 1-17 *Repositorio que contiene toda la información de los diagramas*[9].

1.5.3 Soporte de navegación

La herramienta deba hacer fácil navegar por el modelo, para rastrear un elemento de un diagrama a otro o expander la definición del elemento. Un elemento debe tener hiperligas, que tal vez no sean visibles en el diagrama, pero que son accesibles a través de la herramienta. Una manera común en que las herramientas hacen esto, es dando click derecho sobre un elemento, el cual despliega un menú contextual que permite acceder a las opciones de navegación relativas a ese elemento.

1.5.4 Soporte multiusuario

La herramienta debe permitir que varios usuarios trabajen en un modelo, y debe controlar que su trabajo no se interfiera o se convierta en inconsistente un modelo. En los proyectos de la actualidad esta característica es muy importante debido a que por la complejidad de los mismos intervienen normalmente en un diseño distintas personas a las cuales hay que controlar.

1.5.5 Generación de código

Las herramientas modernas deben brindar la facilidad de generar código, donde toda la información del modelo es traducida en esqueletos de código que son usados como base para la fase de implementación. Los lenguajes en los que típicamente son traducidos los

[9] Eriksson, Martin con Penker, Magnus. UML Toolkit p. 38

modelos son lenguajes como C++ o Java que por la naturaleza de los métodos son lenguajes obviamente Orientados a Objetos, sin embargo algunas herramientas también tienen la capacidad de generar código en lenguaje SQL o IDL. Cuando es generado el lenguaje son colocadas algunas marcas que diferencian el código producido por la herramienta del código producido por los programadores, esto con la finalidad de que si algo cambia en el modelo el código de los programadores se vea afectado lo menos posible.

1.5.6 Ingeniería en reversa

Una herramienta avanzada debe poder leer el código existente y producir modelos a partir de él. Posibilitando de esta manera, la creación de modelos a partir del código existente, además de permitirle al desarrollador interactuar entre el modelo y el código. Típicamente las herramientas solo extraen la estructura estática de un sistema, la información dinámica tiene que se extraída del código por algún desarrollador. La calidad de los modelos producidos mediante Ingeniería en Reversa obviamente dependerá de la forma en como esté estructurado el código de la aplicación, debido a esto, en algunos casos se puede quedar totalmente decepcionado y sorprendido.

1.5.7 Integración con otras herramientas

Las herramientas de modelado están "finalmente" convirtiéndose o soportando mayor integración con otras herramientas utilizadas en el desarrollo de sistemas, como lo muestra la Figura 1-18. Ejemplos de estas otras herramientas son:

- Ambientes de Desarrollo
- Sistemas de Configuración y Control de Versiones
- Herramientas de Documentación
- Herramientas para Pruebas
- Constructores de Interfases Gráficas para el Usuario (GUI)
- Herramientas de especificación de requerimientos
- Y finalmente, herramientas de administración de proyectos

1.5.8 Intercambio de modelos

Esta característica se refiere a la posibilidad de poder exportar e importar modelos desde diferentes herramientas. Para esto UML sirve de gran ayuda, ya que estandariza la notación de diferentes modelos, pero la forma de almacenarlos dentro de las distintas herramientas es lo que dificulta esta tarea. Sin embargo, esta es una característica deseable en este tipo de herramientas.

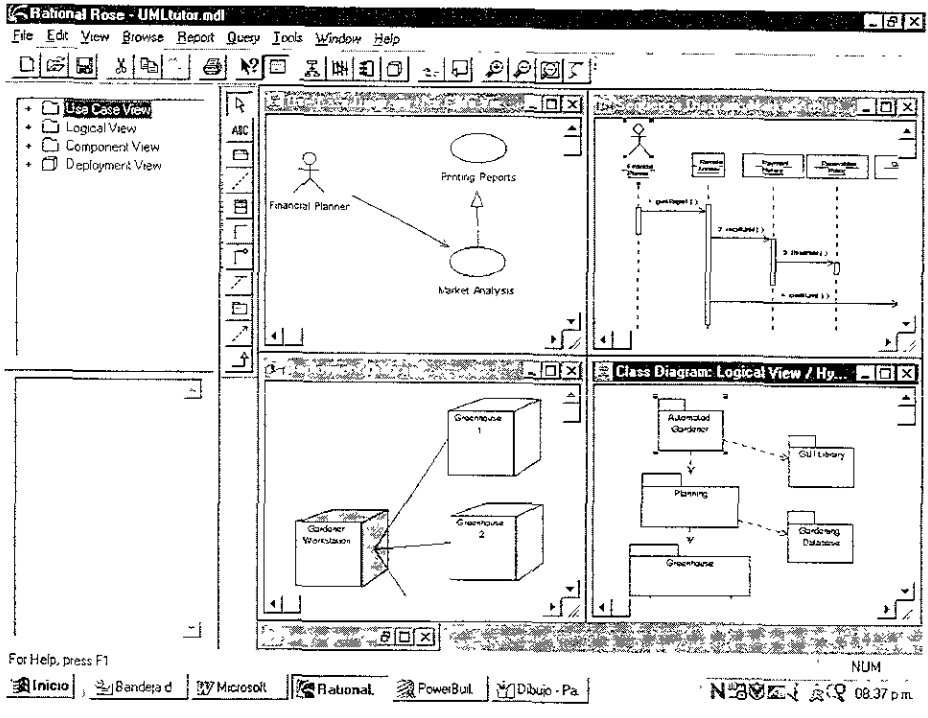

Figura 1-18 *Ejemplo de una herramienta CASE como es Rational Rose.*

CAPÍTULO 2

2. EL MODELO DE CASOS DE USO

"El *modelo de casos de uso especifica la funcionalidad que debe ofrecer el sistema* desde el punto de vista del usuario final, sin definir como será implementada dicha funcionalidad dentro del sistema"[10] Está orientado al usuario final, por lo que debe ser lo suficientemente claro para que personas sin conocimientos en informática, ni del lenguaje UML, lo puedan entender y sobre él realizar los cambios que consideren necesarios, hasta que el modelo represente lo que realmente se quiere que el sistema haga

El *modelo de casos de usos* fue creado por Ivar Jacobson basado en su experiencia como desarrollador en AXE Systems en Ericsson, específicamente, en los métodos OOSE y Objectory. Los casos de uso han despertado gran interés en la comunidad de orientación a objetos y han afectado muchos de los métodos de esta disciplina.

Los componentes principales de este modelo son los casos de uso, los actores y el sistema a modelar. La funcionalidad del sistema es representada mediante un conjunto de casos de uso, donde cada caso de uso especifica una funcionalidad completa del sistema, desde que es iniciado por un actor externo hasta que es obtenido el resultado esperado. Un caso de uso siempre devuelve algún valor a un actor, siendo el valor cualquier cosa. Un actor es cualquier entidad externa que interactúa con el sistema. Comúnmente, un actor es una persona, pero puede también ser otro sistema o algún tipo de hardware que necesite interactuar con el sistema.

---

[10] Jacobson, Ivar. [et al] Object-Oriented Software Engineering. p 157

En el modelo de casos de usos, el sistema es visto como "una caja negra" que contiene un conjunto de casos de uso, ¿cómo son implementados dichos casos de uso? y ¿cómo trabajan internamente?, no es importante en este modelo. De hecho, cuando el modelo de casos de uso es elaborado en las primeras etapas del proyecto, los desarrolladores no tienen una idea de cómo implementarán dicho modelo.

Los propósitos principales del modelo de casos de uso son:

- Decidir y describir los requerimientos funcionales del sistema.
- "Servir como acuerdo o contrato entre el usuario y el desarrollador, en el que ambas partes están de acuerdo en que el sistema a construir realizará la funcionalidad especificada en los casos de uso.
- Dar una descripción clara y consistente de qué debe hacer el sistema, ya que este modelo es la base para los demás modelos de las fases de análisis y diseño"[11].
- Ser el guión sobre el cual se prueben los modelos realizados en las demás fases.
- Servir como base para transformar los requerimientos funcionales en clases y operaciones dentro del sistema.

El trabajo requerido para crear el modelo de casos de uso consiste en definir el sistema, encontrar los actores y casos de uso, describir los casos de uso, definir las relaciones entre casos de uso y finalmente validar el modelo. Es un proceso altamente interactivo en el que deben participar activamente los clientes del sistema y las personas que representan los actores.

El modelo de casos de uso consiste de diagramas de casos de uso, los cuales muestran a los actores, los casos de uso y sus relaciones. Estos diagramas dan un panorama del modelo, pero la descripción real de los casos de uso es típicamente textual. Los modelos visuales no pueden mostrar toda la información necesaria en un modelo de casos de uso, por lo que son utilizadas ambas cosas: los diagramas de casos de uso y las descripciones textuales.

Diferentes personas tienen interés en el modelo de casos de uso. El usuario está interesado porque este modelo especifica la funcionalidad del sistema y describe como será utilizado. Es importante para el desarrollador para entender qué es lo que debe hacer el sistema y sirve como base para el desarrollo de los demás modelos. Es importante para los integradores y el equipo de pruebas, ya que deben asegurarse de que el sistema provee la funcionalidad especificada en los casos de uso. Y, finalmente, cualquiera que se encuentre envuelto en las actividades conectadas con la funcionalidad del sistema puede tener algún interés en este modelo, éstas personas pueden ser los departamentos de mercadotecnia, ventas, soporte y documentación.

El modelo de casos de uso representa la vista de casos de usos del sistema. Está vista es muy importante, ya que afecta a las demás vistas del sistema. La arquitectura lógica y física es influenciada por los casos de uso, porque la funcionalidad especificada en el modelo de casos de uso es implementada en estas arquitecturas.

---

[11] Jacobson, Ivar. [et. al.] Object-Oriented Software Engineering. p. 156-157.

El modelo de casos de uso no es únicamente utilizado para capturar los requerimientos de un nuevo sistema, también puede ser utilizado cuando una nueva versión de un sistema es desarrollada. Cuando sé esta desarrollando una nueva versión de un sistema previamente creado, la nueva funcionalidad es agregada al modelo de casos de uso actual (si es que lo hay), agregando nuevos casos, actores o simplemente alargando la descripción de los casos de uso existentes.

2.1 Diagrama de casos de uso

En UML el modelo de casos de uso es descrito a través de un conjunto de diagramas de casos de uso. Un diagrama de casos de uso contiene los siguientes elementos de modelado: el sistema, los actores y los casos de uso. Además este diagrama muestra las distintas relaciones entre los elementos como la generalización, asociación y dependencia. "El propósito de este diagrama es presentar un tipo de diagrama de contexto, en el cual uno puede rápidamente entender quienes son los actores externos del sistema y las principales maneras en las que estos actores utilizan el sistema"[12].

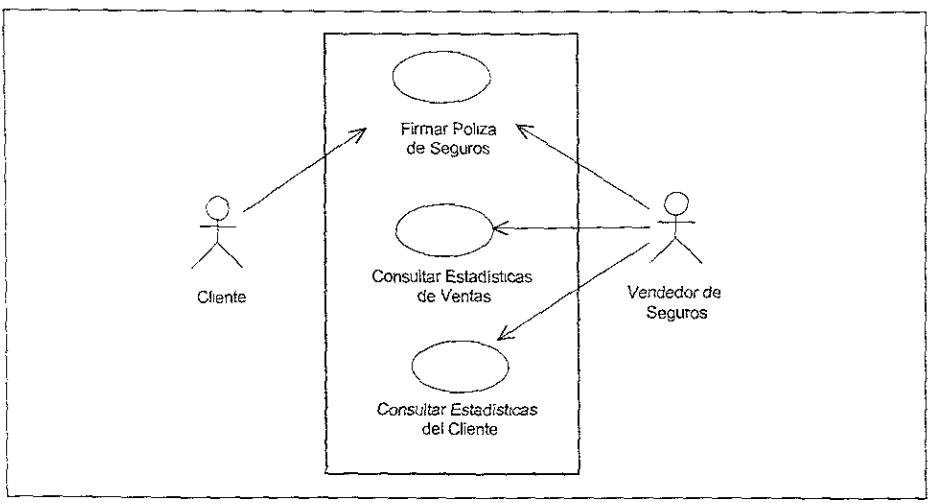

Figura 2-1 *Diagrama de casos de uso para una compañía de seguros que muestra el sistema, actores, casos de uso y sus relaciones.*

La descripción real de los casos de uso es hecha mediante una descripción textual de la funcionalidad que debe desempeñar el caso de uso. En las herramientas CASE, esta descripción es tratada como la propiedad "documentación" del elemento en particular.

Como una alternativa para describir los casos de uso, se puede utilizar un diagrama de secuencia. Sin embargo es importante recordar que un caso de uso debe ser fácilmente

[12] Larman, Craig Appling UML and Patterns An Introduction to Object-Oriented Analysis and Design. p. 55.

comunicable al usuario final y que su estructura formal, como es la representada por el diagrama de secuencia, puede intimidar a las personas no acostumbradas a interpretarlos.

2.2 El sistema

Como parte del modelado de casos de uso deben definirse las fronteras del sistema. Definir las fronteras y la responsabilidad del sistema no es una tarea fácil, porque no es siempre obvio que tareas son mejor automatizadas por el sistema y cuales son mejor manejadas manualmente o por otros sistemas. Otra consideración es que tan largo debe de ser el sistema. Es tentador ser ambiguo en las primeras versiones, pero esa falta de objetivos puede hacer que el sistema sea demasiado grande y que el tiempo de entrega sea mucho más largo. Una mejor idea es identificar la funcionalidad básica del sistema y concentrarse en definir una arquitectura del sistema estable y bien definida a la cual se le puede agregar más funcionalidad en futuras versiones del modelo o del mismo sistema.

Es esencial recopilar un catalogo con los conceptos centrales (entidades) que puedan ser definidos claramente en las primeras etapas del desarrollo. Esto no es un modelo del dominio de objetos (domain object model) sino un intento de describir la terminología del sistema. El catalogo puede también ser usado para empezar el análisis del dominio que sigue en etapas posteriores.

En los diagramas de casos de uso el "Sistema" es representado por una caja, en donde el nombre del sistema aparece en la parte superior de la caja.

2.3 Actores

Un actor es alguien o algo que interactúa con el sistema, es el "quien" o el "que" usa el sistema. El interactuar con el sistema significa que el actor envía y recibe mensajes del sistema o intercambia información con el mismo. Un actor puede ser alguna persona u otro sistema (como otra computadora o algún tipo de dispositivo de hardware). "Los actores modelan a los prospectos de usuario, es decir, son tipos o categorías de usuarios, cuando un usuario hace algo, el o ella actúan como una ocurrencia de ese tipo"[13].

Un actor es un tipo (una clase), no una instancia. El actor representa un rol, no a un usuario en particular. Por ejemplo: si Francisco González quiere obtener una póliza de seguro de una compañía de seguros, su rol será el ser "un comprador" de las pólizas de seguro. De hecho, una persona puede tener diferentes roles en el sistema, dependiendo del momento y lo que quiera obtener del sistema.

El actor se comunica con el sistema enviado y recibiendo mensajes, los cuales son similares a los observados en la programación orientada a objetos, pero con la diferencia de que no son tan formalmente especificados en los casos de uso. Un caso de uso es siempre inicializado por un actor que le envía un mensaje, el cual es conocido como

[13] Jacobson, Ivar. [et. al.] Object-Oriented Software Engineering. p. 157.

estímulo. Cuando un caso de uso es llevado a cabo, el caso de uso puede enviar mensajes a uno o más actores. Estos mensajes pueden ir dirigidos al actor que inicio el caso de uso o algún otro.

Los Actores pueden ser clasificados como:

- *Actores Primarios,* que son los que usan la funcionalidad principal del sistema. Por ejemplo: en un sistema de alguna aseguradora, un actor primario puede ser el que maneja el registro y administración de los seguros.
- *Actores Secundarios,* son los que usan la funcionalidad secundaria del sistema, es decir, aquella funcionalidad que mantiene al sistema, como la administración de las bases, la comunicación, los respaldos y algunas otras tareas administrativas. Un ejemplo de un actor secundario podría ser un administrador o algún analista que utiliza el sistema para sacar estadísticas acerca del negocio o de la compañía.

Ambos tipos de actores son modelados para asegurar que la funcionalidad completa del sistema sea descrita. Sin embargo "los actores primarios gobernarán la estructura del sistema, cuando se identifiquen casos de uso, se empezará con los actores primarios, asegurando de esta manera que la arquitectura del sistema se adapta a los usuarios mas importantes. También los cambios en el sistema vendrán principalmente de estos actores"[14]

Los Actores también pueden ser clasificados como activos o pasivos.

- Un actor activo, es el que inicializa un caso de uso.
- Un actor pasivo nunca inicializa un caso de uso, pero participa en uno o más casos de uso.

2.3.1 ¿Cómo encontrar actores?

Para encontrar los actores, se establece aquellas entidades interesadas en usar o interactuar con el sistema. Es entonces posible tomar la posición del actor y tratar de identificar los requerimientos del sistema y que casos de uso necesita.

Las siguientes preguntas podían ser útiles para encontrar a los actores:

- ¿Quién usará la funcionalidad principal del sistema (actores primarios)?
- ¿Quién requiere soporte del sistema para desarrollar sus actividades diarias?
- ¿Quién debe actualizar, administrar y mantener el sistema trabajando (actores secundarios)?
- ¿Qué dispositivos de hardware requiere manejar el sistema?
- ¿Con qué otros sistemas interactúa el sistema a modelar?
- ¿Quién tiene algún interés en los resultados (los valores) que el sistema produce?

[14] Jacobson, Ivar. [et. al.] Object-Oriented Software Engineering. p. 159.

Cuando se busca a los actores del sistema, no se debe considerar únicamente los individuos sentados frente a la pantalla de la computadora, hay que recordar que un actor puede ser alguien o algo que directa o indirectamente interactúa con el sistema y utiliza los servicios del mismo para alcanzar un objetivo. Hay que tener en mente que el modelo de casos de uso es hecho para modelar un negocio, por lo que los actores usualmente son los clientes de dicho negocio. Por lo tanto, ellos no son usuarios en el sentido de sentarse detrás de una computadora.

Recordemos que un actor es un rol (una clase) no una instancia en particular. Para darnos cuenta que una entidad realmente es un actor y no una instancia o usuario, se debe intentar encontrar ejemplos de instancias del actor, es decir diferentes personas dentro del sistema que puedan jugar ese rol, lo que nos permitirá determinar que el actor realmente existe. Un actor debe tener alguna asociación con al menos un caso de uso, aunque el actor puede que no inicialice ningún caso de uso, el actor debe comunicarse en algún punto al menos con un caso de uso.

El nombre del actor debe ser escogido de tal manera que refleje su rol en el sistema.

2.3.2 Actores en UML

Teniendo en mente que los actores son clases en UML con el estereotipo <<actor>> y que el nombre de la clase es el nombre del actor, una clase actor puede tener atributos y comportamiento así como una propiedad que contenga su documentación. Una clase actor cuenta con un estereotipo estándar que es un pequeño "muñeco", con el nombre del actor debajo de la figura como se muestra en la siguiente figura:

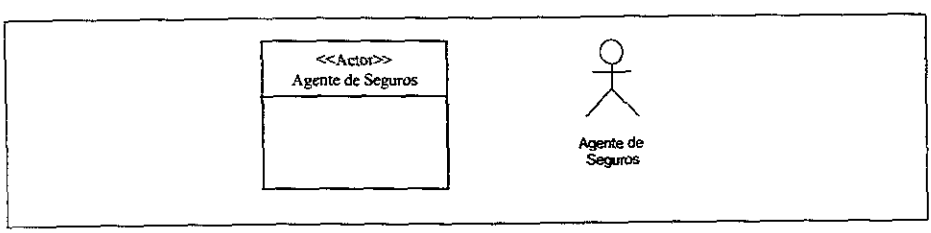

Figura 2-2 *Un actor es una clase, y es representado por un rectángulo de clase con el estereotipo <<actor>>. El estereotipo estandar del muñequito es normalmente mostrado en los diagramas de casos de uso.*

2.3.3 Relaciones entre actores

Debido a que los actores son clases, pueden tener las mismas relaciones de una clase. Sin embargo, en los diagramas de casos de uso, solo las relaciones de generalización son usadas para describir algún comportamiento similar entre diferentes actores.

2.3.3.1 GENERALIZACIÓN

Cuando diferentes actores tienen algún comportamiento en común, dicho comportamiento puede ser descrito en un actor que generalice dicho comportamiento. De está manera al ser extendido el actor general, los actores especializados heredan el comportamiento de

su superclase y extenderán el comportamiento de alguna manera La generalización entre actores es representada en UML como una línea con un triángulo hueco al final el cual apunta hacia la clase más general, como se muestra en la Figura 2-3. Esta notación es la misma que se usa para la generalización de cualquier clase.

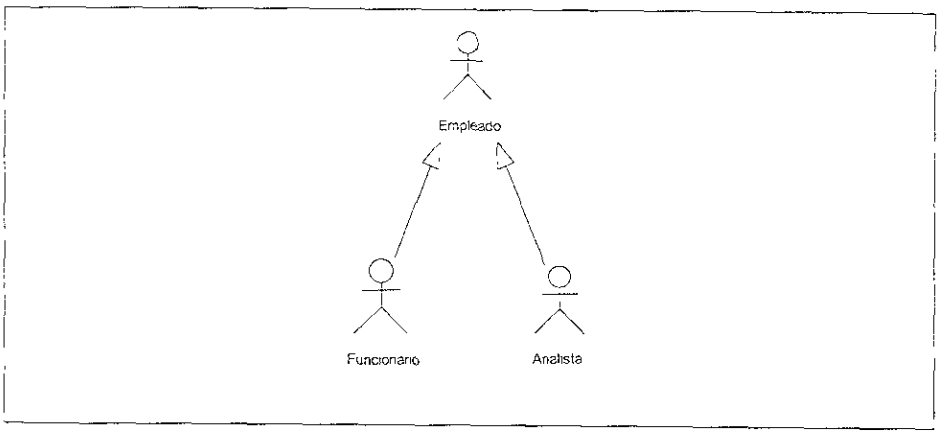

Figura 2-3 *Generalización de un actor. En el ejemplo se muestra la generalización del actor empleado, los actores funcionario y analistas heredan las características del actor empleado*

2.4 Casos de uso

Después de haber definido las fronteras del sistema y qué es lo que existe fuera de él, podemos definir la funcionalidad dentro del mismo. Esto es hecho a través de la especificación de casos de uso. "Un caso de uso es una manera específica de usar el sistema al desempeñar alguna parte de su funcionalidad"[15]. Cada caso de uso constituye una secuencia completa de eventos inicializada por un actor especificando la interacción entre el actor y el sistema. "Un caso de uso es entonces, una secuencia especial de transacciones relacionadas entre sí, que son hechas por el actor y el sistema en un dialogo El conjunto de todos los casos de uso especifica todas las posibles maneras de utilizar el sistema"[16].

"Un caso de uso debe tener las siguientes características:

- *Ser siempre inicializado por un actor.* Un caso de uso es siempre iniciado por un actor El actor debe directa o indirectamente ordenar al sistema desarrollar el caso de uso Ocasionalmente, el actor no será conciente de la inicialización del caso de uso.

[15] Jacobson, Ivar [et al] Object-Oriented Software Engineering p 159

[16] Idem

- *Debe regresar un valor a un actor.* Un caso de uso deber entregar algún tipo de valor tangible al usuario. El valor no tiene que ser siempre un valor de salida, pero debe ser discernible.
- *Debe ser siempre completo*: Un caso de uso debe ser una descripción completa. Un error común es dividir un caso de uso en casos de uso más pequeños que implementan alguna funcionalidad mediante el llamado de otros casos de uso, como si fuera el llamado de funciones en un lenguaje de programación. Un caso de uso no está completo hasta que un valor final es producido, aún cuando diferentes comunicaciones (como diálogos con el usuario) ocurran durante su ejecución"[17].

Los casos de uso son conectados a los actores mediante asociaciones, las cuales se llaman *"asociaciones de comunicación"*. Estas asociaciones muestran con qué actores se comunica el caso de uso, incluyendo el actor que inicializa la ejecución del caso de uso. Dicha asociación es normalmente una relación uno-a-uno, sin dirección, esto significa que una instancia de un actor se comunica con una instancia del caso de uso, y que la comunicación puede ser en ambas direcciones. El nombre del caso de uso es el nombre de la instancia que ejecuta, por ejemplo Firma de la póliza de seguro, Actualización del registro, etc., y es comúnmente una frase más que una sola palabra.

Un caso de uso es una clase, no una instancia. Describe la funcionalidad como un todo, incluyendo posibles alternativas, errores y excepciones que pueden ocurrir durante su ejecución. Una instancia de un caso de uso es llamada un escenario, y representa un uso en específico del sistema. Por ejemplo: un escenario del caso de uso "Firma de la póliza de seguro" podría ser "Juan Pérez contacta el sistema por teléfono y solicita asegurar un carro Jetta que acaba de comprar".

2.4.1 Encontrando casos de uso

La razón por la que se especifican primeramente los actores, es porque ellos serán la mejor herramienta para encontrar los casos de uso. Cada actor utilizará un conjunto de casos de uso en el sistema. Al ir a través de todos los actores y definir todas las posibles acciones que podrá desarrollar cada actor, se puede definir la funcionalidad completa del sistema.

Para encontrar los casos de uso se pueden hacer las siguientes preguntas por cada actor identificado:

- ¿Qué funciones requiere el actor por parte del sistema?, ¿Qué necesita hacer el actor?
- ¿Necesita leer, crear, destruir, modificar o almacenar algún tipo de información en el sistema?
- ¿Necesita ser notificado acerca de eventos en el sistema, o necesita notificar al sistema acerca de algo? ¿Qué representan esos eventos en términos de funcionalidad?
- ¿Pudiera ser simplificado el trabajo diario de algún actor por alguna funcionalidad nueva en el sistema que fuese más eficiente?

[17] Eriksson, Martin con Penker, Magnus. UML Toolkit. p. 51-52.

Otras preguntas que no están relacionadas directamente con los actores, podrían ser.

- ¿Qué entradas/salidas necesita el sistema?, ¿De dónde vienen estas entradas/salidas?
- ¿Cuáles son los mayores problemas con la actual implementación del sistema?

Estas últimas preguntas no significan que los casos de uso no tienen un actor, solo que el actor es reconocido después de identificar el caso de uso. Ya que como se mencionó anteriormente un caso de uso debe estar conectado al menos a un actor.

2.4.2 Casos de uso en UML

Un caso de uso es representado en UML como una elipse con el nombre del caso de uso dentro de la elipse o debajo de ella Es normalmente puesto dentro de las fronteras del sistema y es conectado a los actores a través de una asociación, como se muestra en la siguiente figura.

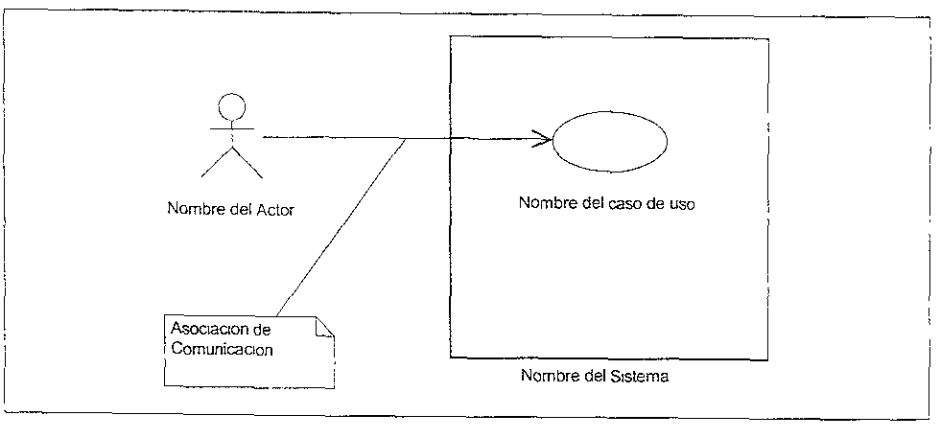

Figura 2-4 *Los casos de uso en UML son representados por elipses dentro de la frontera del sistema y tienen asociaciones con los actores.*

2.4.3 Relaciones entre casos de uso

Existen tres tipos de relaciones entre casos de uso: extensión (extends), uso (uses) y agrupamiento (grouping). La extensión y uso son formas diferentes de herencia. El agrupamiento es una forma de poner juntos en un paquete los casos de uso relacionados.

2.4.3.1 Relación de Extensión

Es un tipo de relación de generalización. Cuando un caso de uso extiende a otro, significa que primero puede incluir algún comportamiento del caso de uso que está siendo extendido. No tiene que incluir todo el comportamiento; puede elegir que partes del comportamiento quiere reutilizar El caso de uso que está siendo extendido debe ser completo. Debido a que los casos de uso son descritos en texto, puede ser difícil definir

que partes están siendo reutilizadas del caso de uso general, cuales son redefinidas y cuales son agregadas al caso de uso.

Un caso de uso extendido puede manejar excepciones que son casos específicos del caso de uso general, las cuales no son fácilmente descritas en el caso de uso general, o son hechas en el proceso de desarrollo.

A continuación se muestra una relación de extensión entre casos de uso, que como es de suponerse se representa en UML como una generalización (una línea con un triángulo hueco apuntando al caso de uso que está siendo extendido) con el estereotipo <<extends>>.

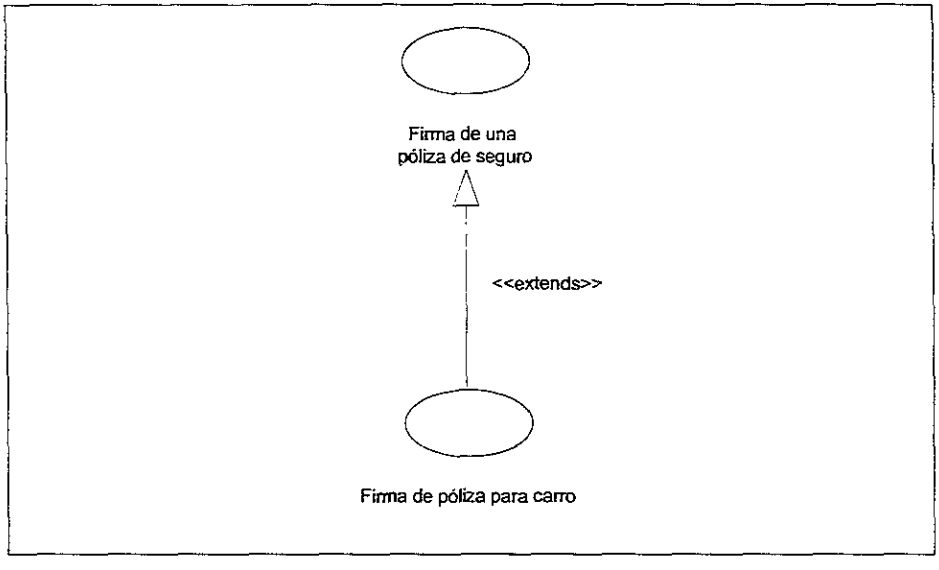

Figura 2-5 *Relación de extensión, donde el caso de uso "Firma de póliza para carro" extiende el comportamiento del caso de uso "Firma de una pòliza de seguro".*

2.4.3.2 Relación de Uso

Es otro tipo de relación de generalización. Cuando un conjunto de casos de uso tiene algún comportamiento en común, este comportamiento puede ser modelado en un solo caso de uso que es usado por los otros. Cuando un caso de uso "usa" otro, todo el caso de uso debe ser usado (aunque las actividades del caso de uso "usado" no tienen que realizarse en la misma secuencia, pueden ser mezcladas con las actividades del caso de uso que se está describiendo). Si el caso de uso "usado" nunca es usado por sí mismo, se dice que es un caso de uso abstracto.

A continuación se muestra una "relación de uso" entre casos de uso, que como es también de suponerse, se representa en UML como una generalización (una línea con un triángulo hueco apuntando al caso de uso general o que está siendo usado) con el estereotipo <<uses>>.

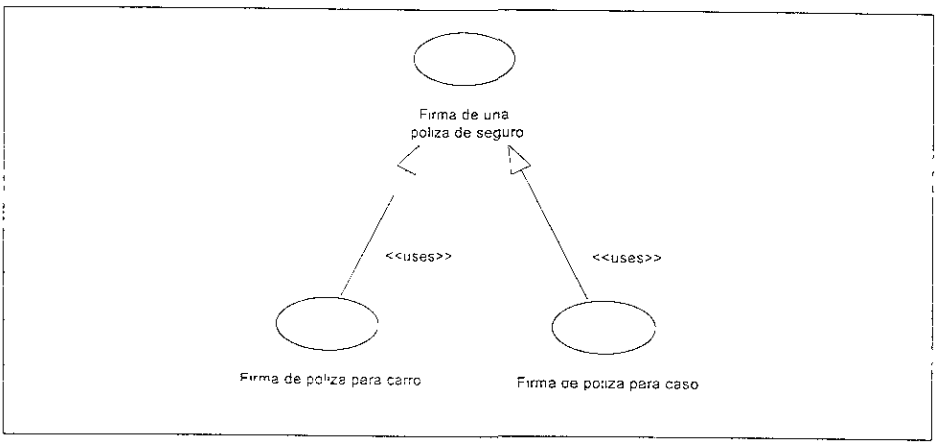

Figura 2-6 *Relación de uso "uses" entre casos de uso. El ejemplo muestra que los dos casos de uso de abajo tienen un comportamiento común descrito en el caso de uso superior*

2.4.3.3 Relación de Agrupamiento

Cuando un conjunto de casos de uso maneja funcionalidad similar o está relacionado de alguna manera, puede ser agrupado en UML como un paquete. Un paquete agrupa elementos de modelado relacionados, y puede ser expandido o colapsado en un icono, permitiéndole al desarrollador ver un paquete a la vez. Los paquetes no tienen otro significado semántico.

2.4.4 Descripción de casos de uso

Como se mencionó anteriormente, la descripción de un caso de uso es hecha normalmente mediante una descripción textual. La cual es una especificación simple y consistente de cómo los actores y los casos de uso (el sistema) interactúan. Se concentra en el comportamiento del sistema e ignora como son hechas las cosas realmente dentro del sistema. El lenguaje y terminología usada en la descripción es la misma que la usada por el cliente/usuario del sistema.

La descripción textual debe incluir los siguientes aspectos:

- *Objetivo o propósito del caso de uso*: ¿Cuál es el objeto del caso de uso?, ¿Qué se pretende alcanzar? Los casos de uso están orientados a objetivos, y el objetivo de cada caso de uso debe ser fácilmente identificable.
- *Como es inicializado el caso de uso* ¿Qué actor inicia la ejecución del caso de uso, y en qué situaciones?
- *El flujo de mensajes entre los actores y el caso de uso:* ¿Qué mensajes o eventos intercambian el caso de uso y el actor para notificar, actualizar o recuperar información entre ellos? ¿Cuál es la secuencia de eventos entre el actor y el sistema para completar un proceso o alcanzar un objetivo?

- *El flujo alternativo en el caso de uso:* Un caso de uso puede tener ejecuciones alternativas dependiendo de las condiciones o excepciones que se presenten. En la descripción de los casos de uso es útil mencionarlos, pero se debe tener cuidado de no describirlos con demasiado detalle, debido a que puede esconder el flujo principal de acciones. Las descripciones específicas de manejo de errores son descritas en escenarios.
- *El valor que es regresado al actor cuando termina el caso de uso.* Describe el momento en el que el caso de uso es considerado terminado y el tipo de valor que es entregado al actor.

Es importante recordar que la descripción identifica lo que es relevante que sea hecho de acuerdo a los actores externos, no como son hechas las cosas dentro del sistema. El texto debe ser claro y consistente para que el cliente lo pueda entender y validarlo (estar de acuerdo que representa lo que él o ella quiere del sistema. Hay que evitar sentencias complicadas que puedan hacer difícil interpretarlo y que se presten a malos entendidos.

2.4.4.1 FORMATO DE LA DESCRIPCIÓN TEXTUAL

UML no especifica un formato rígido para la descripción de casos de uso, el formato que se use debe estar enfocado a cumplir el espíritu de la documentación –claridad en la comunicación–. A continuación se presenta un ejemplo de estructura que pudiese ser usada para la descripción de un caso de uso[18]. El objetivo del formato en el que se presenta es hacer fácilmente identificables cada uno de los componentes con los que debe contar todo caso de uso.

El caso de uso de ejemplo describe el proceso de comprar cosas en una tienda cuando es utilizada una terminal "punto de venta". Para evitar complejidad en el ejemplo, únicamente se maneja pago en efectivo y se ignora el manejo de inventarios.

Caso de Uso:	**Comprar cosas en efectivo.**
Actores:	Cliente (inicio), Cajero
Propósito:	Capturar una venta y su pago en efectivo.
Descripción:	Un cliente llega a la caja con las cosas que desea comprar. El cajero registra las cosas a comprar y recoge el pago en efectivo. Al completarse, el cliente se retira con las cosas. Tipo: primario y esencial.
Tipo:	Primario.
Referencias cruzadas:	R1.1, R1.2, R1.3, R1.7, R1.9, R2.1

Curso típico de eventos

[18] Esta estructura es propuesta en el libro. Craig Larman, Applying UML and Patterns, An Introduction to Object-Oriented Analysis and Design. p. 49-70.

Actores	Sistema
1 Este caso de uso comienza cuando un Cliente llega a la caja con las cosas que desea comprar	
2 El Cajero introduce los identificadores de cada producto. Si el Cliente desea comprar más de un artículo del mismo producto, el Cajero puede introducir también la cantidad.	3. Determina el precio del producto y agrega la información del producto a la transacción actual de venta. La descripción y el precio del producto actual son presentadas
4. Al terminar de introducir los productos, el Cajero le indica al sistema que ha terminado de introducir los productos.	5 Calcula y presenta el total de la venta.
6. El Cajero le indica al Cliente el total de la venta.	
7 El Cliente le da el pago al cajero -posiblemente más que el total-.	
8. El Cajero introduce la cantidad recibida.	9 El sistema le muestra el cambio que le debe regresar al Cliente. Genera el recibo.
10 El Cajero deposita el efectivo recibido y extrae el recibo para el Cliente. El Cajero le da su cambio y recibo al Cliente.	11. Graba la venta como completada
12. El Cliente se marcha con los productos comprados	

Cursos Alternativos

Línea 2: Código de producto introducido invalido: Indicar error

Línea 7: El cliente no tiene suficiente dinero: Cancelar la transacción de venta

2.4.4.1.1 Explicación del formato propuesto

"Caso de Uso:	Nombre del caso de uso
Actores:	Lista de actores (agentes externos), indicando quien inicializa el caso de uso.
Propósito:	Intención del caso de uso.
Descripción:	Descripción a grandes rasgos del caso de uso o resumen.
Tipo:	Primario, secundario o opcional. Categoríza la importancia del caso de uso, esta categorización es útil al momento de desarrollar o implementar en el sistema los casos de uso. De esta manera los casos primarios serán atacados en un primer ciclo del desarrollo del sistema, después los secundarios y finalmente los opcionales.
Referencias cruzadas	Casos de uso relacionados o funcionalidad expresada en un documento de requerimientos.

La sección intermedia, **"Curso típico de eventos"** es el corazón de la descripción del caso de uso, describe en detalle la conversación de interacción entre los actores y el sistema. Un aspecto crítico de esta sección es que describe la secuencia de eventos más común o típica. Las situaciones alternativas no son descritas en esta sección.

La sección final, **"Curso alternativo de eventos"**, describe alternativas importantes o excepciones que pueden presentarse durante la ejecución típica de eventos. Si es complejo algún curso alternativo, se debe describir por sí mismo como otro caso de uso"[19].

2.5 Realización de los casos de uso

Los casos de uso son descripciones de la funcionalidad del sistema independientes de su implementación. Después de especificar la funcionalidad que deberá desempeñar el sistema hay que modelar como será "realizada" dicha funcionalidad, es decir como serán realizados los casos de uso.

Los principios de UML para realizar los casos de uso son:

- Un caso de uso es realizado en una colaboración: Una colaboración muestra una implementación interna del caso de uso en términos de clases/objetos y sus relaciones (llamadas el contexto de colaboración) y la interacción para alcanzar la funcionalidad deseada (llamada la interacción de colaboración). El símbolo de una colaboración es una elipse que contiene el nombre de la colaboración dentro o debajo de ella.

[19] Larman, Craig. Appling UML and Patterns. An Introduction to Object-Oriented Analysis and Design. p. 51-52.

- Una colaboración es representada en UML como un conjunto de diagramas mostrando tanto el contexto como la interacción entre los participantes en la colaboración: Los participantes en la colaboración son un conjunto de clases (ó los objetos si se habla de la instancia de la colaboración). Los diagramas utilizados son: de colaboración, secuencia y actividad. El tipo de diagrama utilizado para describir una pintura completa de una colaboración depende del caso de uso. En algunos casos, un diagrama de colaboración puede ser suficiente; mientras que en otros, una combinación de diferentes diagramas puede ser necesaria.
- Un escenario es una instancia de un caso de uso o una colaboración: El escenario es una ejecución específica (un flujo especifico de eventos) que representa una instancia específica del caso de uso (un uso del sistema). Cuando un escenario es visto como un caso de uso, solo el comportamiento externo del sistema en relación a los actores es descrito. Cuando un escenario es visto como una instancia de una colaboración, la implementación interna de las clases involucradas, sus operaciones y la comunicación entre ellas es descrita.

La tarea de realizar un caso de uso consiste en transformar los diferentes pasos y acciones descritas en los casos de uso en clases, operaciones de las clases, y las relaciones entre ellas. Esto es descrito como alojar la responsabilidad de cada paso en el caso de uso dentro de las clases participantes de la colaboración. En este punto, es encontrada una solución que implementa el comportamiento externo descrito en el caso de uso, en términos de una colaboración dentro del sistema. (Ver la figura siguiente)

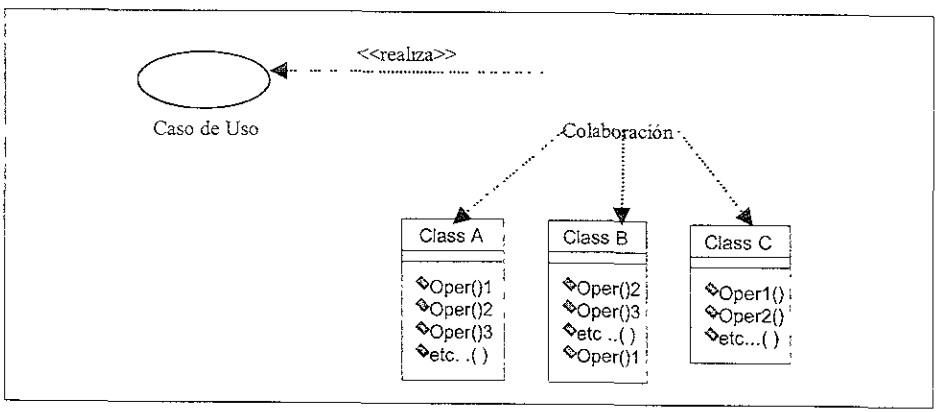

Figura 2-7 *La realización de un caso de uso expresada en una colaboración en donde diferentes clases participan en la realización del caso de uso.*

Cada paso en la descripción del caso de uso es transformada en operaciones de las clases que participan en la colaboración. Un paso en el caso de uso es transformado en un conjunto de operaciones dentro de las clases; comúnmente no son relaciones uno a uno entre un paso en la descripción del caso de uso y una operación de las clases que participan en la colaboración. También es importante notar que una clase puede participar en la realización de diferentes casos de uso. La responsabilidad total de la clase es la integración de todos los roles que juega en los casos de uso.

La relación entre un caso de uso y su implementación en términos de una colaboración es mostrada a través de cualquiera de las siguientes formas: Una *relación de refinamiento*, la cual es representada mediante una línea punteada con una flecha, o; Mediante una hiperliga que es invisible en una herramienta CASE. Una hiperliga en una herramienta CASE hace posible navegar entre la vista del diagrama de casos de uso y la vista del diagrama de colaboración. Las hiperligas también son usadas para navegar entre un caso de uso y un escenario (típicamente cualquiera de los modelos dinámicos: diagramas de actividad, diagramas de secuencia, o diagramas de colaboración).

Alojar o delegar las responsabilidades a las clases exitosamente es una tarea que requiere de cierta experiencia. Y como siempre, cuando la orientación a objetos está envuelta, este trabajo es una trabajo altamente iterativo. El desarrollador prueba varias posibilidades, mejorando gradualmente su solución hasta que el modelo desempeñe la funcionalidad y sea lo suficientemente flexible para permitir cambios futuros.

Jacobson utiliza un método para delegar las responsabilidades a las clases. Utiliza tres tipos de estereotipos de objetos (clases): objetos de frontera, objetos de control y objetos de entidad. Para cada caso de uso, estos objetos son usados para describir una colaboración que implemente el caso de uso. La responsabilidad de cada uno de estos estereotipos es la siguiente:

- *"Objetos de Interfaz:* Este tipo de objetos caen cerca de la frontera del sistema (pero aún dentro del sistema). Son a través de los cuales los actores externos se comunican con el sistema intercambiando mensajes entre los actores y otros objetos dentro del sistema"[20].
- *"Objetos de entidad:* Este tipo de objetos representan una entidad del dominio del sistema. Son típicamente pasivos en el sentido que no inician interacciones por sí solos. En sistemas informáticos, este tipo de objetos son normalmente persistentes y almacenados en una base de datos. Este tipo de objetos participan típicamente en muchos casos de uso"[21].
- *"Objetos de control:* En este tipo de objetos se pone la funcionalidad que es difícil de situar en cualquiera de los otros dos tipos de objetos. Normalmente este tipo de objetos controlan la interacción entre un grupo de objetos"[22].

Diferentes métodos sugieren diferentes maneras de asignar las responsabilidades a las clases. Unos métodos sugieren que se haga primeramente un análisis del dominio, que muestre todas las clases del dominio con sus relaciones y que después el desarrollador tome cada caso de uso y determine la responsabilidad de las clases encontradas dentro del modelo de análisis, algunas veces modificando o agregando nuevas clases. Otros métodos sugieren que los casos de uso se conviertan en la base para encontrar las clases, por lo que el modelo de análisis del dominio es hecho gradualmente mientras se asignan las responsabilidades.

---

[20] Jacobson, Ivar. [et. al.] Object-Oriented Software Engineering. p. 176.

[21] Idem. p.184.

[22] Idem. p. 190-191.

Es importante recordar y hacer énfasis en que el modelado de sistemas es un trabajo iterativo. Cuando las responsabilidades son asignadas a las clases, se pueden encontrar errores u omisiones en los diagramas de clases lo cual significa una modificación de dichos diagramas. Nuevas clases serán encontradas para soportar los casos de uso. En algunos otros casos, puede ser inclusive necesario modificar el diagrama de casos de uso, debido a un mejor entendimiento del sistema por parte del desarrollador que descubre que un caso de uso fue descrito incorrectamente

Los casos de uso nos ayudan a concentrarnos en la funcionalidad del sistema. Un problema con algunos métodos de orientación a objetos que no utilizan los casos de uso es que se concentran en las estructuras estáticas de las clases y objetos (a veces llamado modelo conceptual) mientras que ignoran la funcionalidad y el aspecto dinámico del sistema que esta siendo desarrollado

CAPÍTULO 3

3. EL MODELO ESTÁTICO

El *modelo estático del sistema, también llamado vista estática*, "es el modelo fundamental de UML. Los elementos de este modelo son los conceptos fundamentales de la aplicación que incluyen conceptos del mundo real, conceptos abstractos, conceptos de implementación, conceptos computacionales y todos los tipos de conceptos encontrados en un sistema de software"[23].

En el modelado orientado a objetos, las clases, objetos y sus relaciones son los elementos de modelado primarios. Las clases y los objetos modelan lo que existe dentro del sistema que se está tratando de describir y las relaciones entre ellos revelan como están estructurados en términos de los demás objetos que componen el sistema.

El modelo estático o modelo de objetos captura la estructura estática de un sistema mostrando los objetos en el sistema, las relaciones entre ellos y los atributos y operaciones que caracterizan cada clase de objetos. El modelo estático es el más importante, porque enfatiza la construcción de un sistema alrededor de objetos, más que alrededor de la funcionalidad del mismo. Un modelo orientado a objetos corresponde más cercanamente al mundo real que un modelo orientado a las funciones o procesos de un sistema y consecuentemente es más perecedero a los cambios. El modelo de objetos provee una representación gráfica intuitiva del sistema y es valioso para la comunicación con el cliente y la documentación de la estructura del sistema.

---

[23] Rumbaugh, James. [et al.] The Unified Modeling Language Reference Manual. p. 41.

3.1 Clases y Objetos

El propósito fundamental del modelado orientado a objetos es precisamente describir objetos. Juan Pérez, un Ferrari rojo placas 123 ABC, la casa #5 de la calle de Dalias, la ventana de la sala de dicha casa, una computadora Compac Proliant 7000 No. de serie 9855445, son todos ellos ejemplos de objetos. Un objeto es simplemente algo que tiene sentido en el contexto de una aplicación.

Definimos **objeto** como un concepto, abstracción o cosa dentro de las fronteras y significado del problema que tenemos en mano. Los objetos tienen dos propósitos: Promover un entendimiento del mundo real y servir como base práctica para la implementación computacional.

La palabra "objeto" es utilizada en casi todos los contextos. En el modelado orientado a objetos un objeto es una entidad capaz de conservar un estado (información) y que ofrece una serie de operaciones (comportamiento) que examinan o afectan su estado. Un objeto esta caracterizado por un conjunto de operaciones y un estado que recuerda el efecto de esas operaciones.

Todos los objetos tienen una identidad y son distinguibles, dos figuras con el mismo color, forma y textura son aún individuos u objetos diferentes, dos personas gemelas, son también individuos diferentes, aún cuando puedan parecerse mucho.

Booch propone la siguiente definición para un objeto: "Un objeto tiene estado, comportamiento e identidad; la estructura y comportamiento de objetos similares es definida en su clase común; los términos objeto e instancia son intercambiables"[24]. Para completar esta definición a continuación se presentan las definiciones de estado, comportamiento e identidad que el mismo Booch propone.

- *"Estado:* El estado de un objeto consta de todas las propiedades de ese objeto más su valor actual para cada una de esas propiedades"[25].
- "*Comportamiento:* Es cómo un objeto actúa y reacciona en términos de sus cambios de estados y transferencia de mensajes"[26]
- "*Identidad:* Es aquella propiedad de un objeto que lo hace distinguible entre todos los demás objetos"[27]

Una **clase** describe un grupo de objetos con propiedades similares (atributos), comportamiento en común (operaciones) y relaciones en común con otros objetos.

"Una clase es un conjunto de objetos que comparten una estructura y un comportamiento en común"[28].

---

[24] Booch, Grady. Object-Oriented Analysis and Design, with Applications. p. 83.

[25] Idem. p. 84.

[26] Idem. p. 86.

[27] ídem p. 91.

Persona, Carro, Casa, Ventana, Computadora son ejemplos de clases.

Los objetos de una clase tienen los mismos patrones en cuanto atributos y comportamiento, aunque cada objeto o "instancia de la clase" contiene sus propios valores para sus atributos y se relaciona con otras clases de manera diferente Sin embargo, pueden existir objetos cuyos valores de sus atributos y relaciones con otros objetos sean iguales.

Todos los objetos de una clase comparten un propósito semántico en común

Una clase es una descripción de un tipo de objeto. Todos los objetos son instancias de una clase, donde la clase describe las propiedades y comportamiento de un tipo de objeto. La relación de un objeto con una clase es similar a la de una variable que se relaciona con un tipo de variable (entero, cadena, etc).

Cuando modelamos y construimos sistemas de información, máquinas, u otros sistemas, los conceptos del dominio del problema deben ser utilizados para hacer los modelos entendibles y fáciles de comunicar Si construimos un sistema para una compañía de seguros, el sistema debe estar basado en los conceptos de una aseguradora. Un sistema basado en los conceptos primarios de un negocio puede ser fácilmente rediseñado para cumplir con las nuevas reglas, estrategias, etc., porque solo se tendrán que hacer los ajustes necesarios entre el nuevo y viejo negocio. Cuando los modelos están basados en como se ven y actúan sus contrapartes del mundo real, y cuando dichas contrapartes aparecen en el dominio del problema, la orientación a objetos cuadra perfectamente. La base de la orientación a objetos son las clases, objetos y las relaciones entre ellos.

Ejemplos de clases en un sistema de algún negocio son: Cliente, Contrato, Factura, Deuda, etc

Ejemplos de clases en un sistema operativo de software son: Archivo, programa ejecutable, dispositivo, icono, ventana, etc.

3.2 Encontrando Clases

Encontrar las clases es un trabajo altamente creativo y debe ser realizado por expertos en el dominio del problema. Las clases deben venir del dominio del problema y sus nombres deben representar lo que en la vida real representan. Cuando se está buscando por las clases las siguientes preguntas podrían ser útiles para identificarlas:

- ¿Existe información que debe ser guardada o analizada? Si existe cualquier información que deba ser almacenada, trasformada, analizada o manejada de alguna manera, entonces dicha información es posible candidata a ser una clase. La información pueden ser conceptos que deban ser registrados en el sistema o eventos o transacciones que ocurran en un momento especifico.

[28] Booch, Grady Object-Oriented Analysis and Design with Applications p 103.

- ¿Existen sistemas externos? Si los hay, dichos sistemas son normalmente de interés en el modelo. Los sistemas externos pueden ser vistos como clases que contiene el sistema a modelar o clases con las que se debe interactuar.
- ¿Existen patrones, librerías de clases, componentes, etc. de proyectos anteriores? Si existen entonces son candidatas a ser clases
- ¿Hay dispositivos que el sistema deba manejar? Cualquier dispositivo conectado al sistema es candidato a convertirse en una clase que maneje dicho dispositivo.
- ¿Están involucradas las partes de la organización? Normalmente las estructuras organizacionales suelen convertirse en clases, especialmente en los sistemas para los negocios.
- ¿Qué roles juegan los actores en el sistema? Estos roles pueden ser vistos como clases; por ejemplo: el usuario, el operador del sistema, el cliente, etc.

Si se cuenta con una especificación de requerimientos o un análisis del negocio, deben ser utilizados junto con el modelo de casos de uso, como base para el descubrimiento de clases.

3.3 Diagrama de clases

"Un diagrama de clases es la representación gráfica de la vista estática del sistema, la cual muestra una colección de elementos de modelado estáticos como lo son las clases, tipos y sus contenidos, y las relaciones existentes entre estos elementos"[29]. Aunque tiene sus similitudes con los modelos de datos, hay que recordar que las clases no solo muestran las estructuras de la información, sino que también describen comportamiento. Uno de los propósitos del diagrama de clases es servir como fundamento para otros diagramas donde otros aspectos del sistema son mostrados (como los estados de los objetos y la colaboración entre ellos, que son mostrados en los diagramas dinámicos). Una clase en un diagrama de clases puede ser directamente implementada en un lenguaje de programación orientado a objetos.

Para crear un diagrama de clases, primeramente hay que identificar y describir las clases, cuando ya se cuenta con un conjunto de clases, estas pueden ser relacionadas con otras utilizando un conjunto de relaciones que mas adelante se describirán. Una clase es representada con un rectángulo, dividido en tres compartimentos: el compartimiento del nombre, el compartimiento de los atributos y el comportamiento para las operaciones como se muestra en la siguiente figura.

[29] Rumbaugh, James. [et. al.] The Unified Modeling Language Reference Manual. p. 190

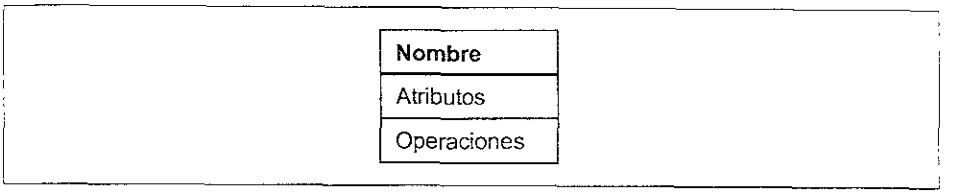

Figura 3-1 *Notación de una clase en UML.*

3.3.1 Compartimientos de una clase

3.3.1.1 Compartimiento del Nombre

El compartimiento superior del rectángulo de una clase es donde se especifica el nombre de la clase; el nombre de la clase es escrito con letras negritas y centrado. Nuevamente, el nombre debe ser derivado del dominio del problema y debe ser lo menos ambiguo posible. Debido a esto debe ser un sustantivo, por ejemplo: factura, deuda. El nombre de una clase no debe tener un prefijo o sufijo.

3.3.1.2 Compartimiento de los Atributos

Las clases tienen atributos que describen las características de los objetos. La Figura 3-2 muestra la clase Carro con los siguientes atributos: número de registro, color, modelo, velocidad y dirección Los atributos capturan la información que describe e identifica una instancia específica de la clase. Sin embargo, solo los atributos de interés para el sistema deben ser incluidos Aún más, el propósito del sistema también influye en que atributos deben ser usados.

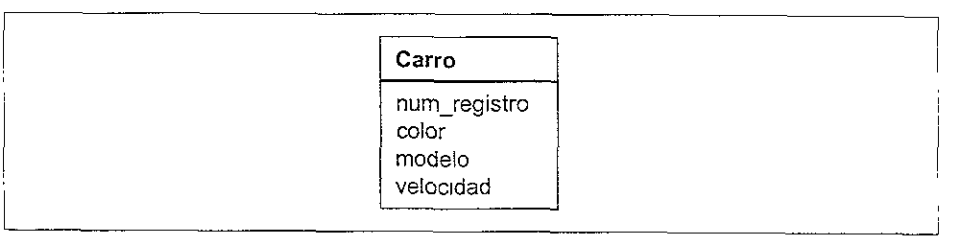

Figura 3-2 *La clase Carro con sus atributos: num_registro, color, etc.*

Un atributo tiene un tipo, el cual indica que tipo de atributo es, Los tipos de atributos típicos son: Entero, Booleano, Real, Puntero, Caracter y Enumeración, los cuales son llamados *tipos primitivos*. Dichos tipos dependen del lenguaje de programación, sin embargo, cualquier tipo puede ser usado, incluyendo otras clases.

Los atributos pueden tener diferente *visibilidad*. La visibilidad describe si el atributo podrá ser visto y podrá ser referenciado por otras clases. Si un atributo tiene la visibilidad de "publico", puede ser usado y visto por otras clases . Si un atributo tiene la visibilidad de "privado", no podrá ser accedido por otras clases. Otro atributo de visibilidad es "protegido", cuando un atributo es protegido solo las subclases de la clase donde se

encuentran podrán leerlo y utilizarlo. Adicionalmente existen diferentes tipos de visibilidad que son definidos para un lenguaje de programación en particular, pero los atributos de visibilidad publico y privado son los únicos necesarios en los diagramas de clases. Los atributos públicos son expresados con un signo mas (+) y los atributos privados son expresados con un signo menos (-). Si no es especificado ningún signo, significa que la visibilidad es indefinida (no hay visibilidad por default). Un atributo puede ser definido con un "ámbito de clase" lo que significa que será *compartido* (*shared*) por todas las instancias de ésa clase, este tipo de atributo es también llamado, variable de clase. Un atributo compartido es representado en UML, por el nombre subrayado del atributo.

Una propiedad de cadena puede ser utilizada para identificar explícitamente que valores son permitidos para un atributo. Esto es usado para especificar tipos enumerados como: color, status, dirección, etc. una propiedad de cadena es escrita dentro de llaves y los posibles valores de la enumeración son escritos utilizando una coma después de cada uno de ellos.

La sintaxis formal para la descripción de un atributo es la siguiente:

```
<visibilidad> nombre: tipo < = valor inicial <{propiedad de cadena}> >
```

Sólo el nombre y el tipo son obligatorios

En la siguiente figura se muestra una clase con sus atributos, el tipo de cada atributo y su visibilidad.

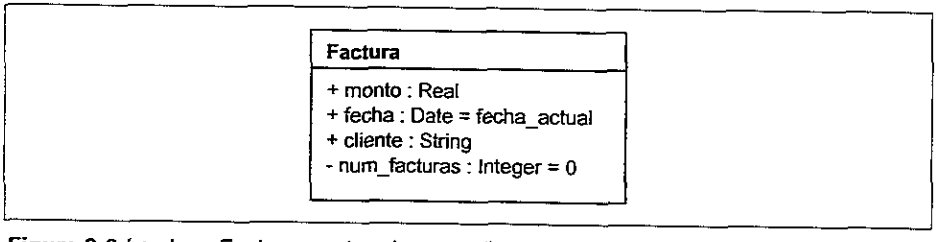

Figura 3-3 *La clase Factura mostrando sus atributos públicos y privados y el tipo de cada atributo.*

3.3.1.2.1 Implementación en Java

Una clase puede ser directamente implementada en un lenguaje de programación orientado a objetos como Java.

El siguiente código es la implementación de la clase de la Figura 3-3

```
public class Factura
{
      public double monto;
      public Date fecha = new Date();
      public String cliente;
      static private int num_facturas = 0;
```

```
        //Constructor, llamado cada vez que un objeto es creado
        public Factura()
        {
                //Otras instrucciones de inicialización
                num_facturas ++;
        }
        // Aquí irían otros métodos
}
```

3.3.1.3 Compartimiento de Operaciones

El tercero y último de los compartimientos de una clase es el destinado para las operaciones. Las operaciones son utilizadas para manipular los atributos de la clase, para interactuar con otras clases o para regresar valores a las clases que los llamaron o al usuario mismo. Las operaciones de una clase describen lo que una clase puede hacer, es decir, los servicios que puede ofrecer, por lo que pueden ser vistas como la interfase de la clase.

Una operación es descrita con un tipo de retorno, nombre y uno o más parámetros, estos tres datos juntos son llamados la signatura de la operación.

Como los atributos, las operaciones pueden tener visibilidad y ámbito (scope).

La sintaxis formal para describir una operación es la siguiente:

```
<visibilidad> nombre ( <lista de parámetros> ) <: expresión de tipo de
retorno <{ cadena de propiedades }>>
```

Donde cada uno de los elementos de la lista de parámetros utiliza la siguiente sintaxis

```
nombre : tipo de expresión < = valor por default>
```

La visibilidad es la misma que para los atributos (+ para pública, - para privada) No todas las operaciones requieren tener un valor de retorno, parámetros o cadena de propiedades, pero requieren tener siempre una signatura única. (tipo de retorno, nombre, parámetros)

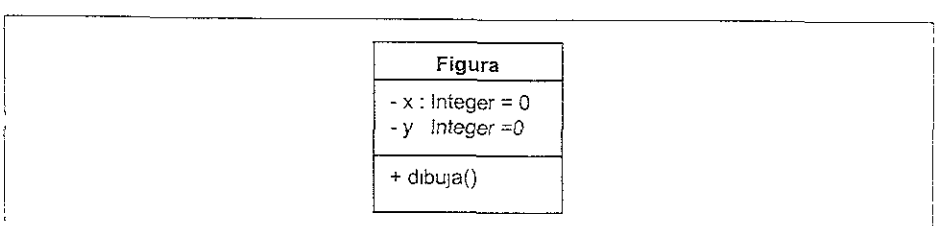

Figura 3-4 *La clase Figura, que cuenta con dos atributos y la operación dibuja()*

3.3.1.3.1 Implementación en Java

El código de Java para la implementación de la clase mostrada en la Figura 3-4 es el siguiente:

```
public class Figura
{
      private int x = 0;
      private int y = 0;
      // Constructor de la clase
public Figura()
{
      // Código de inicialización de la clase
      dibuja()
}
public void dibuja()
{
      // Código para dibujar la figura
}
public void escalarFigura( porcentaje : integer = 25)
{
      // Código para cambiar de tamaño la figura de acuerdo al porcentaje
}
public regresaPosición () : Coordenadas;
{
// Código para regresar las coordenadas de la figura, las cuales serán
regresadas   en un objeto del tipo Coordenadas
      }
}
```

3.3.2 Relaciones

Los diagramas de clases consisten de clases y las relaciones entre ellas. "Una relación es una conexión semántica entre elementos de modelado"[30]. Las relaciones que pueden existir son: asociaciones, generalizaciones, dependencias y refinamientos.

3.3.3 Asociaciones

Una asociación es una conexión entre clases, una conexión semántica (liga) entre objetos de las clases envueltas en la asociación. Una asociación es normalmente bidireccional, lo que significa que si un objeto esta asociado con otro, ambos conocen de la existencia del otro e interactúan entre si. Una asociación representa que los objetos de dos clases tienen una liga entre ellos, por ejemplo: que ellos "conocen acerca de los otros", "que están conectados a ", "por cada X existe un Y", etc. Las clases y asociaciones son muy útiles cuando se modelan sistemas complejos.

[30] Rumbaugh, James. [et. al.] The Unified Modeling Language Reference Manual. p. 411.

3.3.3.1 Asociaciones Normales

La asociación más común es solo una conexión entre clases. Es dibujada como una línea entre dos clases como se muestra en la Figura 3-5 La asociación tiene un nombre que se escribe cerca de la línea que representa la asociación, y que normalmente es un verbo, aunque los sustantivos o inclusive alguna frase también son permitidos. Cuando un diagrama de clases es modelado, debe reflejar el sistema que esta siendo construido, lo que significa que los nombres de las asociaciones deben ser los nombres de las relaciones de los objetos de la vida real

Es posible indicar la dirección de una asociación agregando una pequeña flecha al nombre de la asociación. La flecha indica que la asociación puede ser usada solo en la dirección de la flecha. Sin embargo, las asociaciones pueden tener dos nombres, indicando el nombre de la asociación para cada sentido de las flechas.

Para expresar cuantos objetos intervienen en una asociación, se utiliza la multiplicidad, un rango que indica cuantos objetos de cada clase están relacionados Los rangos pueden ser: cero a uno (0 .1), cero a muchos (0 * ó solo *), uno a muchos (1..*), dos (2), cinco a once (5..11), etc. No especificar la multiplicidad implica que la multiplicidad será de uno (1). La multiplicidad es escrita cerca del final de la asociación, en la clase donde es aplicable La Figura 3-5 muestra un ejemplo donde un carro puede tener uno o mas dueños y una persona puede ser dueña de cero o mas carros.

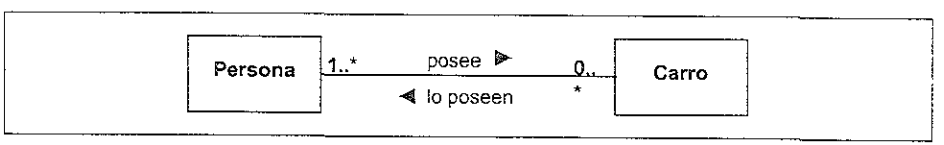

Figura 3-5 *Asociación entre la clase Persona y la clase Carro, mostrando la cardinalidad de la asociación.*

Cuando se modelan sistemas complejos es muy importante poder comunicar los resultados del modelo. Si comunicamos el resultado de un modelo será posible verificarlo, validarlo y consolidarlo. Un diagrama de clases es menos ambiguo que una descripción textual. Cuando se crea un diagrama, se deben realizar muchas decisiones que de otra manera no podrían ser realizadas. Inclusive un modelo pequeño contiene mucha información y es posible traducir el modelo en nuestro lenguaje habitual Por ejemplo, el modelo de la Figura 3-6 puede ser expresado de la siguiente manera:

- Una compañía de seguros tiene contratos de seguros, que se refieren a uno o muchos clientes.
- Un cliente tiene cero o muchos contratos de seguros, los cuales se refieren a una compañía de seguros.
- Un contrato es entre una compañía de seguros y uno o muchos clientes. El contrato de seguros se refiere a los clientes (uno o muchos) y a la compañía de seguros.
- El contrato de seguros es expresado en (cero o una) póliza de seguros
- La póliza de seguro se refiere a un contrato de seguros.

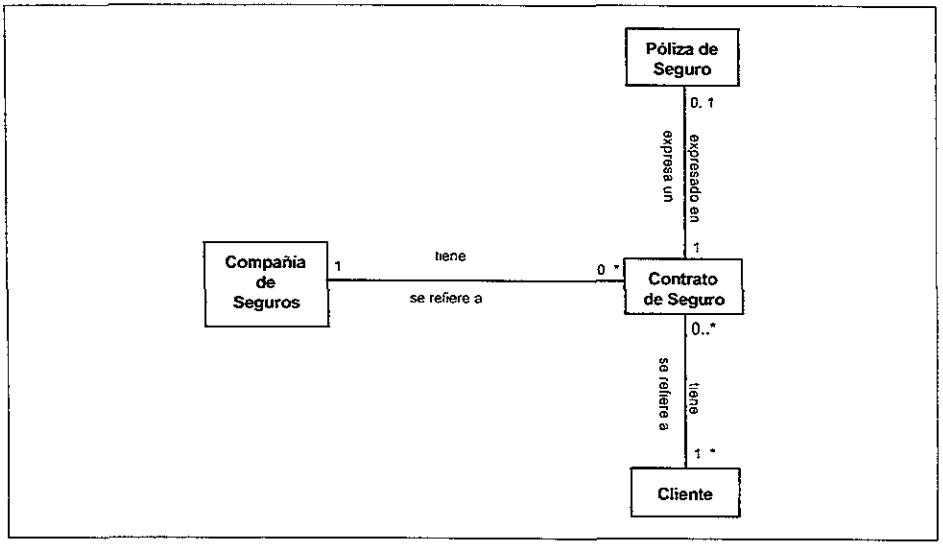

Figura 3-6 *Diagrama de clases de un sistema para una compañía de seguros.*

3.3.3.2 Asociación Recursiva

Es posible conectar a una clase a sí misma, lo que representa una conexión semántica entre objetos, pero esta vez serán los objetos de una misma clase. Una asociación de una clase a si misma es llamada una asociación recursiva y este tipo de asociaciones son utilizadas para modelar sistemas complejos.

A continuación se muestra un ejemplo de este tipo de asociaciones, en la Figura 3-7 se muestra un diagrama de clases de un sistema de red. Y en la Figura 3-8 se muestra un diagrama de objetos con una posible configuración de la red, donde se ejemplifica la relación de objetos de la misma clase.

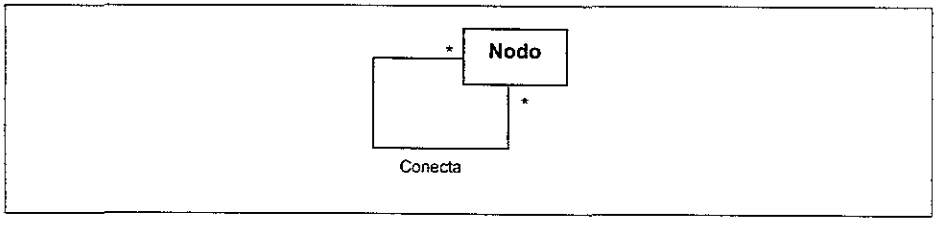

Figura 3-7 *Ejemplo de una asociación recursiva.*

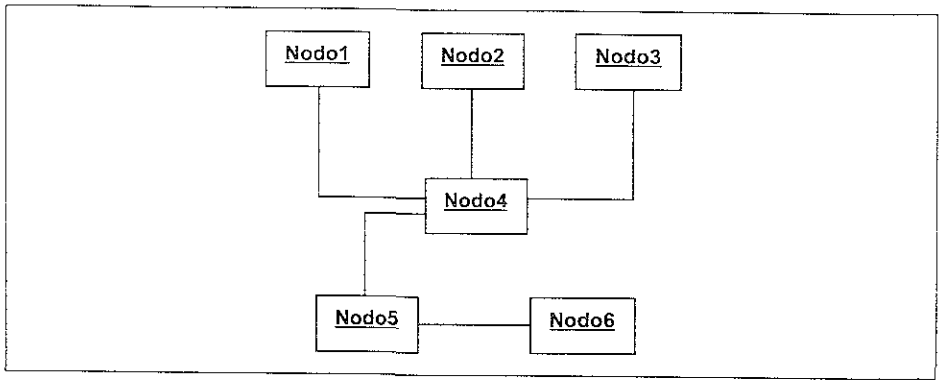

Figura 3-8 *Diagrama de objetos, que muestra un posible esenario de la asociación recursiva del diagrama anterior.*

3.3.3.2.1 Implementación en Java

A continuación se muestra la implementación en Java del siguiente diagrama.

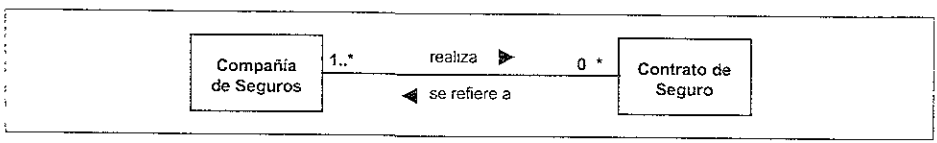

Figura 3-9 *Diagrama de clases que muestra la relación entre la clase "Compañía de Seguros" y la clase "Contrato de Seguro".*

```
// Compañia_seguros.java file
public class Compañia_Seguros
{
      /* Métodos */
      private Contrato_SeguroVector contracts;
}
// Contrato_Seguro.java file
public class Contrato_Seguro
{
      /* Métodos */
      private Compañia_Seguros refer_to,
}
```

Implementar una asociación bidireccional uno a uno o uno a muchos es sencillo, pero no es tan sencillo implementar una asociación muchos a muchos. Para esto hay que descomponer la asociación en dos asociaciones y utilizar una clase auxiliar de tipo "asociativa" para tener solo relaciones de uno a muchos, justo como sucede cuando se normaliza un diagrama entidad relación en Bases de Datos.

3.3.3.3 Roles en una Asociación

Una asociación puede tener roles conectados a cada clase envuelta en la asociación. El nombre del rol es una cadena colocada cerca del final de la asociación y cerca de la clase a la que le aplica ese rol. El nombre del rol indica el rol que juega la clase en términos de la asociación. Los roles son una técnica muy útil para especificar el contexto de una clase y de sus objetos. Los nombres de los roles son parte de la asociación y no parte de las clases. El uso de roles es opcional.

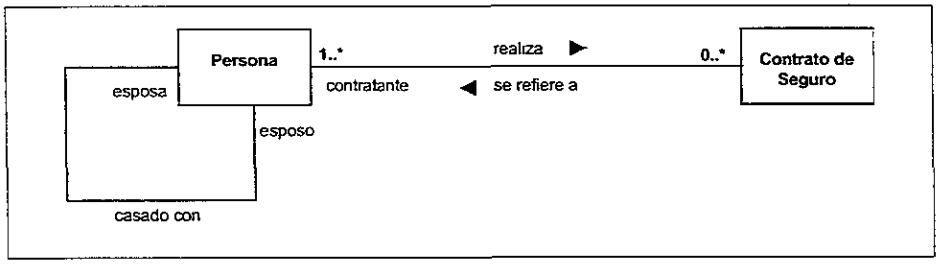

Figura 3-10 *Ejemplo de una asociación donde la clase Persona juega distintos roles en diferentes relaciones.*

En el ejemplo anterior se puede observar que una clase puede jugar distintos roles en diferentes asociaciones.

3.3.3.4 Clase de Asociación

Una clase puede ser vinculada a una asociación, en cuyo caso la clase es llamada "clase de asociación" (Association Class). Las clases de este tipo no están conectadas al final de ninguna asociación, están conectadas a la asociación en sí. Este tipo de clases tienen atributos, operaciones y otras asociaciones como cualquier clase normal. Las clases de asociación son usadas para agregar información adicional a la asociación; por ejemplo, la hora en que la asociación fue creada. Cada ocurrencia de la asociación está relacionada a un objeto de la clase de asociación.

3.3.3.5 Agregación (Aggregation)

Una agregación es un caso especial de asociación. La agregación indica que una clase está compuesta por otras y que las otras son partes de un todo. Un ejemplo de agregación es un carro que consiste de cuatro puertas, un motor, un chasis, una caja de velocidades, etc. Otro ejemplo, es un árbol binario que consiste de uno o dos árboles nuevos. Una agregación es usada comúnmente para describir diferentes niveles de abstracción. Las palabras claves para identificar una agregación son "consiste de", "contiene", "es parte de"; es decir, palabras que indican una relación en que las clases que intervienen son partes de un todo.

Una agregación es representada en UML con un diamante apuntando a la clase que contiene o consiste de las diferentes partes (el todo).

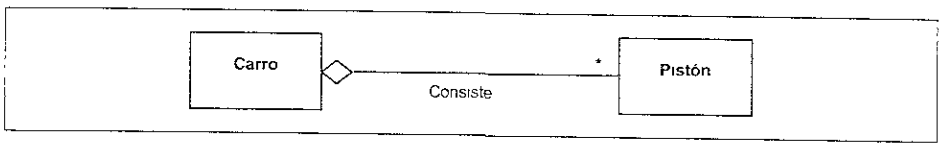

Figura 3-11 *Relación de agregación.*

3.3.3.6 AGREGACIÓN COMPARTIDA

Una agregación compartida (shared aggregation) es aquella en que las partes que componen la agregación pueden ser partes de diferentes todos. La agregación es compartida si la multiplicidad en la clase que es componente del "todo" es diferente de uno. La agregación compartida es un caso especial de una agregación normal.

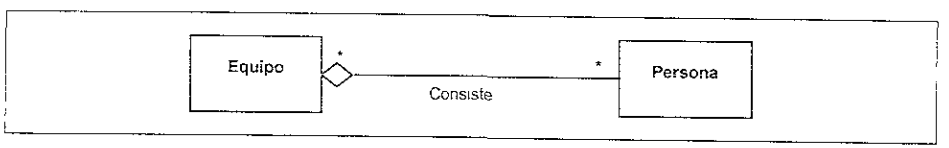

Figura 3-12 *Relación de agregación compartida, donde una persona puede formar parte de muchos equipos.*

3.3.3.7 AGREGACIÓN DE COMPOSICIÓN

Una agregación de composición es aquella en que las partes que componen el todo viven dentro de la clase que las agrupa (el todo). Es decir, que su creación y destrucción dependen de la clase que las agrupa. Este tipo especial de agregación es representado por un diamante relleno. La multiplicidad de este tipo de relaciones es de 0..* del lado de los componentes y no puede ser de mas de uno del lado de la clase que agrupa a los componentes.

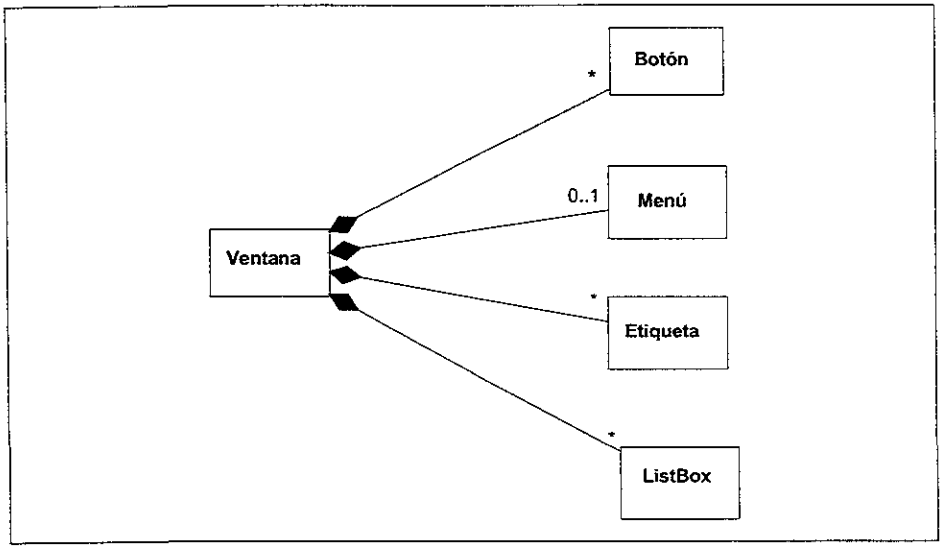

Figura 3-13 *Ejemplo de una agregación de composición, donde la clase Ventana está compuesta de botones, un menú, etiquetas y varias cajas de selección.*

3.3.4 Generalización

La definición de generalización en UML es la siguiente: "Una relación taxonómica[31] entre un elemento general y un elemento más específico. El elemento más específico es totalmente consistente con el elemento general y contiene información adicional. Una instancia del elemento más específico puede ser usada donde el elemento general es utilizado"[32]. Por esto, la generalización (algunas veces llamada herencia) permite que los elementos sean especializados en nuevos elementos, donde los nuevos elementos o extensiones pueden ser fácilmente manejados como elementos separados.

[31] La taxonomía es la ciencia de clasificar cosas.

[32] Rumbaugh, James. [et. al.] The Unified Modeling Language Reference Manual. p. 287

La generalización es usada para las clases y los casos de usos y algunos otros elementos como los paquetes. La generalización es usada únicamente en tipos, nunca en instancias (una clase puede heredar otra clase, pero un objeto nunca puede heredar otro objeto) aun cuando las instancias son afectadas indirectamente por la herencia de sus tipos. La relación de generalización es llamada algunas veces una relación de "es un" (is a), lo que permite decir que un elemento de una clase específica es un elemento específico de una clase genérica. Por ejemplo un carro "es un" vehículo, un gerente de ventas "es un" empleado, etc.).

Existen diferentes variantes de generalización, a continuación se describen cada una de ellas.

3.3.4.1 Generalización Normal

La generalización es una relación entre una clase general y una específica. La clase específica es llamada la subclase, la cual hereda las características de la clase general, llamada superclase. Los atributos, operaciones y todas las asociaciones son heredadas Los atributos y operaciones con visibilidad pública en la superclase serán públicos en la subclase también. Los miembros (atributos y operaciones) que tengan una visibilidad privada serán heredados, pero no serán accesibles dentro de la subclase. Para proteger los atributos u operaciones del acceso fuera de la superclase o de la subclase, se pueden marcar estos miembros como protegidos Un miembro protegido no puede ser accedido desde otras clases, pero esta disponible desde la clase y cualquiera de sus subclases Un miembro privado está representado por un signo menos precediendo a su nombre, un miembro publico utiliza un signo más (+) y un miembro protegido es precedido su nombre por un signo de número (#).

Una clase puede ser tanto una superclase como una subclase, si se encuentra en una jerarquía de clases Una jerarquía de clases es una gráfica dónde las clases están conectadas vía una relación de generalización Una clase puede estar heredada de otra clase (en cuyo caso, es una subclase) y al mismo tiempo puede heredar su comportamiento a otra clase (en cuyo caso, es una superclase).

La generalización es representada en UML mediante una línea que parte de la clase más específica y llega a la clase más general, en el extremo de la clase más general se dibuja un triangulo que apunta hacia la clase más general

```
        }
}
```

3.3.5 Relaciones de dependencia y refinamiento

Además de las relaciones de asociación y generalización, existen en UML dos tipos más de relaciones, relaciones de dependencia y relaciones de refinamiento. La relación de dependencia es una conexión semántica entre dos elementos de modelado, un elemento independiente y un elemento dependiente. Un cambio en el elemento independiente afecta al elemento dependiente. Como en el caso de las relaciones de generalización, un elemento de modelado puede ser una clase, un caso de uno, un paquete, etc. Algunos ejemplos de dependencia son: Cuando una clase toma un objeto de otra clase como parámetro; Cuando una clase accede a un objeto global de otra clase, o cuando una clase llama una operación de ámbito (class scope operation) de otra clase. En todos estos casos hay una dependencia de una clase a la otra, aún cuando no hay una asociación explícita entre ellas.

La relación de dependencia es especificada en UML como una línea punteada con una flecha (y posiblemente una etiqueta) entre los elementos de modelado.

Figura 3-16 *Relación de dependencia entre clases. El tipo de dependencia se representa como un estereotipo.*

Una relación de refinamiento es una relación entre dos descripciones de la misma cosa, pero en diferentes niveles de abstracción. Una relación de refinamiento puede ser entre un tipo y una clase que la realiza, en cuyo caso es llamada una realización. Otras relaciones de refinamiento son las que ocurren entre las clases del análisis y las clases del diseño, o entre una descripción de alto nivel y una descripción de bajo nivel (por ejemplo la relación entre un diagrama general de una colaboración y el diagrama detallado de esa misma colaboración). Las relaciones de refinamiento pueden también ser utilizadas para modelar diferentes implementaciones de la misma cosa (una que es una implementación simple y una implementación más compleja, pero a la vez más eficiente).

Una relación de refinamiento en UML es representada con una línea punteada con un triangulo hueco entre los dos elementos de modelado.

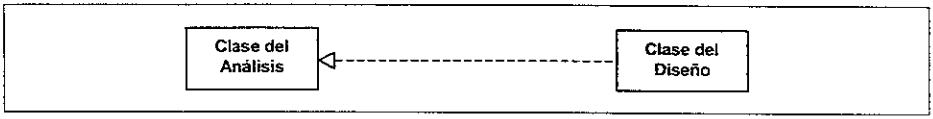

Figura 3-17 *Relación de refinamiento.*

Las relaciones de refinamiento son utilizadas en la coordinación de un proyecto, en proyectos muy grandes en los que diferentes modelos deben ser coordinados. La coordinación de modelos puede ser usada para:

- Mostrar como están relacionados los modelos en los diferentes niveles de abstracción.
- Mostrar como están relacionados los modelos en las diferentes etapas de un proyecto (especificación de requerimientos, análisis, diseño, implementación, etc.).
- Colaborar en el seguimiento de un modelo

3.3.6 Interfases

"Una interfase es un conjunto nombrado de operaciones que caracteriza el comportamiento de un elemento"[33].

Un paquete, componente o clase que tiene una interfase conectada a él se dice que implementa o soporta la interfase especificada, lo que implica que soporta el comportamiento definido en la interfase. Las interfase juegan un rol importante cuando se construyen sistemas bien estructurados y pueden ser vistas como contratos de colaboración entre un conjunto de elementos de modelado. Una interfase es descrita sólo con operaciones abstractas, que son un conjunto de signaturas que juntas especifican el comportamiento que un elemento puede elegir soportar al implementar la interfase.

La interfase es mostrada como un pequeño circulo con un nombre. La interfase es conectada al elemento de modelado mediante una línea (realmente es una asociación que siempre tiene una multiplicidad 1:1) Una clase que usa la interfase como implementación en una clase especifica es conectada vía una relación de dependencia al círculo de la interfase. La clase dependiente es solo dependiente de las operaciones de la interfase especifica, y de nada más en la clase (en cuyo caso, la dependencia debería de ir directamente a la clase). La clase dependiente puede llamar las operaciones publicadas en la interfase, las cuales no son mostradas directamente en el diagrama.

Para mostrar las operaciones en la interfase, la interfase debe ser especificada como una clase con el estereotipo <<interface>> usando el rectángulo normal de una clase.

Una interfase puede ser especializada como cualquier otra clase. La herencia entre interfases es representada en un diagrama de clases utilizando los mismos símbolos que para las clases Todas las interfases tienen el estereotipo <<interface>>.

[33] Rumbaugh, James [et al.] The Unified Modeling Language Reference Manual. p 310

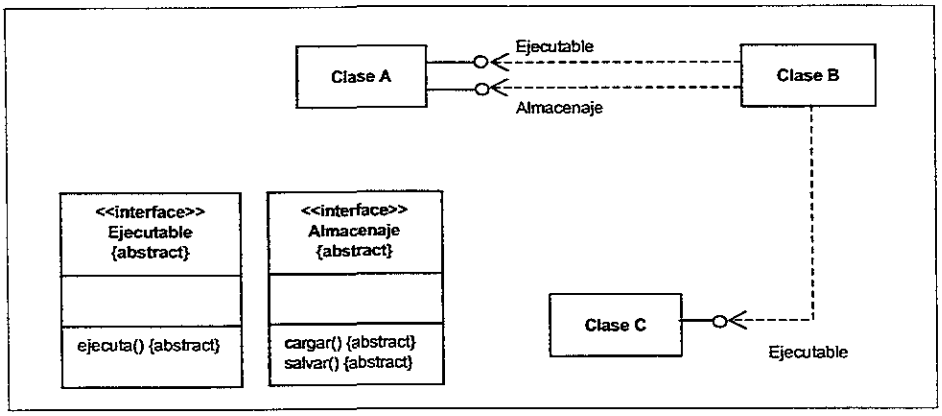

Figura 3-18 *En este ejemplo la "Clase A" implementa las interfases "Ejecutable" y "Almacenaje. La "Clase C" implementa la interfase "Ejecutable". La "Clase B" usa las interfases "Ejecutable" y "Almacenaje" de la "Clase A" y "Ejecutable" de la "Clase C". Las interfases son especificadas como clases con el estereotipo <<interface>> y contienen las operaciones abstractas que las clases que la utilizan deben implementar.*

3.3.6.1.1 Implementación en Java

```
interface Almacenaje
{
      public void Salvar();
      public void Cargar();
}
public class Person implements Almacenaje
{
      public void Salvar()
      {
            // Implementación de la operación de Salvado para una persona
      }
      public void Cargar()
      {
            // Implementación de la operación de Cargado para una persona
      }
}
```

3.3.7 Paquetes

Un paquete es un mecanismo de organización que permite agrupar cualquier tipo de elementos.

Un paquete se define en UML como: "Un mecanismo de propósito general para organizar elementos en grupos semánticamente relacionados"[34]. Todos los elementos de modelado que se encuentran dentro del paquete o referenciados por él son llamados contenido del paquete. Debido a que los paquetes son únicamente un mecanismo para la organización de un modelo, las instancias de un paquete no tienen ningún significado, debido a esto un paquete solo existe durante el trabajo de modelado y no es necesario traducirlo en código ejecutable durante la fase de codificación del sistema. Algunos métodos denominan a los paquetes como subsistemas.

Un paquete es dueño de sus elementos de modelado, lo que significa que un elemento de modelado no puede pertenecer a más de un paquete, sin embargo, los paquetes pueden importar elementos de modelado de otros paquetes. Cuando se importan elementos de un paquete a otro, es necesario indicar a que paquete pertenecen cuando quieran ser utilizados en el paquete a donde se importaron.

Como cualquier elemento de modelado, los paquetes cuentan con relaciones entre si. Sin embargo, las relaciones solo serán entre tipos, no entre instancias, debido a que como se mencionó anteriormente las instancias de los paquetes no tienen ningún significado semántico. Las relaciones permitidas para los paquetes son; dependencia, refinamiento y generalización.

Un paquete es representado como un rectángulo con una pequeña pestaña en su lado superior izquierdo simulando un separador (tab).

Un paquete tiene similitudes con una agregación. Si un paquete es dueño de su contenido, es una agregación de composición; y si cuenta con referencias de su contenido (es decir, importa sus elementos de otro paquete) es una agregación compartida (shared aggregation).

[34] Rumbaugh, James [et al.] The Unified Modeling Language Reference Manual. p. 378

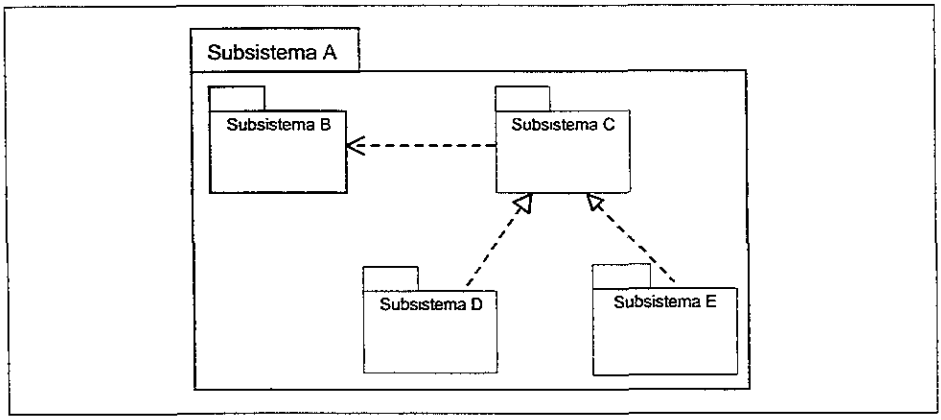

Figura 3-19 *Ejemplos de subsistemas en donde D y E son una especialización del subsistema C. B,C,D, y E son subsistemas contenidos en A. Y C depende de B.*

Un paquete tiene diferentes valores de visibilidad como las clases, determinando si otros paquetes tienen acceso a su contenido. En UML existen cuatro niveles de visibilidad para los paquetes: privada, protegida, pública e implementación. El valor de visibilidad por default es publico.

Un paquete puede tener una interfase que publique su comportamiento. La interfase es representada con un pequeño círculo conectado mediante una línea al paquete, como se hace con las clases. De esta manera una o más clases dentro del paquete pueden implementar la interfase.

CAPÍTULO 4

4. EL MODELO DINÁMICO

Todo sistema tiene una estructura estática y un comportamiento dinámico, UML cuenta con diagramas para capturar y describir ambos aspectos. Como ya se explicó en el capítulo anterior, los diagramas de clases son usados para documentar y expresar la estructura estática de un sistema –las clases, objetos y sus relaciones–. Mientras los diagramas de estado, secuencia, colaboración, y actividad son utilizados para expresar el comportamiento dinámico de un sistema, para modelar como interactúan los objetos en diferentes momentos durante la ejecución de un sistema.

Como se mencionó en el capitulo anterior los diagramas de clases modelan las cosas físicas o inteligibles y las relaciones entre ellas. Describiendo la estructura estática del sistema podemos saber qué cosas contiene el sistema y cómo están relacionadas entre si, pero el diagrama de clases no nos permite saber como esas cosas cooperan entre si para manejar sus tareas y brindar la funcionalidad del sistema.

Los objetos dentro del sistema se comunican entre si mediante mensajes. Por ejemplo: el objeto cliente Juan Pérez envía un mensaje de que desea comprar algo al objeto vendedor Bill. Un mensaje es típicamente una llamada a una operación que un objeto invoca de otro. Cómo se comunican los objetos y los efectos de dicha comunicación es conocido como el comportamiento dinámico del sistema. Además de esta colaboración entre objetos, el cambio de estados de dichos objetos, también constituye el comportamiento dinámico de un sistema.

La comunicación entre un conjunto de objetos con el propósito de brindar alguna funcionalidad, es conocida como interacción, la cual puede ser descrita por tres tipos de

diagramas: diagramas de secuencia, colaboración o actividad. En este capítulo se presentan estos tres tipos de diagramas, además del diagrama de estados, el cual como ya se mencionó también constituye parte del modelo dinámico.

- *Diagrama de estados*: Describe que estados puede tener un objeto durante su tiempo de vida, el comportamiento de esos estados y los eventos que pueden causar los cambios de estado; por ejemplo: una factura puede ser pagada (estado de pagada) o no pagado (estado de no pagada).
- *Diagrama de secuencia*: Describe como interactúan y se comunican los objetos. El enfoque primario en un diagrama de secuencia es el tiempo. Estos diagramas muestran como son enviados y recibidos una secuencia de mensajes entre un conjunto de objetos para desempeñar una función.
- *Diagrama de colaboración*: También describe como interactúan los objetos, pero en estos diagramas el aspecto principal es el espacio. Esto significa que las relaciones entre los objetos (en el espacio) son de particular interés y por eso son mostradas explícitamente en los diagramas.
- *Diagrama de actividad*: Es otra manera de mostrar interacciones, pero su enfoque principal es en el trabajo. Cuando los objetos están interactuando entre sí, desempeñan trabajo en términos de actividades, estas actividades y su orden son descritas en los diagramas de actividad.

Debido a que los diagramas de secuencia, colaboración y actividad todos muestran interacciones, comúnmente es necesario tomar una decisión acerca de que diagrama utilizar cuando se está documentando una interacción. La decisión depende de que aspecto se considere el más importante.

4.1 Interacción entre objetos (mensajes)

En la programación orientada a objetos, la interacción entre dos objetos es desarrollada como un mensaje enviado de un objeto a otro, En este contexto, es importante que la palabra "mensaje" no sea tomada tan literalmente, en el sentido de que un mensaje es "enviado" como en un protocolo de comunicación. Un mensaje es muy comúnmente implementado por una simple llamada a una operación, cuando un objeto llama a una operación de otro objeto. Cuando la operación ha sido ejecutada, el control es regresado al objeto que llamó la operación junto con un valor de retorno.

Los mensajes son mostrados en todos los diagramas dinámicos (secuencia, colaboración, estado y actividad) como medios de comunicación entre objetos. Un mensaje es dibujado como una línea con una flecha entre el emisor y el receptor del mensaje. El tipo de flecha indica el tipo de mensaje. Los tipos de mensaje en UML son:

- *Simple:* Representan un flujo de control plano. Muestran como es pasado el control de un objeto a otro sin describir ningún detalle acerca de la comunicación. Este tipo de mensaje es utilizado cuando los detalles acerca de la comunicación no son conocidos o no se considera relevante en el diagrama. Es también usado para mostrar el retorno de un mensaje síncrono, es decir, es dibujado del objeto que atiende el mensaje de regreso al que lo llamó para mostrar que el control es pasado de regreso.

- *Síncrono*: Un flujo de control anidado, típicamente implementado como una llamada a una operación. La operación que maneja el mensaje es completada (incluyendo cualquier otro mensaje anidado que haya sido enviado como parte del mensaje) antes de que el objeto que llamo la operación continúe con la ejecución. El mensaje de retorno puede ser mostrado como un mensaje simple o puede ser implícito cuando el mensaje haya sido manejado.
- *Asíncrono*: Un flujo de control asíncrono, donde no hay un retorno explícito al objeto que llama la operación y donde el que envía continua ejecutándose después de enviar el mensaje sin esperar a que el mismo haya sido manejado.

Los mensajes simple y síncrono se pueden combinar un una sola línea de mensaje, con la flecha del mensaje síncrono en un extremo y la flecha simple de retorno en el otro extremo. Esto indica que el retorno es casi inmediato después de la llamada a la operación.

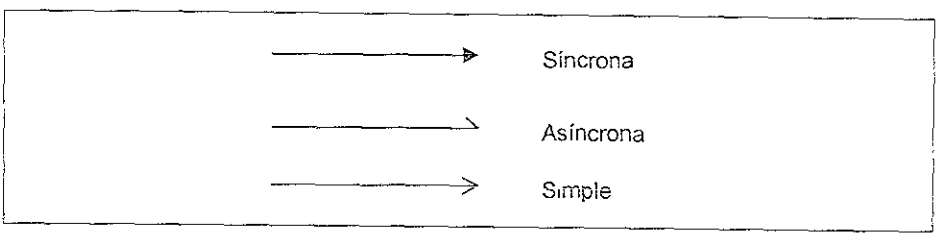

Figura 4-1 *Notación de los tipos de mensajes.*

4.2 Diagrama de estados

"Un diagrama de estados representa el ciclo de vida de los objetos, sistemas y subsistemas de un modelo, muestra los estados que un objeto puede experimentar y como son afectados dichos estados por los eventos (mensajes recibidos, condiciones cumplidas, errores, lapsos de tiempo, etc) que un objeto puede recibir"[35].

Las clases que tengan estados claramente identificables y un comportamiento complejo deben de contar con un diagrama de estados que especifique el comportamiento de los objetos de la clase y como difiere dicho comportamiento en cada uno de los estados de los objetos. Un diagrama de estados también ilustra que eventos harán cambiar de estado a los objetos de la clase.

4.2.1 Estados y transiciones

"Un estado describe un periodo de tiempo durante el tiempo de vida de un objeto de una clase. Puede estar caracterizado de tres maneras que pueden ser complementarias: por un conjunto de valores del objeto que son cualitativamente similares en algún aspecto, como un periodo de tiempo durante el cual un objeto está esperando a que un evento

[35] Martin, Fowler con Kendall, Scott UML Distilled: Applying the Standard Object Modeling Language p. 121

ocurra; o como un periodo de tiempo durante el cual un objeto está desempeñando una acción"[36].

Todo objeto cuenta con estados, un estado es el resultado de actividades previamente desarrolladas por el objeto y es típicamente determinado por los valores de sus atributos y las ligas hacia otros objetos. Una clase puede tener un atributo específico que indique su estado o dicho estado puede ser determinado por los valores de los atributos "normales" en el objeto.

Los siguientes son ejemplos de estados de objetos:

- La factura (el objeto) está pagada (el estado)
- El carro (objeto) está parado (estado)
- El motor (objeto) está encendido (estado)
- Juan (objeto) esta jugando el rol de cajero (estado)
- María (objeto) está casada (estado)

Un objeto cambia de estado cuando algo sucede, lo que es conocido como un evento; por ejemplo, alguien paga la factura, alguien comienza a manejar el carro, o María contrae nupcias.

En este contexto existen dos dimensiones de dinamismo: la interacción y los cambios internos de estado. La interacción describe el comportamiento externo del objeto y como interactúa con otros objetos (enviándoles mensajes o ligándose o desligándose de ellos). Los cambios internos de estado describen los cambios de comportamiento dentro del objeto, por ejemplo los cambios en los valores de sus atributos, y la ejecución de operaciones internas.

Los diagramas de estado tienen un punto de partida y pueden tener diferentes puntos de finalización. El punto de partida (estado inicial) es representado con un círculo negro y los puntos de finalización son representados por un círculo negro con una circunferencia a su alrededor. Los estados son representados mediante un rectángulo con las puntas redondeadas. Entre los estados están las transiciones de estados, las cuales son representadas por una línea con una flecha que indica la transición de un estado a otro.

Los estados pueden contener tres tipos de compartimientos. El primer compartimiento es obligatorio y es donde se muestra el nombre del estado, por ejemplo: pagado, detenido y en movimiento. El segundo compartimiento, es opcional y contiene las variables de estado y los valores asignados a dichas variables. El tercer compartimiento, también opcional, muestra las actividades y eventos asociados al estado.

Existen tres tipos de eventos estándar que pueden ser utilizados en el compartimiento de actividades: "entry" (entrada), "exit" (salida) y "do" (durante). El evento "entry" (de entrada) puede ser utilizado para especificar acciones a desarrollarse a la entrada en el estado, por ejemplo: asignar un valor a una variable o enviar un mensaje. El evento "exit" (de salida) puede ser utilizado para especificar acciones a desarrollarse en la salida del estado. El

[36] Rumbaugh, James. [et. al.] The Unified Modeling Language Reference Manual. p. 70.

evento "do" (durante) puede ser utilizado para especificar una acción a desarrollarse mientras el objeto se encuentra en el estado; por ejemplo: mandar un mensaje, esperar o calcular algo. La sintaxis formal para el compartimiento de actividades es la siguiente:

```
nombre_del_evento < lista de argumentos > / acción
```

El nombre del evento puede ser cualquier evento, incluyendo los eventos estándar. La acción indica que acción será realizada Y la lista de argumentos es opcional. Los eventos estándar no tienen ningún argumento

En la siguiente figura se muestra un estado con los tres compartimientos antes descritos.

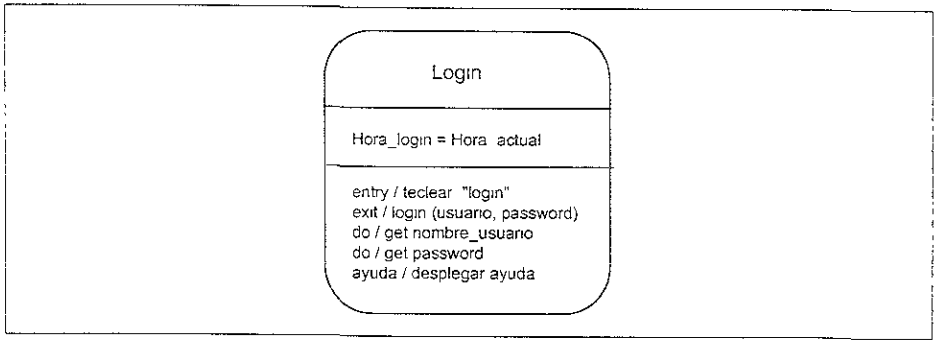

Figura 4-2 *Un estado llamado Login, donde a "Hora_login" se le asigna la hora actual, y donde algunas acciones son ejecutadas en los eventos estandar entry, exist y do El evento "ayuda / desplegar ayuda" es un evento definido por el usuario (en este caso el desarrollador).*

La transición entre estados es representada mediante una flecha que indica el flujo de un estado a otro Una transición de estado puede tener un evento asociado, pero no es obligatorio. Si tiene un evento asociado, la transición será ejecutada cuando el evento ocurra Una acción "durante" puede ser interrumpida por un evento asociado a una transición.

Si una transición de estado no tiene un evento asociado, dicha transición será disparada cuando las acciones internas del estado inicial relacionado hayan concluido.

La sintaxis formal para especificar una transición de estado es la siguiente:

```
signatura_del_evento < [ condición ] >  < /  expresión de acción >
```

donde la sintaxis de signatura del evento es la siguiente:

```
nombre_evento ( < parametro1, parametro2, ..> )
```

Los siguientes son ejemplos de signaturas de eventos:

```
dibuja (f: Figura, c: Color)
redibuja()
```

```
print(factura)
```

La **condición** es opcional y significa que la transición ocurrirá si y solo sí el evento expresado en la signatura se cumple y se cumple también la condición expresada. Los siguientes son ejemplos de condiciones:

```
[t = 15 sec]
[número_de_facturas = n]
retiro (cantidad) [saldo >= cantidad]
```

La **expresión de acción** es también opcional y es una expresión procedural ejecutada cuando la transición es disparada. Puede ser escrita en términos de operaciones y atributos dentro del objeto representado (el objeto que tiene los estados que se están modelando). También es posible tener más de una acción, en cuyo caso las acciones se separan por una diagonal (/). Dichas acciones serán ejecutadas de izquierda a derecha. Ejemplos de **expresión de acciones** son los siguientes:

```
incrementar() / n:= n+1 / m:= m+1
agregar(n) / sum: sum + n
```

A continuación se muestra un diagrama de estados que contiene los elementos aquí explicados.

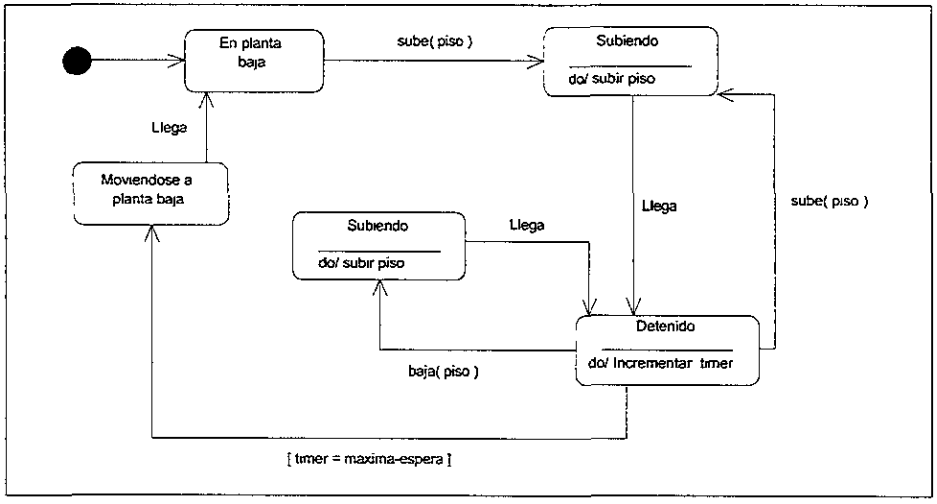

Figura 4-3 *Diagrama de estados de un elevador.*

4.2.2 Eventos

"Un evento es algo que sucede y que puede causar una acción. Ocurren en un punto en el tiempo y no tiene duración y es únicamente modelado cuando su ocurrencia tiene algún resultado o implicación"[37]. Por ejemplo, cuando alguien presiona el botón "Play" de un reproductor de CD, el reproductor empieza a tocar. El evento es el que alguien haya presionado el botón de Play y la acción es que el reproductor haya empezado a tocar.

En UML existen tres tipo de eventos:

- Los que ocurren cuando una condición se convierte en verdadera
- Los derivados de la recepción de una señal explícita enviada por otro objeto.
- Los que suceden cuando se cumple un periodo de tiempo.

Cabe señalar que los errores son también eventos y pude ser útil modelarlos. UML no da un soporte específico o explícito para la representación de eventos de error, pero pueden ser modelados como un evento con un estereotipo, como se muestra en el siguiente ejemplo:

```
<<error>> falta_de_memoria
```

Es importante conocer la semántica básica acerca de los eventos. Primero, los eventos son disparadores que activan la transición de estados; estos eventos son procesados uno a la vez. Si un evento puede activar más de una transición de estado, solo una de las transiciones será disparada (la que sea definida). Si un evento ocurre y la condición de la transición del estado es falsa, el evento es ignorado, es decir, no se guarda para dispararse cuando la condición se convierta en verdadera.

4.2.3 Implementación en Java

Los diagramas de estado son en algunos casos información redundante, dependiendo si se han especificado algoritmos para la especificación de las operaciones. En otras palabras, un comportamiento puede ser especificado dentro de los algoritmos de las operaciones o explícitamente en los diagramas de estados (o en ambos). Cuando un diagrama de estados es implementado en los lenguajes de programación orientados a objetos, son implementados directamente en algoritmos (sentencias de control, etc.) o son implementados con mecanismos separados, como una maquina de estados finitos o tablas de función.

[37] Rumbaugh, James. [et. al] The Unified Modeling Language Reference Manual p 68.

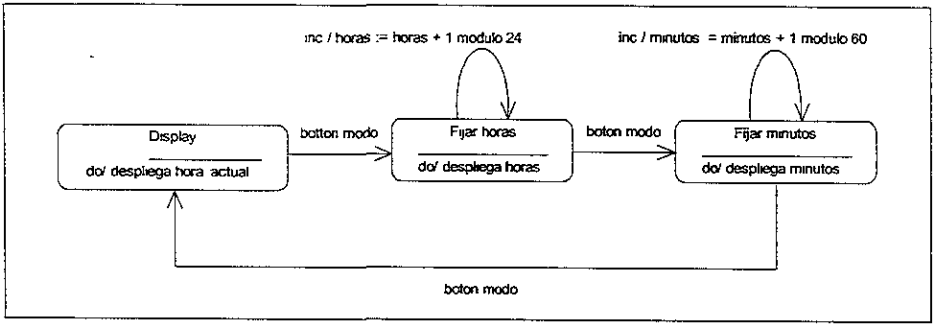

Figura 4-4 *Diagrama de estados de un reloj digital.*

El código de Java para implementar el diagrama de estado de la figura anterior se vería como sigue:

```
public class Estado
{
      public final int Display = 1;
      public final int Fijar_horas = 2;
      public final int Fijar_minutos = 3;
      public int valor;
}
public class Reloj
{
      private Estado estado = new Estado();
      private DisplayDigital LCD = new DigitalDisplay();
public Reloj()
{
            estado.valor = Estado.Display;
            LCD.desplegar_hora();
      }
      public void boton:modo()
      {
            switch (estado.valor)
            {
            case Estado.Display :
                  LCD.desplegar_hora();
                  estado.valor = Estado.Fijar_horas;
                  break;
            case Estado.Fijar_horas :
                  LCD.desplegar_horas();
                  estado.valor = Estado.Desplegar_minutos;
                  break;
            case Estado.Fijar_minutos :
                  LCD.desplegar_hora();
```

```
                estado.valor = Estado.Desplegar;
                break;
        }
        public void inc()
        {
                switch (estado.valor)
                {
                case Estado.Display :
                        ;
                        break;
                case Estado.Fijar_horas:
                        LCD.inc_hours();
                        break;
                case Estado.Fijar_minutos:
                        LCD.inc_minutos;
                        Break;
        }
}
```

4.3 Diagramas de secuencia

Los diagramas de secuencia ilustran como interactúan los objetos entre sí. Se concentran en las secuencias de mensajes en el tiempo, es decir, en como y cuando son enviados y recibidos los mensajes entre un conjunto de objetos, para lograr un fin específico Los diagramas de secuencia tienen dos ejes; el eje vertical muestra el tiempo y el eje horizontal muestra el conjunto de objetos. Un diagrama de secuencia también revela la interacción para un escenario específico –una interacción específica entre objetos que sucede en algún momento durante la ejecución del sistema (cuando una función específica es usada).

En el eje horizontal se encuentran los objetos que intervienen en la secuencia, cada objeto es representado por un rectángulo que contiene el nombre del objeto y/o de la clase subrayado. Debajo de cada objeto se dibuja una línea punteada en posición vertical, llamada la línea del tiempo de vida del objeto, la cual indica la ejecución del objeto durante la secuencia, es decir, los mensajes enviados o recibidos y la activación del objeto. La comunicación entre los objetos es representada mediante líneas horizontales de mensaje que viajan entre las líneas del tiempo de vida de los objetos. La flecha especifica si el mensaje es síncrono, asíncrono o simple.

Para leer el diagrama de secuencia, se comienza por la parte de arriba del diagrama y se va leyendo hacia abajo para ver el intercambio de mensajes que está tomando lugar mientras transcurre el tiempo.

4.3.1 Forma genérica y de instancia

Los diagramas de secuencia pueden ser utilizados de dos maneras diferentes: La forma genérica o la forma de instancia.

La forma de instancia describe un escenario específico en detalle; documenta una posible interacción. Esta forma no tiene ninguna condición, rama o ciclos.; muestra la interacción para solo el escenario escogido.

La forma genérica describe todas las posibles alternativas en un escenario, por lo cual en este tipo de diagrama se pueden encontrar ramas, condiciones y ciclos. Por ejemplo, el escenario de "abrir una cuenta" describiría todas las posibles alternativas: cuando la apertura es un éxito, cuando la apertura no es posible, cuando el dinero es inmediatamente depositado en la cuenta, etc.

"Un mensaje es una comunicación entre objetos que intercambian información con la esperanza de que una acción sea tomada. El recipiente de un mensaje es normalmente considerado un evento"[38]. Los mensajes pueden ser señales, llamadas a operaciones o algo similar (por ejemplo: Remote Procedure Calls en C++). Cuando un mensaje es recibido, inicia una actividad en el objeto receptor, lo cual es llamado activación. La activación muestra el objeto que tiene el control en ese momento, es decir, que objeto(s) se están ejecutando en un momento determinado. Un objeto activado puede estar ejecutando su propio código o esperando el retorno de otro objeto al cual le ha enviado un mensaje. La activación es representada en el diagrama mediante un rectángulo alargado sobre la línea de vida del objeto.

La línea del tiempo de vida representa la existencia de un objeto en un momento particular, es dibujada como una línea punteada que se extiende de arriba a abajo del diagrama. Los mensajes son dibujados como flechas (síncronas, asíncronas o simples) entre las líneas de vida de los objetos. Cada mensaje puede tener una signatura con un nombre y los parámetros, por ejemplo:

```
imprimir (file : File)
```

Los mensajes pueden también tener números de secuencia, los cuales indican explícitamente la secuencia de ejecución de los mismos. El retorno de un mensaje síncrono (como una llamada a una operación) es mostrado con una flecha simple, aunque no todos los retornos son siempre mostrados. Cuando deben ser mostrados los mensajes de retorno y cuando no, depende de si el mostrarlos clarifica la secuencia de eventos del diagrama, usualmente son mostrados únicamente cuando la llamada a otro objeto implica una secuencia de eventos antes de que el control sea regresado al objeto que llamó originalmente al método en cuestión.

Los mensajes pueden también tener condiciones. Una condición debe ser verdadera para que el mensaje sea enviado y recibido. Las condiciones son usadas para modelar ramificaciones o para decidir si enviar o no el mensaje. Si las condiciones son usadas para describir ramas, diferentes flechas de mensajes son dibujadas con las condiciones que ejecutan cada una, en otras palabras, solo un mensaje es enviado a la vez como se puede observar en la siguiente figura:

[38] Rumbaugh, James. [et. al.] The Unified Modeling Language Reference Manual. p. 333.

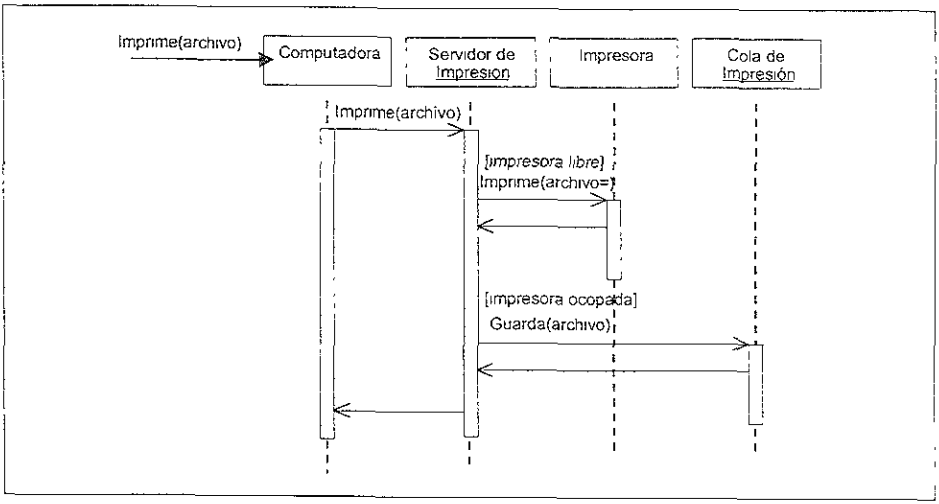

Figura 4-5 *Diagrama de secuencia con la interacción de objetos que intervienen en la impresión de un archivo, cuando se utiliza un servidor de impresión.*

4.3.2 Creación y destrucción de objetos

Los diagramas de secuencia pueden mostrar como son creados y destruidos los objetos como parte del escenario documentado. Un objeto puede crear a otro vía un mensaje. El objeto creado es dibujado con su símbolo de objeto puesto en el momento en el que es creado, es decir en el eje vertical del tiempo. El mensaje que crea o destruye un objeto es normalmente un mensaje síncrono (una flecha sólida), cuando un objeto es destruido, es marcado con una X larga; esto es indicado al dibujar la X en la línea del tiempo de vida del objeto en el punto en el que el objeto es destruido.

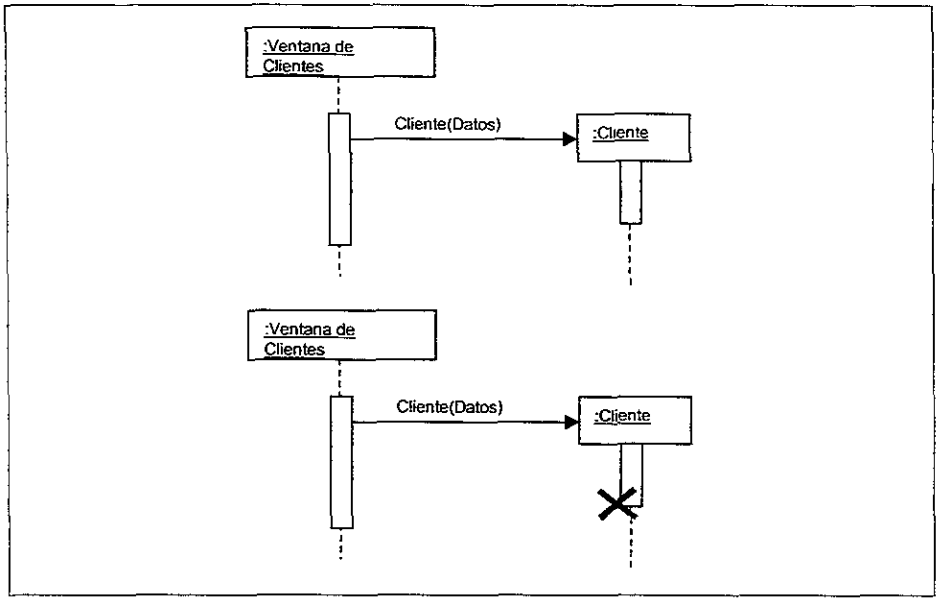

Figura 4-6 *Ejemplo de la creación y destrucción de un objeto en un diagrama de secuencia.*

4.4 Diagrama de colaboración

El diagrama de colaboración se enfoca en dos cosas fundamentalmente, la interacción y las ligas entre un conjunto de objetos colaborativos (una liga es una instancia de una asociación). Los diagramas de secuencia y colaboración ambos muestran interacciones, pero el diagrama de secuencia se enfoca principalmente en el tiempo mientras el diagrama de colaboración se enfoca principalmente en el espacio. Las ligas muestra los objetos y como están relacionados entre si. En este tipo de diagramas un objeto puede ser mostrado con su estructura interna. Como los diagramas de secuencia, los diagramas de colaboración pueden ser utilizados para ilustrar la ejecución de una operación, la ejecución de un caso de uso, o simplemente un escenario de interacción en el sistema.

Los diagramas de colaboración muestran objetos y sus ligas con otros, así como los mensajes que son enviados entre los objetos de la relación. Los objetos son dibujados de la misma manera que las clases, pero sus nombres son subrayados. Las ligas son dibujadas con líneas (las cuales son como asociaciones, pero sin multiplicidad). En una liga se puede agregar una etiqueta que indique entre otras cosas el número de secuencia para el mensaje.

"Numerar los mensajes hace más difícil de ver la secuencia de eventos que poner líneas verticales, como se hace en los diagramas de secuencia, pero en cambio la distribución

espacial permite mostrar otras cosas más fácilmente, como lo son las ligas entre los objetos y el rol que cada uno juega en la relación"[39].

Un diagrama de colaboración inicia con un mensaje que inicializa la interacción o colaboración por ejemplo una llamada a una operación.

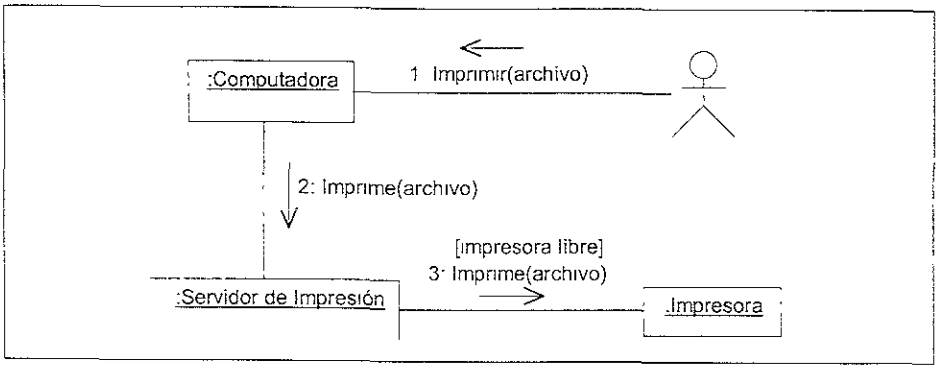

Figura 4-7 Diagrama de colaboración que ilustra la ejecución del caso de uso "Imprimir Archivo".

4.4.1 Flujo del mensaje

Una etiqueta de mensaje en un diagrama de colaboración es especificada con la siguiente sintaxis:

```
<predecesor> < [condición-de-guarda] > expresión-de-secuencia < valor-de-
retorno := signatura >
```

donde el predecesor es especificado con la siguiente sintaxis:

```
numero-de-secuencia ,  .. /
```

El predecesor es una expresión para sincronización de threads o rutas, lo que significa que un mensaje conectado a un numero de secuencia debe ser realizado y manejado antes de que el mensaje actual sea enviado. La lista de números de secuencia es separada por una coma.

La cláusula de condición es normalmente expresada en seudo código o en un lenguaje de programación. UML no prescribe una sintaxis específica.

La expresión-de-secuencia tiene la siguiente sintaxis:

```
[entero | nombre] [recurrencia] ':'
```

[39] Martin, Fowler con Kendall, Scott UML Distilled: Applying the Standard Object Modeling Language. p 108

El entero es un numero de secuencia que especifica el orden del mensaje. El mensaje 1 siempre inicia una secuencia de mensajes; el mensaje 1.1 es el primer mensaje anidado dentro del mensaje 1; el mensaje 1.2 es el segundo mensaje anidado dentro del mensaje 1, y así sucesivamente. El nombre representa un thread concurrente de control. Por ejemplo, 1.2a y 1.2b son mensajes concurrentes enviados en paralelo. La expresión de secuencia debe ser terminada con el signo de dos puntos (:).

El valor de retorno debe ser asignado a una signatura de mensaje. Una signatura de mensaje esta compuesta de un nombre del mensaje un una lista de argumentos. El valor de retorno muestra el valor regresado como resultado de una llamada a una operación (mensaje). Ejemplos de etiquetas de mensajes son los siguientes:

```
1.4.5:x: = calc(n)
1: display()
```

4.4.2 Ligas

Una liga es una conexión entre dos objetos. Cualquier nombre de rol de los objetos de la liga puede ser mostrado al final de la liga, junto con los calificadores en la liga. Ambos, calificadores y roles son también especificados en el diagrama de clases que contiene las clases de los objetos.

4.4.3 Tiempo de vida de un objeto

Los objetos que son creados durante una colaboración son designados con el estereotipo {new}. Los objetos que son destruidos durante la colaboración son especificados mediante el estereotipo {destroyed}. Los objetos que son creados y destruidos durante la misma colaboración son designados como {transitorios}, lo que es equivalente a {new}{destroyed}.

4.4.4 Uso de los diagramas de colaboración

Los diagramas de colaboración pueden ser usados para mostrar interacciones complejas entre objetos. Sin embargo, aprender el esquema de numeración de los mensajes puede tomar algún tiempo, pero una vez aprendido es mas bien fácil de usar. La principal diferencia entre los diagramas de colaboración y los de secuencia es que los diagramas de colaboración muestran los objetos actuales y sus ligas (es decir la red de objetos que participan en la colaboración), lo que en muchas situaciones puede hacer más fácil el entendimiento de una interacción. La secuencia en el tiempo puede ser mas fácilmente vista en un diagrama de secuencia, en donde se puede leer de arriba a abajo. Para decidir que diagrama utilizar para mostrar una interacción, la guía general es seleccionar el diagrama de colaboración cuando los objetos y sus ligas faciliten el entendimiento de la interacción y el diagrama de secuencia cuando solo la secuencia necesita ser mostrada.

El siguiente diagrama muestra la siguiente colaboración. Una ventana de Estadísticas de ventas (mensaje 1), crea un objeto de Resumen de Estadísticas (1.1), el cual recabara las estadísticas para desplegarse en la ventana. Cuando el objeto Resumen de Estadísticas es creado, iterará en todos los vendedores para obtener el total de cada vendedor (1.1.1) y el presupuesto (1.1.2) para cada vendedor. Cada objeto vendedor obtiene su resumen al iterar en todas sus ordenes y obteniendo el monto de cada una (1.1.1.1) y sumándolas

juntas, y obtendrá su presupuesto al obtener el monto del objeto de presupuesto de ventas (1.1.2.1). Cuando el objeto Resumen de Estadísticas haya iterado en todos los vendedores, será creado el mensaje de retorno (mensaje 1.1). Después la ventana de Estadísticas de Ventas obtiene el resultado del objeto Resumen de Estadísticas y muestra cada línea en su ventana. Cuando todas las líneas han sido leídas, la operación de mostrar en la ventana de Estadísticas de Ventas regresa y la colaboración es terminada.

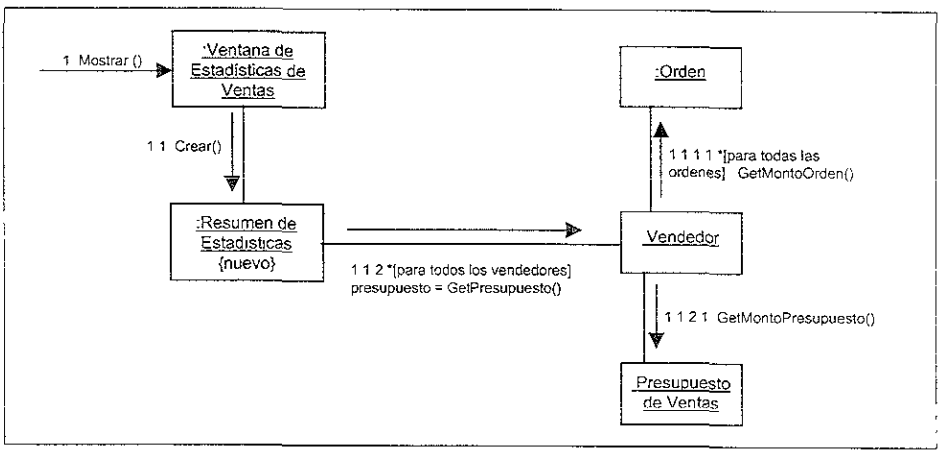

Figura 4-8 *Diagrama de colaboración que obtiene un resumen de los resultados de ventas.*

4.5 Diagrama de Actividad

"Los diagramas de actividad son una de las partes más inesperadas de UML. A diferencia de otras técnicas en UML, estos diagramas no tienen sus orígenes en los trabajos previos de ninguno de los "tres amigos" (Booch, Jacobson, Rumbaugh). Combinan las ideas de diferentes técnicas: los diagramas de eventos de Jim Odell, técnicas de modelado de estados SDL y las redes Petri"[40].

Los diagramas de actividad modelan acciones y sus resultados, se enfocan en el trabajo realizado en la implementación de una operación (un método) o caso de uso. Son una variante de los diagramas de estado, con una sutil diferencia de propósito, el cual es modelar acciones (trabajo o actividades que serán desempeñadas) y sus resultados en términos de cambios de estado de los objetos. Los estados en los diagramas de actividad, llamados estados de acción, cambian al siguiente estado directamente cuando la acción es completada, sin necesidad de especificar ningún evento que lo haga cambiar, como sería el caso de los diagramas de estados. Otra diferencia es que las acciones pueden ser agrupadas en swimlanes (carriles). Un swimlane agrupa actividades en relación a quién es el responsable de ellas. Otra característica importante de este tipo de diagramas

[40] Martin, Fowler con Kendall, Scott. UML Distilled: Applying the Standard Object Modeling Language p 129

es que permiten indicar que un conjunto de acciones sean realizadas simultáneamente o en paralelo, lo cual es muy común cuando se modelan flujos de trabajo.

Un diagrama de actividad es una manera alternativa de describir interacciones, con la posibilidad de expresar en que orden son realizadas, que sea realiza en cada acción (cambio de estado de los objetos), cuando ocurren (secuencia de acciones) y donde toman lugar.

Los diagramas de actividad pueden ser utilizados para distintos propósitos, como por ejemplo:

- Para capturar las acciones (trabajo) que serán realizadas cuando una operación se encuentre en ejecución (la instancia de la implementación de una operación). El cual es el uso más común para los diagramas de actividad.
- Para capturar o representar el trabajo interno de un objeto.
- Para mostrar como un conjunto de acciones relacionadas pueden ser ejecutadas y cómo dichas acciones afectarán los objetos alrededor de ellas.
- Para mostrar cómo una instancia de un caso de uso puede ser realizada en términos de acciones y cambio de estado de los objetos.
- Para representar cómo un negocio trabaja en términos de trabajadores (actores), flujos de trabajo, organizaciones y objetos.

4.5.1 Acciones y transiciones

Una acción es realizada para producir un resultado, la implementación de una operación puede ser descrita como un conjunto ordenado de acciones, las cuales serán más tarde traducidas en líneas de código. Un diagrama de actividad tiene, al igual que un diagrama de estados, un punto de inicio y un punto de terminación, el punto de inicio es representado mediante un circulo relleno, y el punto de terminación es representado por otro circulo relleno pero esta vez rodeado de un circulo más grande. Las acciones son representadas mediante rectángulos con las esquinas redondeadas, justo como se hace con los estados en un diagrama de estados.

Dentro del rectángulo redondeado de una acción se coloca una cadena de texto que especifica la acción o acciones a tomar. La transición entre acciones tiene la misma notación que en un diagrama de estados (una flecha), pero en este caso no se requiere especificar ningún evento para que la transición ocurra. El único lugar donde se pueden especificar eventos es en la transición del punto inicial hacia la primera acción. En la flecha de una transición se pueden también especificar condiciones de guarda, pero normalmente no se especifica nada, lo que indica que la transición será realizada tan pronto como la acción termine.

En la flecha de una transición se puede especificar una condición de guarda, la cual deberá ser cierta para que la transición ocurra. La notación de las condiciones de guarda es la misma que en los diagramas de estados. De esta manera las decisiones son hechas utilizando las condiciones de guarda, por ejemplo [si] y [no]. Otro símbolo que puede ser utilizado en este tipo de diagramas es un diamante, el cual representa una actividad de decisión, la cual puede tener una o mas transiciones de entrada y dos o mas transiciones de salida cada una de las cuales tendrá una condición de guarda asociada a ella Normalmente una de esas condiciones será siempre cierta, de lo contrario el flujo se quedaría en la actividad de decisión y no sabría hacia donde dirigirse.

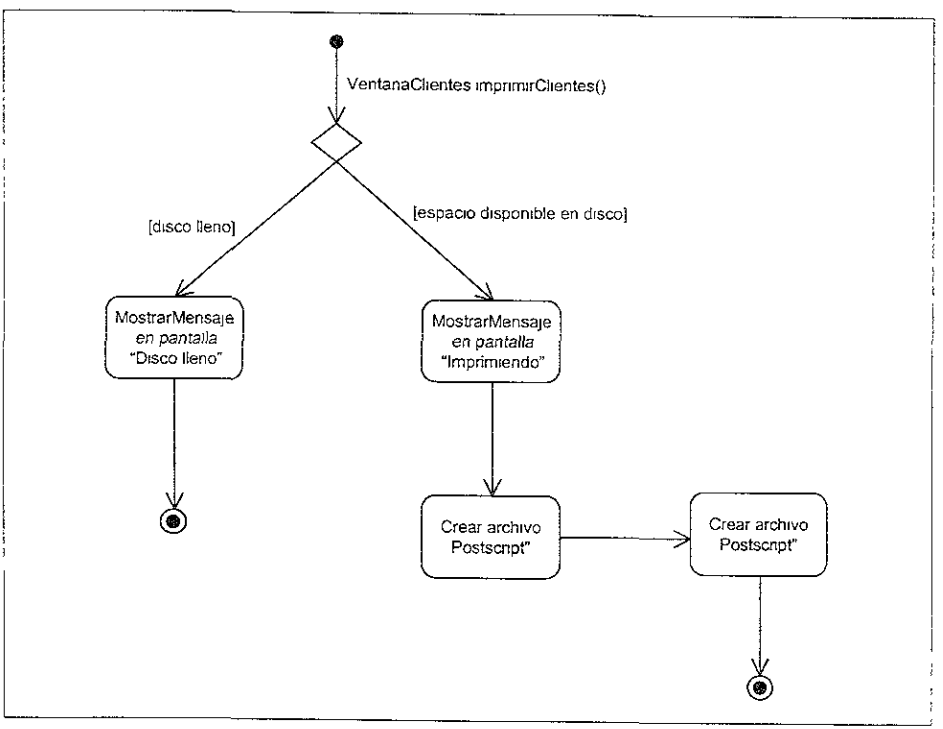

Figura 4-9 *Diagrama de actividades.*

Una transición puede ser dividida en dos o más transiciones, las cuales darán como resultado acciones en paralelo. Lo que quiere decir que las acciones se ejecutarán concurrentemente o que no importa su orden, lo importante es que todas las acciones en paralelo sean ejecutadas antes de que se vuelvan a unir en una transición, si es que en algún momento hay una transición que las una. Una línea en negrita y más gruesa es dibujada para representar que una transición es dividida en diferentes ramas y muestra que la transición ha sido dividida, dando como resultado que diferentes acciones sean ejecutadas de manera paralela, como se muestra en el siguiente diagrama. La línea gruesa y en negrita es también utilizada para indicar la unificación de distintas ramas.

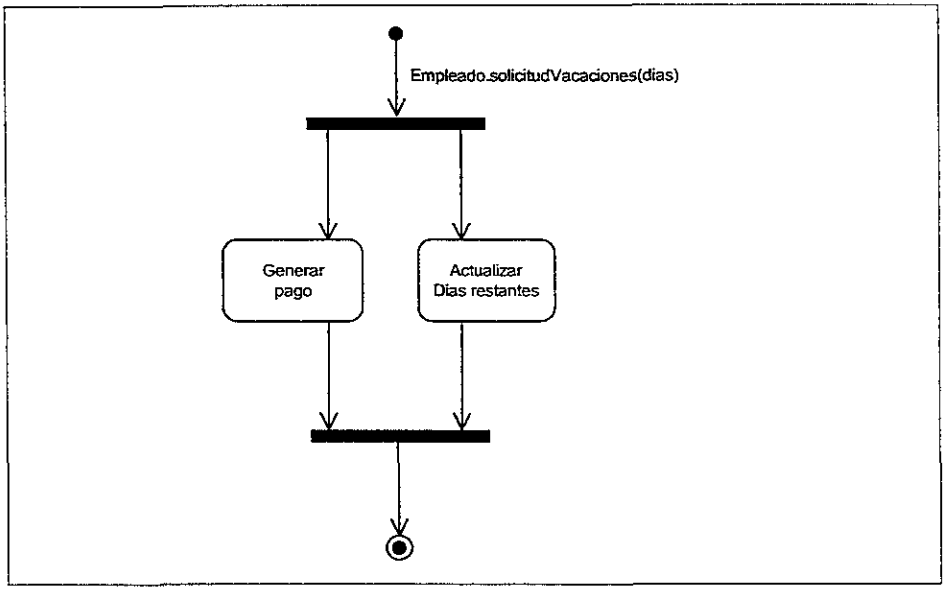

Figura 4-10 *Diagrama de actividades con acciones en paralelo. En este caso indicando que las acciones "Generar pago" y "Actualizar Días restantes" puden ser ejecudadas simultaneamente o que no importa el orden en el que sean ejecutadas.*

4.5.2 Swimlanes (Carriles)

"Un swimlane es la división de un diagrama de actividades que agrupa dichas actividades respecto a sus responsables"[41]. Son usados para distintos propósitos, como por ejemplo, para indicar explícitamente donde son ejecutadas las acciones (en que objetos), o para mostrar en que parte de una organización es desarrollado el trabajo (cuando los diagramas de actividad se utilizan para modelar un flujo de trabajo en alguna organización). Los swimlanes son dibujados como rectángulos verticales que agrupan las acciones que pertenecen al mismo swimlane. El nombre del swimlane es puesto en la parte de arriba del rectángulo como se muestra en la siguiente figura.

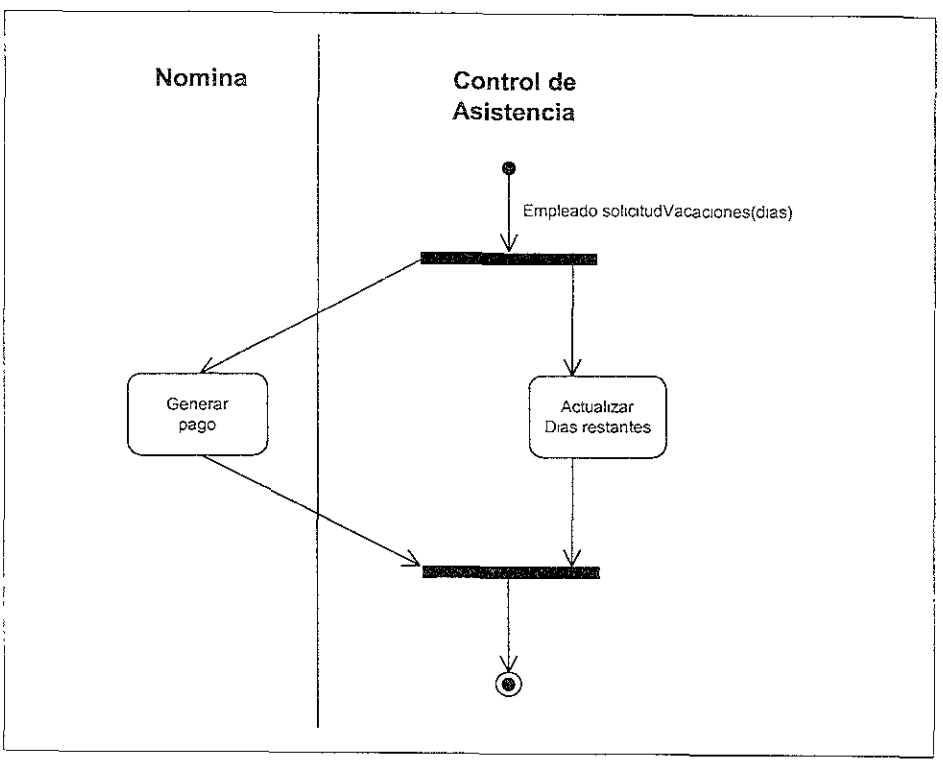

Figura 4-11 *Diagrama de actividades dividido mendiante swimlanes que delimitan la responsabilidad de cada objeto en este caso "Nomina" y "Control de Asistencia" son objetos, pero muy bien podrían ser los departamentos de una organización.*

[41] Rumbaugh, James. [et. al.] The Unified Modeling Language Reference Manual p 461

4.5.3 Objetos en un diagrama de actividad.

Se puede dibujar objetos en un diagrama de actividad, lo cual indica que el objeto es la entrada o salida de una acción o que simplemente dicho objeto es afectado por una acción específica. Los objetos son dibujados con su notación habitual (un rectángulo con el nombre del objeto o de la clase). Cuando un objeto es la entrada de una acción, se muestra una flecha punteada que se extiende del objeto a la acción; cuando el objeto es la salida de una acción, se dibuja una flecha punteada que va de la acción al objeto. Cuando el objeto es afectado por una acción, se dibuja una línea punteada entre la acción y el objeto. Opcionalmente se puede mostrar el estado del objeto, debajo del nombre del mismo, poniendo el nombre del estado encerrado entre paréntesis rectangulares como [lleno], [comprado], [casado], etc.

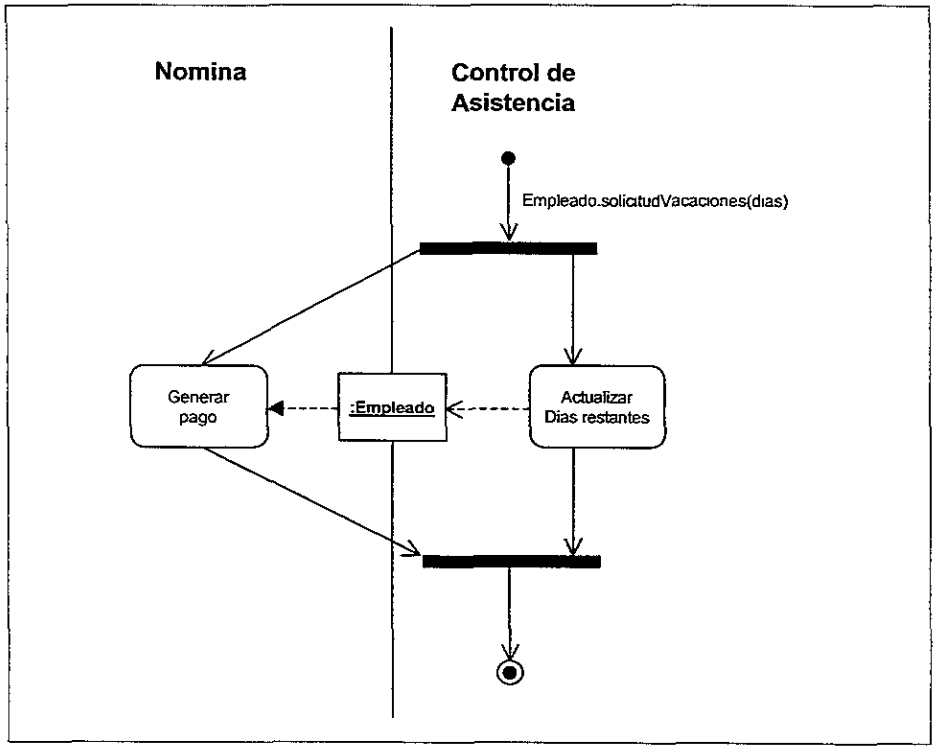

Figura 4-12 *Diagrama de actividades en donde el objeto "Empleado" recibe datos de la actividad "Actualizar Dias restantes" y proporciona datos a la actividad "Generar pago".*

CAPÍTULO 5

5. ARQUITECTURA

"La arquitectura del sistema es un bosquejo de todas las partes que en su conjunto definen al sistema: su estructura, interfases y los mecanismos utilizados para comunicarse"[42]. Al definir una arquitectura apropiada se hace mas fácil navegar por el sistema, encontrar una función o concepto específico, o identificar la ubicación en donde agregar una nueva funcionalidad o concepto que embone en toda la arquitectura. La arquitectura debe ser lo suficientemente detallada para que pueda ser traducida en las líneas de código del sistema. Una arquitectura que es fácil de navegar y suficientemente detallada, debe ser también escalable, lo que significa que puede ser vista en diferentes niveles. La arquitectura, por ejemplo, debe brindar una vista de alto nivel que incluya solo algunas partes y de ahí el desarrollador debe ser capaz de seleccionar una parte y examinar su arquitectura interna la cual consiste de partes mas detalladas. Usando una herramienta CASE, es posible navegar dentro la arquitectura de tal manera que al seleccionar un elemento de la vista de alto nivel se pueda ir al detalle de dicho elemento en un siguiente nivel de detalle.

Una arquitectura bien definida permite la inserción de nuevas funciones y conceptos sin imponer problemas en el resto del sistema (como en los antiguos sistemas monolíticos, en donde un pequeño cambio representaba un gran cambio para el sistema debido a la complejidad de las relaciones entre las diferentes partes del sistema).

[42]Eriksson, Martin con Penker, Magnus UML Toolkit p. 197

La arquitectura debe servir como un mapa para los desarrolladores, donde se describe como está construido el sistema y donde se encuentran funciones y conceptos específicos. Durante el desarrollo este mapa puede que se tenga que modificar debido a descubrimientos importantes y a experiencias que se obtengan al ir desarrollando. La arquitectura debe "vivir" con el sistema mientras el sistema está siendo desarrollado y debe constantemente reflejar la construcción del sistema en todas sus fases y generaciones. Naturalmente, la arquitectura base es definida en la primer versión del sistema, por lo cual la calidad de la arquitectura inicial es vital para permitirle a los desarrolladores cambiarla, extenderla y actualizar la funcionalidad del sistema.

La definición en UML de arquitectura es la siguiente:

"La Arquitectura es la estructura organizacional de un sistema, incluyendo su descomposición en partes, su conectividad, sus mecanismos de interacción, y los principios guías que informan el diseño de un sistema. Una arquitectura puede ser recursivamente descompuesta en partes que interactúan a través de interfases, relaciones que conectan las partes y restricciones para las partes conectadas."[43]

Una arquitectura de software es una descripción de los subsistemas y componentes de un sistema de software y las relaciones entre ellos. Los subsistemas y componentes son típicamente especificados en diferentes vistas para mostrar funcionalidad relevante y propiedades no funcionales de un sistema de software. La arquitectura de software de un sistema es un artificio, resultado de las actividades del diseño de software.

La arquitectura de un sistema es descrita mediante un conjunto de vistas, donde cada vista se concentra en un aspecto específico del sistema. La foto completa del sistema solo puede ser hecha mediante la definición de todas las vistas. En UML, estas vistas son usualmente definidas como la vista de casos de uso, la vista lógica, la vista de concurrencia, la vista de componentes y la vista de distribución o entrega. Una más amplia clasificación divide la arquitectura en dos partes, en la arquitectura lógica y en la arquitectura física del sistema. La arquitectura lógica principalmente especifica las propiedades funcionales del sistema y está dirigida por los requerimientos funcionales del sistema; la arquitectura física principalmente especifica los aspectos no funcionales, como la confiabilidad, compatibilidad, uso de recursos y distribución del sistema.

En resumen una buena arquitectura debe cumplir con las siguientes características:

- Una correcta descripción de las partes que definen el sistema, en términos de la arquitectura lógica y física.
- Un mapa del sistema en el que el desarrollador pueda fácilmente saber donde una funcionalidad o concepto ha sido implementado. La funcionalidad o concepto puede ser orientada a la aplicación (es decir, un modelo de algo dentro del dominio de la aplicación) o orientada al diseño (es decir, alguna implementación técnica). Esto también implica que los requerimientos del sistema deben poder ser rastreados hasta el código que los implementa.
- Los cambios y extensiones deben ser fáciles de hacer en una localización específica en el sistema , sin que el resto del sistema se vea negativamente afectado.

[43] Rumbaugh, James. [et. al.] The Unified Modeling Language Reference Manual. p. 150.

- Debe ser simple, con interfases bien definidas y dependencias claras entre las diferentes partes, para que un arquitecto pueda desarrollar una parte específica sin tener un entendimiento completo de todos los detalles del sistema entero.
- Debe poder ser reutilizada tanto por otras partes del mismo sistema como por otros sistemas.

Una arquitectura que cumpla con todas estas cualidades no es fácil de diseñar. Pero definir una buena arquitectura de base es uno de los pasos más importantes en el desarrollo de un sistema exitoso. Si no se define concienzudamente, la arquitectura será definida de abajo hacia arriba es decir surgirá a partir del código, lo que tiene como resultado un sistema que será difícil de cambiar, extender, mantener y entender.

5.1 Arquitectura lógica

La arquitectura lógica representa la funcionalidad del sistema, contiene las funciones de las diferentes partes del sistema y especifica en detalle como trabaja la solución La arquitectura lógica contiene la lógica de la aplicación, pero no contiene la distribución física de dicha lógica en procesos diferentes, programas o computadoras. La arquitectura lógica da un claro entendimiento de la construcción del sistema que hace más fácil su administración y coordina el trabajo (para usar los recursos de desarrolladores lo más eficientemente posible). No todas las partes de la arquitectura lógica tienen que ser desarrolladas dentro del proyecto; usualmente se pueden comprar librerías de clases, componentes binarios y/o patrones.

La arquitectura lógica responde preguntas como:

- ¿Qué funcionalidad debe proveer el sistema?
- ¿Qué clases existen y como están relacionadas esas clases?
- ¿Cómo esas clases y objetos colaboran para proporcionar la funcionalidad?
- ¿Cuales son las restricciones de tiempo en las funciones del sistema?
- etc.

En UML, los diagramas usados para describir la arquitectura lógica son. los diagramas de casos de uso, los de estado, actividad, colaboración y secuencia.

Una arquitectura comúnmente usada es la de estructura de tres capas donde el sistema es dividido en una capa de interfase, una capa con los objetos de lógica del negocio y una tercer capa de base de datos. A continuación se muestra un ejemplo de este tipo de arquitectura lógica que incluye una posible arquitectura interna de cada capa. Desde los paquetes se pueden alcanzar o desplegar otros diagramas que describen las clases en cada paquete y su colaboración interna.

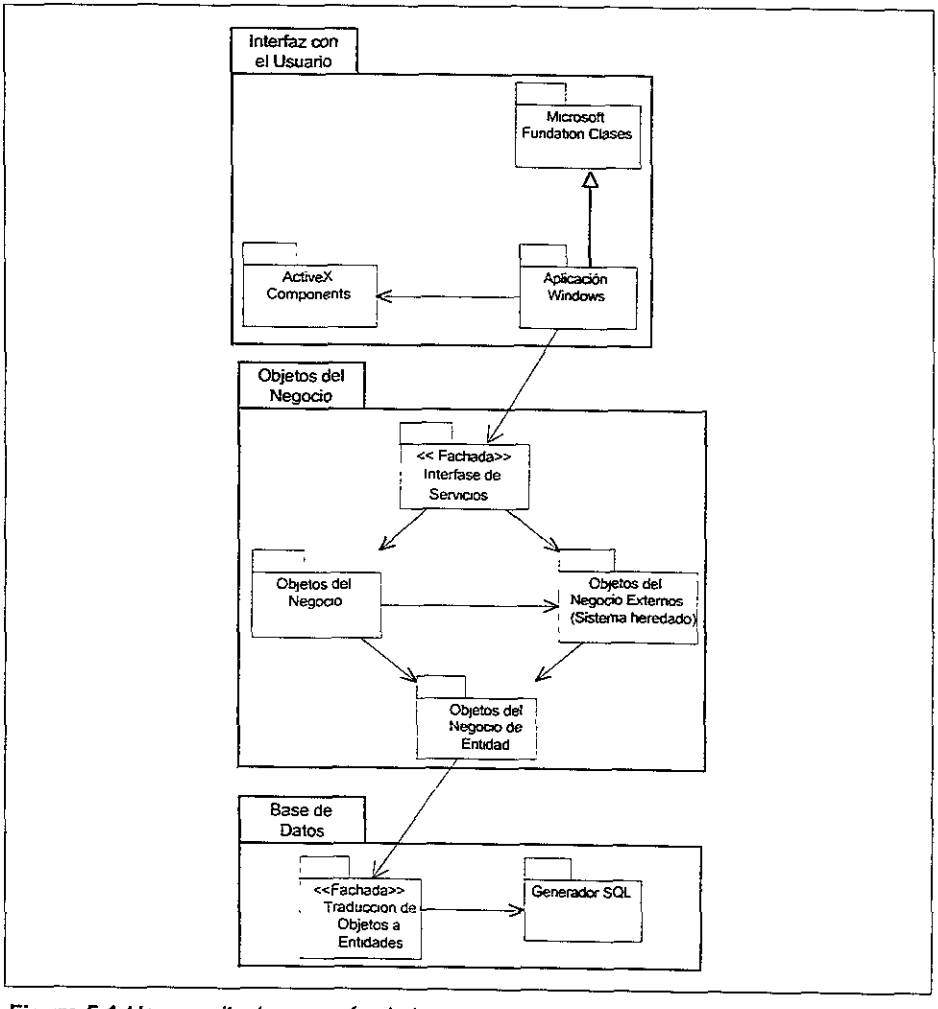

Figura 5-1 *Una arquitectura común de tres capas mostrada como paquetes de UML y mostrando las dependencias entre ellos.*

5.2 Arquitectura física

La arquitectura física contiene una descripción detallada del sistema en términos del hardware y software que el sistema contiene. Revela la estructura del hardware, incluyendo los diferentes nodos y como están conectados dichos nodos entre si. También ilustra las dependencias de los módulos de código que implementan los conceptos definidos en la arquitectura lógica y la distribución del el software en tiempo de ejecución en términos de procesos, programas y otros componentes. La arquitectura física trata de hacer uso eficiente de los recursos de hardware y software.

La arquitectura física responde preguntas como:

- ¿En qué programas o procesos están físicamente localizadas las clases y objetos?
- ¿En qué computadoras se ejecutan los programas y procesos?
- ¿Qué computadoras y otros dispositivos de hardware existen en el sistema y cómo están conectados entre sí?
- ¿Cuáles son las dependencias entre los diferentes archivos de código? si un archivo específico es cambiado, ¿Qué otros archivos tienen que ser recompilados?

En UML mediante el uso de herramientas CASE se establece una conexión entre la arquitectura lógica y la arquitectura física, de tal manera que se pueda localizar los *componentes, procesos y computadoras en la arquitectura física* donde se encuentran las clases de la arquitectura lógica. Esta conexión le permite al desarrollador seguir una clase desde la arquitectura lógica y encontrar su implementación física, o viceversa, rastrear la descripción de un programa hasta llegar a su arquitectura lógica.

Como ya se menciono anteriormente, la arquitectura física está relacionada con la implementación del sistema y como tal, es modelada en diagramas de implementación. Los diagramas de implementación en UML son los diagramas de componentes y de distribución. Los diagramas de componentes contienen los componentes de software: las unidades de código y la estructura de los archivos (el código fuente y los binarios). El diagrama de distribución muestra la arquitectura en tiempo de ejecución del sistema, cubriendo los dispositivos físicos y el software localizado en ellos.

5.2.1 Hardware

Los conceptos de hardware en la arquitectura física pueden ser divididos de la siguiente manera:

- *Procesadores:* Los cuales son las computadoras que ejecutan los programas del sistema Un procesador puede ser de cualquier tamaño, desde un microprocesador hasta una supercomputadora
- *Dispositivos:* Son los dispositivos de soporte como impresoras, ruteadores, lectoras, etc. Están típicamente conectados a un procesador que los controla.
- *Conexiones*: Los procesadores cuentan con conexiones con otros procesadores También cuentan con conexiones a los dispositivos. Las conexiones representan un mecanismo de comunicación entre dos nodos y pueden ser descritas como el medio físico (por ejemplo, un cable óptico) o un protocolo de comunicación (por ejemplo, TCP/IP).

5.2.2 Software

Tradicionalmente el software en la arquitectura de un sistema es definido como un conjunto de paquetes, módulos, componentes o subsistemas Una palabra común para cualquiera de estas unidades modulares es subsistema, que es un sistema en miniatura dentro de un sistema más grande Tiene una interfase y está internamente compuesto por otros subsistemas más detallados o por clases y objetos. Los subsistemas pueden ser

asignados a procesadores en los cuales se ejecutarán (y los procesadores pueden ser alojados a computadoras donde se ejecutarán).

En UML un subsistema es modelado como un paquete de clases. Un paquete organiza un conjunto de clases en un grupo lógico.

En el diseño es común definir uno o más componentes como la fachada del subsistema. El componente "fachada" provee la interfase para comunicarse con el subsistema (paquete) y es el único componente visible para el resto del sistema. Al utilizar una interfase, el paquete se convierte en una unidad muy modular, en la que el diseño interno se encuentra oculto y solo la fachada tiene dependencias con otros módulos en el sistema. Al ver el paquete, solo el componente de fachada es el que le interesa a los que desean utilizar los servicios del paquete y algunas veces sólo se necesita mostrar la fachada del paquete en el diagrama.

Cabe recordar que los paquetes son utilizados tanto en el diseño lógico, donde un conjunto de clases pueden ser incluidas dentro de una unidad, como en la arquitectura física, donde un paquete encapsula un conjunto de componentes.

Los principales conceptos utilizados para describir el software son:

- Componente: Un componente en UML es definido como un componente reutilizable que provee la envoltura física de un conjunto de instancias de elementos de modelado". Esto significa que un componente es una implementación física (por ejemplo, un archivo con código fuente) que implementa elementos del modelo lógico de la manera en que estén definidos en los diagramas de clases o diagramas de interacción.
- Procesos y threads: Un proceso representa un flujo de control pesado, mientras que un thread representa un flujo de control ligero.
- Objetos: Los objetos son aquellos que por si mismos no cuentan con un thread de ejecución. Solamente se ejecutan cuando alguien más les envían un mensaje. Pueden ser asignados a un proceso, a un thread o directamente a un componente ejecutable.

5.2.3 Diagrama de componentes

"El diagrama de componentes muestra la organización y dependencia entre tipos de componentes. Muestra las dependencias entre componentes de software, incluyendo componentes de código fuente, componentes de código binario y componentes ejecutables"[44]. Los componentes son la implementación en la arquitectura física de los conceptos y funcionalidad definida en la arquitectura lógica (clases, objetos y sus relaciones). Los componentes son típicamente los archivos de implementación en el ambiente de desarrollo.

Un componente puede ser cualquiera de las siguientes cosas:

- *Un componente fuente*: Típicamente el archivo con el código fuente de una o más clases. Este tipo de componentes son útiles en tiempo de compilación.

[44] Rumbaugh, James. [et. al.] The Unified Modeling Language Reference Manual. p. 222.

- *Un componente binario*: Es el código binario resultado de la compilación del componente fuente. Puede ser un archivo con el código de un objeto, una librería estática o una librería dinámica Estos componentes son utilizados en la fase de ligado de una aplicación.
- *Un componente ejecutable*: Es un programa ejecutable que es el resultado del ligado de todos los componentes binarios (estático y dinámicos). Un componente ejecutable representa la unidad ejecutable que es ejecutada por un procesador.

En UML un componente es representado por un rectángulo con una elipse y dos pequeños rectángulos en el lado izquierdo (este símbolo es usado en el método de Bush para representar un módulo). El nombre del componente se escribe debajo o dentro del rectángulo

Los componentes son tipos, pero solo los componentes ejecutables pueden tener instancias, las cuales son creadas cuando el programa que representan es ejecutado en un procesador. Un diagrama de componentes muestra solo los componentes como tipos. Para mostrar instancias de un componente, se debe utilizar un diagrama de distribución, donde todas las instancias de los componentes ejecutables son alojadas en instancias de nodos en las cuales se ejecutan.

Las relaciones de dependencia entre componentes se dibujan como líneas punteadas con una flecha en su extremo, lo que significa que un componente necesita a otro para tener una definición completa. Una dependencia de un componente fuente A con otro componente fuente B significa que existe una dependencia específica de lenguaje de A a B. En un lenguaje de compilación significaría que un cambio en B requerirá que A sea recompilado, debido a que las definiciones en el componente B son utilizadas cuando se compila el componente A. Si los componentes son ejecutables, las conexiones de dependencia pueden ser utilizadas para identificar qué librerías dinámicas y programas ejecutables necesitan poder ser ejecutados

Un componente puede definir interfases que sean visibles a otros componentes. Las interfases pueden ser definidas al nivel de código fuente (como en Java) o ser interfases binarias utilizadas en tiempo de ejecución (como un componente OLE). Una interfase es representada con una línea que parte del componente con un circulo al final. El nombre de la interfase es puesto al lado del circulo La dependencias entre componentes pueden apuntar a la interfase del componente que se requiera utilizar.

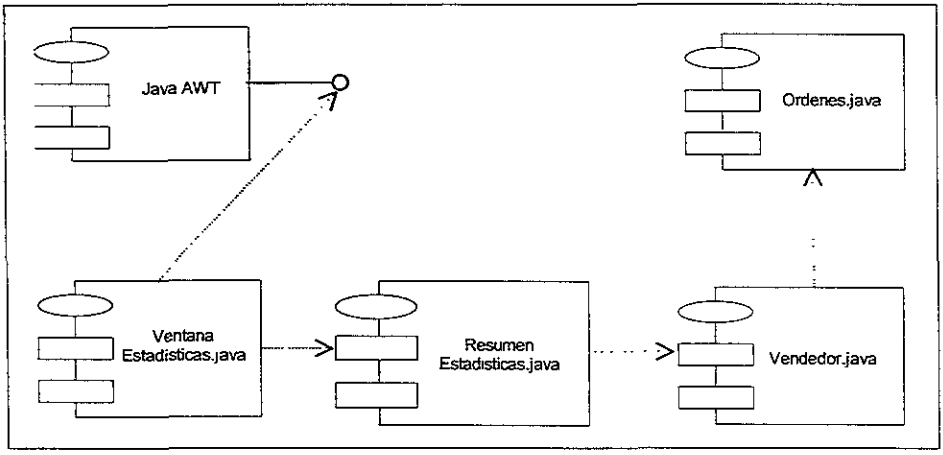

Figura 5-2 *Diagrama de componentes, en donde se puede observar la dependencia entre componentes y el uso de una interfaz para el paquete Java AWT.*

5.2.4 Componentes de tiempo de compilación

Estos componentes son los componentes fuentes que contienen el código producido en los proyectos. Otros componentes como componentes de tiempo de ligado "link-time" y componentes de tiempo de ejecución son generados a partir de los componentes de tiempo de compilación.

Algunos estereotipos que pueden ser utilizados para representar este tipo de componentes son:

- <<archivo>> Una representación de que el archivo contiene código fuente.
- <<página web>> Una representación de que el archivo es una página Web.
- <<documento>> Una representación de un documento (los cuales contienen usualmente la documentación)

Una dependencia de un componente de tiempo de compilación con otro del mismo tipo revela que otros componentes son necesarios para tener la definición completa del componente.

5.2.5 Componentes de tiempo de ligado

Estos componentes son utilizados cuando el sistema es ligado. Son típicamente el resultado de la compilación de uno o varios de los componentes anteriores. Son utilizados por los componentes de tiempo de ejecución. Un tipo especial de esta clase de componentes son las DLLs (dynamic link library) las cuales son ligadas al ejecutable al tiempo de ejecución más que a tiempo de ligado.

5.2.6 Componentes de tiempo de ejecución

Estos componentes representan los componentes utilizados cuando se ejecuta el sistema. Son generados a partir de los componentes de tiempo de ligado y en algunas ocasiones directamente de los componentes de tiempo de compilación. El estereotipo <<aplicación>> representa un programa ejecutable, y el estereotipo <<tabla>> representa una tabla de una base de datos que en algunas ocasiones es vista como un componente en tiempo de ejecución.

Solo los componentes de tiempo de ejecución pueden tener instancias, las cuales son asignadas a nodos (las unidades del diagrama de distribución).

5.3 Diagrama de distribución

"El diagrama de distribución describe la arquitectura en tiempo de ejecución de los procesadores, dispositivos y de los componentes de software que se ejecutan en dicha arquitectura"[45]. Es la última descripción física de la topología del sistema, la cual describe la estructura de las unidades de hardware y el software que se ejecuta en cada unidad. Como en toda arquitectura es posible examinar un nodo especifico en la topología, ver cuales componentes están siendo ejecutados en dicho nodo y que elementos lógicos (clases, objetos, etc) están siendo implementados en dicho componente. y finalmente rastrear dichos elemento hasta el análisis inicial de requerimientos del sistema.

5.3.1 Nodos

"Un nodo es un objeto físico en tiempo de ejecución que representa un dispositivo computacional, que generalmente tiene memoria y capacidad de procesamiento"[46]. Los nodos son objetos físicos (dispositivos) que tienen algún tipo de recurso computacional, esto incluye computadoras con procesadores, dispositivos como impresoras, lectoras de códigos, dispositivos de comunicación, etc.

Un nodo puede ser representado como un tipo o como una instancia (un nodo es una clase), donde un tipo describe las características de un tipo de procesador o dispositivo y una instancia representa las ocurrencias reales (maquinas) de ese tipo. Las características específicas de cada nodo pueden ser definidas dentro de las propiedades de cada nodo. Un nodo es representado mediante un cubo tridimensional con su nombre dentro de él, y al igual que la notación utilizada para las clases y objetos, si el símbolo representa una instancia, el nombre del nodo es subrayado.

[45] Rumbaugh, James. [et. al.] The Unified Modeling Language Reference Manual p. 252.

[46] Idem. p 357

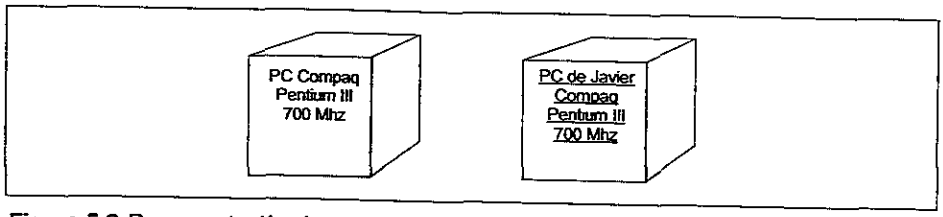

Figura 5-3 *Representación de un nodo. El cubo de la izquierda representa un tipo de nodo y el de la derecha representa una instancia de ese tipo.*

5.3.2 Conexiones

Los nodos son conectados mediante asociaciones de comunicación. Estas asociaciones son dibujadas como una línea que une un nodo con otro, indicando que existe algún tipo de comunicación entre ellos. Los nodos intercambian objetos o envían mensajes a través de dichas asociaciones. El tipo de comunicación es representado mediante un estereotipo que identifica el protocolo de comunicación o red utilizada.

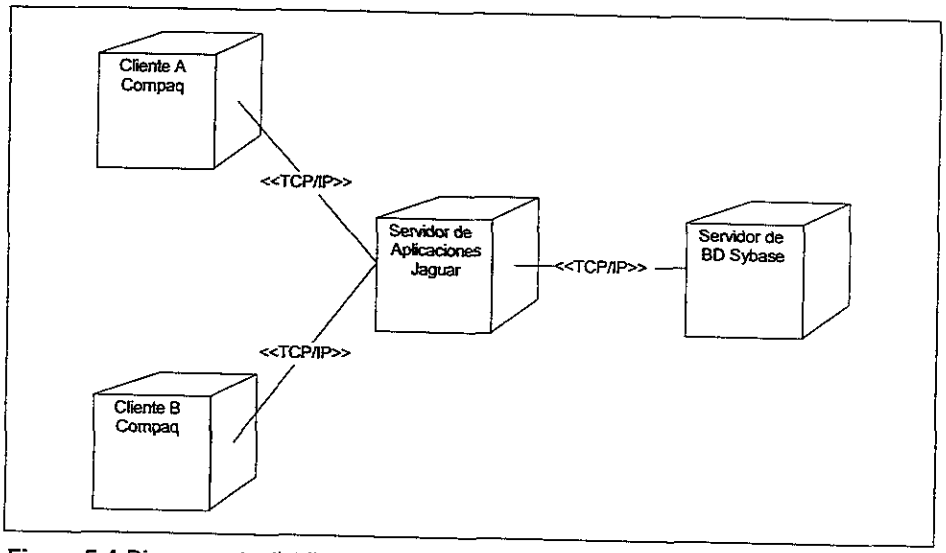

Figura 5-4 *Diagrama de distribución. En el diagrama se muestra la comunicación entre nodos con el estereotipo "<<TCP/IP>>" lo que indica cual es el protocolo de comunicación utilizado.*

5.3.3 Componentes

Dentro de los símbolos de instancia de los nodos pueden existir instancias de componentes ejecutables, lo que representa que dichos componentes se ejecutan en los nodos donde aparece su nombre. Dentro de un tipo de nodo se puede especificar que dicho tipo de nodo soporta algún tipo específico de componente, lo que significa que en

tiempo de ejecución una instancia del tipo de componente especificado será ejecutada por una instancia del tipo de nodo del que se trata. Por ejemplo, no es posible ejecutar un componente Windows en un nodo AS/4000, por lo que el tipo de nodo AS/400 no soporta ese tipo de componente.

Los componentes se conectan entre sí mediante líneas de dependencia punteadas, justo como en el diagrama de componentes, lo que significa que un componente utiliza los servicios del otro, aunque es importante recordar que en un diagrama de distribución solo deben aparecer los componentes ejecutables.

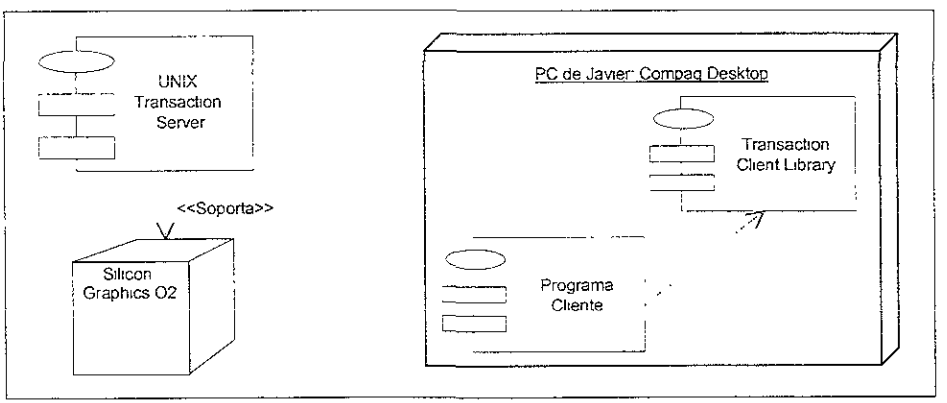

Figura 5-5 *Diagrama que ejemplifica la asociación de componenetes a nodos.*

5.3.4 Objetos

Un objeto es puesto dentro de una instancia de un nodo para indicar donde reside. El objeto puede estar contenido dentro de otro objeto o dentro de un componente, lo que se representa anidando sus símbolos, si esto se torna muy complicado, el objeto puede ser *dibujado directamente dentro de un nodo sin mostrar los componentes en los que es* implementado. La información acerca de a qué componente pertenece es definida mediante la propiedad de localización del objeto o puede permanecer indefinida en el diagrama de distribución.

En un sistema distribuido, un objeto se puede mover entre diferentes nodos durante la ejecución del sistema. Esto es técnicamente hecho mediante objetos distribuidos como COM o CORBA, donde los objetos pueden ser transmitidos a través de la red y pueden cambiar su localización en el sistema. Un objeto que cambia su localización durante su tiempo de vida en el sistema puede ser incluido en todos los nodos donde es posible que exista.

5.3.5 Modelado complejo de nodos

Como ya se mencionó anteriormente, los nodos son definidos como clases dentro de UML, por lo tanto, para definir relaciones más complejas entre nodos se pueden utilizar diagramas de clases, como se muestra en la siguiente figura. Este mecanismo es usado

cuando se describe una familia de nodos, donde la generalización es usada para definir una configuración general de nodos y la especialización es utilizada para capturar casos especiales.

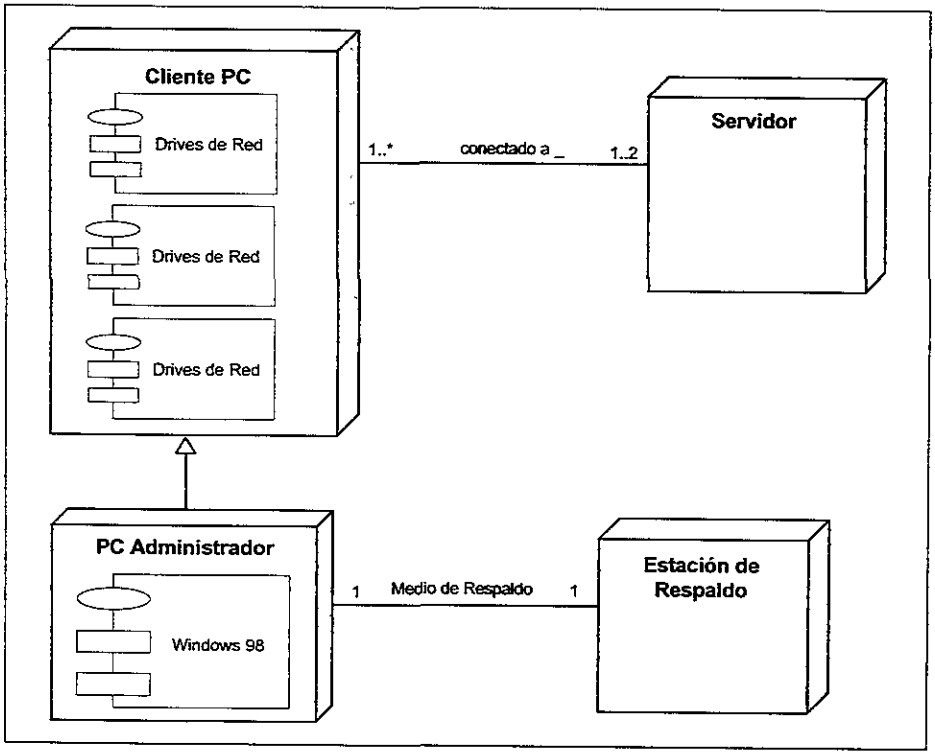

Figura 5-6 *Diagrama de distribución, que muestra algunos de los componenetes necesarios en cada nodo.*

5.3.6 Asignación de componentes a nodos

Las clases y colaboraciones definidas en el diseño lógico son asignadas a componentes en las que son implementadas. Esta asignación es dirigida por el lenguaje de programación utilizado, por ejemplo, en C++ la implementación de una clase se hace en dos archivos: un archivo de cabecera (.h) que contiene la especificación y un archivo de implementación (.cpp) que contiene la implementación de las operaciones. Java por otra parte implementa una clase en un solo archivo que contiene la especificación y la implementación. Por lo tanto el lenguaje de programación define las reglas para la asignación de clases a los componentes.

Los procesos son asignados a los componentes en los que se ejecutan. La asignación es dirigida por la necesidad de objetos activos o por la necesidad de distribuir geográficamente el sistema.

Finalmente, los componentes son asignados a nodos. Una instancia de un componente se ejecuta en al menos una instancia de un nodo y posiblemente en varios. La asignación de componentes a los nodos puede afectar la topología del sistema, Existe un conjunto de *aspectos a considerar cuando se asignan componentes a los nodos*:

- *Uso de recursos*: Uno de los principales objetivos cuando se determina la arquitectura física y la asignación de componentes es el uso de los recursos de hardware. El hardware debe ser utilizado de manera eficiente, utilizando la capacidad completa de cada nodo, pero sin sobrecargarlo ya que esto podría resultar en un desempeño pobre del sistema.
- *Localización Geográfica*: Se requiere tomar decisiones respecto donde es requerida una funcionalidad especifica (en que nodos), y que funcionalidad debe estar disponible localmente (debido a cuestiones de desempeño, o si debe estar disponible aunque otros nodos no estén operando).
- *Acceso a dispositivos*: ¿Cuáles son los requerimientos de dispositivos de un nodo en especifico?, ¿Puede la impresora se conectada a ese servidor?, O ¿cada cliente necesita una impresora local?.
- *Seguridad*: Este factor es importantisimo en el diseño de una arquitectura física, sobre todo cuando la información que se maneja es confidencial. Por lo tanto se debe de tomar decisiones que redunden en un sistema que maneja la seguridad de manera eficiente y confiable.
- *Desempeño*: La necesidad de un alto desempeño puede algunas veces afectar la localización de un componente.
- *Extensibilidad y portabilidad:* Cuando diferentes nodos tienen sistemas operativos diferentes o la arquitectura de las maquinas es diferente, se deben considerar qué componentes pueden ser dependientes de un sistema operativo y cuales deben ser portables a otros sistemas operativos. Esto puede afectar la localización de esos componentes y tal vez el lenguaje de programación para su implementación.

En el diagrama de distribución como en muchos otros se requiere un diseño iterativo, que fructifique en el descubrimiento de la mejor arquitectura física. Para esto es muy probable que se requiera probar distintas soluciones, primero discutiéndolas en la etapa de modelado y más tarde implementándolas en prototipos. Idealmente el sistema debe ser lo suficientemente flexible para que un componente pueda ser movido entre los diferentes nodos.

CAPÍTULO 6

6. CASO DE ESTUDIO

El objetivo de este capítulo es presentar un caso de estudio mediante el cual se ejemplifique el uso de UML en el desarrollo de un sistema. El caso de uso que se presenta es muy sencillo y no pretende ser la base de ningún desarrollo futuro. Se decidió incluir un ejemplo muy sencillo debido a que la explicación del modelado con UML de un sistema completo de la vida real muy bien podría ser el tema de otra tesina.

El problema planteado es el de una biblioteca, que presta libros y revistas, sin embargo para mantener la simplicidad del ejemplo, no se manejan todas las posibles alternativas que se pueden presentar, sino únicamente las que sirvan para ejemplificar el uso de los elementos de modelado de UML

Los diagramas que se presentan solo son unos cuantos, del total de diagramas que deberían aparecer en un sistema aún así de simple Al igual que el código que se presenta, no es el de la totalidad del sistema.

La secuencia de pasos que se sigue, es la secuencia típica que se seguiría en un método de análisis y diseño orientado a objetos. Sin embargo en un método formal de este tipo se deberían completar un conjunto de actividades con mucho más detalle que el que aquí se presenta

La herramienta CASE utilizada para la elaboración de los diagramas es Rational Rose 4.0. Cualquier discrepancia en la notación de los diagramas presentados en este capítulo con la notación explicada a lo largo de la tesina, hay que tomar la de los capítulos anteriores como la correcta, debido a que es la notación formal aunque en Rose existan

ligeros cambios sobre todo en la iconografía utilizada para la signatura de los atributos y funciones.

6.1 Requerimientos

Este es un extracto de lo que podría ser una definición de requerimientos expresados por los usuarios del sistema o por el cliente que solicita el desarrollo (es decir, quién lo pagará).

- El sistema será un sistema de soporte para la operación de una biblioteca.
- La biblioteca presta libros y revistas a sus usuarios. Tanto los libros y revistas, como los usuarios de la biblioteca deberán estar registrados en el sistema.
- El sistema deberá manejar la compra de libros nuevos para la biblioteca. Los títulos más utilizados son comprados en varias copias. Los libros viejos son removidos de la biblioteca cuando se encuentran desactualizados o en malas condiciones.
- El bibliotecario es un empleado de la biblioteca el cual interactúa con los usuarios de la misma y cuyo trabajo será soportado por el sistema.
- Un usuario de la biblioteca puede reservar un libro o revista que no se encuentre disponible físicamente en ese momento, para que cuando se encuentre disponible, se le notifique al usuario que su reservación ya está disponible. La reservación es cancelada cuando el usuario obtiene el libro o a través de una solicitud explícita de cancelación.
- El sistema debe poder fácilmente crear, actualizar y borrar la información de los títulos, estudiantes, préstamos y reservaciones.
- Debido a que la biblioteca cuenta con distintas plataformas, el sistema deberá poder correr en diferentes plataformas (Unix, Windows, OS/2, etc). Además de que deberá contar con una interfaz gráfica (GUI).
- Además de lo anterior y debido a que toda la funcionalidad requerida no será implementada en la primera versión, el sistema deberá ser fácilmente extendible.

En la primera versión el sistema no tendrá que manejar el envío de mensajes a los estudiantes con reservaciones, ni tendrá que verificar la disponibilidad de un título.

6.2 Análisis

En el análisis se pretende capturar y describir todos los requerimientos del sistema y hacer un modelo que defina las principales clases del dominio del sistema (lo que será manejado por el sistema). El propósito es clarificar el entendimiento de los requerimientos y facilitar la comunicación entre los desarrolladores y las personas que establecen los requerimientos (usuarios/clientes). Debido a lo anterior el análisis debe ser elaborado en estrecha colaboración con el usuario o cliente.

El análisis no debe estar restringido por cuestiones técnicas o detalles. El desarrollador no debe pensar en términos de código o programas durante esta fase. Esta fase es en realidad el primer paso para el real entendimiento de los requerimientos del sistema.

6.2.1 Análisis de requerimientos

El primer paso en el análisis es definir los casos de uso, los cuales describen la funcionalidad que el sistema debe ofrecer --los requerimientos funcionales del sistema. Un análisis de casos de uso consiste en leer y analizar las especificaciones, así como discutir el sistema con los clientes y usuarios potenciales del sistema.

Los actores identificados en la biblioteca son: los bibliotecarios y los usuarios de la *biblioteca. Los* **bibliotecarios** son los usuarios del sistema y los **usuarios** son los clientes de la biblioteca, es decir, las personas que reservan los libros y los solicitan en calidad de préstamo. Los usuarios no interactúan directamente con el sistema, las funciones de *préstamo y reservación son ejecutadas por el bibliotecario.*

Los casos de uso en el sistema de la biblioteca son los siguientes:

- Prestar libro
- Devolver libro
- Hacer reservación
- Cancelar reservación
- Agregar título
- Actualizar o borrar título
- *Agregar libro*
- Borrar libro
- *Agregar usuario*
- Actualizar o borrar usuario

No incluido en esta lista está el caso de uso "Mantenimiento", el cual es un caso de uso más general que utiliza otros casos de uso. Se podría discutir si realmente es un caso de uso por si mismo, pero para separar claramente las tareas de mantenimiento de las funciones clave del sistema se definirá como tal.

También hay que notar la presencia de dos conceptos: libro y título. Un libro es una copia física de un título, por lo cual pueden existir distintas copias (libros) de un mismo título Cabe señalar que es posible agregar un título antes de contar con una copia del mismo (libro), con la finalidad de que los usuarios puedan hacer reservaciones de libros que están próximos a adquirirse.

Así mismo hay que observar que se hizo una generalización de los términos libro y revista, únicamente aparece el término libro, debido a que esencialmente su *comportamiento y características son los mismos, a excepción* de que los tiempos máximos de préstamo para una revista difieren de los tiempos máximos de préstamo para un libro, pero esto se verá reflejado en el diagrama de clases.

El análisis del sistema es documentado en UML a través de un diagrama de casos de uso como se muestra en la siguiente figura.

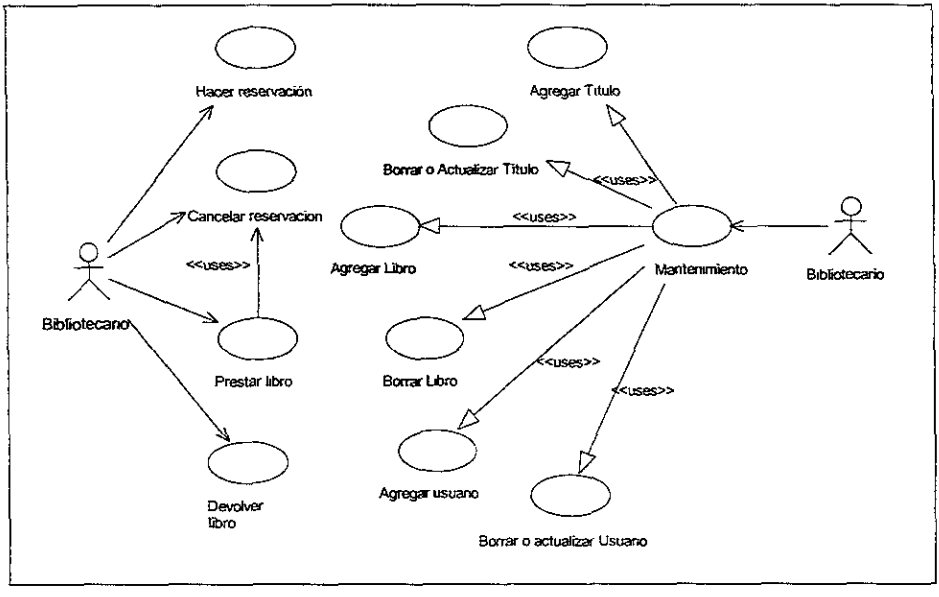

Figura 6-1 *Diagrama de casos de uso de un sistema para una biblioteca.*

Cada caso de uso es documentado mediante una descripción textual donde se indica la interacción del sistema con los actores y la funcionalidad que se desea obtener. Aquí es muy importante recordar que el texto de los casos de uso debe ser discutido y obtenido con los usuarios del sistema.

A continuación se presenta como ejemplo la descripción del caso de uso "Préstamo".

Caso de Uso:	**Prestar libro.**
Actores:	Bibliotecario
Propósito:	Registrar el préstamo de un libro.
Descripción:	Un usuario llega al mostrador con los libros que desea le sean prestados. El bibliotecario registra el préstamo. Al completarse, el usuario se retira con sus libros.
Tipo:	primario.

Curso típico de eventos

Actores	Respuesta del sistema
13. Este caso de uso comienza cuando un usuario llega al mostrador con los libros que desea obtener.	
14. El bibliotecario introduce el identificador del usuario.	15. Despliega la información del usuario, como su nombre, dirección, etc. Despliega los títulos que tenga reservados el usuario indicando cuales ya se encuentran disponibles.
16. El bibliotecario introduce los identificadores de los libros que el usuario desea obtener.	17. El sistema registra los prestamos realizados, cargándole a la cuenta del usuario los libros obtenidos. Si alguno de los libros que se obtuvo se tenía reservado, se elimina la reservación.

Para efectos de este ejemplo la descripción textual de los demás casos de uso no se especificará, pero cabe mencionar que es necesaria para cada uno de los casos de uso encontrados. También se puede optar por realizar la especificación de los casos de uso mediante diagramas de secuencia pero hay que recordar que deben quedar lo suficientemente claros y sencillos ya que son un instrumento de comunicación fundamental entre los desarrolladores y los usuarios y estos últimos deben ser capaces de entenderlos.

6.2.2 Análisis del dominio

Durante el análisis del dominio se descubren las principales clases del sistema. Para realizar el análisis del dominio hay que leer la especificación de requerimientos y los casos de uso para ver que conceptos deben ser manejados por el sistema. También se deben organizar sesiones de lluvia de ideas con los usuarios y expertos del dominio para tratar de identificar todos los conceptos que deben ser manejados y las relaciones entre ellos.

Las clases del dominio en el sistema de la biblioteca son: Usuario, Título, Libro, Reservación y Préstamo. Estas clases son documentadas en un diagrama de clases junto con sus relaciones como se muestra en la siguiente figura.

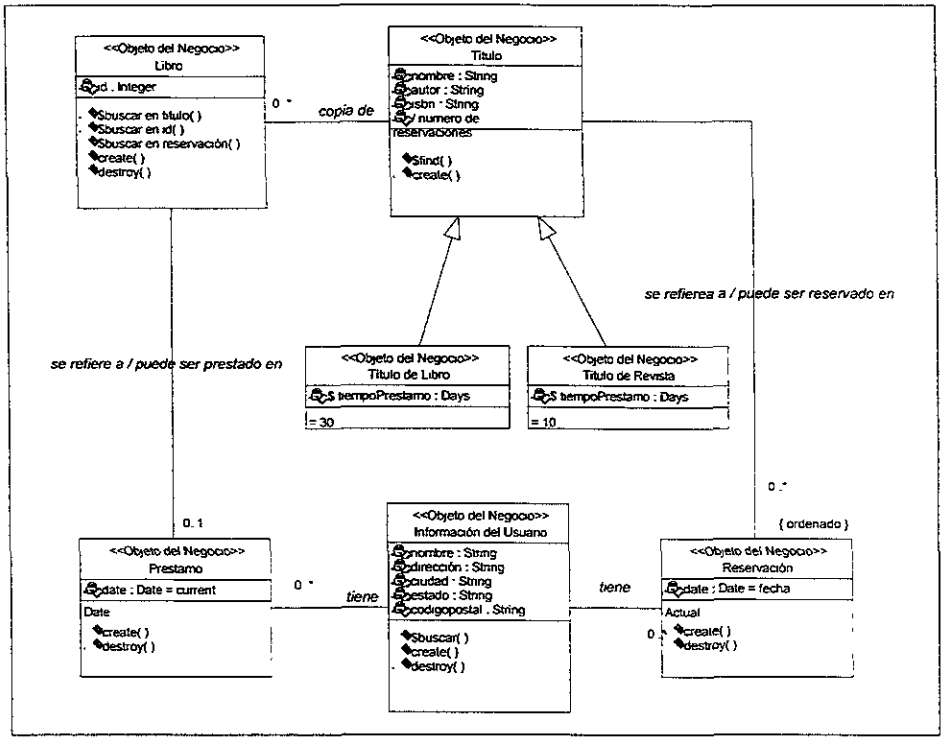

Figura 6-2 *Diagrama de clases del dominio del sistema de Biblioteca.*

Las clases del dominio son definidas con el estereotipo <<Objeto del Negocio>> que es un estereotipo definido por el desarrollador que especifica que ese objeto es una clase que es parte del dominio principal y debe ser almacenada "persistentemente"[47] en el sistema.

Aquí me gustaría enfatizar que las clases del dominio han sido solamente esquematizadas hasta este momento, es decir que los atributos y operaciones no son los finales, solamente han sido definidos los que parecen los apropiados para estas clases en este momento. Algunas de las operaciones han sido definidas al buscar en la definición de los casos de uso.

A dos de las clases aquí encontradas se les han definido diagramas de estados para ilustrar los diferentes estados que los objetos de esas clases pueden tener, junto con los eventos que harán que cambien su estado. Las clases que tienen diagramas de estado son Libro y Título. A continuación se muestran los diagramas de estado de estas dos clases.

[47] Persistentemente significa que se almacenará de manera permanente en algún dispositivo de almacenamiento como puede ser una base de datos, un archivo en disco duro, etc.

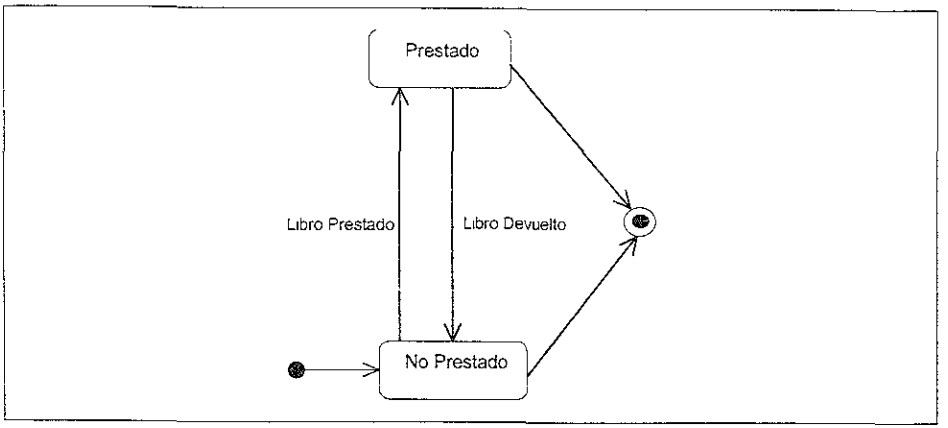

Figura 6-3 *Diagrama de estados de la clase Libro.*

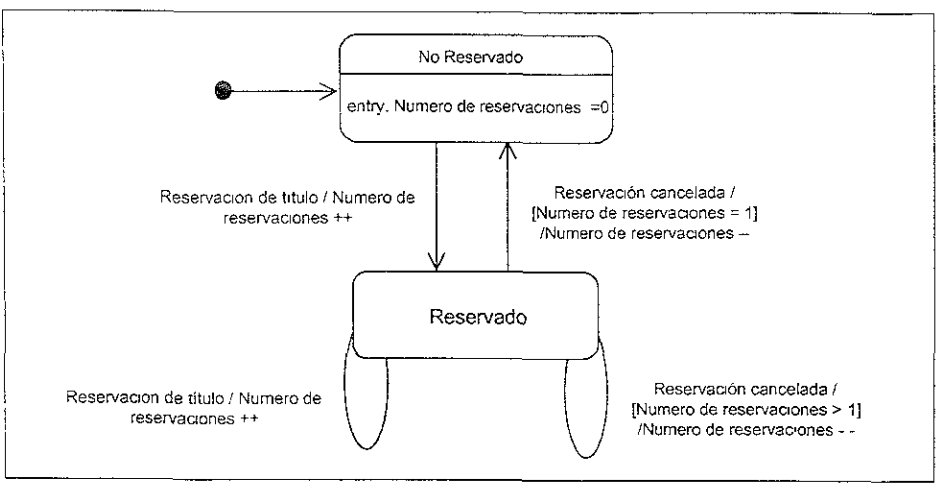

Figura 6-4 *Diagrama de estados de la clase Título.*

Para describir el comportamiento dinámico de las clases del dominio, se pueden utilizar cualquiera de los diagramas dinámicos en UML, es decir, diagramas de secuencia, de colaboración o de actividad. Para efectos de este caso de estudio he decidido utilizar los diagramas de secuencia. Las bases para los diagramas de secuencia son los casos de uso, donde cada caso de uso ha sido descrito con su impacto en las clases del dominio, para ilustrar como las clases del dominio colaboran para realizar el caso de uso en el sistema Naturalmente, cuando se modela estos diagramas de secuencia, se descubren nuevas operaciones, las cuales se deben agregar a las clases originalmente encontradas. Nuevamente en este punto todavía las operaciones no son definidas en detalle por lo que no aparecen sus signaturas. Los objetivos del análisis en este punto es facilitar la comunicación entre el usuario y crear un entendimiento del sistema a construir, en este momento no se tiene una solución en detalle.

Un diagrama de secuencia para el caso de uso "Prestar libro" es mostrado en la siguiente figura.

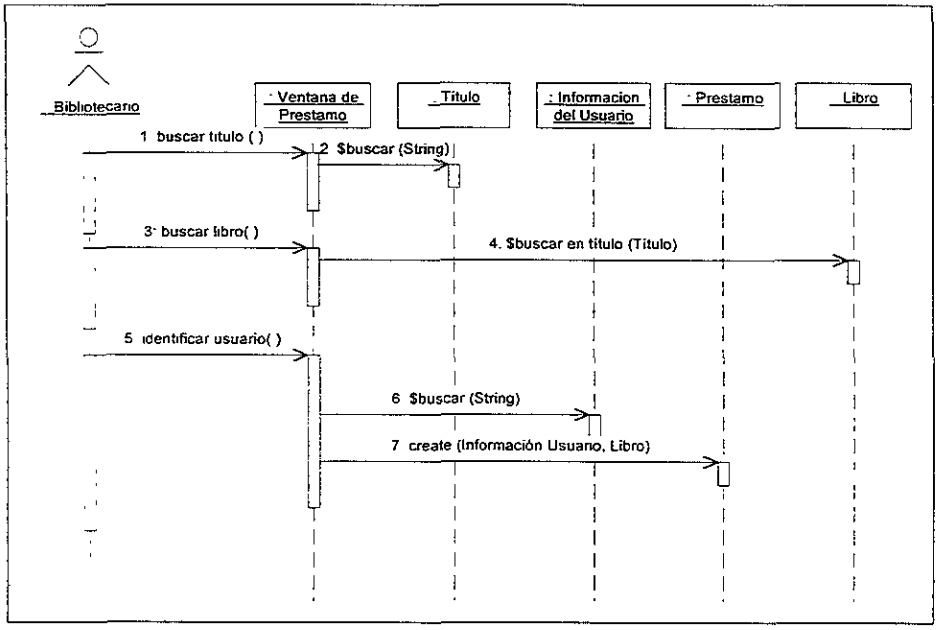

Figura 6-5 *Diagrama de secuencia del caso de uso Prestamo de Libro.*

Al modelar los diagramas de secuencia, se vuelve obvio la necesidad de ventanas y cuadros de dialogo que sirvan de interfaz con los actores. Algunos métodos sugieren que es una buena idea en esta fase crear un prototipo de las ventanas y cuadros de dialogo necesarios y enseñárselos al usuario para ver si realmente eso es lo que el quiere. Las ventanas que podemos hasta este momento visualizar de nuestro caso de estudio son: una ventana para el préstamo de libros, una ventana para la reservación, una ventana para regresar los libros, una ventana para el mantenimiento de usuarios, de títulos y libros, etc. Cabe recordar que en esta fase las interfases son solamente definidas a grandes rasgos, ya que una definición detallada de las mismas se hace dentro de la fase de diseño.

Para separar las clases de las ventanas en el análisis de las clases del dominio, las clases de ventanas fueron agrupadas en un paquete llamado GUI Package.

6.3 Diseño

La fase de diseño y el modelo UML resultante expande y detalla el modelo producido durante el análisis tomando en cuenta todas las implicaciones técnicas y restricciones. El propósito del diseño es especificar una solución que pueda ser fácilmente traducida en código de programación. Las clases definidas en el análisis son detalladas y son

agregadas clases nuevas para manejar las áreas técnicas como una base de datos, una interfase con el usuario, comunicaciones, dispositivos y más.

La fase de diseño puede ser dividida en dos segmentos:

- *El diseño de la arquitectura:* Este es el diseño de alto nivel donde los paquetes (subsistemas) son definidos, incluyendo las dependencias y mecanismos primarios de comunicación entre paquetes. Naturalmente, el propósito es una arquitectura simple, clara y extendible, donde sean pocas las dependencias y las dependencias bidireccionales sean evitadas al máximo.
- *El diseño detallado:* En esta parte se detalla el contenido de los paquetes. Todas las clases son descritas con suficiente detalle para dar una especificación clara al programador que codificará las clases. Los modelos dinámicos de UML son usados para demostrar como se comportan los objetos de las clases en situaciones específicas.

6.3.1 Diseño de la arquitectura

Una arquitectura bien diseñada es el fundamento para un sistema extendible y modificable Los paquetes se pueden concentrar ya sea en manejar un área funcional especifica o un área técnica especifica. Es vital separar la lógica de la aplicación (las clases del dominio) de la lógica técnica, para que los cambios en cualquiera de estos segmentos puedan ser fácilmente hechos sin demasiado impacto en la otra parte.

Las principales tareas que hay que hacer cuando se define una arquitectura es identificar y definir las reglas para la dependencia entre paquetes (subsistemas) evitando en todo lo posible dependencias bidireccionales ya que este tipo de dependencias convierten a dos paquetes en altamente integrados, lo que dificulta la modificación de uno sin afectar al otro. Otra de las tareas que hay que hacer en esta fase es identificar la necesidad de librerías estándar y encontrar las existentes para utilizarlas. Hoy en día existen librerías estándar en el mercado para cubrir las áreas técnicas como interfases con el usuario, bases de datos o comunicaciones, y se espera que próximamente emerjan librerías especificas a un tipo de aplicación.

Los paquetes, subsistemas o capas en el caso de estudio son:

- *Paquete de interfaz con el usuario*: En este paquete se encuentran las clases para permitir al usuario ver los datos del sistema e ingresar nuevos datos. Estas clases están basadas en el paquete AWT de Java, el cual es una librería estándar en Java para escribir aplicaciones gráficas de interfaz con el usuario. Este paquete colabora con el paquete de objetos del negocio. El paquete de interfaz con el usuario llama operaciones de los objetos del negocio para regresar información e insertar nuevos datos
- *Paquete de objetos del negocio*: Este paquete incluye las clases del dominio detectadas desde el modelo de análisis como la clases Libro, Título, Préstamo, etc. Estas clases son detalladas en el diseño por lo que sus operaciones ahora están completamente definidas y el soporte para el almacenamiento Persistente es agregado. El paquete de objetos del negoció coopera con el paquete de base de datos en el sentido de que todas las clases de los objetos del negocio heredan el comportamiento de la clases Persistente del paquete de base de datos.

- *Paquete de Base de Datos.* Este paquete provee los servicios para las clases de los objetos del negocio para que estos últimos puedan almacenar Persistentemente su información. En la versión actual, la clase Persistente almacenará los objetos de sus subclases en archivos dentro del sistema de archivos.
- *Paquete de utilerías:* Este paquete contiene servicios que son usados en otros paquetes. Actualmente la clase ObjId es la única en este paquete. Es utilizada para referirse a través del sistema a los objetos Persistentes y es utilizada en los otros tres paquetes.

A continuación se muestra el diseño de estos tres paquetes.

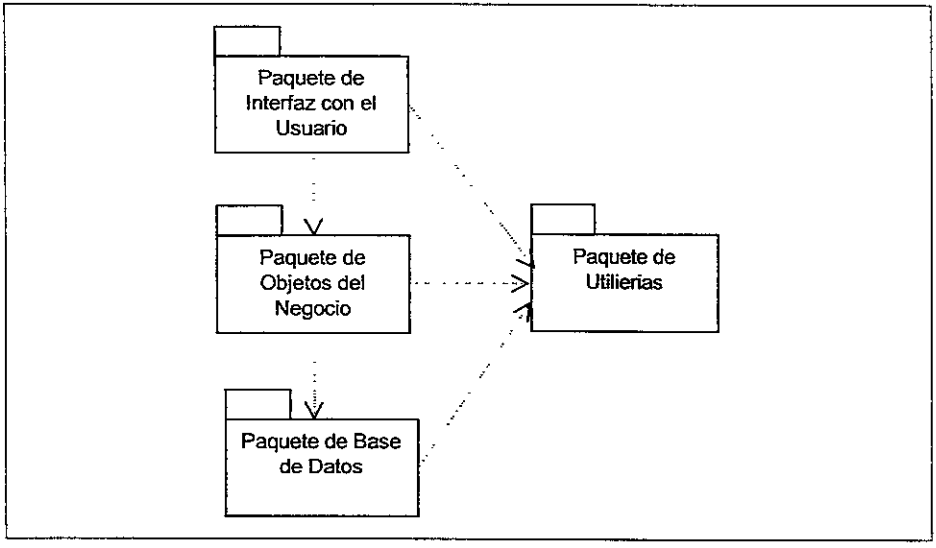

Figura 6-6 *Arquitectura del sistema de biblioteca.*

6.3.2 Diseño detallado

El propósito del diseño detallado es describir las nuevas clases técnicas, es decir, las clases de los paquetes de interfaz con el usuario y de base de datos y expandir y detallar la descripción de las clases de los objetos del negocio con las que actualmente se han solo esquematizado en el análisis. Esto es hecho creando nuevos diagramas de clases, diagramas de estados y diagramas dinámicos (como de secuencia, colaboración y diagrama de actividad). Estos son los mismos diagramas usados en el análisis, pero aquí son definidos en mayor detalle y nivel técnico. La descripción de casos de uso del análisis es utilizada para verificar que los casos de uso son manejados en el diseño; y los diagramas de secuencia son usados para ilustrar como cada caso de uso es técnicamente realizado en el sistema.

6.3.2.1 Paquete de Base de Datos.

La aplicación debe tener objetos guardados Persistentemente, por eso es necesario agregar una capa que provea este servicio. La solución natural en una aplicación de la vida real sería utilizar una base de datos comercial, ya sea una base de datos orientada a objetos o una tradicional base de datos relacional con una capa de traducción de objetos a tabla. Pero debido a que la aplicación del caso de estudio pretende ser portable y no requerir licencias de un vendedor específico, he decidido una solución más simple. Los objetos serán simplemente guardados en archivos en el disco. Detalles acerca del almacenamiento serán transparentes para la aplicación, que tendrá simplemente que llamar operaciones comunes como: guardar(), actualizar(); borrar(); buscar(), etc. Toda la implementación del almacenamiento Persistente será manejado por la clases llamada Persistente, la cual es una clase abstracta, la cual deberán heredar todas las clases que requieran almacenamiento Persistente. La clase Persistente requiere que sus subclases implementen las operaciones escribir() y leer() con código de cómo escribir sus atributos a un objeto archivo y como leer sus atributos de una objeto archivo.

Un factor importante en el almacenamiento es la clase ObjId, cuyos objetos son usados para referirse a cualquier objeto Persistente en el sistema. La clase ObjId es una técnica común para manejar referencias a objetos elegantemente. La clase ObjId es una clase general usada por todos los paquetes en el sistema (interfaz con el usuario, objetos del negocio y base de datos) por lo tanto se decidió ponerlo en un paquete llamado "utilerías". Este objeto es el único en este paquete y está muy relacionado al paquete de base de datos, pero ponerlo ahí hubiera significado que el paquete de interfaz con el usuario tuviera alguna dependencia con el paquete de base de datos, lo cual no es muy recomendable, hay que tratar de que la interfaz con el usuario solo tenga relación con los objetos del negocio.

La interfaz de la clase Persistente ha sido definida de tal manera que sería fácil cambiar la implementación del almacenamiento Persistente, algunas alternativas serían almacenar los objetos en una base de datos relacional o en una base de datos orientada a objetos En el diseño actual no hay nada que impida implementar esto de manera fácil.

6.3.2.2 Paquete de Objetos del Negocio

Este paquete está basado en el correspondiente paquete en el análisis donde se encontraron las clases del dominio. Las clases, sus relaciones y su comportamiento son preservados, solo se describe a las clases en mayor detalle incluyendo como son implementadas sus relaciones y comportamiento. Entre las cuestiones de implementación que contempla el diseño está el hecho de que todas las clases de objetos del negocio heredan la clase Persistente del paquete de base de datos e implementan las operaciones necesarias de lectura y escritura.

Las operaciones que se descubrieron en el análisis han sido detalladas, lo que significa que algunas de ellas han sido traducidas en diferentes operaciones en el modelo del diseño y algunas han cambiado de nombre. Esto es considerado normal, ya que en el análisis solo se bosquejo las características de cada clase y en el diseño se detalla dicha descripción del sistema Consecuentemente todas las operaciones en el diseño deben estar perfectamente definidas, su signatura y valores de retorno deben ser especificados. En la siguiente figura se muestra la evolución del modelo del análisis al modelo del diseño

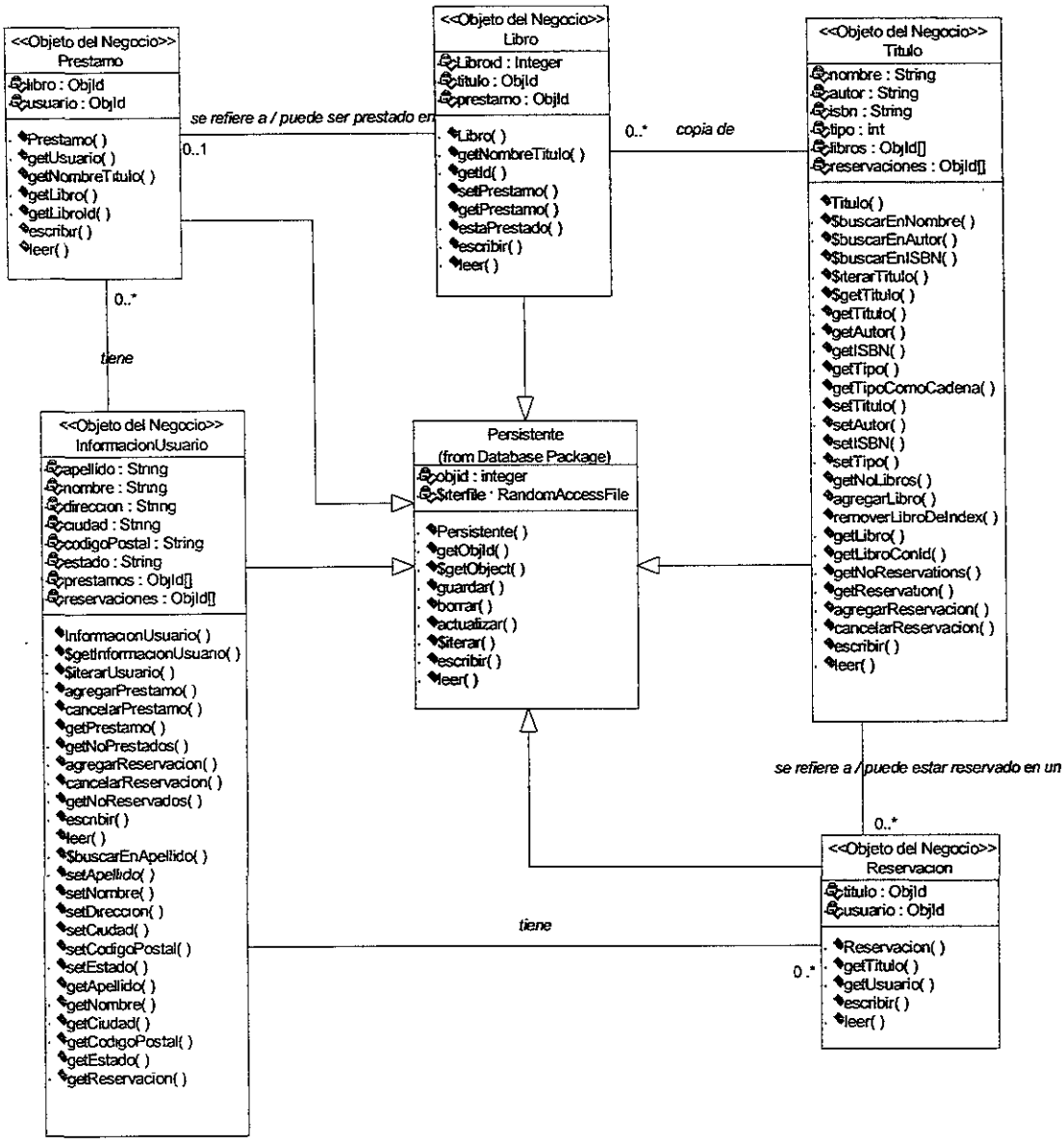

Figura 6-7 *Diagrama de clases resultado de la fase de diseño del sistema de biblioteca.*

Para mantener la simplicidad del ejemplo en el diseño no se consideran algunas cosas que podrían ser necesarias, como por ejemplo:

- No se verifica si los libros son regresados a tiempo.
- No se maneja el orden de las reservaciones.
- No se notifica a las personas con reservación cuando su libro ya está disponible.
- etc.

Los diagramas de estado hechos en el análisis también son detallados en el diseño, mostrando como los estados son representados y manejados en la solución propuesta. El de estados en el diseño para la clase Titulo es mostrado en la siguiente figura. Los estados son implementados en el diseño utilizando un arreglo llamado reservaciones, el cual contiene los identificadores de los objetos reservación asociados (ver también la asociación entre la clase Título y Reservación en el diagrama de clases). Cuando el arreglo tiene un tamaño de cero elementos (es decir, está vació), el objeto Título está en el estado de No Reservado, cuando su tamaño es de uno o más, su estado pasa a ser Reservado. Otros objetos pueden cambiar el estado del objeto Título al realizar una llamada a la operación agregarReservación() y eliminarReservación() como se muestra en el diagrama

Para ver como fue traducido el diagrama de estados del análisis en el correspondiente diagrama en el diseño se pueden comparar los diagramas de las figuras Figura 6-2 y Figura 6-7.

6.3.2.3 Paquete de Interfaz con el Usuario

El paquete de interfaz con el usuario se encuentra sobre los demás paquetes, es el que presenta los servicios e información del sistema al usuario. Está basado en la librería clases estándar de Java AWT (Abstract Window Toolkit), la cual sirve para escribir aplicaciones que tengan interfaz grafica con el usuario en Java Las clases de la librería AWT no son mostradas en el modelo del diseño, ya que ellas constituyen por si mismas un paquete cuya especificación se encuentra en la documentación del ambiente de desarrollo Java JDK.

La librería AWT tiene clases para diferentes tipos de ventanas (frames, cuadros de dialogo, etc) y para diferentes tipos de componentes gráficos como lo son etiquetas, botones, cuadros de edición, menús de selección, etc. También la librería AWT maneja los eventos que son generados por el usuario al interactuar con los objetos de la interfaz como lo son el click del mouse, el presionar cualquier tecla, el presionar la tecla de Intro, etc.

Debido a que toda la interacción con el usuario es iniciada desde algún elemento de la interfaz, los diagramas dinámicos del modelo del diseño han sido alojados en el paquete de Interfaz con el Usuario. Los diagramas dinámicos que se utilizaron fueron diagramas de secuencia, los cuales fueron generados, como en el modelo del análisis, a partir de los casos de uso, pero en esta ocasión se incluyó el detalle exacto de la secuencia de eventos y la interacción de los objetos. Estos diagramas son creados a través de iteraciones que los van detallando conforme van apareciendo más detalles, inclusive son detallados aún más cuando en la fase de implementación, es decir la etapa de codificación, se encuentran nuevos detalles.

A continuación se muestra el diagrama de secuencia para el caso de uso "Agregar Título", las operaciones y signaturas que aparecen son exactamente las mismas que aparecen en el código.

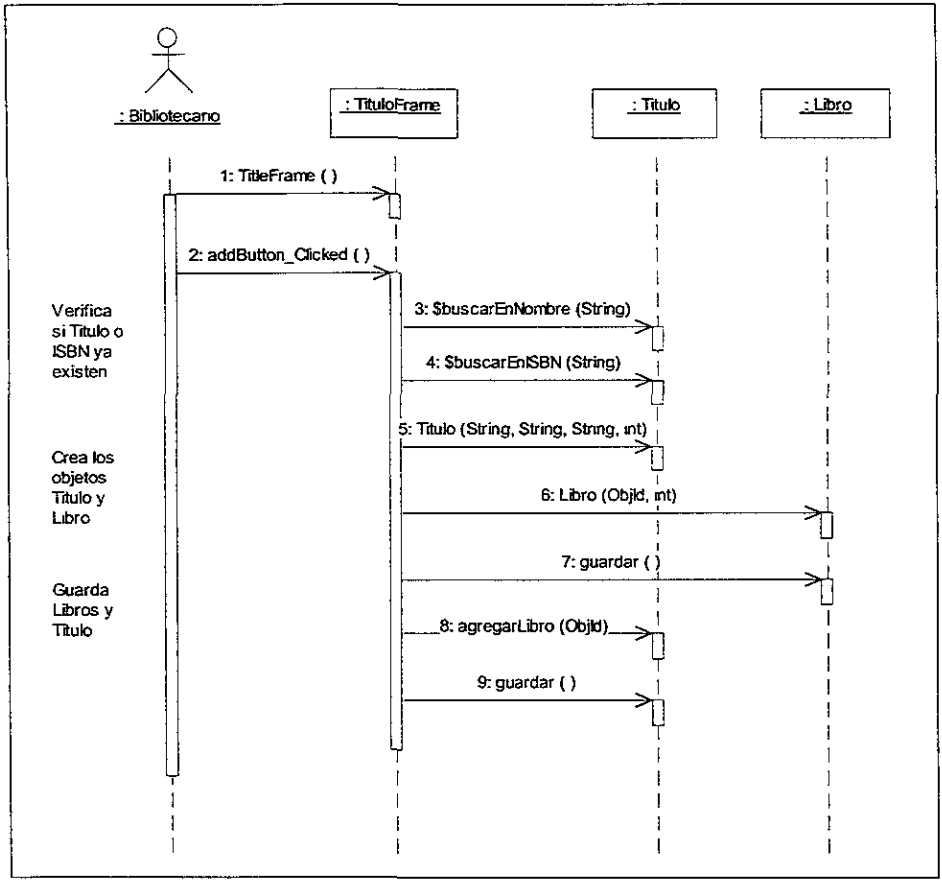

Figura 6-8 *Diagrama de secuencia en el la fase de diseño para el caso de uso "Agregar Título"*

Como alternativa a los diagramas de secuencia, se pueden utilizar diagramas de colaboración, sin embargo Rational Rose 4.0 no soporta la sintaxis completa de UML en los diagramas de colaboración, por esta razón se decidió realizar los diagramas dinámicos mediante diagramas de secuencia. Entre las cosas que no soporta Rose 4.0 es el esquema de numeración que se debe manejar en un diagrama de colaboración (1, 1.1, 1.2, 2, 2.1,*etc), en Rose 4.0 el esquema de numeración es mucho más simple, se utiliza* sencillamente un esquema de numeración secuencial (1,2,3, etc). Por esta razón un diagrama complejo de secuencia sería muy difícil de traducir en Rose 4.0 a un diagrama de colaboración, lo que no ocurriría si se tuviera soporte completo de la notación UML en

los diagramas de colaboración. Si se tuviera dicho soporte no habría ningún problema para traducir un diagrama de secuencia en un diagrama de colaboración.

A continuación se muestra el caso de uso anterior "Agregar título" traducido a un diagrama de colaboración utilizando la sintaxis completa de los diagramas de colaboración.

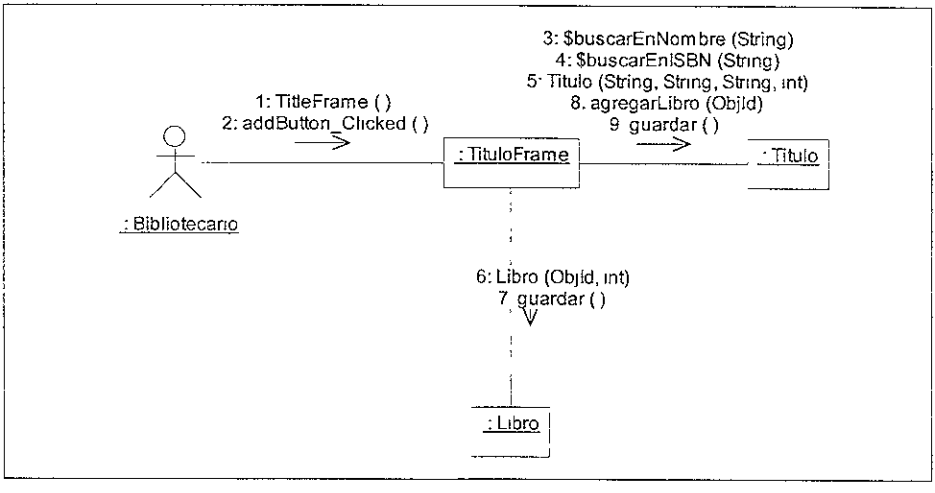

Figura 6-9 *Diagrama de colaboración del caso de uso "Agregar título".*

6.3.3 Diseño de la interfaz con el usuario

Una actividad especial que debe ser realizada paralelamente a las fases de análisis y de diseño es la creación de la interfaz con el usuario. El diseño de dicha interfaz debe seguir algunos lineamientos los cuales están mas haya del alcance de esta tesina, pero que sin embargo deben ser consultados para la creación de una interfaz adecuada. El objeto de dichos lineamientos es la creación de interfases estándar las cuales faciliten el entrenamiento del usuario, al no tener que aprender una interfaz nueva para cada sistema que maneja, otro de los objetivos es conservar la personalización de colores y tamaños que el usuario tenga definida en su estación de trabajo.

La interfaz con el usuario en el caso de estudio que nos ocupa está basada en los casos de uso y ha sido dividida en las siguientes secciones, cada una de las cuales cuenta con una entrada dentro del menú principal de la aplicación.

- *Funciones*: Ventanas para las funciones primarias del sistema que son: préstamo, devolución y reservaciones de títulos.
- *Información*: Ventanas para ver la información contenida en el sistema, la información que se puede consultar es acerca de los títulos y los usuarios.
- *Mantenimiento* Ventanas para realizar las actividades de mantenimiento del sistema como son: Agregar, actualizar y eliminar títulos, usuarios y libros.

Las ventanas de la aplicación pueden ser generadas con un ambiente de desarrollo como PowerJ, el cual permite gráficamente crear las ventanas y agregar los controles necesarios. Mediante dicho ambiente de desarrollo es fácil también agregar los eventos que manejaran las acciones que el usuario tome sobre los controles. Dichos eventos serán traducidos en métodos que serán llamados cuando el usuario realice la acción correspondiente (dispare el evento).

A continuación se muestra un ejemplo de uno de los diagramas de clases dentro del paquete de interfaz con el usuario, el correspondiente al menú de "Funciones".

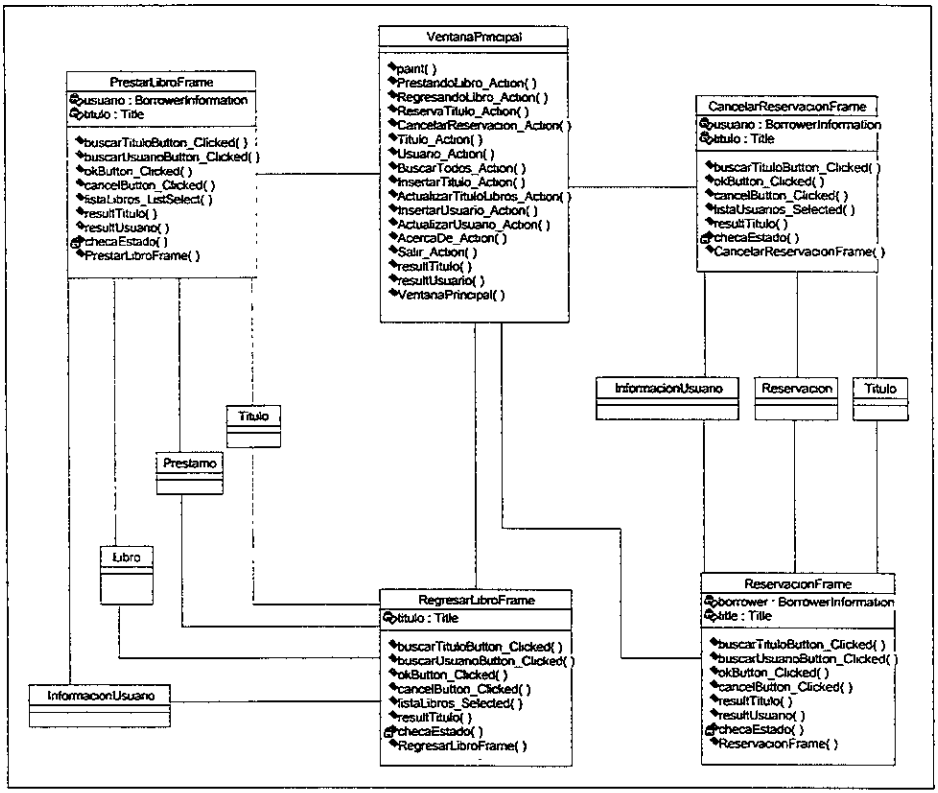

Figura 6-10 *Diagrama de clases del menú de Funciones dentro del paquete de Interfaz con el usuario.*

La interfaz con el usuario resultante está compuesta de una ventana principal que cuenta con un menú con las cuatro secciones principales antes descritas. Cada sección cuenta con un menú desplegable desde los cuales se puede acceder a cada una de las ventanas de la aplicación. A continuación se muestra un ejemplo de cómo se vería la interfaz, la cual como se puede observar, y por la simplicidad del ejemplo, es una interfaz muy sencilla.

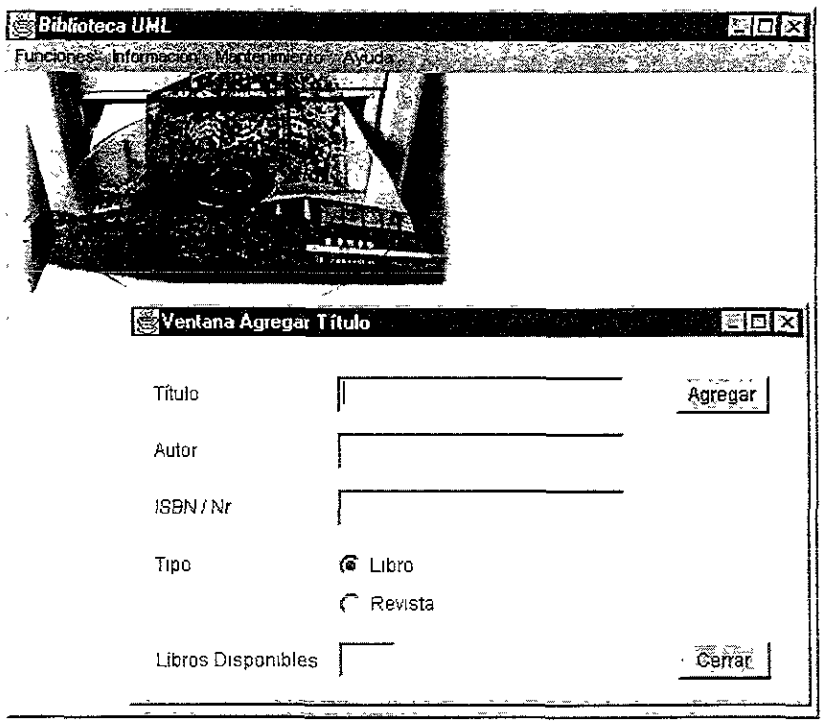

Figura 6-11 *Ejemplo de interfaz del sistema de biblioteca.*

6.4 Construcción (Implementation)

En la fase de construcción es donde son programadas las clases. El lenguaje seleccionado para la implementación fue Java debido a que los requerimientos se especificó que el sistema debería poder correr en diferentes plataformas. En esta *aplicación Java fue utilizado como un lenguaje de programación de propósito general y la* aplicación no ha sido implementada como un applete debido a que un applete no puede acceder al sistema de archivos en el lado del cliente. }

En Java cada una de las clases lógicas es codificada dentro de un archivo fuente, el cual debe tener el mismo nombre de la clase y una terminación .java, los clases ejecutables son traducidas a un archivo con el mismo nombre de la clase pero con la terminación .class. Lo anterior hace muy fácil identificar la relación entre los componentes de código y las clases lógicas.

La siguiente figura ilustra que los diagramas de componentes en el modelo del diseño contienen (en este caso) una simple relación uno a uno entre las clases de la vista lógica y los componentes de la vista de componentes De esta manera cada componente en la vista de componentes contiene una liga a la descripción de la clase en la vista lógica haciendo fácil navegar entre las diferentes vistas. Los paquetes en la vista lógica también

son asociados a paquetes con el mismo nombre en la vista de componentes. Las dependencias entre componentes no son mostradas en los diagramas de componentes, excepto para el paquete de objetos del negocio, ya que las dependencias pueden ser derivadas de los diagramas de clases en la vista lógica.

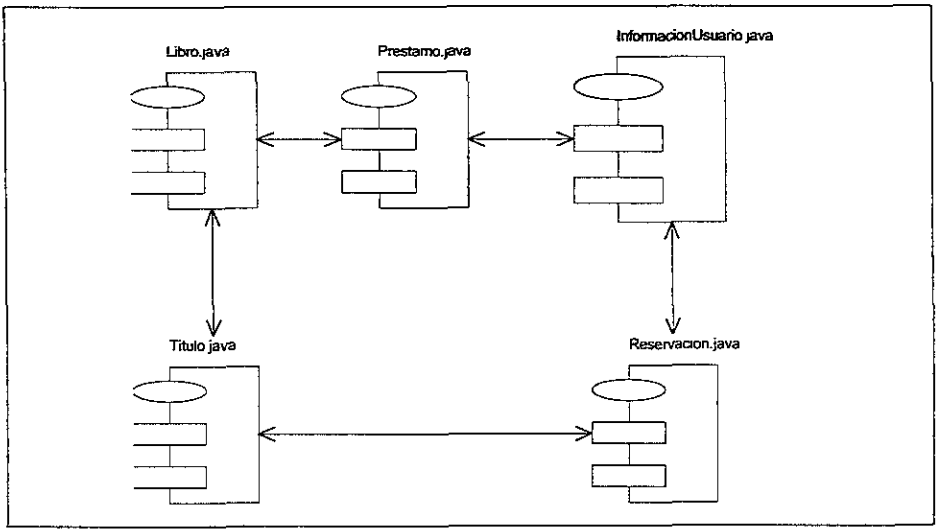

Figura 6-12 *Diagrama de componentes del sistema de biblioteca.*

En el caso de nuestro ejemplo los componentes fueron asociados a cada una de las clases en la vista lógica, debido a que en Java las relaciones entre componentes de software y clases lógicas son uno a uno, sin embargo en otros lenguajes de programación se tendrá normalmente componentes que agrupan a varias clases. Lo más común en estos casos es tener un componente de software que agrupa a todos los elementos de un paquete en la vista lógica.

Para la codificación las especificaciones son tomadas de los siguientes elementos y diagramas del modelo del diseño:

- *Especificación de clases:* En la especificación de cada clase se muestra en detalle los atributos y operaciones necesarias en cada clase.
- *Diagramas de clases:* En los cuales se muestra la relación con otras clases.
- *Diagramas de estados:* En donde se muestran los posibles estados y transiciones que necesitan ser manejados junto con las operaciones que disparan dichas transiciones.
- *Diagramas dinámicos:* (secuencia, colaboración) en donde se muestra la implementación de un método específico de la clase y cómo otros objetos usan los objetos de la clase.
- *Diagramas de casos de uso:* Estos diagramas muestran el resultado que el sistema debe proporcionar y son utilizados por el programador para saber como será utilizado el sistema y que resultados se esperan.

Naturalmente dentro de la fase de construcción serán encontradas algunas deficiencias u omisiones dentro del modelo del diseño, dichas deficiencias deberán ser corregidas por el codificador y el modelo deberá ser actualizado para contemplar dichas diferencias. Está iteración con el modelo del diseño es muy importante, ya que dicho modelo servirá como documentación del sistema.

A continuación se presenta como ejemplo la implementación de la clase Préstamo, la cual es una clase de objetos del negocio que guarda información acerca de un préstamo. Mucha de su funcionalidad es heredada de la clase Persistente del paquete de Base de Datos. Al leer el código con los diagramas UML en mente podemos observar como han sido traducidos los elementos de UML en pedazos de código considerando entre otros los siguientes puntos:

- La especificación del paquete de Java al que pertenece la clase (primera línea), es el equivalente en código de la especificación del paquete en la vista lógica al que pertenece la clase.
- Los atributos privados de las clases en Java corresponden a los atributos especificados en el modelo; y naturalmente, los métodos de Java corresponden a las operaciones del modelo.
- La clase ObjId (object identifiers) es invocada para implementar asociaciones, lo que significa que las asociaciones son normalmente salvadas junto con la clase, ya que la clase ObjId es Persistente.

```
//
// Prestamo.java: representa un préstamo. El préstamo se refiere
// a un título y a un usuario.
//
package bo;
import util.ObjId;
import db.*;
import java.io.*;
import java.util.*;
public class Prestamo extends Persistente
{
      private ObjId libro;
      private ObjId usuario;

      // Constructor de la clase
public Prestamo(ObjId lib, ObjId lec)
      {
            libro = lib;
            usuario = lec;
      }
      public InformacionUsuario getUsuario()
      {
InformacionUsuario ret = (InformacionUsuario) &
Persistente.getObject(usuario);
return ret;
```

```
    }

    public String getNombreTitulo()
    {
        Libro lib = (Libro) Persistente.getObject(libro);
        return lib.getNombreTitulo();
    }
    public Libro getLibro()
    {
        Libro lib = (Libro) Persistente.getObject(libro);
        return lib;
    }
    public int getLibroId()
    {
        Libro lib = (Libro) Persistente.getObject(libro);
        return lib.getId();
    }
    public void escribe(RandomAccessFile out)
        throws IOException
    {
        libro.escribe(out);
        usuario.escribe(out);
    }
    public void lee(RandomAccessFile in)
        throws IOException
    {
        libro = new ObjId();
        libro.lee(in);
        usuario = new ObjId();
        usuario.lee(in);
    }
}
```

6.5 Pruebas y distribución

La aplicación tiene, por supuesto, que ser probada. Los casos de uso originales tienen que ser probados al final de la aplicación para determinar si el sistema los soporta y si se comportan como fue definido en la descripción de casos de uso. La aplicación también debe ser probada, de forma mas informal, por un usuario que interactúe con el sistema y verifique que la funcionalidad sea la que se especificó. En un sistema de la vida real, se requiere una especificación más formal de las pruebas a realizar y de los errores reportados por los usuarios.

Para la distribución del sistema, que no es otra cosa que su liberación, se requiere incluir su documentación. Para nuestro ejemplo la documentación son los modelos presentados en este capitulo, pero en un proyecto de la vida real la documentación debe contener

manuales para el usuario, descripciones conceptuales del producto (de mercadotecnia) y la especificación de requerimientos para ejecutar el sistema. Además de esto se debe incluir un diagrama de distribución que muestra la arquitectura física en la que será distribuido el sistema. A continuación se muestra el diagrama de distribución para nuestro caso de estudio.

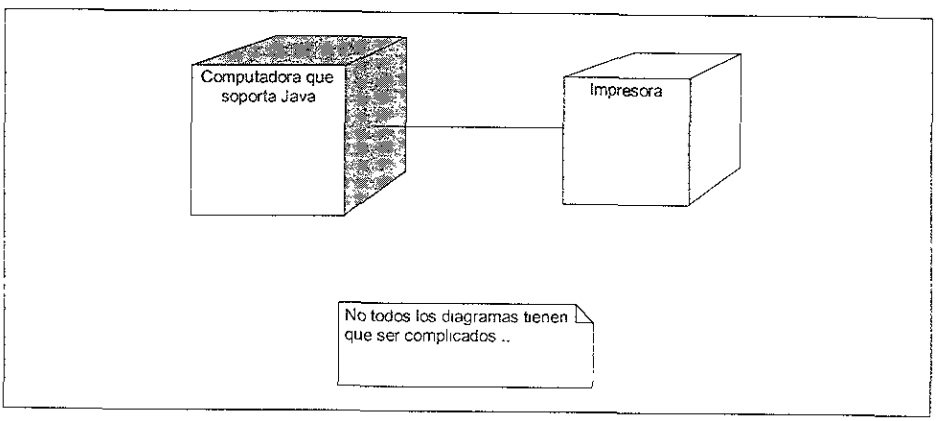

Figura 6-13 *Diagrama de distribución del sistema de bilbioteca.*

CONCLUSIONES

En la actualidad han surgido una gran variedad de métodos de desarrollo de sistemas que intentan dar mayor importancia a las labores fundamentales en el desarrollo de un sistema que son: el análisis y el diseño. Estos métodos implementan los ya no novedosos conceptos de orientación a objetos que además de brindar grandes ventajas en el momento de la codificación, brindan también grandes ventajas en las etapas anteriores a la codificación, que como ya se dijo son las más importantes. Esta variedad de métodos, todos ellos con su propia y única notación y herramientas, han dejado a muchos desarrollados confundidos, ya que muchos de ellos deben enfrentarse no solo a la pronunciada curva de aprendizaje que representa entender y aplicar los conceptos de orientación a objetos sino a una multitud de notaciones utilizadas para cada uno de estos métodos de desarrollo. La falta de una bien establecida notación que pueda ser aplicada entre los diferentes métodos y herramientas habían hecho más difícil aprender como usar un buen método, hasta el surgimiento de UML, que es un lenguaje que sintetiza y estandariza la notación utilizada en los más importantes métodos de desarrollo de sistemas orientados a objetos.

La tendencia de la industria del software se centra en la necesidad de crea modelos de los sistemas a construir, ya que los sistemas tienden cada vez a ser más grandes y complejos. Se hace de vital importancia la creación de modelos que especifique todos estos factores y sirvan como una especie de contrato entre el usuario y el desarrollador, en el que ambas partes están de acuerdo en que el modelo representa el sistema a construir. La importancia de los modelos ha sido ampliamente probada en otras áreas de la ingeniería, como la Ingeniería Mecánica, Ingeniería Civil, Ingeniería Eléctrica, etc. Y la Ingeniería de Software no tiene porque ser una excepción.

UML surge entonces como un intento de resolver los problemas arriba descritos (la necesidad de una notación estándar entre los métodos y la necesidad de creación de modelos). UML es un lenguaje estándar de modelado de sistemas y es "ya" en este momento un estándar de facto en la industria de desarrollo de software.

Sin embargo, UML no es un método, es simplemente un lenguaje de modelado que se puede aplicar a distintas metodologías que se apeguen a este estándar. El proceso o pasos a seguir son especificados por las metodologías. Sin embargo, en fechas recientes, ha surgido un intento por definir un proceso estándar de desarrollo de sistemas que junto con la notación (UML) conformen una metodología de desarrollo de sistemas. Este proceso se llama (RUP) Rational Unified Process, el cual está siendo creado por las mismas personas que desarrollaron UML, que son a su vez los investigadores más reconocidos en materia de metodologías de orientación a objetos.

En mi experiencia laboral, he observado por increible que parezca, que la mayoría de las organizaciones responsables del desarrollo de software en nuestro país, no cuentan con una metodología de desarrollo de sistemas. Cuando tratan de especificar una, tratan de encontrar el hilo negro, especificando su propia metodología. UML y RUP son un lenguaje y procedimiento creados por los más distinguidos investigadores y desarrolladores de sistemas orientados a objetos, han sido probados y refinados en proyectos de gran complejidad y tamaño. En su conjunto podría ser una opción muy importante para dichas organizaciones en su intento de adoptar una metodología de desarrollo de sistemas.

Y como última conclusión, me gustaría resaltar la importancia de que alumnos de la carrera de Matemáticas Aplicadas y Computación, Ingeniería en Computación y afines, estudien este tema. Ya que como lo mencione antes, UML se está convirtiendo en un estándar dentro de la industria de desarrollo de software y el mantenerse actualizado en una industria tan vertiginosamente cambiante como lo es la de las Tecnologías de Información (IT), cobra vital importancia.

BIBLIOGRAFÍA

- RUMBAUGH, JAMES; JACOBSON, IVAR; BOOCH GRADY. "The Unified Modeling Language Reference Manual". *Editorial Addison Wesley Longman, Inc.*, Diciembre, 1998.
- ERIKSSON, HANS-ERIK Y PENKER, MAGNUS. "UML Toolkit". Editorial John Wiley & Sons, Inc., 1998.
- FOWLER, MARTIN con KENDALL SCOTT. "UML Distilled: Applying the Standard Object Modeling Language". Editorial Addison Wesley Longman, Inc., Diciembre, 1997.
- LARMAN, CRAIG. "Applying UML and Patterns. An Introduction to Object-Oriented Analysis and Design". Editorial Prentice Hall PTR., 1998.
- BOOCH GRADY; RUMBAUGH, JAMES; JACOBSON, IVAR. "The Unified Modeling Language User Guide". Editorial Addison Wesley Longman, Inc., Diciembre, 1998
- BOOCH, GRADY. "Object-Oriented Analysis and Design with Applications". 2da Edición. Editorial Addison Wesley Longman, Inc., Noviembre, 1997.
- JACOBSON, IVAR [et. al.] "Object-Oriented Software Engineering: a Use Case Driven Approach". Editorial Addison Wesley Longman Limited, 1996.

- RUMBAUGH, JAMES. [et. al.] "Object-Oriented Modeling and Design". General Electric Research and Development Center Schenectady, New York, Editorial Prentice Hall, Inc., 1991.

- http://www.rational.com/uml/index.jsp