UNIVERSIDAD AUTONOMA DE NUEVO LEON

FACULTAD DE CIENCIAS FISICO-MATEMATICAS

UML (UNIFIED MODELING LANGUAGE) CON APLICACIONES

TESIS

QUE PARA OBTENER EL TITULO DE

LIC. EN CIENCIAS COMPUTACIONALES

PRESENTA

ALFREDO MEZQUITIC CASTILLO

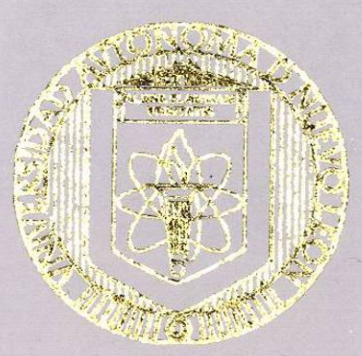
SAN NICOLAS DE LOS GARZA, N. L.

JUNIO DE 1999

TL
QA76
.9
.O35
M49
1999
c.1

UNIVERSIDAD AUTONOMA DE NUEVO LEON

FACULTAD DE CIENCIAS FISICO-MATEMATICAS



UML (UNIFIED MODELING LANGUAGE) CON APLICACIONES

TESIS

QUE PARA OBTENER EL TITULO DE
LIC. EN CIENCIAS COMPUTACIONALES

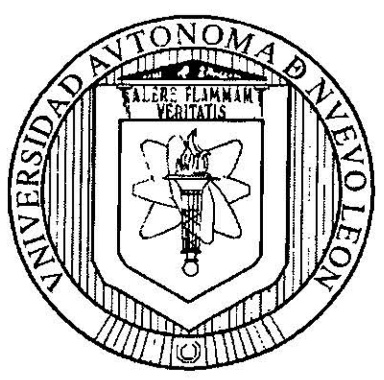
PRESENTA

ALFREDO MEZQUITIC CASTILLO

SAN NICOLAS DE LOS GARZA, N. L.
JUNIO DE 1999

UNIVERSIDAD AUTÓNOMA DE NUEVO LEÓN

FACULTAD DE CIENCIAS FÍSICO MATEMÁTICAS



UML (Unified Modeling Language) con Aplicaciones

TESIS

Que para obtener el título de
Lic. en Ciencias Computacionales

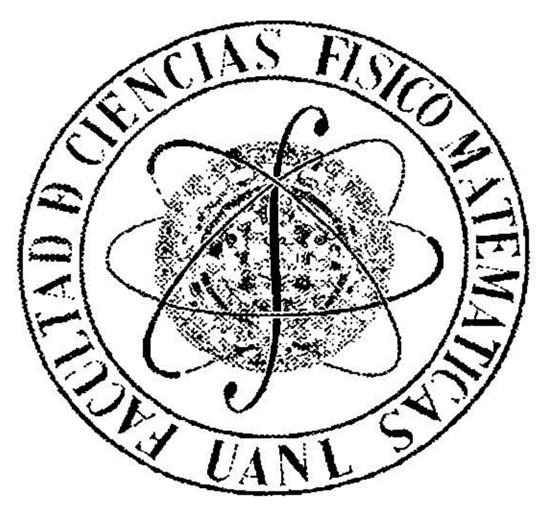
Presenta:
Alfredo Mezquitic Castillo

JUNIO 1999 San Nicolás de los Garza

UANL
FONDO

UNIVERSIDAD AUTÓNOMA DE NUEVO LEÓN

FACULTAD DE CIENCIAS FÍSICO MATEMÁTICAS



UML
(Unified Modeling Language) con Aplicaciones

Alfredo Mezquitic Castillo

Asesor
Ing. Gerardo Manuel Valdés Macías

JUNIO 1999 San Nicolás de los Garza

INDICE

INTRODUCCIÓN **4**

HISTORIA DEL UML **5**

PROLIFERACIÓN DE MÉTODOS ORIENTADOS A OBJETOS 5

SE ACERCAN BOOCH Y OMT 6

UNIFICACIÓN DE MÉTODOS 6

MODELO Y METAMODELO 7

CASOS DE USO **10**

DIAGRAMA DE CASOS DE USO 11

USES Y EXTENDS 12

DIAGRAMA DE CLASES **14**

PERSPECTIVAS 15

ASOCIACIONES 17

ATRIBUTOS 21

OPERACIONES 22

GENERALIZACIÓN 22

DIAGRAMAS DE ACTIVIDAD **24**

DIAGRAMAS DE DESARROLLO **27**

DIAGRAMAS DE ESTADO **30**

DIAGRAMAS DE ESTADO CONCURRENTES 35

DIAGRAMAS DE INTERACCIÓN **37**

DIAGRAMAS DE SECUENCIA 38

Proceso Concurrentes y Activaciones 40

DIAGRAMAS DE COLABORACIÓN 42

COMPARACIÓN DE DIAGRAMAS DE SECUENCIA Y DE COLABORACIÓN 43

DIAGRAMAS DE PAQUETES **44**

CASO PRÁCTICO **48**

Configuración 50

Identificación 50

Modificación de puertas 50

Modificación de una persona ... 51
Buscar puertas accesibles a una persona dada ... 53
Buscar los grupos en los que está una persona ... 53
Buscar las personas contenidas en un grupo ... 54
Modificación de Acceso para un grupo de personas para una puerta. ... 55
Control de Acceso ... *56*
Autorización de Entrada ... 56
Configuración ... *57*
Identificación de supervisor. ... 57
Modificación de puertas ... 60
Modificación de una persona ... 61
Modificación de grupos de personas ... 62
Buscar puertas accesibles a una persona dada ... 63
Buscar los grupos en los que esta contenida una persona ... 64
Buscar las personas contenidas en un grupo ... 65
Modificación de Acceso para un grupo de personas para una puerta. ... 66
Autorización de Entrada ... 68

GUÍA PARA HACER LA TRANSICIÓN DE BOOCH Y OMT A UML ... 71

CONSTRAINTS ... 71
NOTAS ... 72
SUBSITEMAS ... 72
OBJETOS ... 73
LINKS ... 73
MENSAJES ... 74
ETIQUETAS DE MENSAJES ... 75
FLUJO DE DATOS ... 75
CLASES ... 76
Clases Simples ... *76*
Atributos y Operaciones ... *76*
Visibilidad ... *77*
RELACIONES ... 77
Asociaciones ... *77*
Asociaciones de Clases ... *78*
Agregación ... *78*
Dependencia ... *79*
Herencia ... *80*

GENERACIÓN DE CODIGO C++ ... 81

CLASE VACÍA ... 81

CLASE CON ATRIBUTOS Y OPERACIONES 82
ASOCIACIONES 83
Asociación 1 a 1 .... *83*
Asociación N a 1 .... *84*
ASOCIACIÓN DE CLASES 84
ASOCIACIÓN DE CLASES N A N 85
HERENCIA 86
Herencia Simple *86*
HERENCIA MÚLTIPLE 86

GENERACIÓN DE CODIGO JAVA 87

CLASE VACÍA 87
CLASE CON ATRIBUTOS Y OPERACIONES 87
ASOCIACIONES 88
Asociación 1 a 1 .... *88*
Asociación 1 a N *88*
HERENCIA 89
Herencia Simple *89*
Herencia Múltiple *90*

GENERACIÓN DE CODIGO VBASIC 91

CLASE VACÍA 91
CLASE CON ATRIBUTOS Y OPERACIONES 91
ASOCIACIONES 92
Asociación 1 a 1 .... *92*
Asociación N a 1 .... *92*
HERENCIA 93
Herencia Simple *93*

BIBLIOGRAFÍA 94

LIBROS RECOMENDADOS 95

Introducción

El Lenguaje de Modelación Unificado (UML Unified Modeling Language), es una técnica que es una estandarización de los métodos de modelación más importantes que actualmente existen. Utiliza lo mejor de las técnicas de modelación actuales por lo que su aceptación fue muy buena dentro del ámbito de sistemas.

Poco a poco esta tecnología ha entrado a México y se espera que para los próximos años sea una de las principales herramientas para la modelación de sistemas.

Esta Tesis da una vista general de lo que es el UML, muestra sus ventajas, su facilidad de uso y espera fomentar el interés hacia esta nueva tecnología.

Historia del UML

Proliferación de Métodos Orientados a Objetos

A principios de los años 90's se vió un nacimiente de cerca de 50 diferentes métodos Orientados a Objetos (OO). Este crecimiento fue señal de una gran vitalidad de la tecnología OO, pero también trajo de una multitud de interpretaciones de lo que exactamente "es" un objeto. La abundancia de estas metodologías creó una gran confusión, los usuarios optaron por la actitud de "espera y ve" que limito el progreso hecho por estos métodos. La mejor manera de probar algo es desarrollándolo, los métodos evolucionan en base a la retroalimentación de los usuarios.

Afortunadamente, un acercamiento de los métodos dominantes permitió la extracción de un consenso acerca de las principales características de los objetos, compartidos por numerosos métodos, artículos acerca de los conceptos de clases (descritos por James Rumbaugh), partición dentro de subsistemas (Grady Booch), y mostrar los requerimientos basados sobre el estudio de la interacción entre los usuarios y los sistemas (Ivar Jacobson casos de uso).

Finalmente, métodos bien desarrollados, tales como Booch y OMT (Object Modeling Technique), fueron respaldados por experiencia, y adoptaron los elementos de metodología que fueron los más aceptados por los usuarios.

Se acercan Booch y OMT

La segunda generación de los métodos de Booch y OMT, llamados Booch'93 y OMT-2, muy más parecidos uno al otro. Las diferencias encontradas fueron mínimas, y pertenecían principalmente terminología y tecnología. Booch'93 fue influenciado por OMT y adopta asociaciones, diagramas Harey y trazo de eventos. Del otro lado, OMT-2 fue influenciado por Booch e introduce mensajes de flujo, modelos de herencia y subsistemas, y componentes de modelo. Lo más importante, se removieron los diagramas de flujo del modelo funcional.

Unificación de Métodos

La unificación de los métodos de modelación OO llegó a ser posible por la experiencia que permitió la evaluación de varios conceptos propuestos por los métodos existentes. Basados sobre el hecho de que las diferencias entre los métodos llegaron a ser muy pequeñas, y que la guerra entre los métodos no permitía crecer a los métodos, Jim Rumbaugh y Grady Booch decidieron a finales de 1994 unificar su trabajo en un método sencillo: el **Método Unificado**. Un año después se unieron a Ivar Jacobson, el padre de los casos de uso.

Booch, Rumbaugh y Jacobson se propusieron 4 objetivos:

1. Representar sistemas completos (en lugar de mostrar sólo una porción de software).
2. Establecer un explícito acoplamiento entre los conceptos y el código ejecutable.
3. Tomar en cuenta la escala de factores que son inherentes para los sistemas complejos y críticos.
4. Crear un lenguaje de modelación usable entre los humanos y las máquinas.

La primera versión de la descripción del Método Unificado fue presentada en octubre de 1995, en un documento titulado *Unified Method V0.8*. Este documento fue ampliamente distribuido y los autores recibieron miles de comentarios detallados de la comunidad usuaria. Estos comentarios fueron tomados en cuenta en la versión 0.9, liberada en junio de 1996. Sin embargo, la versión 0.91, lanzada en octubre de 1996 fue la que represento una evolución sustancial del Método Unificado.

La principal modificación fué un cambio en la dirección del esfuerzo por la unificación, así que el primer objetivo fue la definición de un lenguaje universal para la modelación OO, y la estandarización de los procesos de desarrollo OO deberían hacerse después. El Método Unificado fué transformado en el **Unified Modeling Language** (UML, Lenguaje de Modelación Unificado). Para el desarrollo OO.

Modelo y Metamodelo

El esfuerzo inicial fue enfocado hacia la definición de conceptos fundamentales de la semántica - la construcción de bloques de modelación OO. Estos conceptos son los artefactos de los procesos de desarrollo, y debe ser intercambiable entre las diferentes partes que envuelven a un proyecto. Para implementar estos intercambios, fue necesario primero, darle importancia relativa a cada concepto, para estudiar las consecuencias de estas opciones, y seleccionar una representación gráfica, de la cual la sintaxis debe ser simple, intuitiva y expresiva.

Para facilitar este trabajo de definición y ayudar a formalizar el UML, todos los diferentes conceptos tienen que ser a sí mismos modelados usando el UML. Esta definición recursiva, llamada **metamodelación**, tiene una doble ventaja, la de permitir la clasificación de conceptos por un nivel de abstracción, por una complejidad y un dominio de aplicación, mientras se garantiza una notación con un poder de expresión tal que puede ser usado para representarse a sí mismo.

Un metamodelo describe formalmente los elementos del modelo, la sintaxis y la semántica de la notación permite su manipulación. El agregar abstracción introducida por la construcción de un metamodelo facilita el descubrimiento de potenciales inconsistencias, y promete generalización. El metamodelo de UML es usado como guía de referencia para la construcción de herramientas, y la partición de modelos entre diferentes herramientas.

Un modelo es una descripción abstracta de un sistema o un proceso, una representación simplificada que promueve su comprensión y permite su simulación. El término **modelación** es frecuentemente usado como un sinónimo de análisis, que es, la descomposición dentro de elementos simples que son más fáciles de comprender. En la ciencia de la computación, la modelación inicia con la descripción de un problema, y entonces se describe su solución. Estas actividades son llamadas respectivamente "análisis" y "diseño".

La forma de un modelo depende de un metamodelo. La modelación funcional descompone tareas dentro de funciones que son más sencillas de implementar. La modelación OO descompone sistemas en objetos de colaboración. Cada metamodelo define elementos del modelo y reglas para la composición de estos elementos de modelo.

El contenido del modelo depende del problema. Un lenguaje de modelación como UML es suficientemente general para ser usado en todos los dominios de la Ingeniería de Software, y más allá -podría aplicarse a la Ingeniería de Negocios por ejemplo.

Un modelo es la unidad básica del desarrollo; es altamente consistente en sí mismo y libremente agrupada con otros modelos por conexiones de navegación. Como una regla, un modelo relata una fase específica del desarrollo y se construye desde los elementos del modelo con sus diferentes representaciones de asociación.

Un modelo no esta directamente visible por los usuarios. Captura las principales semánticas de un problema y contiene acceso a los datos por herramientas para facilitar el intercambio de información, generación de código, navegación, etc. UML define muchos modelos para representar sistemas:

- Los modelos de clases capturan la estructura estática.
- Los modelos de estado expresan el comportamiento dinámico de los objetos.
- El modelo de los casos de uso describe los requerimientos de los usuarios.
- Los modelos de interacción representan los escenarios y flujo de mensajes.
- Los modelos de implementación muestran como trabajan las unidades.
- Los modelos de desarrollo dan detalles que pertenecen a la alojación de los procesos.

Los modelos son visualizados y manipulados por los usuarios por el significado de las representaciones gráficas, las cuales son proyecciones de los elementos contenidos en uno o más modelos. Muchas perspectivas diferentes pueden ser construidas por un modelo base, cada uno puede mostrar todo o una parte del modelo, y cada uno tiene su respectivo diagrama. UML define 9 tipos de diagramas.

- Diagramas de Casos de Uso
- Diagramas de Objetos
- Diagramas de Clases
- Diagramas de Secuencias
- Diagramas de Colaboración
- Diagramas de Actividad
- Diagramas de Estados
- Diagramas de Componentes
- Diagramas de Desarrollo

Diferentes notaciones pueden ser usadas para representar el mismo modelo. Las notaciones Booch, OMT y OOSE usan diferentes sintaxis gráfica, pero todos ellos representan los mismos conceptos de OO.

En resumen, UML es simplemente otra representación gráfica de un modelo de semántica común. Sin embargo, por la combinación de los elementos más usados en los métodos OO, extienden la notación para cubrir nuevos aspectos del desarrollo de un sistema, el UML da una notación comprensiva para todo el ciclo de vida de un proyecto.

Casos de Uso

Además de la introducción de los casos de uso como elemento primario en el desarrollo de Software, Jacobson (1994) también introdujó un diagrama para la visualización de los casos de uso. El **diagrama de casos de uso** es parte del UML.

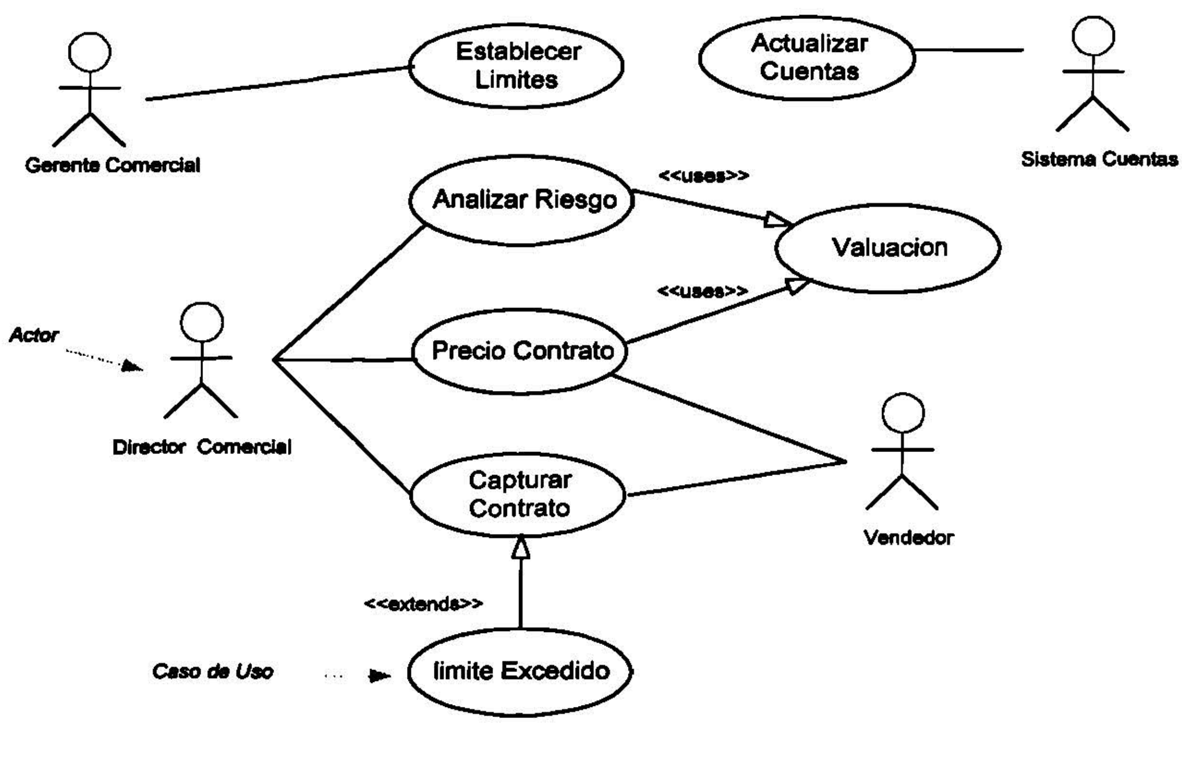


Fig. 1

Diagrama de casos de uso

Empecemos definiendo a los actores.

Actores

Un actor es un rol que un usuario lleva a cabo con respecto a un sistema. En la figura 1 hay cuatro actores. El Director Comercial, el Comerciante, el Vendedor y el Sistema de Cuentas.
Habrá probablemente muchos comerciantes en una organización dada, pero en lo que al sistema concierne, todos ellos jugarán el mismo rol. Un usuario podría jugar más de un rol. Por ejemplo, un Gerente Comercial también podría jugar un rol de Director Comercial y también regularmente de comerciante; un comerciante podría ser un vendedor. Cuando tratas con actores, es mucho más importante pensar en sus roles que en personas o en puestos de trabajo.

Un actor puede llevar a cabo muchos casos de uso; e inversamente, un caso de uso puede tener muchos actores que lo usen.

En la práctica, se encuentra que los actores son más usados cuando intentan especificar los casos de uso. En un sistema grande, frecuentemente puede ser difícil encontrarse con una lista de casos de uso. Es más fácil en esas situaciones que llegue una lista de actores primero, y después intentar trabajar para encontrar los casos de uso para cada actor.

Hay que hacer notar que los casos de uso no necesariamente deben de ser personas, aún y cuando los actores son representados por figuras de rayas en los diagramas de casos de uso. Un actor también puede ser un sistema externo que necesita información del sistema. En la figura 1, podemos ver la necesidad de actualizar las cuentas de un sistema de Cuentas.

El hecho de interactuar con sistemas externos causa muchas confusiones y variaciones de estilo entre los usuarios de los diagramas de casos de uso.[1]

[1] Algunos sienten que se debería de mostrar actores de tipo sistema externo sólo cuando son los únicos que utilizan el caso de uso. Así, si el sistema genera un archivo cada noche que es usado por el Sistema de Cuentas, el Sistema de Cuentas es un actor relevante porque es el único que utiliza el archivo producido.

Cuando uno trabaja con actores y casos de uso, no deberíamos de preocuparnos demasiado acerca de las relaciones que existen entre ellos. Más adelante, lo que importa son los casos de uso; los actores son sólo una manera de obtenerlos. Así que como se vayan obteniendo todos los casos de uso, no deberíamos fijarnos en los detalles de los actores.

Una situación en la que los actores toman importancia, sin embargo, es en la configuración del sistema para diferentes tipos de usuarios. Si tu sistema tiene casos de uso que corresponden con funciones de usuarios de un nivel alto (administradores del sistema), tu puedes usar al actor y las ligas de los casos de uso para los perfiles de cada usuario. Cada usuario debería de tener una lista de actores asociados, la cual deberías de usar para determinar que caso de uso puede el usuario llevar a cabo.

Otra razón para utilizar a los actores es la necesidad que envuelve el conocer quien quiere cual caso de uso. Esto también puede ser usado en la especificación de seguridad.

Una buena forma de identificar los casos de uso son los eventos externos. Pensar en todos los eventos desde el mundo exterior a los cuales uno quiere reaccionar. Un evento dado puede permitir reaccionar al sistema sin que envuelva a los usuarios, ó puede causar una reacción primaria desde los usuarios. Identificando los eventos que se necesitan para responder con alguna acción, ayudará a identificar los casos de uso.

Uses y Extends

Además de las uniones entre actores y casos de uso, existen otros dos tipos de relaciones. Estas son representadas por las relaciones **uses** y **extends** entre los casos de uso.

Uno puede usar la relación de **extend** cuando se tiene un caso de uso similar a otro, pero este hace un poco más.

En el ejemplo mostrado, el caso de uso básico es Capturar un Pedido. Este caso es válido cuando todo sigue un curso normal. Pero que pasa, cuando esto no ocurre normalmente, por ejemplo que un cliente exceda su límite de crédito. Aquí no llevaremos un comportamiento usual asociado para un caso de uso dado; haremos una variación.

Esta variación se podría agregar al caso de uso Capturar Pedido. Sin embargo, esto tendría que hacerse con mucha lógica especial, lo cual podría oscurecer el flujo "normal".

Otra forma de hacer esta variación es poner el comportamiento normal en un caso de uso y el comportamiento inusual en otro más.

La relación **uses** ocurre cuando se tienen algunos comportamientos que son similares a través de más de un solo caso de uso y no se quiere mantener una copia de esta descripción de comportamiento. Por ejemplo, el Analiza Riesgo y Negocia Precio requieren un valor de pedido. Así que para evitar duplicidad en la información, se genera un caso de uso para el Valor Pedido y se referencía desde los casos de uso originales.

Los dos tipos de relaciones implican diferentes cosas en su asociación con los actores. En el caso de los *extends*, los actores tienen una asociación al caso de uso cuando esta siendo extendida. Esto es asumido para un actor dado que llevará a cabo el caso de uso base y todas sus extensiones. Con las relaciones *uses*, frecuentemente no hay actores asociados con estos casos de uso comunes. Y si lo hubiera, el actor no es considera para realizar estos casos de uso.

Las reglas que aplican son las siguientes.

- Usar *extends* cuando se esta describiendo una variación sobre un comportamiento normal.
- Usar *uses* cuando se esta repitiendo elementos en dos o más casos de uso y se quiere evitar la repetición.

Posiblemente han escuchado el término **escenario** junto con los casos de uso. Esta palabra es usada inconsistentemente. Algunas veces, los escenarios son usados como un sinónimo de los casos de uso. Con el UML, el escenario se refiere a una ruta sencilla a través de un caso de uso, uno que muestra una combinación particular de condiciones con esos casos de uso. Por ejemplo, si se quieren ordenar unas donas, deberíamos tener un caso de uso con muchos escenarios: uno en el cual todas las donas llegan bien, uno en el cual no todas llegan bien, y uno en el cual nos rechazaron nuestro pedido.

Diagrama de Clases

La técnica de Diagrama de Clases a llegado a ser básico en la mayoría de los métodos OO. Virtualmente cada método ha incluido algunas variaciones sobre esta técnica.

Además de ser ampliamente usados los diagramas de clases son también los que tienen el más grande rango de conceptos de modelación. Aunque los elementos básicos si necesitados por todos, los conceptos avanzados son usados con una frecuencia menor (a continuación sólo se verán los conceptos básicos).

Un diagrama de clases describe los tipos de objetos en el sistema y los tipos de relaciones estáticas que existen entre ellos. Hay dos principales tipos de relaciones estáticas:

- *Asociaciones* (por ejemplo, un cliente puede rentar muchos videos).
- *Subtipos* (una enfermera es un tipo de persona).

Un diagrama de clases también muestra los atributos y operaciones de una clase y las reglas que aplican a la forma en que los objetos están conectados.

Varios métodos OO usan diferentes (y frecuentemente conflictivos) términos para estos conceptos. Esto es extremadamente frustrante pero inevitable, dado que los lenguajes OO son simplemente como desconsiderados. Es en ésta área que el UML traerá de éstos sus más grandes beneficios para la simplificación de los diferentes diagramas.

Perspectivas

Antes de iniciar con la descripción de los diagramas de clases, veremos un punto importante en la forma en que la gente los usa. Este punto inusualmente es documentado pero tiene un impacto sobre la manera en que se debería interpretar un diagrama, para eso concierne mucho la forma en que se esta describiendo un modelo.

Hay tres diferentes formas en que se puede dibujar un diagrama de clases:

- **Conceptual**: Si se toma una perspectiva conceptual, se dibuja un diagrama que representa los conceptos en el dominio de estudio. Estos conceptos serán naturalmente relatados a las clases que se implementarán, pero de ahí no se utilizan con frecuencia directamente. Ciertamente un modelo conceptual debería ser dibujado con poco o nada para la implementación del software, así puede ser considerado un lenguaje-independiente.

- **Especificación**: Ahora estamos viendo al software, pero estamos viendo las interfaces del software, no la implementación. Nosotros vemos a los tipos más que a las clases. Los desarrolladores OO ponen gran énfasis sobre la diferencia entre interface e implementación, pero es frecuente pasar por alto esto en la práctica porque la noción de clase en un lenguaje OO combina a ambos. Así que frecuentemente se escucha que las interfaces se refieren como tipos y las implementaciones de todas estas interfaces como clases. Un tipo representa interfaces que puede tener muchas implementaciones, diferencias causadas por: llamadas, medios de implementación, características para llevarse a cabo. La distinción puede ser muy importante en un número de técnicas de diseño basadas en la delegación.

- **Implementación**: En esta vista, realmente tenemos clases y estamos construyendo la capa de implementación. Esta es probablemente la perspectiva de mayor uso, pero en muchas la perspectiva de especificacion es la más frecuente de usar.

Entender la perspectiva es crucial para dibujar y leer los diagramas de clases. Desafortunadamente, las líneas entre las perspectivas no estan muy establecidas, y muchos modeladores no se preocupan por tener una perspectiva cuando estan dibujando.

Cuando se dibuja un diagrama, se debería de dibujar en forma simple, con una perspectiva clara. Cuando se lee un diagrama, se debe estar seguro de que se conoce la perspectiva desde la cual ha sido dibujado. Este conocimiento es primordial si se quiere interpretar el diagrama apropiadamente.

La perspectiva no es una parte formal del UML, pero es extremadamente valiosa cuando se modelo y se revisan modelos.

Asociaciones

La fig. 2 muestra un caso simple de modelo de clases. Se describirá cada una de las piezas y se hablará acerca de cómo se deberían interpretar desde varias perspectivas.

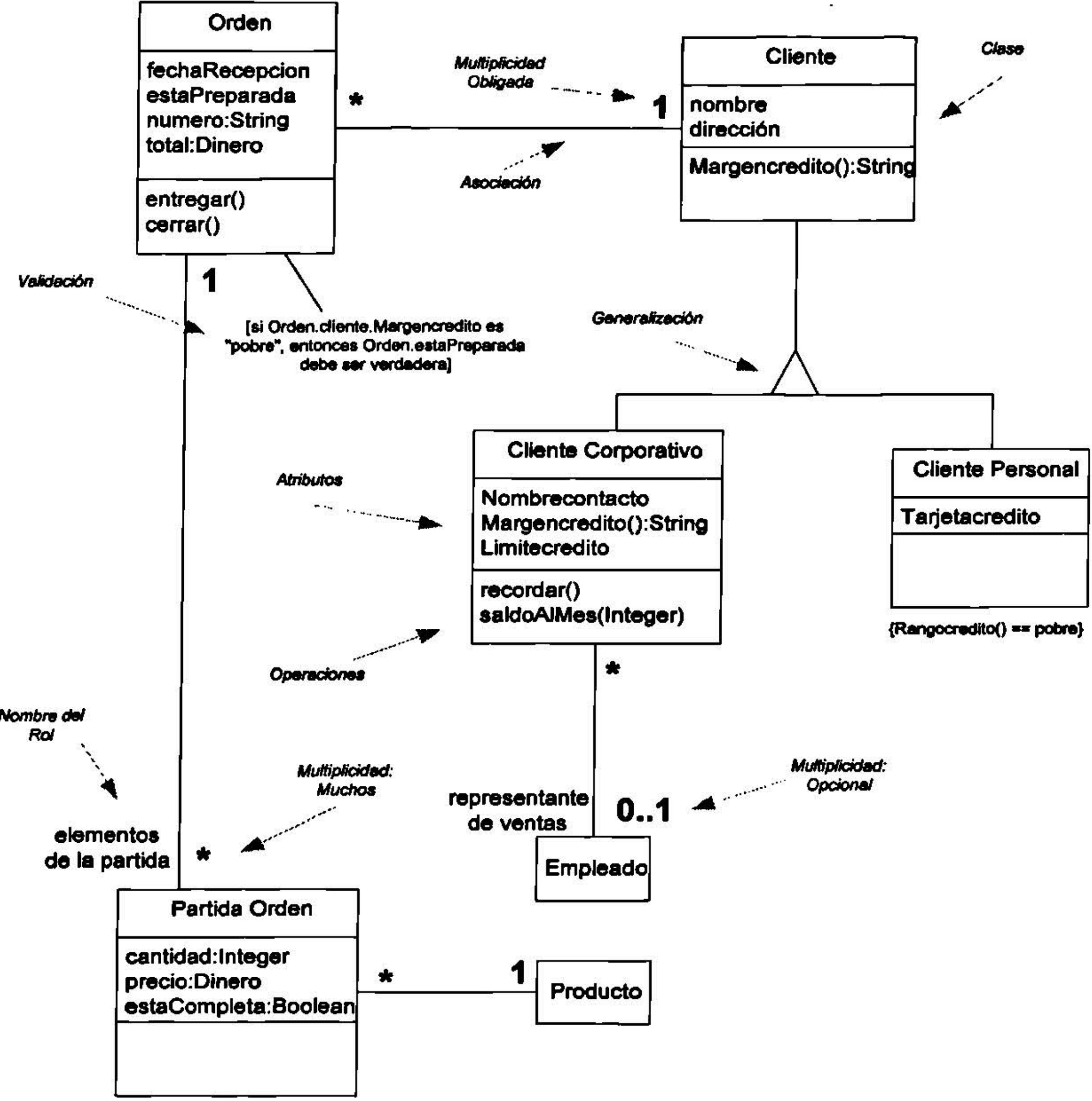


Fig. 2

Iniciaremos con las asociaciones. Las **asociaciones** representan relaciones entre instancias de clases (una persona trabaja para una compañía; una compañía puede tener un número de oficinas, por ejemplo).

Desde la perspectiva conceptual, las asociaciones representan relaciones conceptuales entre las clases. El diagrama indica que una Orden viene de una Cliente y que ese Cliente puede tener muchas Ordenes al mismo tiempo. Cada una de estas Ordenes tiene muchas Partidas de Ordenes, cada una de las cuales referencían a un solo Producto.

Cada asociación tiene dos **roles**; cada rol es una dirección en la asociación. De esta manera, la asociación entre Cliente y Orden contiene dos roles: una de Cliente a Orden; la segunda de Orden a Cliente.

Un rol puede ser explícitamente nombrado con una etiqueta. En este caso, el rol de la dirección Ordenes a Partidas de Ordenes es llamado Elementos Partida. Si no hay etiqueta, se nombra el rol después de la clase a asociar- así el rol desde Ordenes a Clientes debería llamarse Clientes.

Un rol tiene **multiplicidad**, la cual es un indicador de que muchos objetos pueden participar en una relación dada. En la Fig. 2, el * entre Cliente y Orden indica que un Cliente puede tener muchas Ordenes asociadas con él; el **1** indica que una Orden proviene de un Cliente.

En general la multiplicidad indica los limites más bajos y más altos para los objetos participantes. El * actualmente representa el rango *0.. infinito*: un Cliente puede no necesitar generar una Orden, y no hay limite superior para el número de Ordenes que puede generar un Cliente. El **1** puesto en *1..1*: Indica que una Orden debe tener exactamente solo un Cliente.

Las multiplicidades más comunes en la práctica son 1, *, y 0..1. Para multiplicidad más generales, se pueden expresar con un número (tales como 11 jugadores de un equipo de fútbol), un rango (como 2..4 jugadores de canasta), o combinaciones discretas de números y rangos (como 2,4 para puertas de un carro).

Con la perspectiva de especificación, las asociaciones representan responsabilidades.

La Fig. 4-1.implica que hay uno o más métodos asociados con Cliente que dirá que ordenes dado un Cliente le pertenecen. Similarmente, hay métodos con Ordenes que harán conocer el Cliente correspondiente para una Orden dada y que Elementos Partida se comprometieron con la Orden.

Si hay convenciones de estándar para nombrar métodos de filtro, probablemente se puede respetar desde el diagrama a los que esos métodos son llamados. Por ejemplo, se puede tener una convención que nos diga las relaciones de valores-sencillos son implementados con un método que retorna los objetos relatados y relaciones de valores-múltiples son implementados con enumeraciones (consecutivo) dentro de una colección de objetos relatados.

Trabajando con la convención de nombres como en JAVA, por ejemplo la siguiente sería una interface de una clase Orden.

```
class Orden {
  public Cliente cliente();

  // Enumeración de Partidas de ordenes
  public Enumeration ordenPartidas();
```

Si esto fuera un modelo de **implementación**, deberíamos ahora dejar implícito que hay apuntadores en ambas direcciones de las clases relacionadas. El diagrama debería decir que la Orden tiene un campo que es una colección de apuntadores a Partida Ordenes y también tienen un apuntador a Cliente. En Java, podría parecerse a algo así:

```
class Orden {

        private Customer _customer;
        private Vector _orderLines;
}
class Customer {
  private Vector _orders;
}
```

En este caso, no podemos interferir en nada desde las asociaciones acerca de la interface. Las operaciones sobre una clase deberían darnos esta información.

Ahora observemos a la figura 3. Esta es básicamente la misma que la figura 2 excepto que se agrego un par de líneas sobre la punta de las líneas de asociación. Estas flechas indican la **navegabilidad**.

En el modelo de **especificación**, esto indicaría que una Orden tiene la responsabilidad de decir para cual Cliente es, pero el Cliente no tiene la habilidad de decir que Orden tiene. En lugar de responsabilidades simétricas, ahora tenemos responsabilidades de un solo lado. En un diagrama de implementación, uno debería indicar que la Orden contiene un apuntador a Cliente pero Cliente no debería tener un apuntador a Orden.

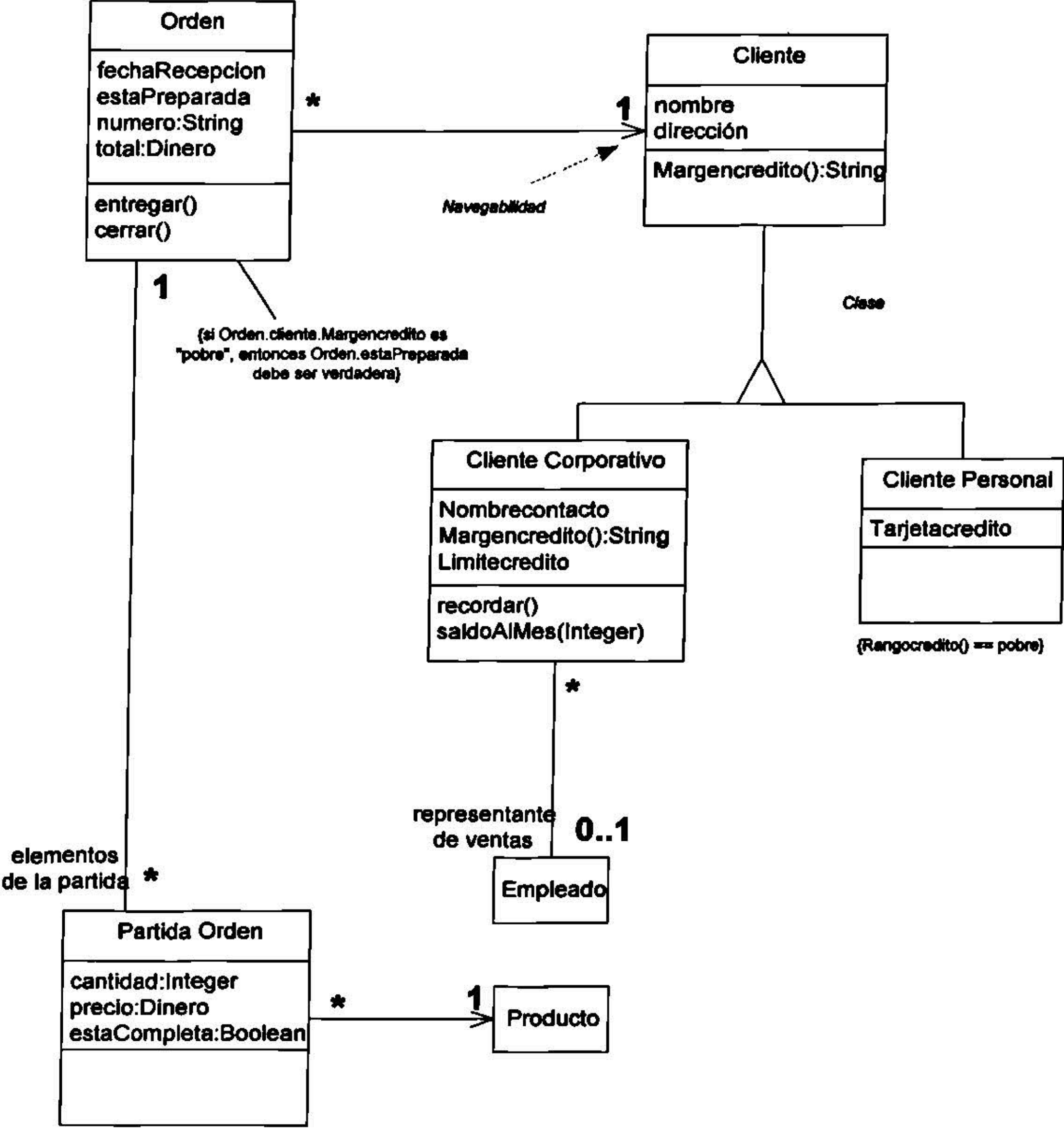


Fig. 3

Como se puede ver, la navegabilidad es una parte importante en los diagramas de implementación y de especificación.

Atributos

Los **Atributos** son similares a las asociaciones.

En el nivel conceptual, el atributo nombre de Cliente indica que un Cliente tiene nombres. Al nivel de especificación, este atributo indica que el objeto Cliente puede decirnos su nombre y de alguna manera ponerle un nombre. Al nivel de implementación, un Cliente tiene un **campo** (también llamado una variable instanciada o un dato miembro) para este nombre.

Dependiendo del detalle sobre los diagramas, la notación para un atributo puede mostrar el nombre, tipo, y el valor por default de un atributo. La sintaxis en UML es:

Visibilidad nombre: tipo = valor por default.

Donde:
visibilidad es lo mismo para las operaciones, que se describen en la siguiente sección.

¿Así que cual es la diferencia entre un atributo y una asociación?
Desde el nivel conceptual, no hay diferencia - un atributo solo lleva a otro tipo de notación que se puede usar si no se ven inconvenientes -. Los atributos son simpre un valor simple. Usualmente, un diagrama no indica si un atributo es opcional u obligatorio (aunque estrictamente hablando, debería ser).

La diferencia ocurre en los niveles de especificación e implementación. Los atributos implican navegabilidad desde un tipo a solo un atributo. Por lo tanto, esto implica que el tipo contiene solamente una copia del atributo del objeto, implicando que cualquier tipo usado como un atributo tiene un valor, más que referencia semántica.

Operaciones

Las **Operaciones** son los procesos que las clases realizan. Estas corresponden más obviamente a los métodos de una clase. Al nivel de especificación, las operaciones corresponden a los métodos publicos de un tipo. Normalmente, no se muestran estas operaciones que simplemente manipulan atributos, porque usualmente pueden ser no referenciados. Sin embargo, se puede indicar si un atributo dado es de sólo lectura o **inmutable**. En el modelo de implementación, lo deseable es mostrar las operaciones privadas y publicas muy bien.

La sintaxis en UML para las operaciones es:

Visibilidad nombre (lista de parámetros): expresion-tipo-a-retornar {propiedad-string}

Donde:
Visibilidad es + (publica), # (protegida), o - (privada).
Nombre es string.
lista de parámetros contiene argumentos (opcionales), donde la sintaxis es la misma que los atributos.
Expresion-tipo-a-retornar es opcional, dependiendo de la especificación del lenguaje.
Propiedad-string indica el valor de la propiedad que aplica a la operación dada.

Con los modelos conceptuales, las operaciones no deberían intentar de especificar la interface de una clase. En lugar de eso, deberían indicar las principales responsabilidades de esa clase.

Generalización

Un típico ejemplo de generalización envuelve al personal y los clientes de un negocio. Hay diferencias pero también hay muchas similitudes. Las similitudes pueden ser situadas en una clase general de Clientes (el supertipo) con Cliente Personal y Cliente Corporativo como subtipos.

Este fenómeno esta también sujeto a diferentes interpretaciones en los diferentes niveles de modelación. Conceptualmente, por ejemplo, podemos decir que un Cliente Corporativo es un subtipo de Cliente si todas las instancias de Cliente Corporativo son también, por definición, instancias de Cliente. Un Cliente Corporativo es entonces un tipo especial de Cliente. Este idea clave es todo lo que se necesita saber acerca de Cliente, la cual es verdadera para Cliente Corporativo.

Con el modelo de especificación, la generalización significa que los métodos de una interface de un subtipo deben incluir todos los elementos desde la interface del supertipo. La interface del subtipo esta hecha **conforme a** la interface del supertipo.

La generalización en la perspectiva de la implementación es asociada con la herencia en lenguajes de programación. Las subclases heredan todos los métodos y atributos de las superclases y puede ser sobrecargados con los métodos heredados.

Diagramas de Actividad

Los **diagramas de actividad** son una de las partes más inesperadas del UML.

En la figura 4, la cual viene en la documentación 1.0 de UML, el símbolo del centro es la **actividad**. La interpretación de este término depende de la perspectiva sobre la cual se esta dibujando el diagrama. En un diagrama conceptual, una actividad es alguna tarea que necesita ser hecha, ya sea por un humano o por una máquina. En un diagrama con perspectiva de especificación, una actividad es un método sobre una clase.

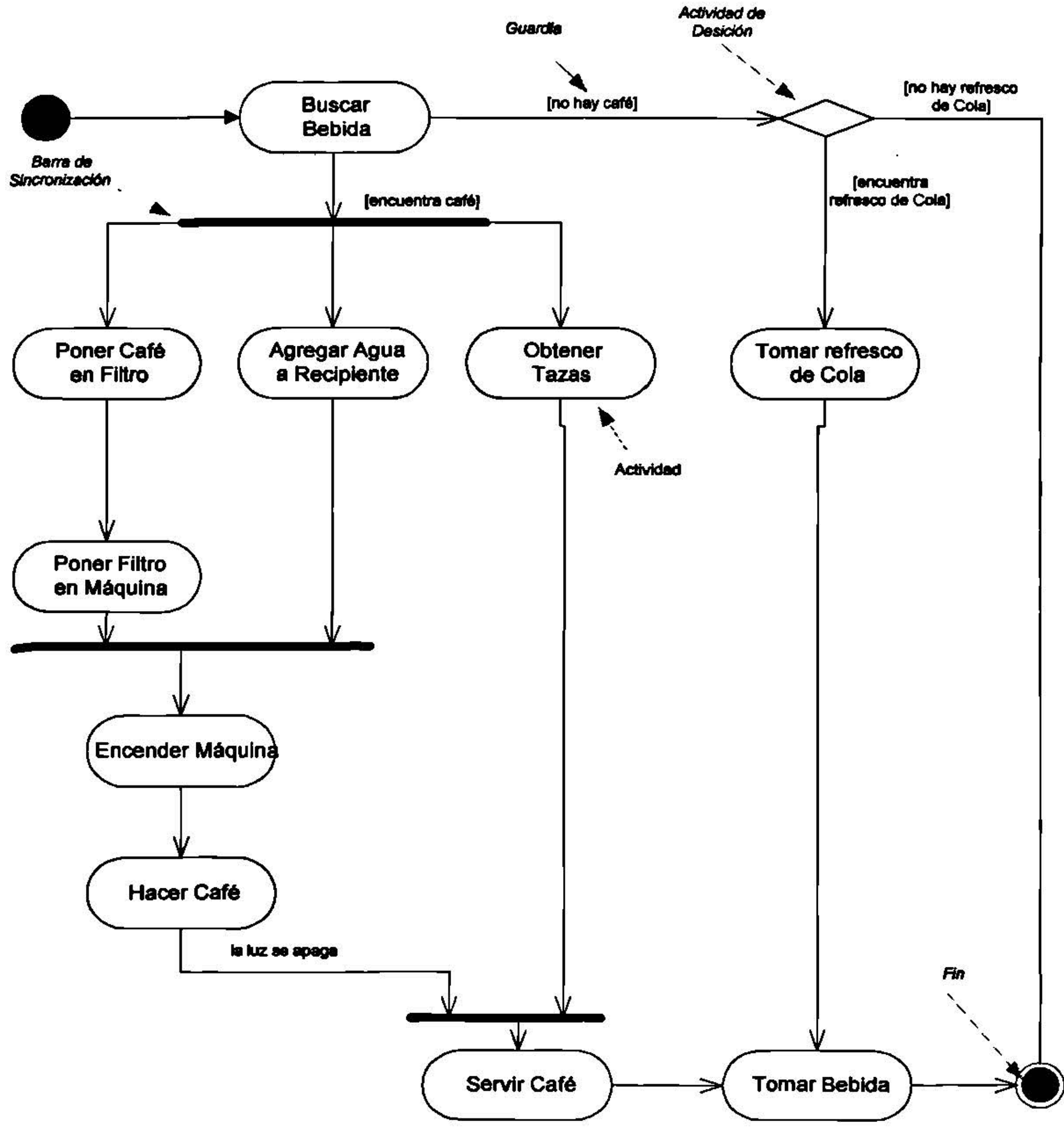


Fig. 4

Cada actividad puede ser seguida por otra actividad. Esto es simple secuencia. Por ejemplo, en la figura 4, la actividad Poner Café en Filtro esta seguida de la actividad Poner Filtro en Máquina. Un poco más allá, el diagrama de actividad se parece a un diagrama de flujo. Exploraremos las diferencias viendo la actividad de Encontrar Bebida.

Encontrar Bebida tiene dos disparadores que salen de él. Cada disparador tiene un **guardia**, una expresión lógica que se evalúa en "verdadero" o "falso", como en un diagrama de estados. En este caso, la persona seguirá actividad de Encontrar Bebida pensando en un café o en un refresco de cola.

Asumiremos que encontró una bolsa de café e iremos por esa ruta. Este disparador encabeza una **barra de sincronización**, adjunta a ella salen disparadores. Estos disparadores irán a las actividades de Poner Café en Filtro, Agregar Agua a Recipiente y Obtener Tazas, respectivamente.

El diagrama dice que las tres actividades pueden ocurrir paralelamente. Esencialmente, esto significa que su orden es irrelevante. Y también se pueden realizar estas actividades intercalándolas.

El diagrama de actividad permite escoger en que orden hacer las cosas. En otras palabras, es establecer las reglas de secuencia que se deben de seguir. Esta es la diferencia clave entre un diagrama de actividad y un diagrama de flujo. Los diagramas de flujo son normalmente limitados por procesos secuenciales; los diagramas de actividad pueden manejar procesos paralelos.

Esto es importante para modelación de negocios. Los negocios frecuentemente no ofrecen procesos secuenciales. Una técnica como esta que aliente a un comportamiento paralelo es valiosa para estas situaciones porque motiva a la gente a moverse de innecesarios procesos secuenciales en su comportamiento y tomar oportunidades para hacer cosas en paralelo. Esto puede ser improvisado en las responsabilidades y eficiencia de procesos de negocios.

Los diagramas de actividad también son muy usados para programas concurrentes desde que pueden gráficamente mostrar que threads tienes y cuando se necesitan sincronizar. Una sincronización es importante en un comportamiento paralelo, ya que el que dos procesos se realicen al mismo tiempo puede producir resultados inesperados.

Diagramas de Desarrollo

Un Diagrama de Desarrollo muestra las relaciones físicas entre componentes de software y de hardware en las partes de un sistema. Un Diagrama de Desarrollo es un buen lugar para mostrar como los componentes y objetos esta ruteados y se mueven a través de un sistema distribuido.

Cada nodo en un Diagrama de Desarrollo representa algún tipo de unidad computacional - en muchos casos, una pieza de hardware. El hardware puede ser un dispositivo sencillo o un sensor o pudiera ser una mainframe.

La siguiente figura muestra a una PC conectada a un servidor de UNIX a través de TCP/IP. Las conexiones entre nodos muestran las rutas de comunicación sobre las cuales el sistema interactuará.

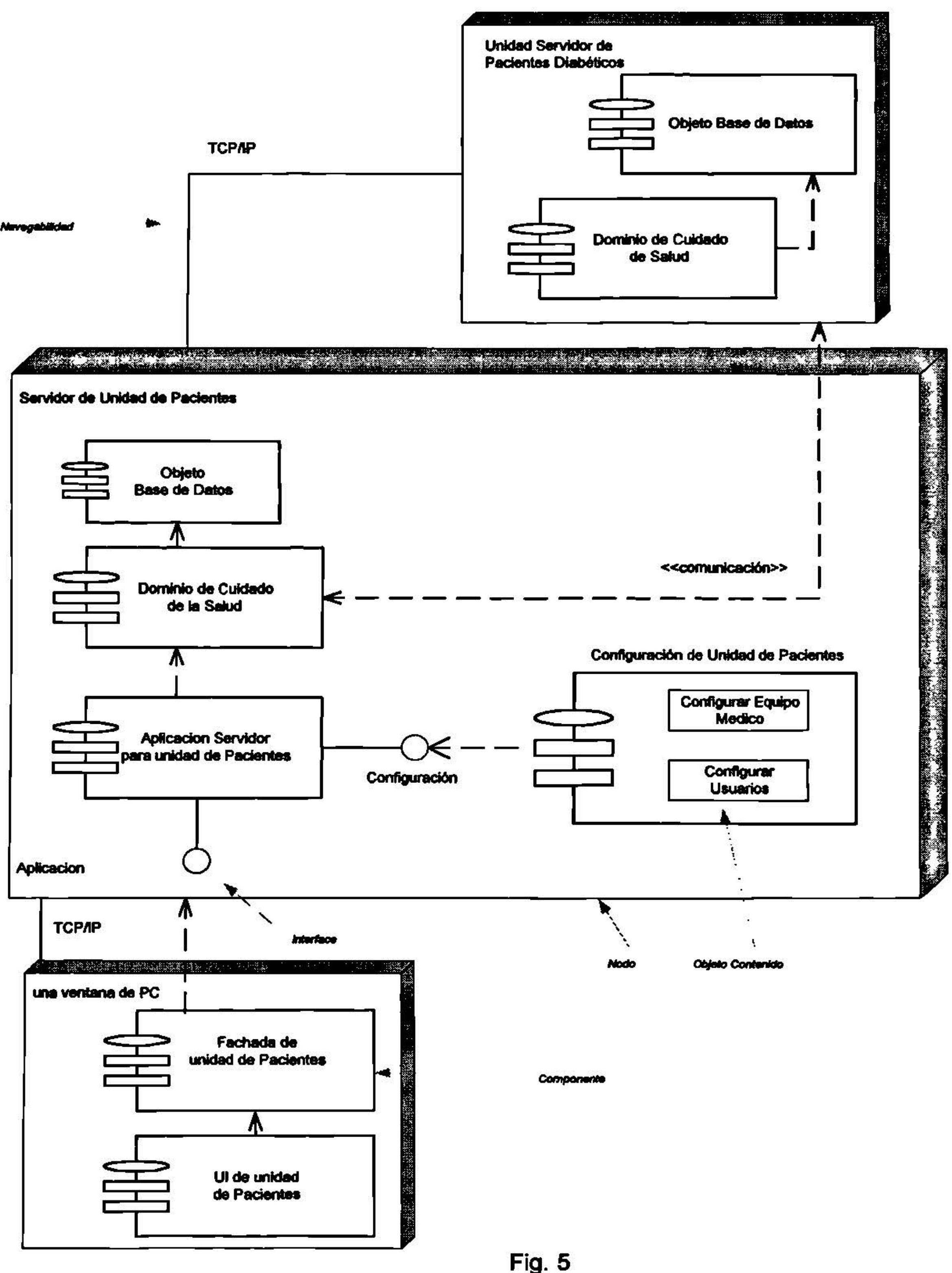


Fig. 5

Los componentes en un Diagrama de Desarrollo representan módulos físicos de código. En la práctica éstos debieran de corresponder exactamente a un Diagrama de Paquetes, así que el Diagrama de Desarrollo muestra donde cada paquete esta corriendo en el sistema.

Las dependencias entre los componentes deberían de ser las mismas que las dependencias entre paquetes. Estas dependencias muestran como los componentes se comunican con otros componentes. La dirección dada la dependencia indica la información en la comunicación.

Un componente puede tener más de una interface, que en ese caso se mostraría que componentes se comunican con que intefaces.

Diagramas de Estado

Los **Diagramas de Estado** son una técnica familiar para describir el comportamiento de un sistema. Describen todos los posibles estados que un determinado objeto puede tener dentro del sistema y como los estados del objeto cambian como resultado de los eventos que puede alcanzar el objeto. En muchas técnicas de OO, los diagramas de estado son dibujados para una clase que muestra el comportamiento de un objeto.

La siguiente figura muestra el Diagrama de Estados de UML para una Orden en un sistema de procesamiento de Ordenes. El diagrama indica los varios estados de una Orden.

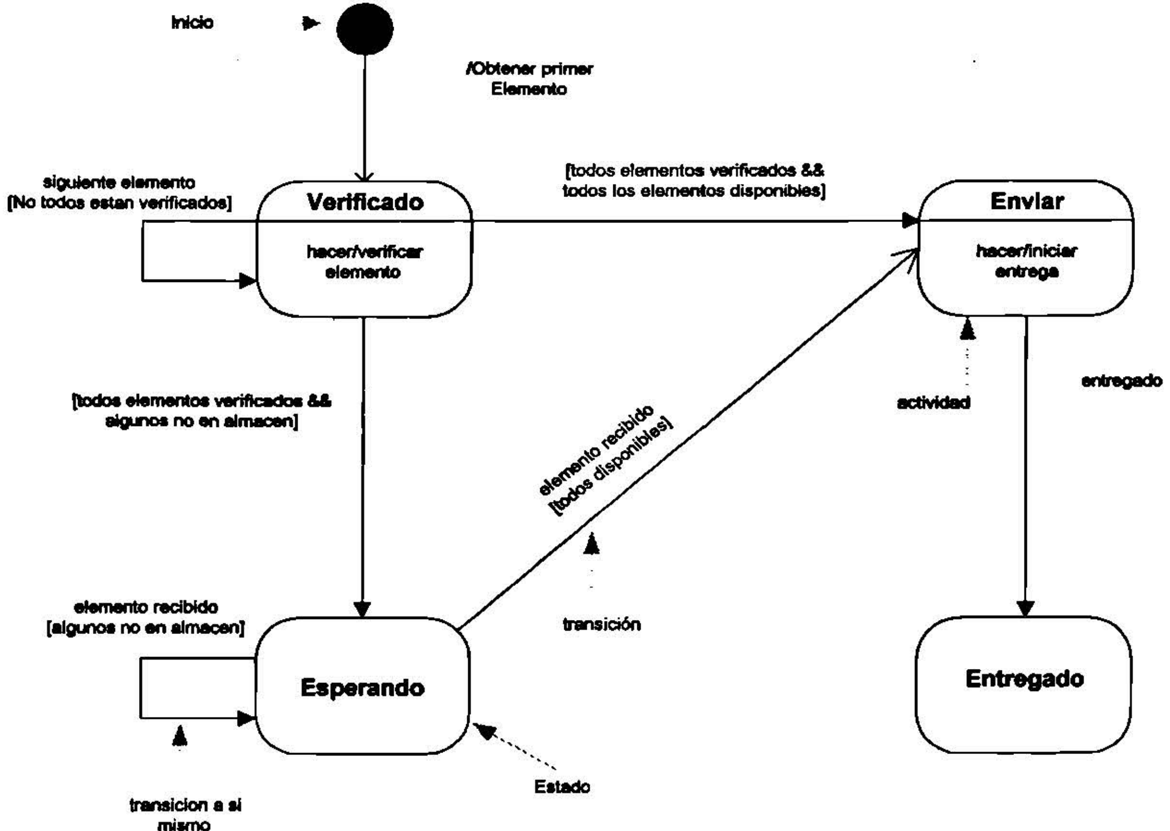


Fig. 6

Comenzaremos con el punto de inicio y muestra una transición inicial dentro del estado de Verificado. Esta transición esta etiquetada "/obtener primer elemento". La sintaxis para la etiqueta de transición tiene tres partes, las cuales son opcionales: *Evento[guardia] / Acción.* En este caso, tenemos sólo la acción "obtener el primer elemento". Una vez efectuada esta acción, entramos al estado de Verificado. Este estado tiene una actividad asociada, indicada por una etiqueta con la sintaxis *hacer/actividad.* En este caso, la actividad es llamada "verificar elemento".

Hay que recalcar que el término "acción" es usado para la transición y "actividad" para el estado. Aunque ambos son procesos, típicamente implementados por algún método sobre Orden, son tratados diferentes. **Acciones** son asociadas con transiciones y son consideradas a ser procesos que ocurren rápidamente y no son interrumpidos. Actividades son asociadas con estados y que pueden tomar más tiempo. Una actividad permite ser interrumpida por algún evento.

Notar que la definición de "rápidamente" depende del tipo de sistema que se esta produciendo. Con un sistema complicado de tiempo real, "rápidamente" debería significar pocas instrucciones en la máquina; para un sistema con información regular, "rápidamente" podía significar tan sólo unos segundos.

Cuando una transición no tiene evento en su etiqueta, significa que la transición ocurre tan pronto como cualquier actividad asociada con el estado dado, esta completa. En este caso, significa que tan pronto como hagamos el verificado. Tres condiciones salen del estado Verificado. Los tres tienen guardias en sus etiquetas. Un guardia es una condición lógica que retornará "verdadero" ó "falso". Una transición guardada ocurre sólo si el guardia se resuelve a "verdadero".
Sólo una transición puede ser tomada de un estado dado, así que proponemos a los guardias a ser mutuamente exclusivos para cualquier evento. En la figura 6 tenemos 3 condiciones.

1. Si no se han verificado todos los elementos, obtenemos el siguiente elemento y regresamos al estado de Verificado para verificarlo.
2. Si verificamos a todos los elementos y los hay en existencia, hacemos la transición al estado de Enviar.
3. Si verificamos todos los elementos pero no todos están en existencia, cambiamos hacia el estado de Espera.

Lo último para direccionar es un estado llamado "Cancelado". Nosotros queremos cancelar una orden en cualquier punto antes de ser enviada. Podríamos hacerlo dibujando transiciones separadas para estado de Verificado, Espera y Envío. Una alternativa usual es crear un **super estado** de los tres estados y entonces dibujar sólo una transición desde ese. Los subestados simplemente heredan cualquier transición de un super estado.

Las figuras 7 y 8 muestran como estos acercamientos reflejan el mismo comportamiento. Aún con sólo tres transiciones duplicadas, la figura 7 luce desordenada. La figura 8 hace todo el dibujo más claro, y si se necesitan cambios más tarde, será difícil olvidarse del evento cancelado.

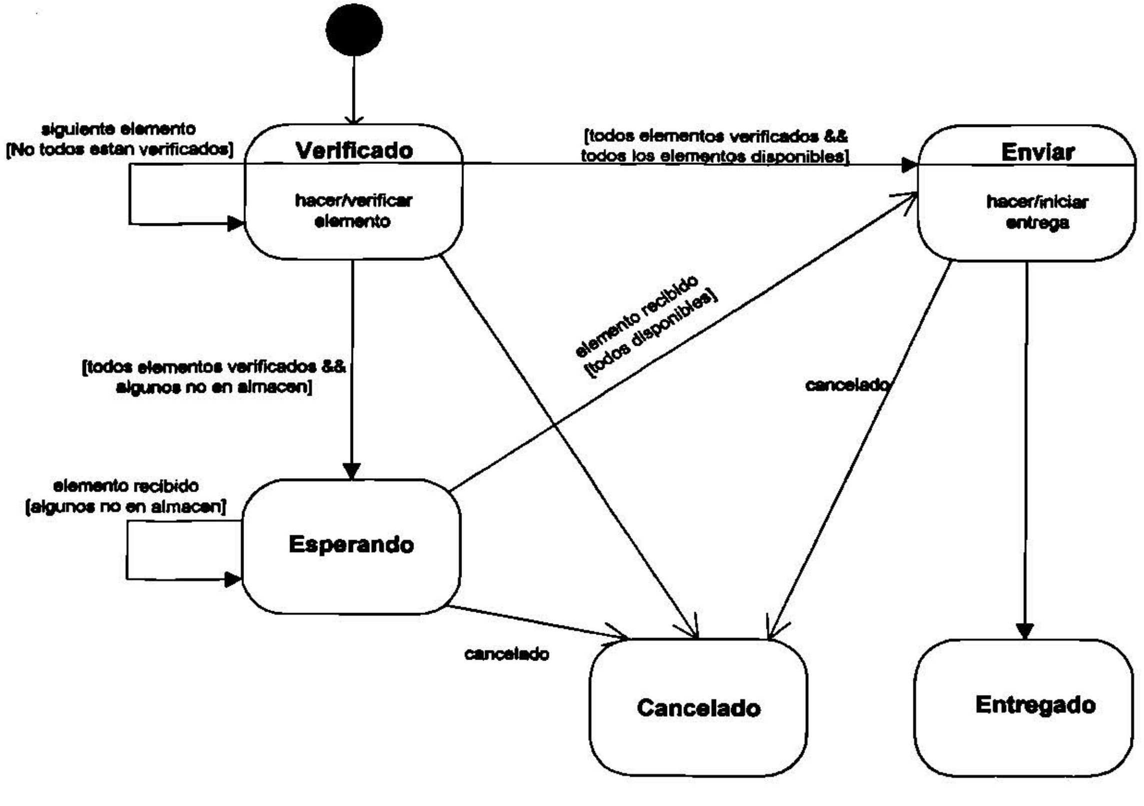


Fig. 7

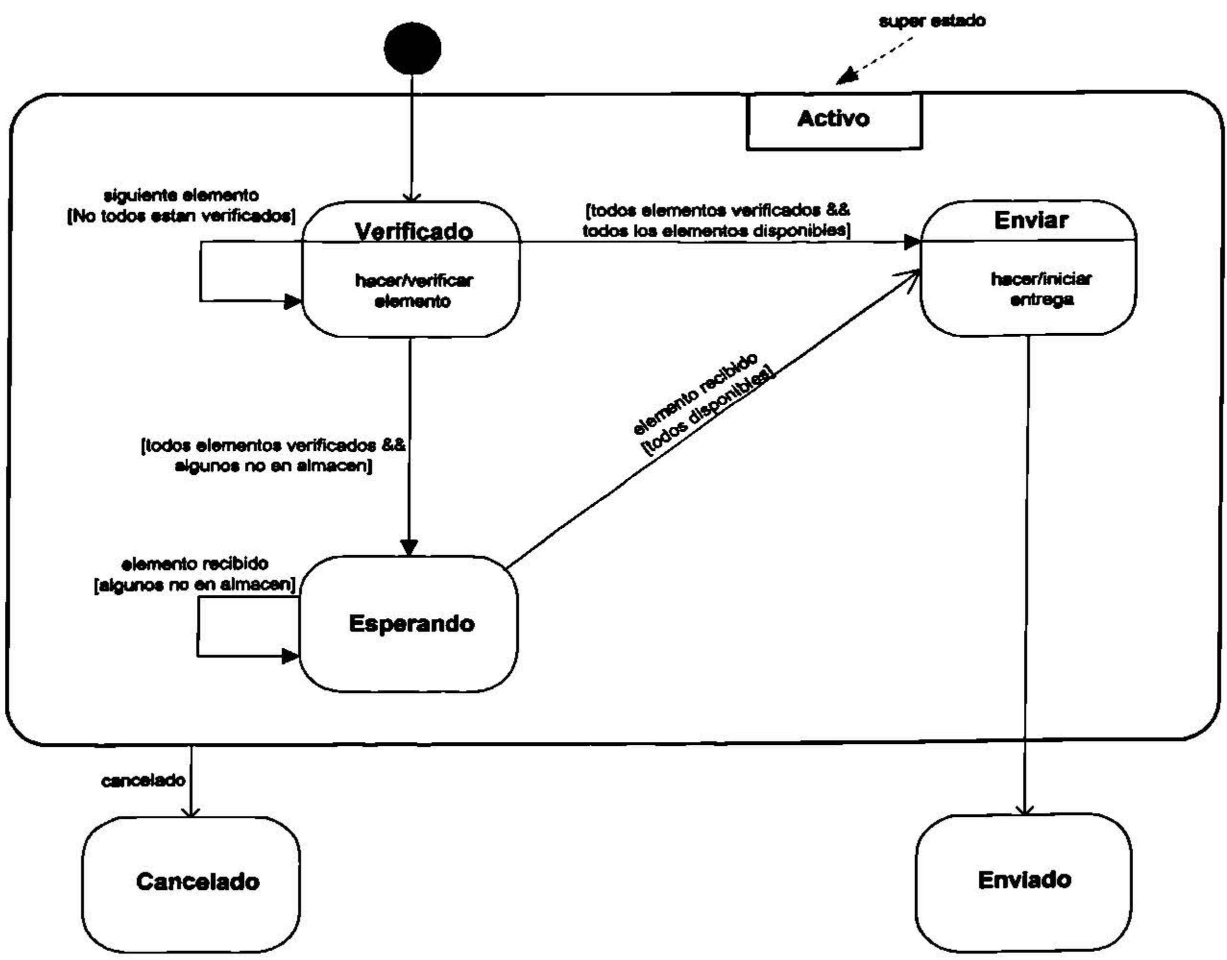


Fig. 8

En el ejemplo actual, se muestra una actividad con un estado, indicado por el texto en la forma *hacer/actividad*. También se pueden indicar otras cosas en un estado.

Si el estado responde a un evento con una acción esto no causa una transición, se puede mostrar esto poniendo un texto en la forma *NombreEvento/NombreAccion* en la caja de Estado.

Hay dos tipos especiales de eventos, entrada y salida. Cualquier acción esta marcada y encadenada a un evento de **entrada** y ejecutada si un estado dado es accionado por vía de transición. El evento de **salida** es ejecutado si el estado dado es dejado por la vía de transición. Si se tiene una transición que se regresa al mismo estado (esto es llamado **transición a sí mismo**) con una acción, la acción de salida debería ser ejecutada primero, entonces la acción de transición y finalmente la acción de entrada. Si el estado esta asociado a una actividad, esa actividad es ejecutada después de la acción de entrada.

Diagramas de estado concurrentes

Además de los estados de una orden basados sobre la disponibilidad de los elementos, hay también estados que son basados en la autorización de pagos. Si miramos estos estados, observaremos un estado como el siguiente.

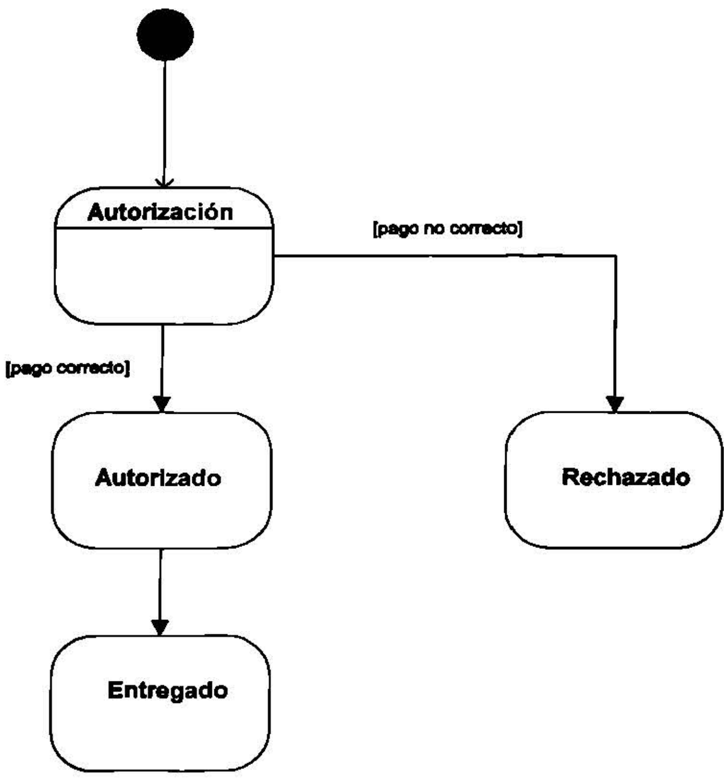


Fig. 9

Aquí empezaremos haciendo una autorización. La actividad de "verificado de pago" termina con la señalización de que el pago es aprobado. Si el pago esta correcto, la orden dada espera en el estado de Autorizado hasta el evento "envío" ocurre. De otra manera, la orden entra al estado Rechazado.

El objeto orden exhibe una combinación de comportamientos mostrados en las figuras 7 y 8. Los estados asociados y el estado Cancelado pueden ser combinados con el **diagrama de estado concurrente** (figura 10).

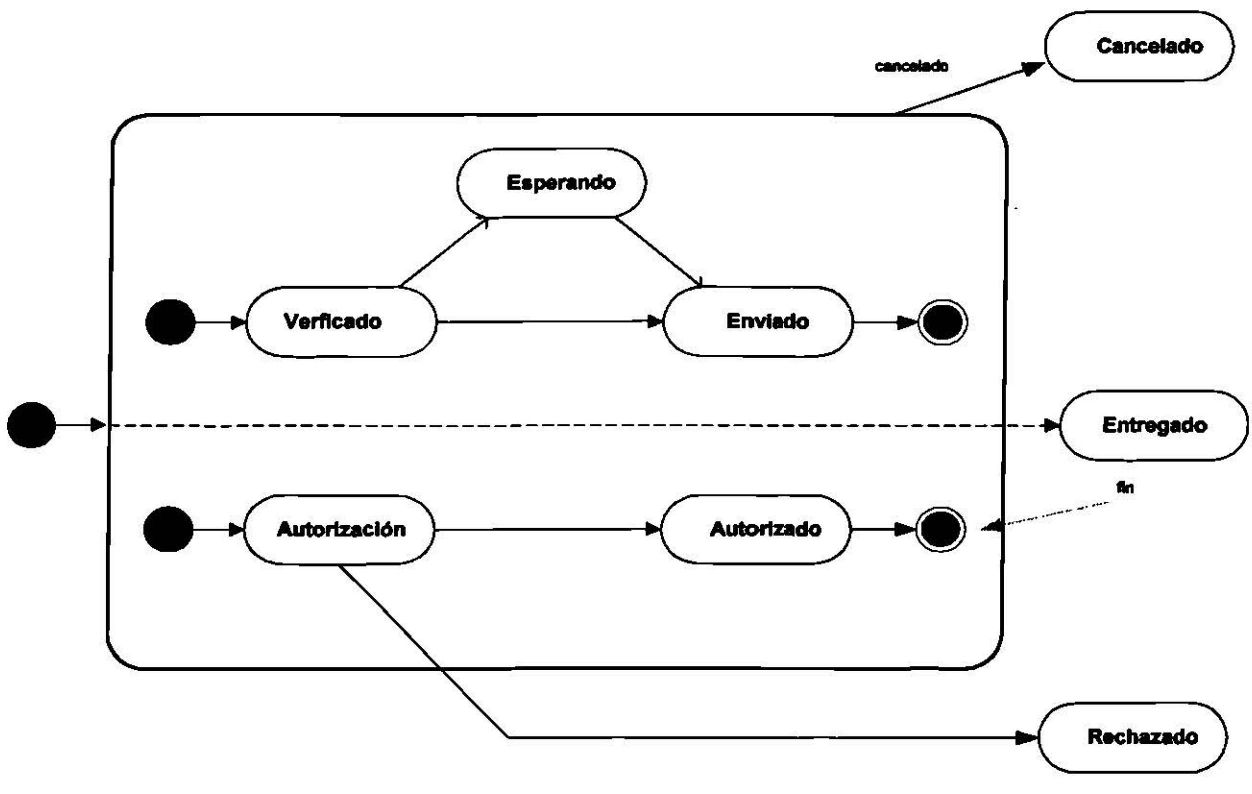


Fig. 10

** Notar que se eliminaron los detalles internos.*

Las secciones concurrentes de diagrama de estados son puestos en cualquier punto, la orden dada esta en dos estados diferentes, uno de cada diagrama. Cuando la orden deja los estados concurrentes, sólo hay un estado. Vemos que la orden inicia en los estados de Verificado y Autorizado. Si la actividad de "verificado de pago" del estado de Autorización es completado primero, la orden estará en el estado de Verificado. Si el evento "cancelar" ocurre, entonces la orden estará en el estado Cancelado.

Los diagramas de estado concurrentes son usados cuando un objeto dado tiene varios comportamientos independientes. Notar, sin embargo, que no se deberían tener muchos comportamientos independientes ocurriendo con un objeto. Si se tienen muchas complicaciones con los diagramas de estado concurrentes para un objeto, se debería considerar dividir al objeto
en objetos separados.

Diagramas de Interacción

Los **Diagramas de Interacción** son modelos que describen como grupos de objetos colaboran en algún comportamiento.

Típicamente, una diagrama de interacción captura el comportamiento de un caso de uso. El diagrama muestra un número de objetos ejemplo y los mensajes que son pasados entre éstos objetos con el caso de uso.

Nos basaremos en el siguiente ejemplo:

1. La ventana de Orden de Entrada envía un mensaje de "preparar" a una orden.
2. La orden envía un mensaje de "preparar" a cada Línea de la Orden sobre la Orden.
3. Cada Línea de la Orden verifica el Elemento en Existencia.
4. Si esta verificación regresa "verdadero", la Línea de la Orden remueve la apropiada cantidad de Elementos de Existencia sobre Existencia.
5. En otro caso, la cantidad de Elementos en Existencia ha fallado en el reorden de nivel de existencia, y el Elemento de Existencia requiere una nueva orden.

Hay dos tipos de diagramas de interacción: Diagramas de Sequencia y Diagramas de Colaboración.

Diagramas de Secuencia

Con un diagrama de secuencia, un objeto es mostrado como una caja al tope de una línea vertical punteada.

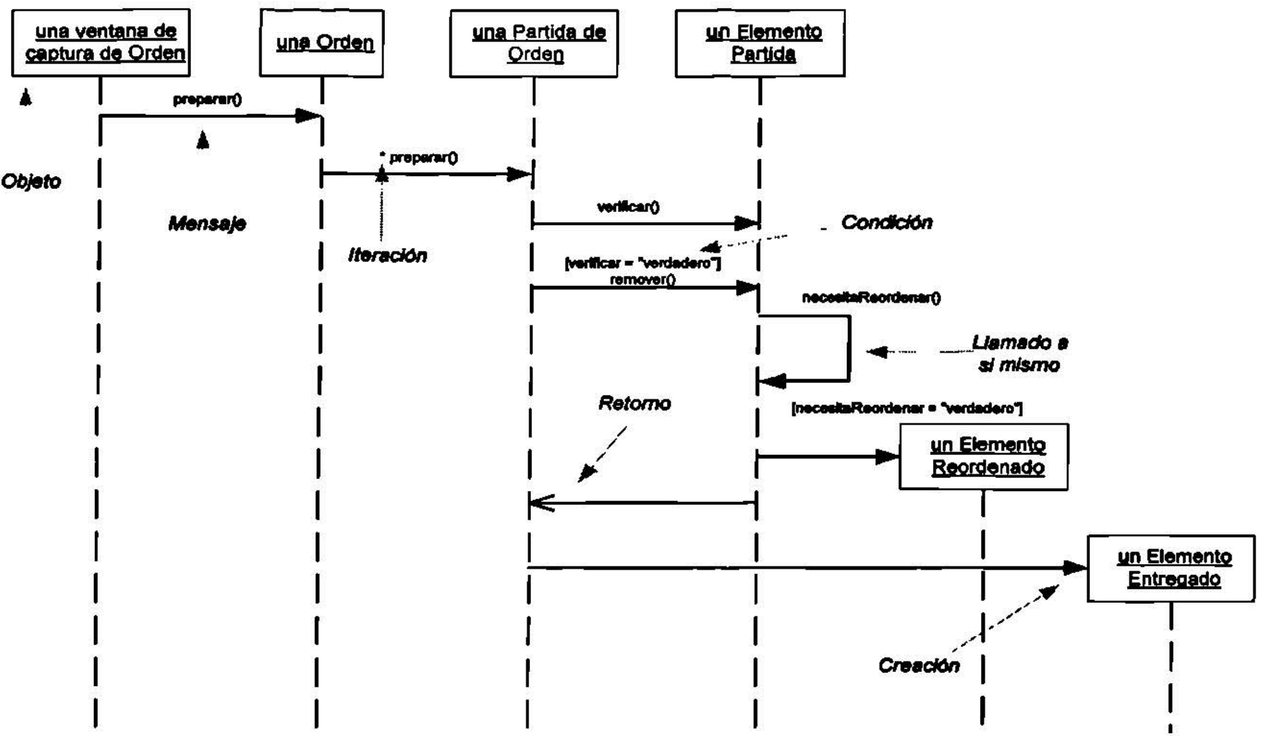


Fig. 11

Esta línea vertical es llamada **línea de vida del objeto**. La línea de vida representa la vida del objeto durante la interacción.

Cada mensaje es representado por una flecha ente las líneas de vida de dos objetos. El orden en el cual estos mensajes ocurren es mostrado de arriba abajo sobre la página. Cada mensaje es etiquetado al mínimo con el nombre del mensaje; se puede incluir los argumentos y alguna información de control, y se puede mostrar una **delegación a sí mismo**, un mensaje que se envía a sí mismo, por el envío de una flecha de regreso a la misma línea de vida.

Dos cosas del control de información son valiosas. Primero, hay una condición, la cual indica cuando un mensaje es enviado (por ejemplo, *[NecesarioReordenar() == "verdadero"]*). Este mensaje es enviado sólo si la condición es verdadera. El segundo marcador de control usado es el **marcador de iteración**, el cual muestra cuantas veces los mensajes son enviados a los objetos receptores, como podría pasar cuando se esta haciendo una iteración sobre una colección (tal como **preparar*).

Una de las cosas más difíciles de entender en un programa orientado a objetos es el flujo del control de todo el sistema. Un buen diseño tiene muchos métodos pequeños en diferentes clases, y a veces puede ser engañoso mostrar toda la secuencia de un comportamiento. Se puede llegar al fin del código para tratar de entender el programa. Esto es particularmente cierto para todos los objetos nuevos. Los diagramas de secuencia ayudan a entender esa secuencia.

Este diagrama incluye un **retorno**, el cual indica el retorno del mensaje, no un nuevo mensaje. Retorno difiere de los mensajes regulares por la cabeza de la flecha, para retorno son sólo un par de líneas, no esta rellena.

Proceso Concurrentes y Activaciones

Los diagramas de secuencia también son valiosos para procesos concurrentes.

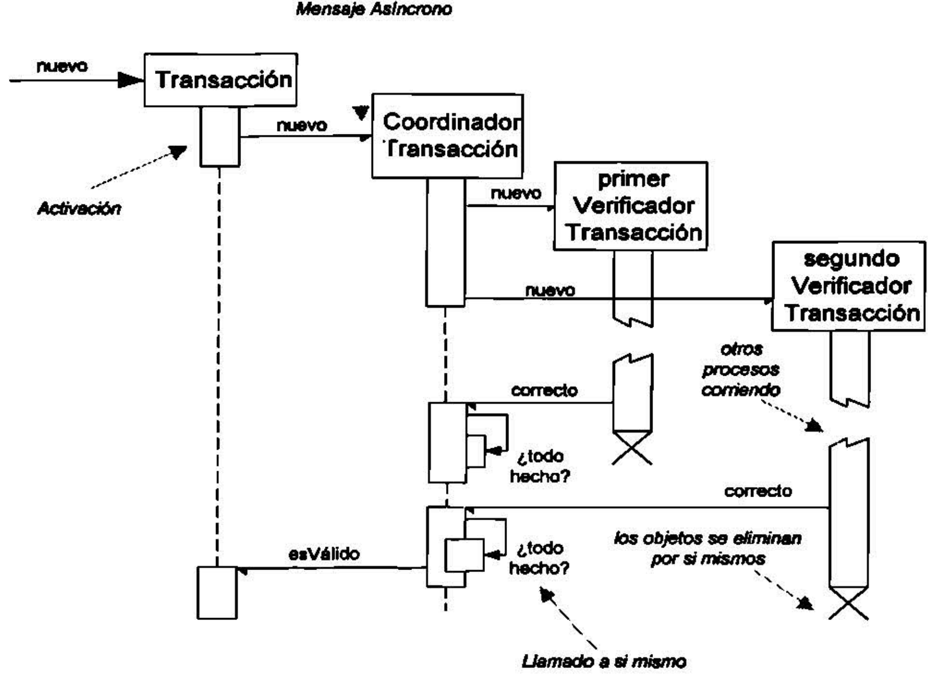


Fig. 12

Cuando una Transacción es creada, se crea un Coordinador de Transacciones para coordinar la verificación de la Transacción. El coordinador crea un número (en este caso dos) de objetos Verificadores de Transacción, cada uno es responsable de una verificación particular. Este proceso hace fácil agregar diferentes procesos de verificado porque cada verificador es llamado asíncronamente y procesa en paralelo.

Cuando un Verificador de Transacción completa, notifica al Coordinador de Transacciones. El coordinador observa si todos los verificadores han terminado. Si no, el coordinador no hace nada. Si terminaron, y todos ellos no tuvieron errores, como en este caso, entonces el notificador notifica a la transacción que todo salió bien.

La figura 13 introduce un número de nuevos elementos para los diagramas de secuencia. Primero, se ven las **activaciones**, las cuales aparecen explícitamente cuando un método esta activo porque también se esta ejecutando o esperando por una subrutina para retornar. Muchos diseñadores usan activaciones todo el tiempo.

Las flechas con media cabeza indican un mensaje asíncrono. Un mensaje asíncrono no bloquea al que lo llamo, así que puede llevárselo consigo en su procesamiento. Un mensaje asíncrono puede hacer tres cosas.

1. Crear un nuevo "thread", en determinado caso se encadena al inicio de una activación.
2. Crear un nuevo objeto.
3. Comunicarse con un thread que esta corriendo.

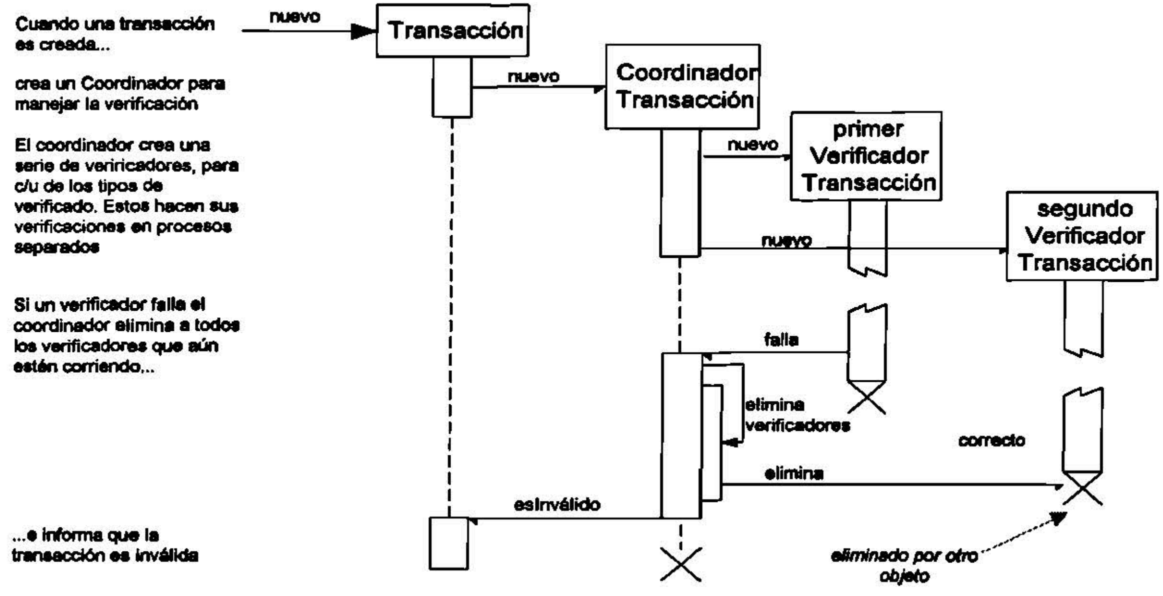


Fig. 13

Diagramas de Colaboración

La segunda forma de diagramas de interacción es el **diagrama de colaboración**.

Con un diagrama de colaboración, el ejemplo de los objetos es mostrado como iconos. Como en el diagrama de secuencia, la flecha indica los mensajes enviados con el caso de uso dado. Esta vez, sin embargo, la secuencia es indicada por mensajes enumerados.

La numeración de mensajes hace más difícil de ver la secuencia que poniendo las líneas hacia debajo de la hoja. Por otro lado, el especial esquema permite mostrar otras cosas más fácilmente. Se puede ver como los objetos están relacionados y usar el esquema para recubrir paquetes u otra información.

Se puede usar alguno de muchos esquemas de numeración para los diagramas de colaboración. El más simple es ilustrado en la siguiente figura 14. Otro tipo de esquema es mostrado en la figura 15.

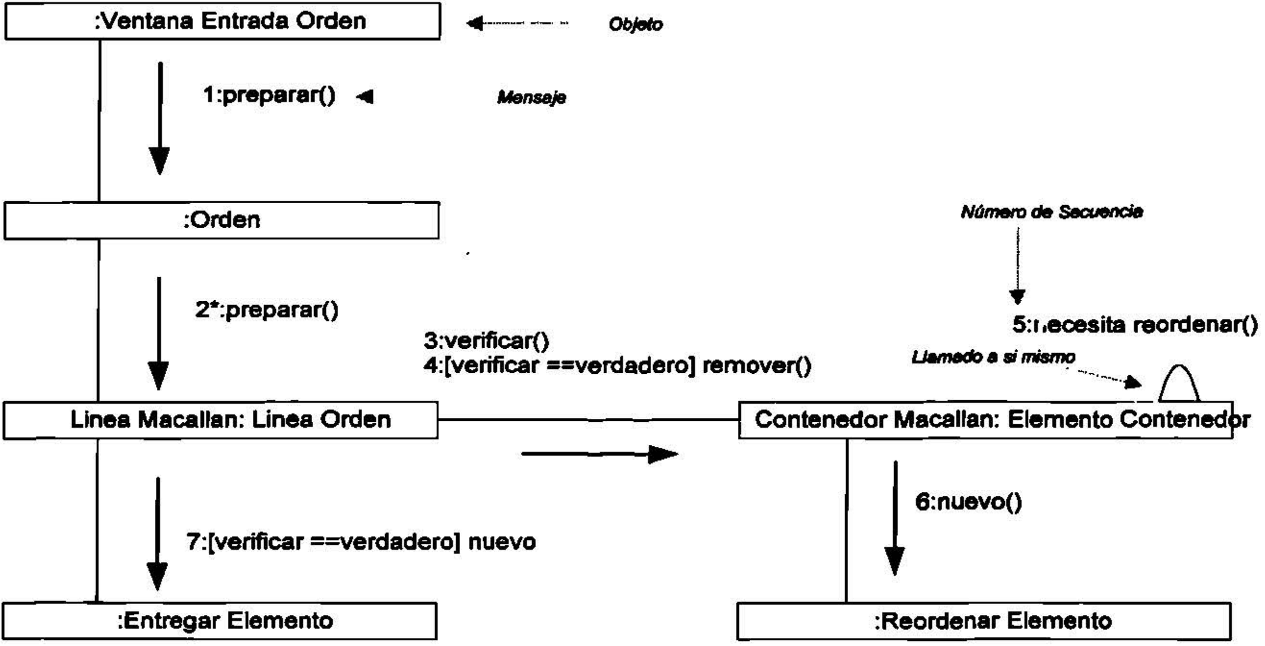


Fig. 14

En el pasado, mucha gente usaba el simple esquema de numeración. El UML usa el esquema decimal porque hace más claro cual operación es llamada por cual operación, aunque puede ser más difícil de ver toda la secuencia.

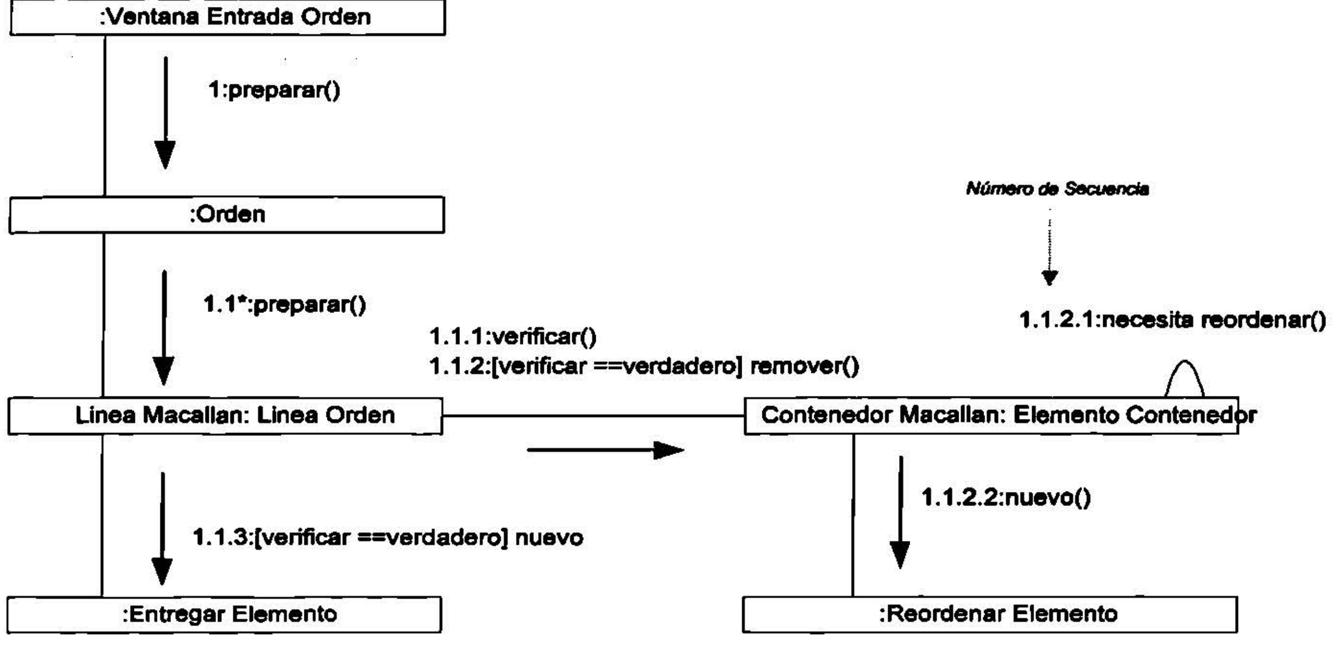


Fig. 15

A pesar de que esquema de numeración veamos, se puede agregar el tipo de control de información que se mostró en un diagrama de secuencia.

Comparación de Diagramas de Secuencia y de Colaboración

Diferentes desarrolladores tiene diferentes preferencias cuando comienzan a escoger la forma del diagrama de interacción que van a usar. Usualmente se podría escoger el diagrama de secuencia porque el énfasis se pone en la secuencia; esto es más fácil de ver para ordenar que cosas ocurren. Otros prefieren los diagramas de colaboración porque usan el esquema para indicar como los objetos están relacionado estáticamente.

Una de las principales características de las formas de un diagrama de interacción es su simplicidad. Se pueden ver fácilmente los mensajes con sólo ver el diagrama. Sin embargo, si se trata de representar algo más que un sencillo proceso de secuencia sin muchas condiciones o un ciclo, la técnica no será de mucha ayuda.

Diagramas de Paquetes

Una de las más viejas cuestiones en métodos de software es: ¿Cómo descomponer un sistema grande dentro de sistemas más pequeños? Nos preguntamos esto porque los sistemas tienden a agrandarse, y resulta difícil entenderlos y hacer cambios sobre ellos.

Los métodos estructurados usaron descomposición funcional, en la cual todo el sistema era hecho como una función y descompuesto dentro de pequeños subsistemas, los cuales a su vez, quizá, eran descompuestos dentro de sub-sub-sistemas, y así en adelante. Las funciones eran como los casos de uso en los sistemas orientados a objetos en que las funciones representaban algo del sistema como un todo.

Todos esos fueron días cuando los procesos y los datos estaban separados. Así que además de la descomposición funcional, ahí hubo también datos estructurados. Esto tomo segundo lugar, aunque algunas técnicas de Ingeniería de Información agrupaban registros de datos dentro de áreas y matrices para mostrar como las funciones y los registros de datos interactuaban.

Es desde este punto de vista que nosotros vemos el más grande cambio en los objetos con los que estamos trabajando. La separación de procesos y datos esta hecha, la descomposición funcional esta dada, pero la vieja pregunta aún esta vigente. Una idea es agrupar clases dentro de unidades de un nivel más alto. Esta idea, aplicada muy vagamente, aparece en muchos métodos de objetos. En el UML, este mecanismo de agrupación es llamado el **paquete**.

La idea de un paquete puede ser aplicada a cualquier elemento del modelo, no sólo a clases. Sin alguna regla para tener a las clases juntas, la agrupación llega a ser arbitraria; la más usada, y más estresante, en el UML es la dependencia. Por lo regular se usa el término de **diagrama de paquetes**, para un diagrama que muestra paquetes de clases con dependencias entre ellos.

Estrictamente hablando, paquetes y dependencias son elementos sobre un diagrama de clases, así que el diagrama de paquetes es sólo una forma de diagramas de clases.

Una **dependencia** existe entre dos elementos si al cambiar la definición de los elementos puede causar cambios en el otro. Con clases, las dependencias existen por varias razones: una de las clases envía un mensaje a otra; una de las clases tiene a otra como parte de sus datos; una clase menciona a otra como parte de sus parámetros para una operación. Si una clase cambia su interface, entonces cualquier mensaje que envíe posiblemente no será válido.

Idealmente, sólo los cambios de la interface deberían afectar a otra clase. El arte del diseño a gran escala envuelve minimizar dependencias -de esta manera, los efectos de los cambios son reducidos y el sistema requiere de menos esfuerzo para los cambios-.

En la figura 16, tenemos un dominio de clases que modela los negocios de una empresa, son agrupados dentro de dos paquetes: Ordenes de compra y Clientes. Ambos paquetes son parte de un paquete de dominio general. La aplicación de Captura de Ordenes tiene una dependencia con estos dos paquetes. La Orden de Captura UI tiene dependencias con la aplicación de Orden de Captura y la AWT (una herramienta GUI de Java).

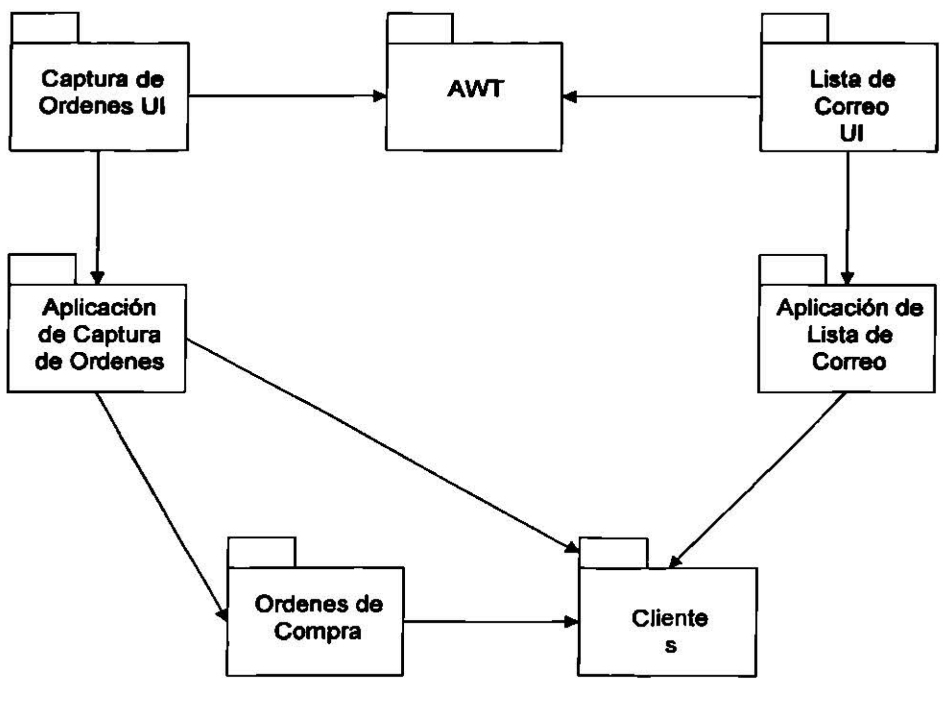


Fig. 16

Una dependencia entre dos paquetes existe si cualquier dependencia existe entre dos clases de los paquetes. Por ejemplo, si cualquier clase en el paquete de Liste de Correo es dependiente sobre cualquier clase en el paquete de Clientes, entonces una dependencia existe entre sus correspondientes paquetes.

Hay una obvia similitud entre las dependencias de paquetes y la compilación de paquetes. De hecho, hay una vital diferencia: Con los paquetes, las dependencias no son transitivas.

Un ejemplo de una relación transitiva es por ejemplo, Jim tiene una barba más grande que Grady, y Grady tiene una barba más grande que Ivar, así que nosotros podemos deducir que Jim tiene una barba más grande que Ivar. Otros ejemplos incluyen relaciones tales como "es norte de", y "es más alto que". Por otra parte "es amigo de" no es una relación transitiva.

Para ver porque son importantes las dependencias, miremos la figura 16 otra vez. Si una clase del paquete de Ordenes de Compra cambia, esto no indica que el paquete de Ordenes de Compra UI necesite ser cambiada. Esto sólo indica que el paquete de aplicación Captura de Ordenes necesita ser observada si algo cambia. Sólo si la interface

del paquete de aplicación de Ordenes de Captura es alterada entonces hace que el paquete de Captura de Ordenes UI necesite ser cambiada.

Este comportamiento es el propósito clásico de una arquitectura por capas. Efectivamente, estos son semánticos del comportamiento de los "imports" de Java pero no así los "includes" de C/C++. Los includes son transitivos, lo cual significa que la Captura de Ordenes UI debería depender del paquete de Ordenes de Compra. Una dependencia transitiva hace difícil limitar el alcance de los cambios en la compilación.

¿Qué significa dibujar dependencia a un paquete que contiene otros subpaquetes?. Los diseñadores usan diferentes convenciones. Algunos asumen que dibujar una dependencia a un paquete "contenedor" da visibilidad sobre los paquetes contenidos. Otros dicen que sólo se deben ver los paquetes que están conteniendo, no las clases con paquetes anidados (esto es, la vista se opaca).

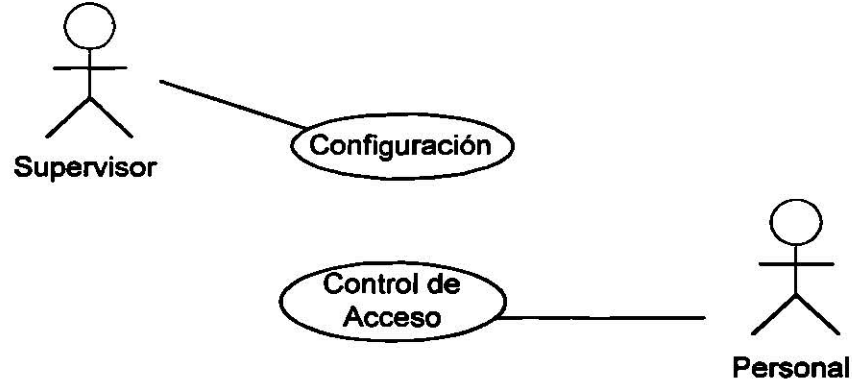
Caso Práctico

El caso práctico tomado es un breve ejemplo de cómo se podría utilizar el UML para el diseño de un sistema. Este no muestra todas las características del UML ya que solo ve la parte práctica en la experiencia del autor.

El caso que se mostrará a continuación esta basado en el acceso de personas a un edificio. Cada persona está asignada a uno o varios grupos de personas que tienen acceso a diferentes partes del edificio.

De lo anterior tenemos lo siguiente:

Dos actores uno de los cuales está encargado de la configuración del sistema (supervisor) y el otro actor es la gente que accesará el edificio (en este caso usuario).



Para este ejemplo estudiaremos los siguientes escenarios.

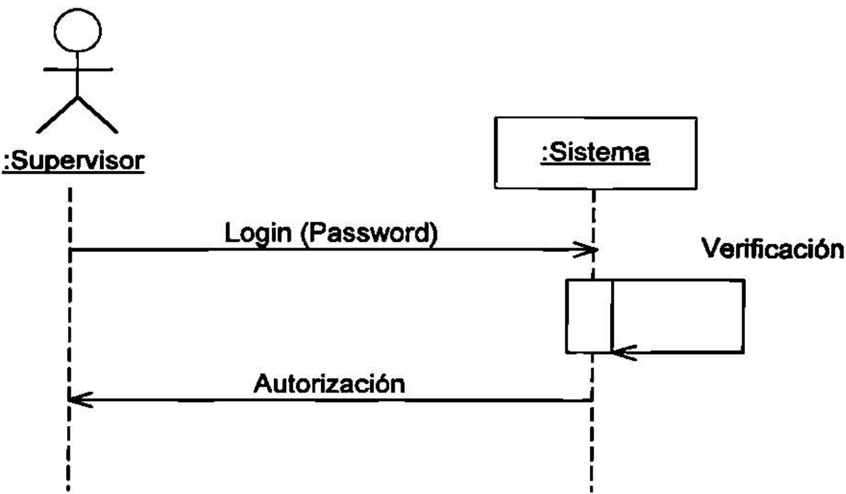
Configuración	Identificación Modificación de puertas Modificación de una persona Modificación de grupos de personas Buscar puertas accesibles a una persona dada Buscar los grupos en los que está contenida una persona Buscar las personas contenidas en un grupo Modificación de Acceso para un grupo de personas para una puerta.
Control Acceso	Autorización de Entrada.

En la siguiente parte se mostrará gráficamente con diagramas de secuencia, los escenarios asociados con el caso de uso de configuración.

Configuración

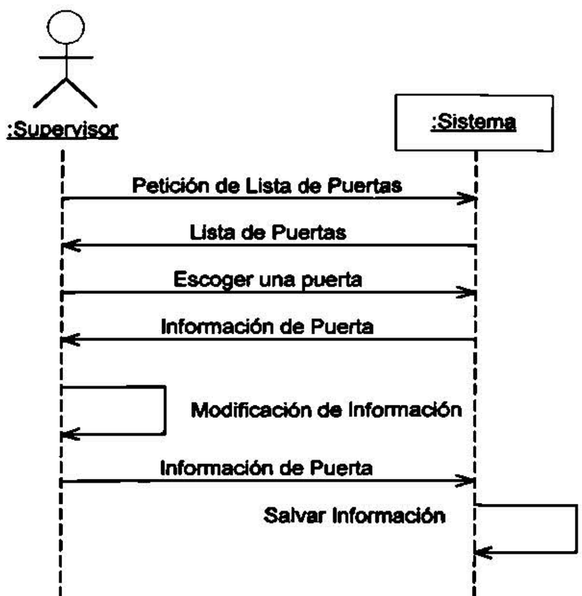
Identificación

El supervisor se conecta al sistema y da su password.
El sistema verifica el usuario y su password.
El sistema permite la conexión.



Modificación de puertas

El supervisor pide una lista de puertas.
El sistema muestra la lista de puertas.
El supervisor escoge una puerta.
El sistema despliega su información.
El supervisor modifica la información de la puerta.
El sistema registra los cambios.



Modificación de una persona

El supervisor pide una lista de personas.

El sistema muestra la lista de personas.

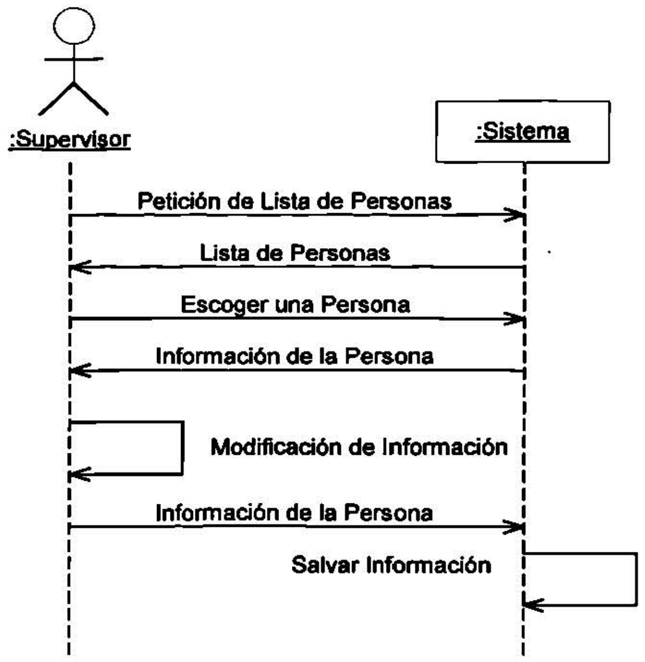
El supervisor escoge una persona.

El sistema despliega su información.

El supervisor modifica la información de la persona.

El sistema registra los cambios.

Modificación de un grupo de personas



El supervisor pide una lista de grupos de personas.

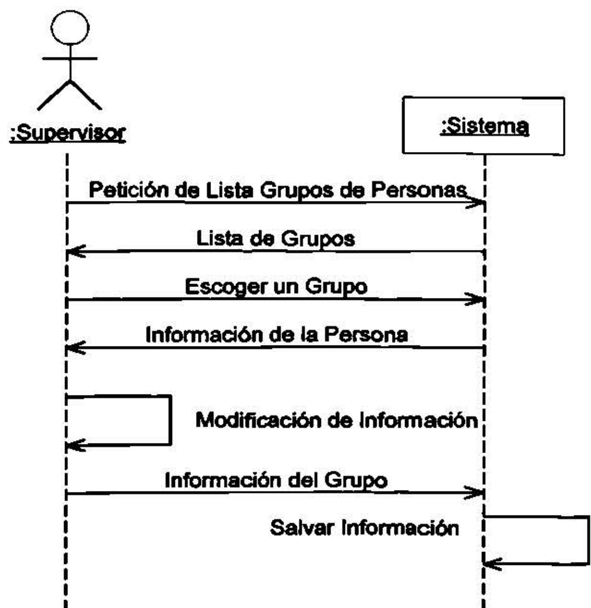
El sistema muestra la lista de grupos de personas.

El supervisor escoge un grupo de personas.

El sistema despliega su información.

El supervisor modifica la información de los grupos de personas.

El sistema registra los cambios.



Buscar puertas accesibles a una persona dada

El supervisor pide una lista de personas.

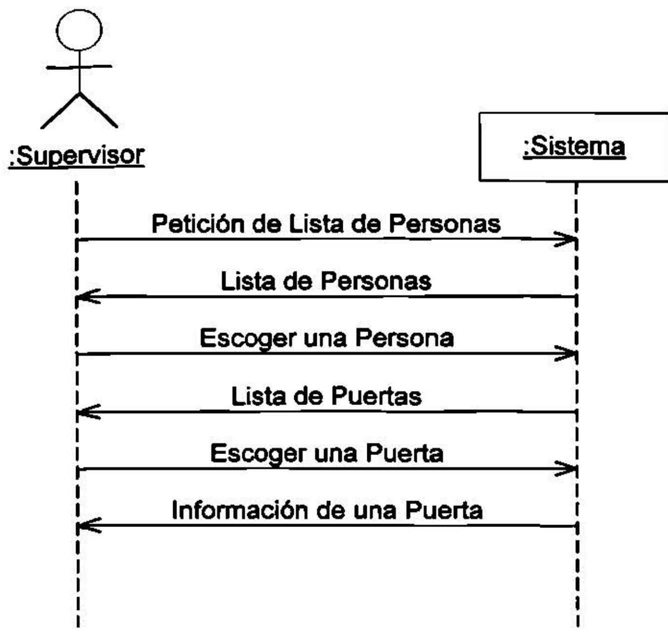
El sistema despliega a las personas que están registradas.

El supervisor escoge a una persona.

El sistema despliega la lista de puertas accesibles.

El supervisor escoge una persona.

El sistema despliega la información de la puerta.



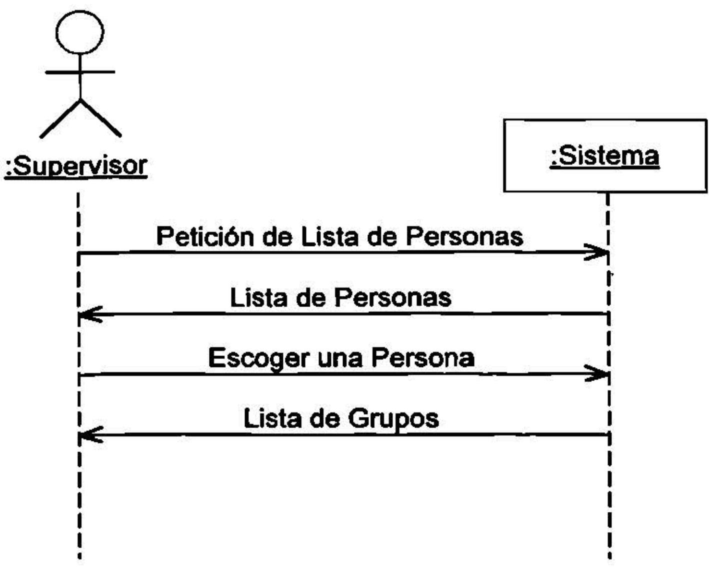
Buscar los grupos en los que está una persona

El supervisor pide una lista de personas.

El sistema despliega la lista de puertas accesibles.

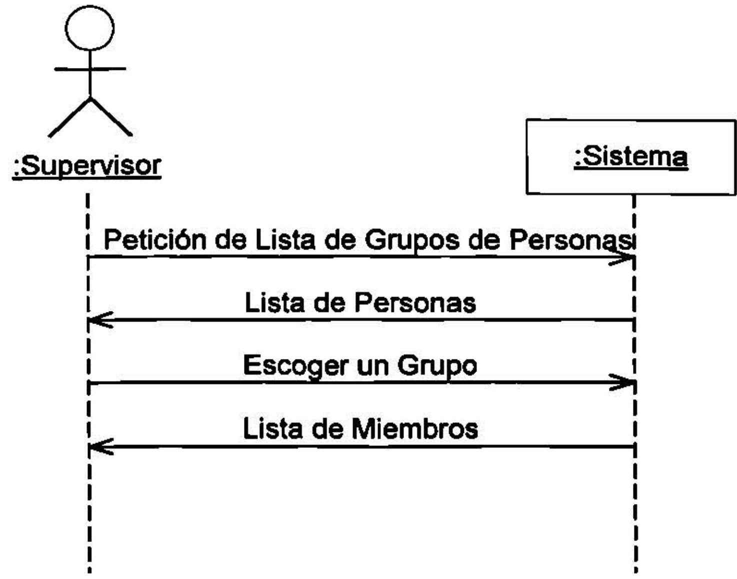
El supervisor escoge una persona.

El sistema despliega los grupos en los que esta la persona.



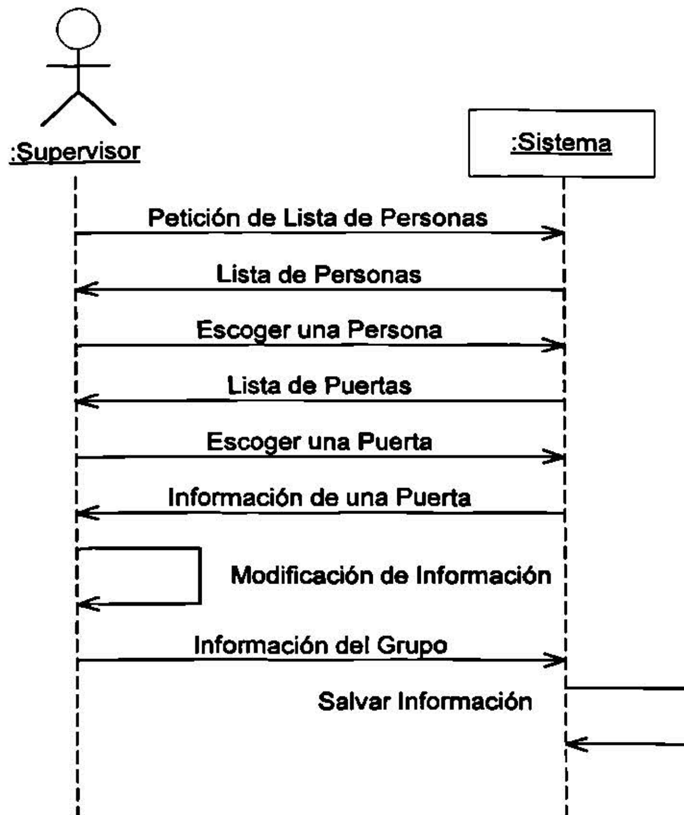
Buscar las personas contenidas en un grupo

El supervisor pide una lista de grupos de personas.
El sistema muestra la lista de grupos de personas.
El supervisor escoge un grupo de personas.
El sistema despliega la lista de personas que contiene ese grupo.



Modificación de Acceso para un grupo de personas para una puerta.

El supervisor pide una lista de grupos de personas.
El sistema muestra la lista de grupos de personas.
El supervisor escoge un grupo de personas.
El supervisor escoge una puerta de una lista.
El sistema despliega la información de la puerta.
El supervisor modifica la información existente.
El sistema registra la información.

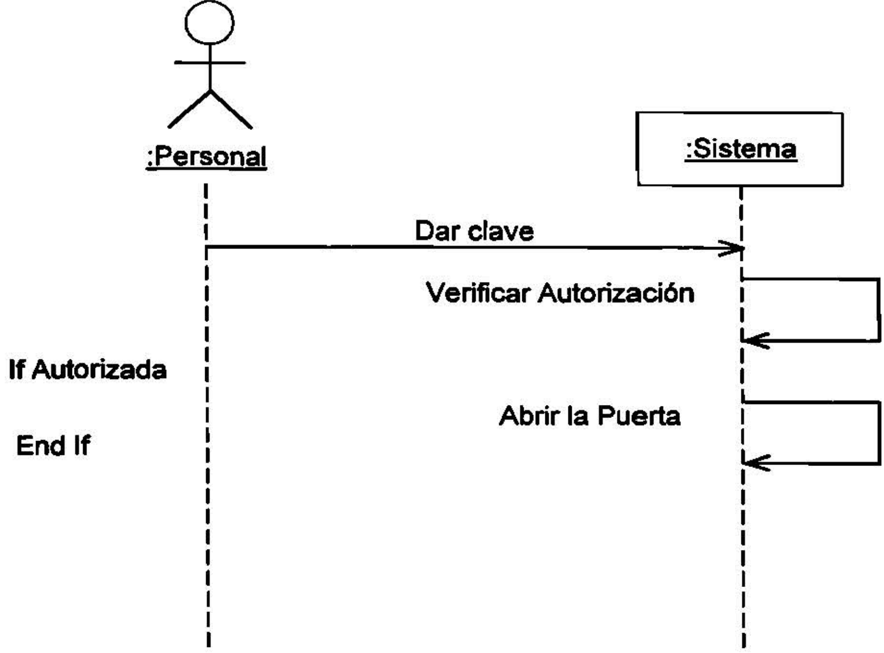


Control de Acceso.

Para el control de Acceso sólo hay un escenario, el cual es tener acceso.

Autorización de Entrada

La persona da su clave.
El sistema determina si tiene acceso.
Si el acceso es autorizado.
El sistema abre la puerta.



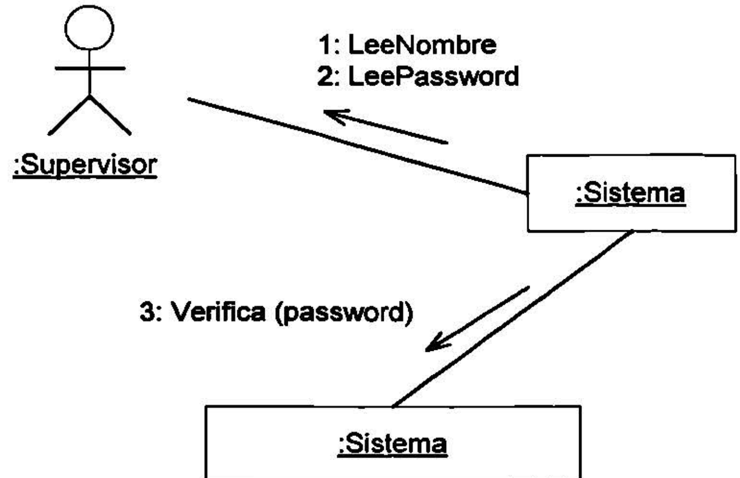
Los diagramas de colaboración agregan más objetos al sistema que por lo regular son las interfaces gráficas que el sistema tendrá para interactuar con el sistema, así que revisaremos estos diagramas para cada uno de los escenarios antes vistos.

El primer escenario se verá a detalle, los siguientes escenarios se mostrarán sólo en forma de diagramas.

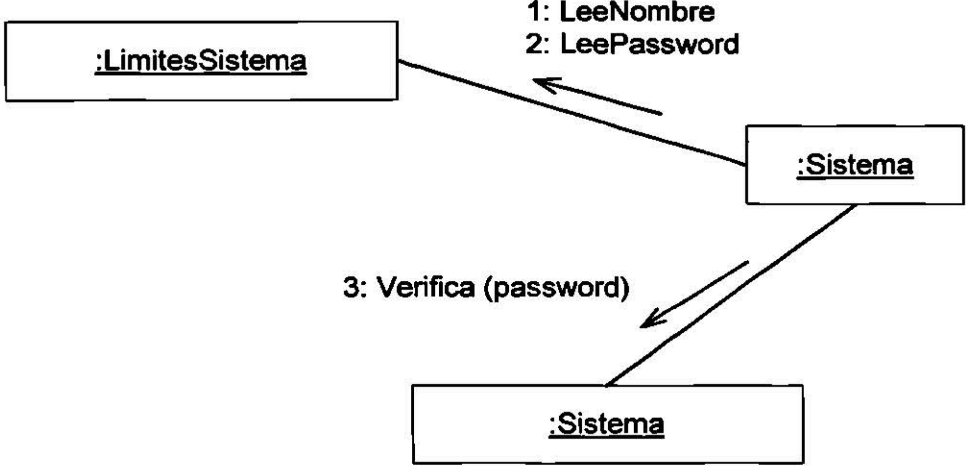
Configuración

Identificación de supervisor.

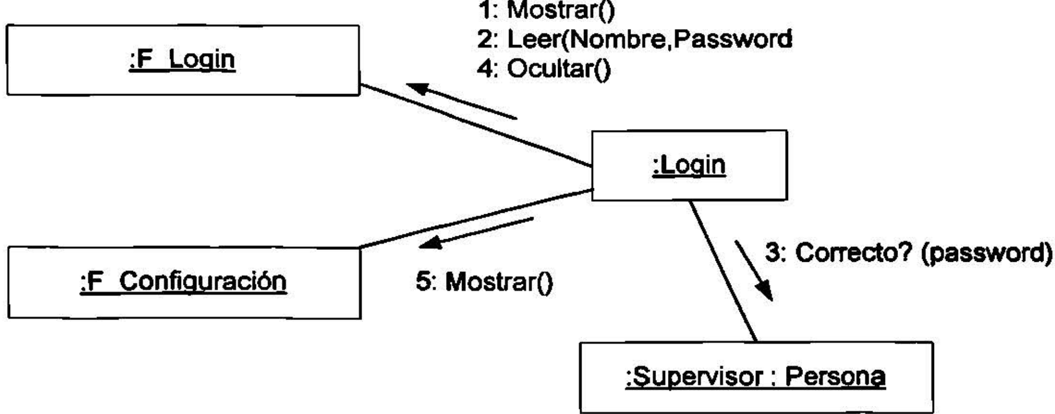
Primero, la noción de un actor debe ser reservada, para representar la interface del usuario de forma compacta.



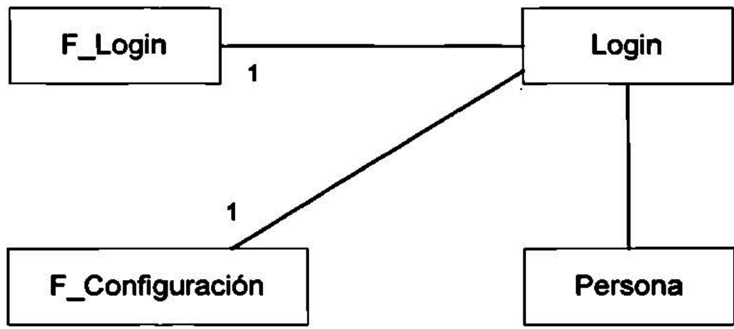
Alternativamente, es posible crear una clase LimitesSistema y usarla en lugar del actor.



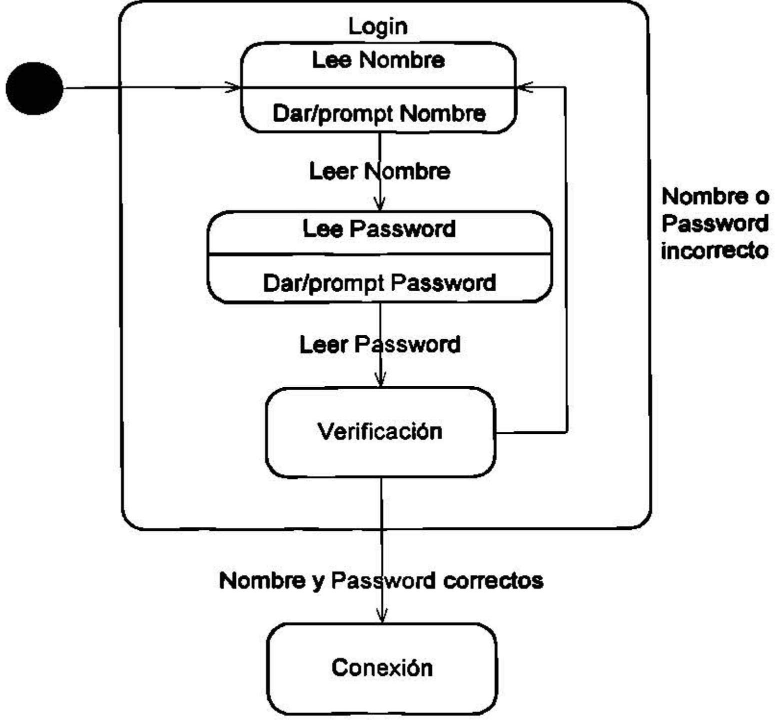
Cada objeto de dominio es representado por un objeto de interface del mismo nombre, con un prefijo **F**. En el siguiente diagrama, el objeto de la clase **Login** es accesible gráficamente por el objeto gráfico **F_Login**.



Después mostramos su diagrama de clases, y como vemos la multiplicidad no es un factor que influye hasta ahora.



El comportamiento de la clase **Login** es mostrado en un diagrama de estados a continuación.



Modificación de puertas

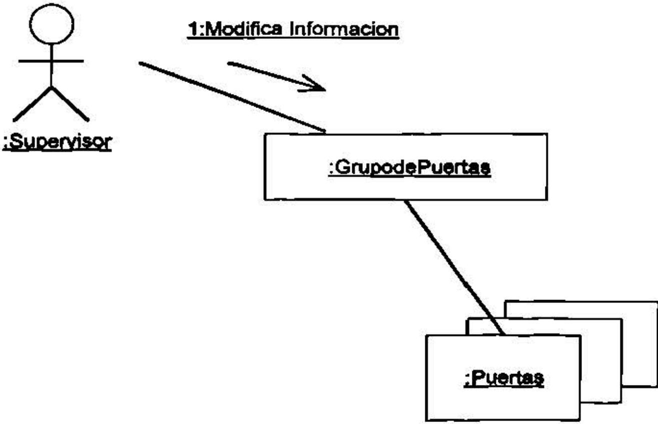


Diagrama de colaboración

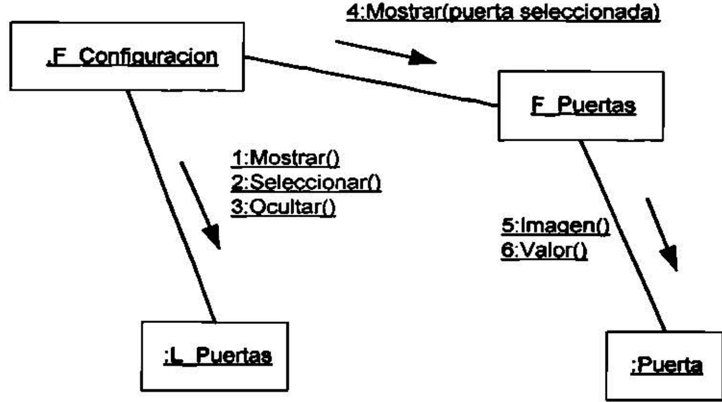
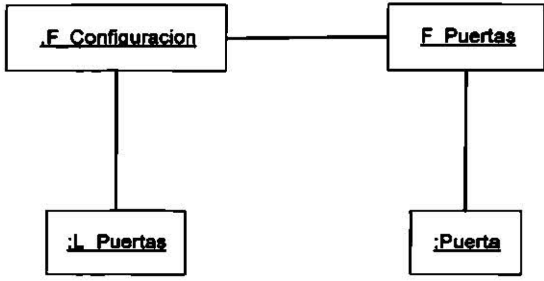


Diagrama de clases



Modificación de una persona

Diagrama de colaboración

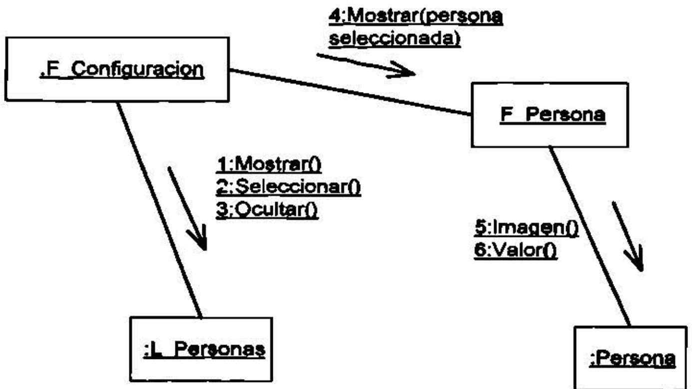
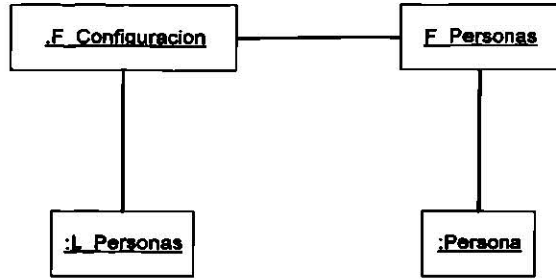


Diagrama de clases



Modificación de grupos de personas

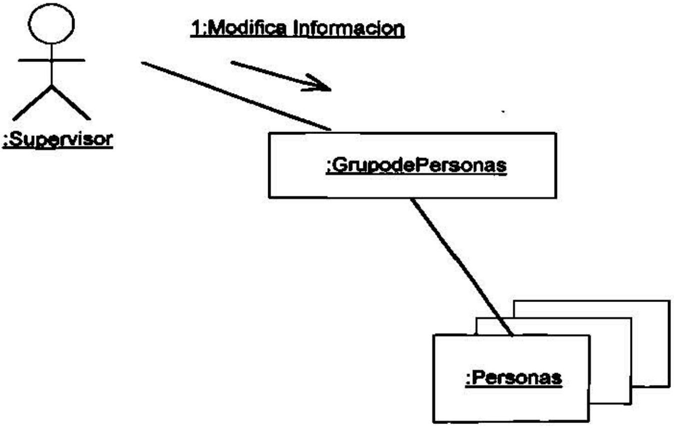


Diagrama de colaboración

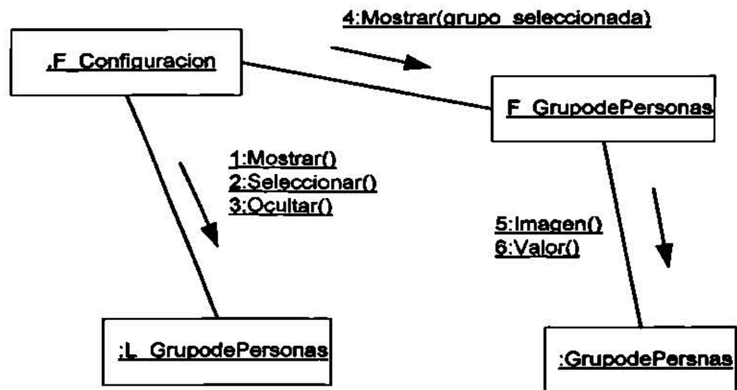
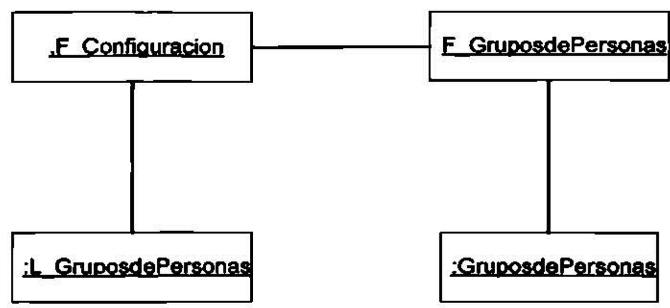


Diagrama de clases



Buscar puertas accesibles a una persona dada

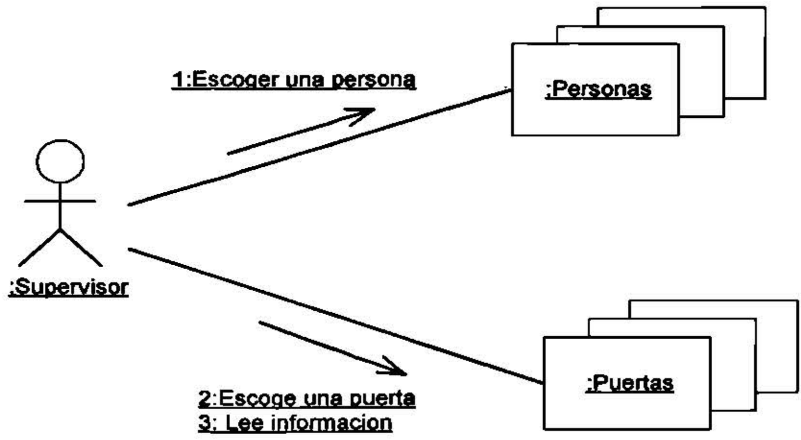


Diagrama de colaboración

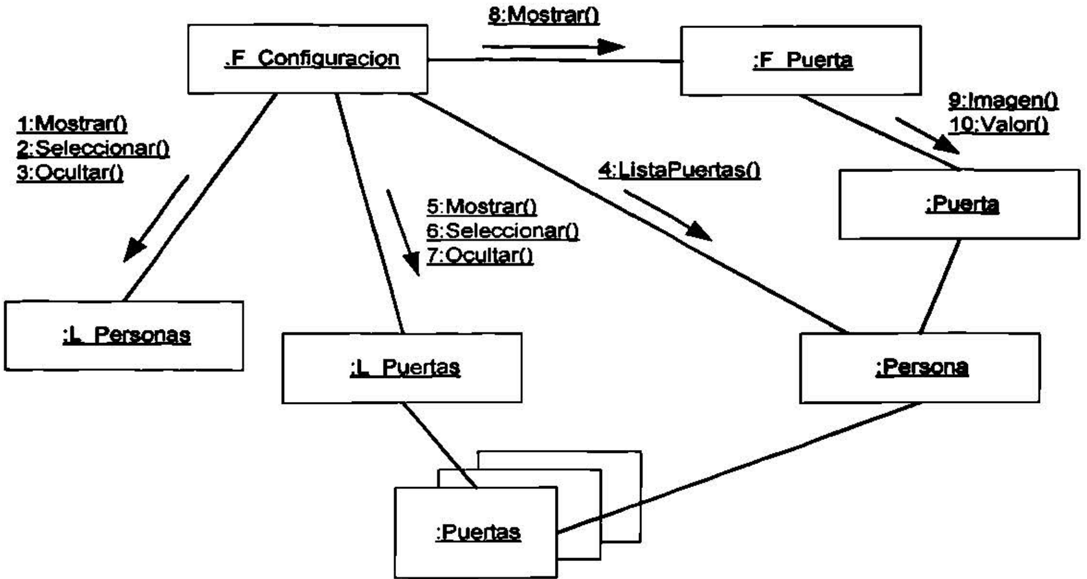
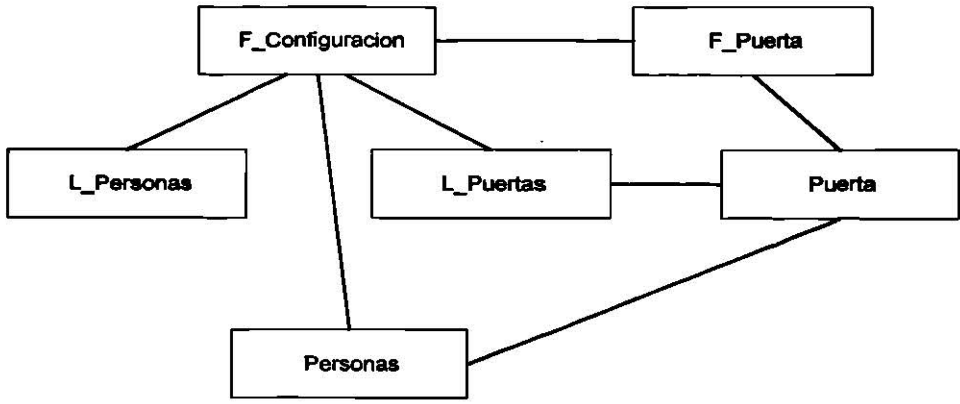


Diagrama de clases



Buscar los grupos en los que esta contenida una persona

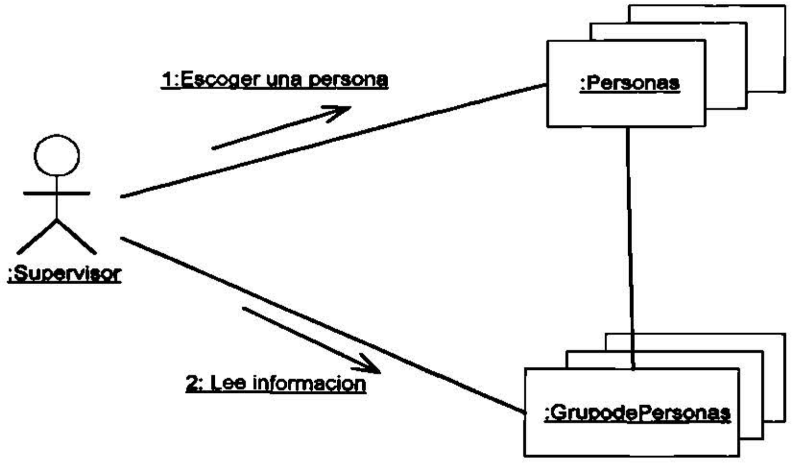


Diagrama de colaboración

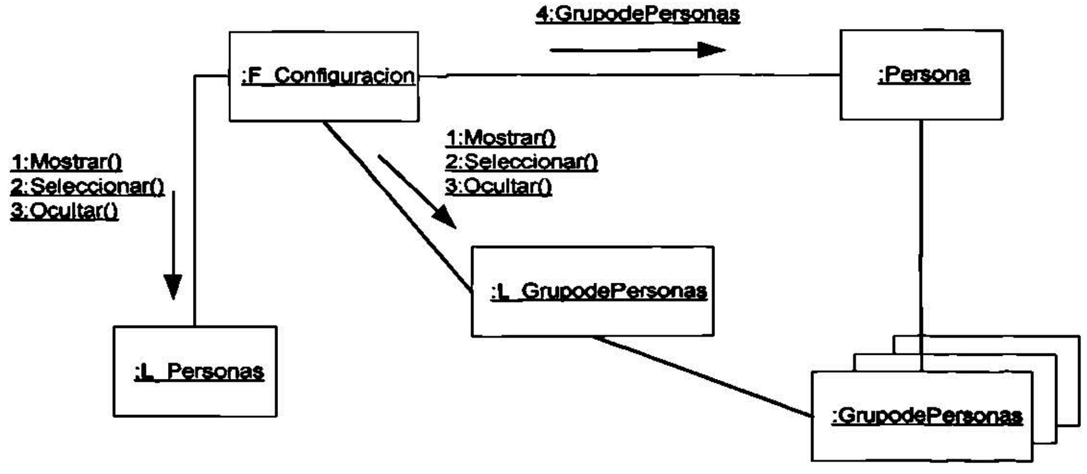
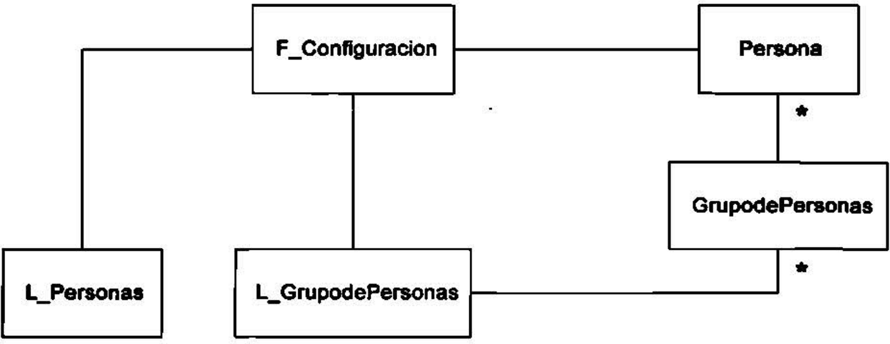


Diagrama de clases



Buscar las personas contenidas en un grupo

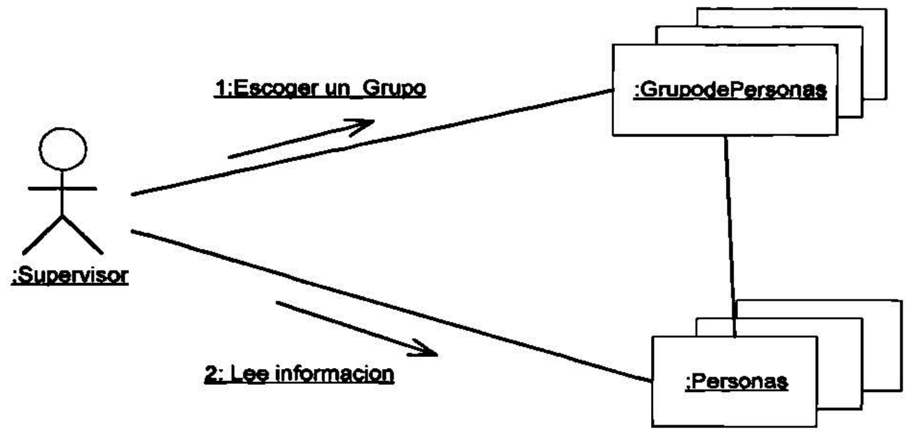


Diagrama de colaboración

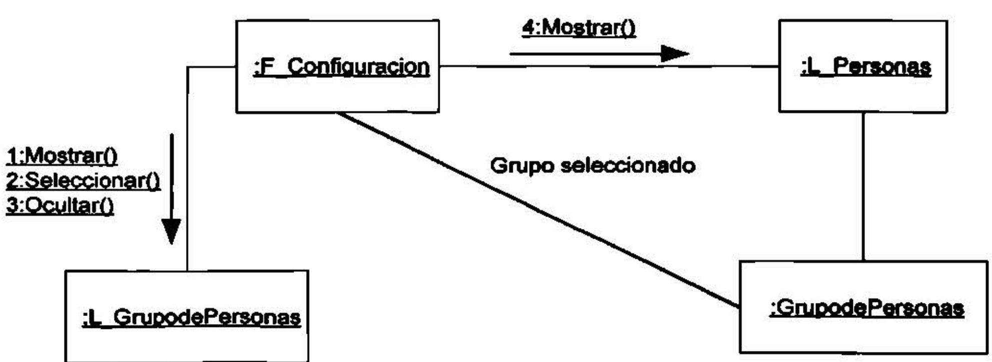
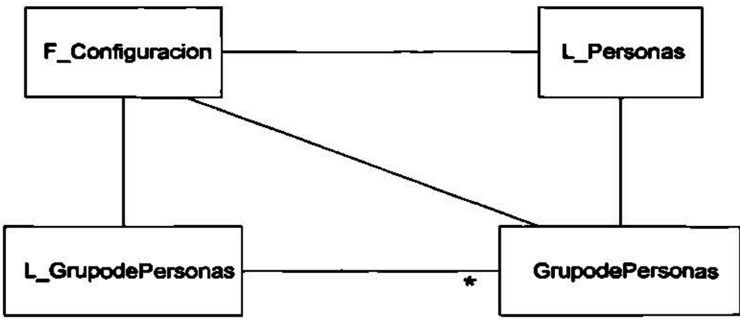


Diagrama de clases



Modificación de Acceso para un grupo de personas para una puerta.

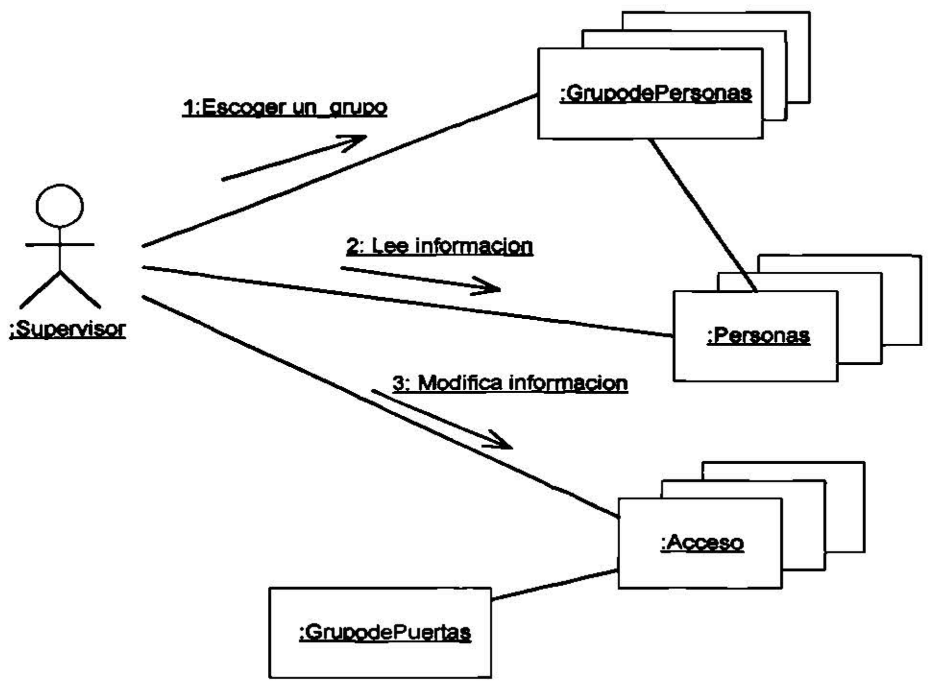


Diagrama de colaboración

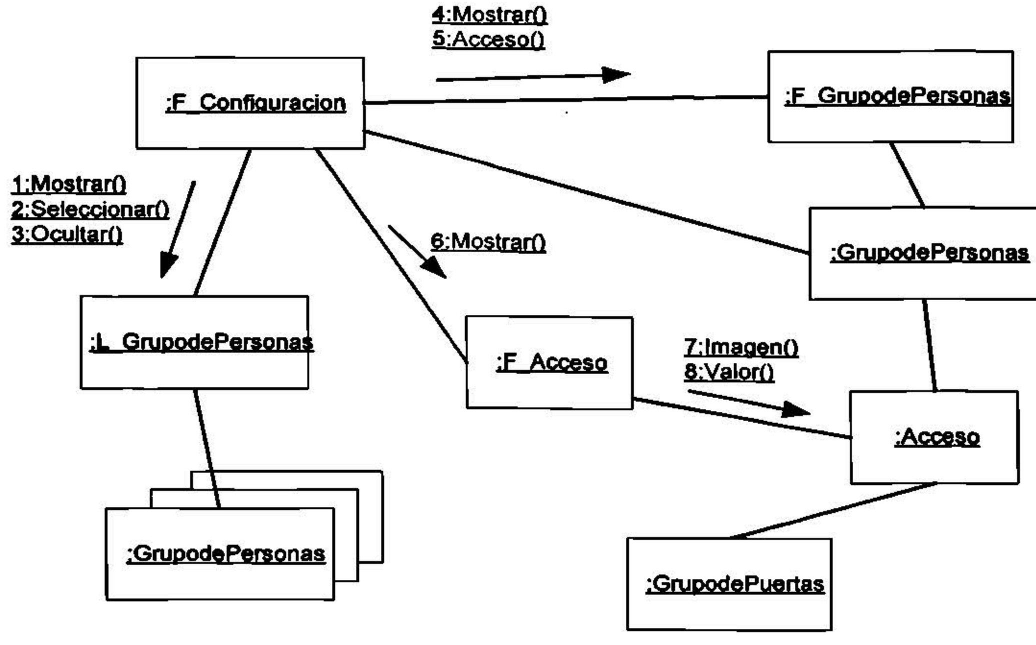
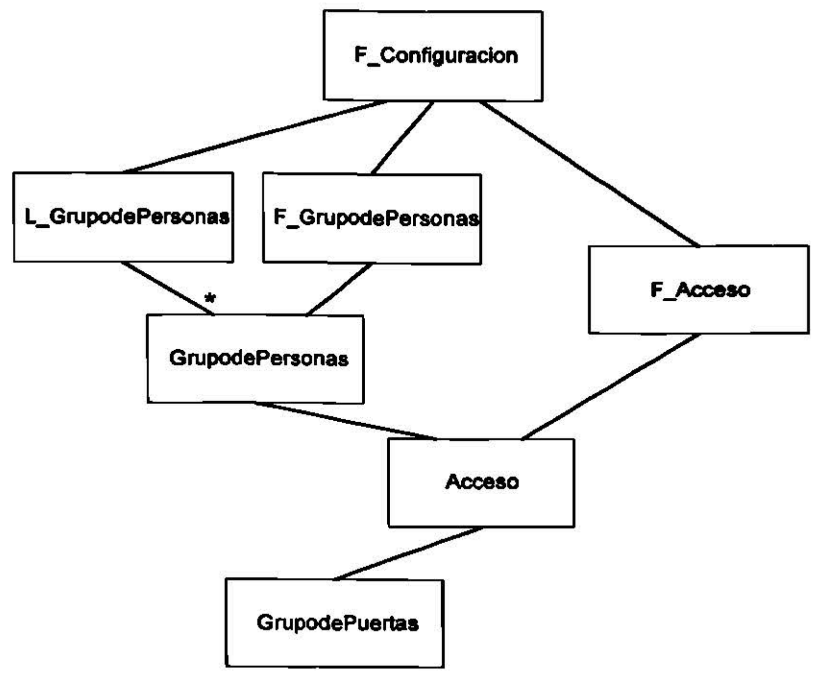


Diagrama de clases



Autorización de Entrada

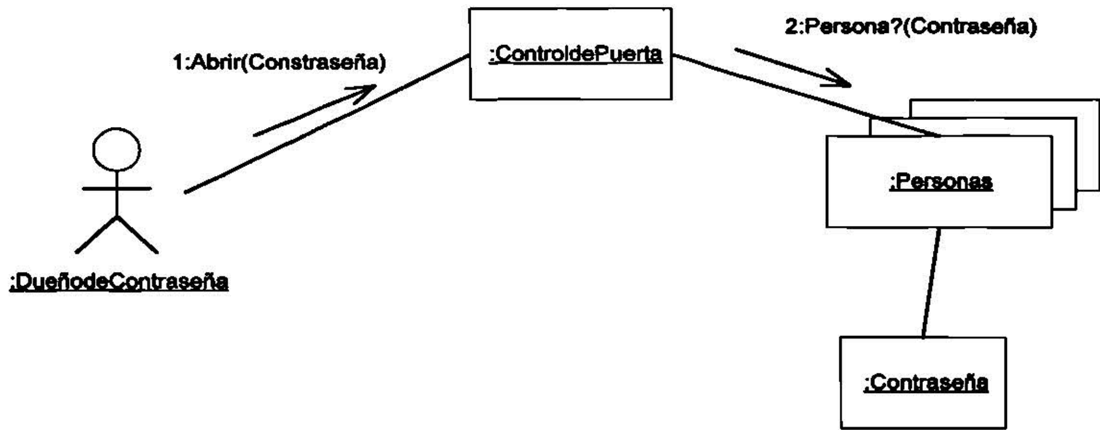


Diagrama de colaboración

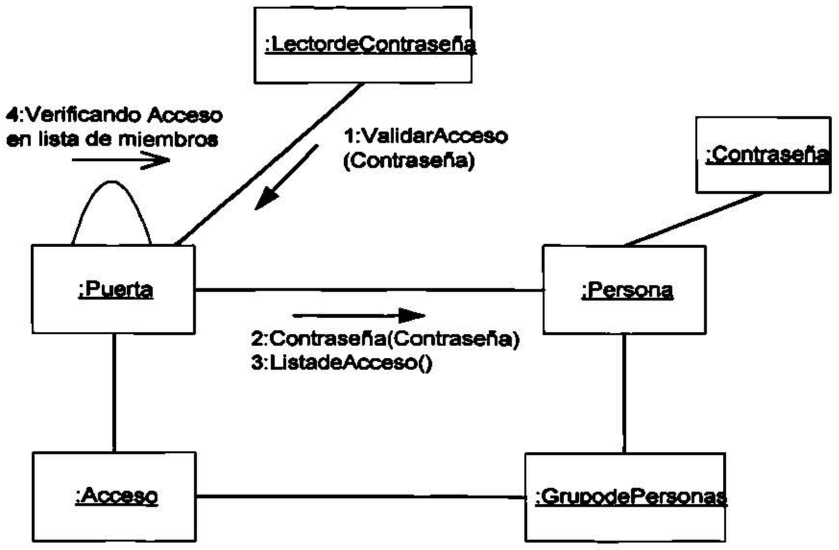
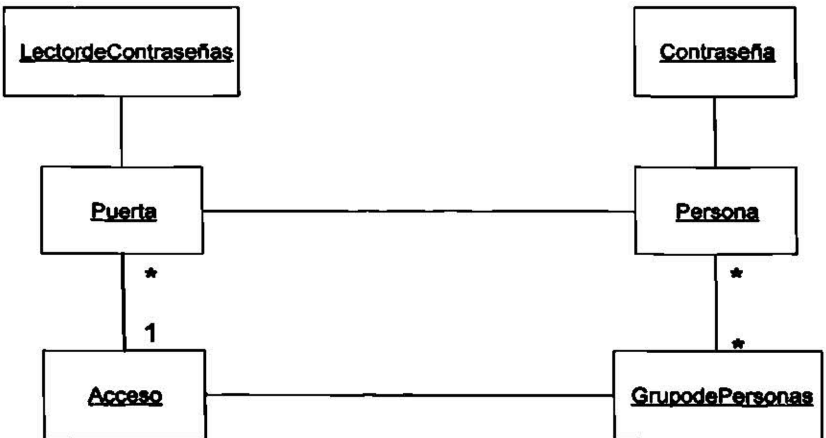


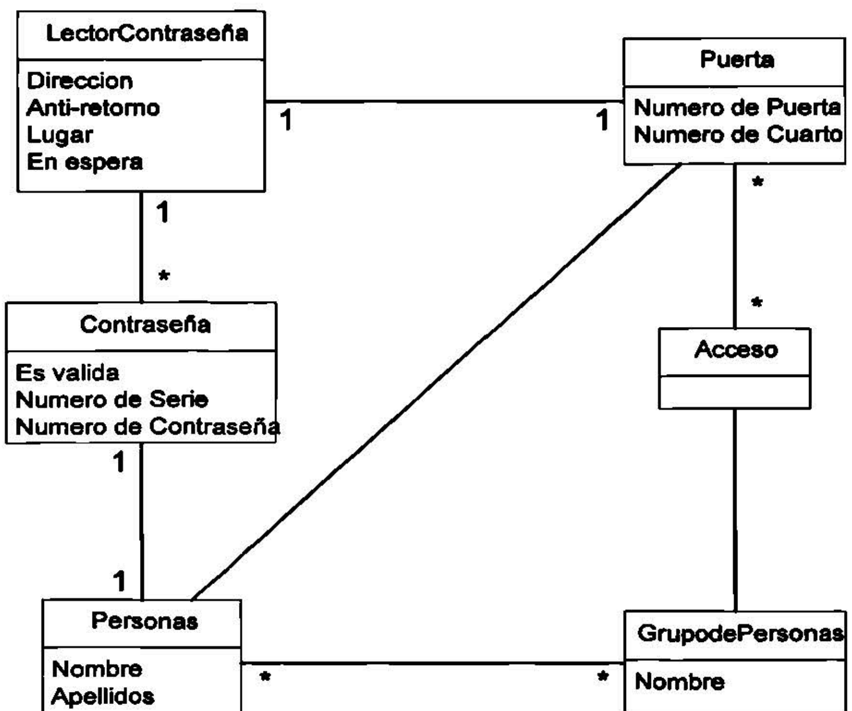
Diagrama de clases



Los casos de uso dividen los requerimientos de acuerdo al punto de vista de cada actor a la vez. La descripción dada por los casos de uso es puramente funcional, y es importante tener cuidada al no iniciar una descomposición funcional en lugar una descomposición orientada a objetos. Los casos de uso deben ser vistos como clases de comportamiento.

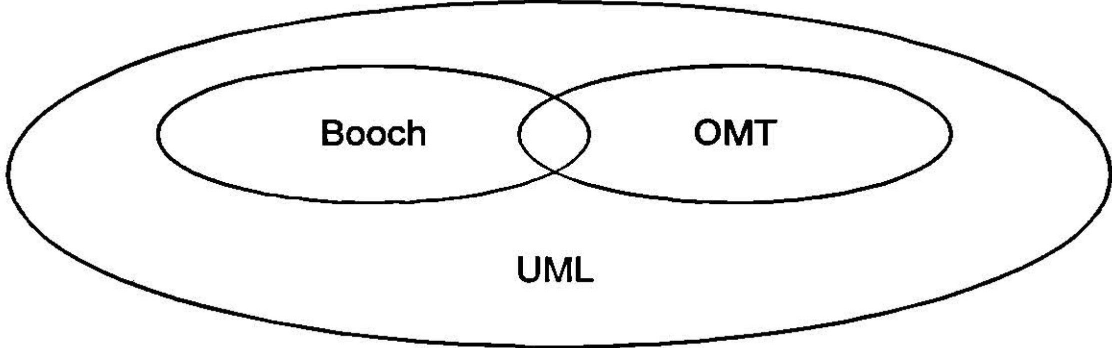
El UML implementa los casos de uso para ayudar a la colaboración entre los objetos que vienen desde el diagrama de dominio. Cada colaboración combina el contexto de un objeto y las interacciones entre estos objetos. El contexto del objeto es expresado en una forma específica en los diagramas de colaboración y en una forma general en los diagramas de clases.

A continuación se muestra el diagrama de clases que representa al sistema tomado para este ejemplo.



Guía para hacer la transición de Booch y OMT a UML

Gráficamente, UML es más parecido a OMT que a Booch, desde que los iconos de nubes fueron abandonados y remplazados por rectángulos, los cuales son más fáciles de dibujar. Más allá de las consideraciones gráficas, UML, puede ser considerado como el contenedor de las otras dos notaciones.



Las siguientes tablas ilustran las diferencias entre notaciones pertenecientes a los conceptos principales de objetos.

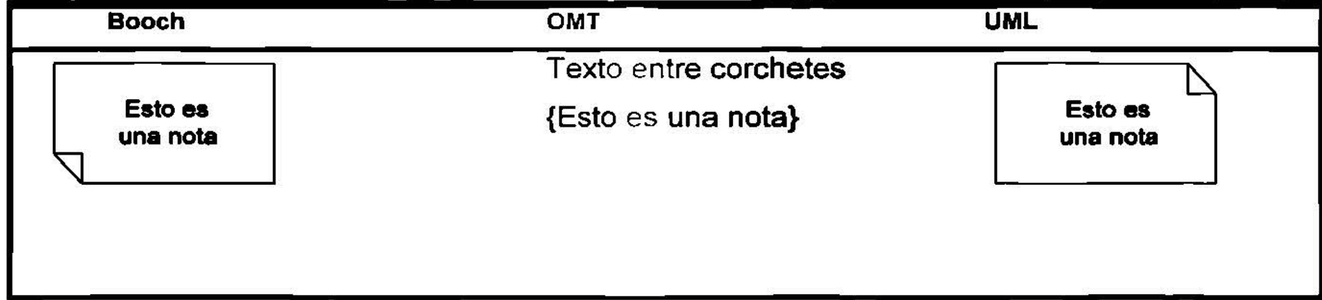
Constraints

En las tres notaciones, los constraints son representados por expresiones entre corchetes.

Booch	OMT	UML
Texto entre corchetes	Texto entre corchetes	Texto entre corchetes
{Esto es un constraint}	{Esto es un constraint}	{Esto es un constraint}

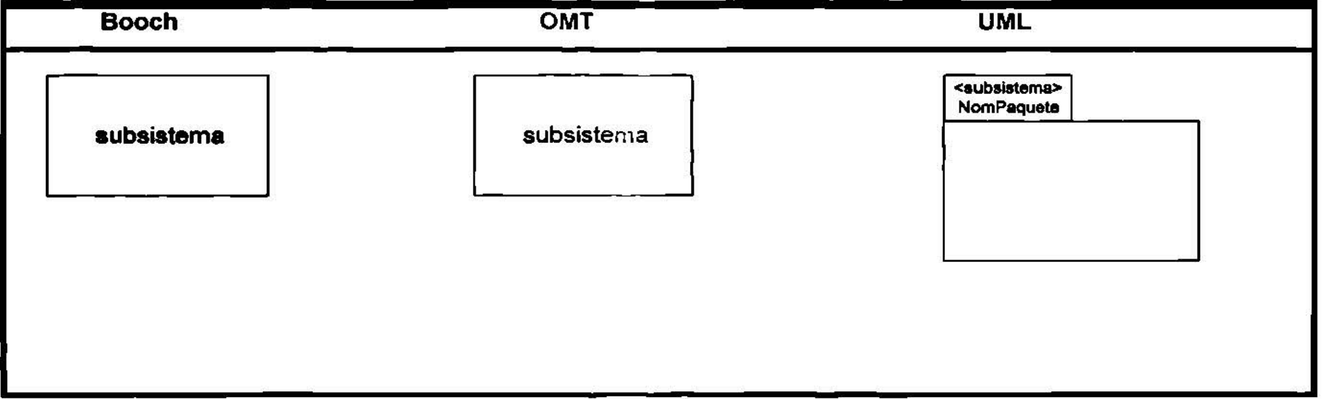
Notas

En Booch y UML, las notas son representadas por rectángulos con una esquina doblada. En OMT, las notas son representadas como los constraints. En Booch, la esquina inferior izquierda esta doblada, mientras que en UML la esquina superior derecha esta doblada.



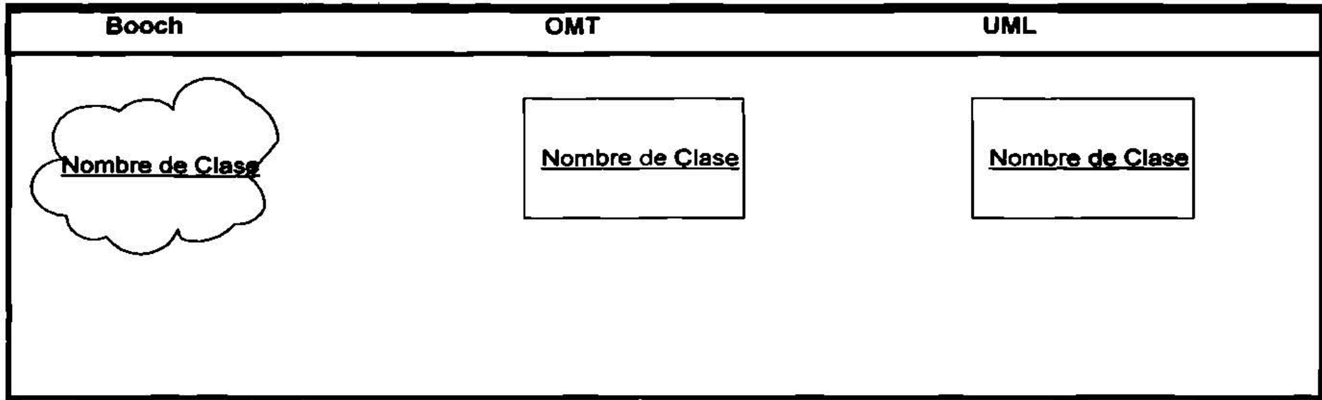
Subsitemas

En Booch y OMT, los subsistemas son parte de los elementos de un modelo. En UML, los subsistemas son implementados por estereotipación de paquetes.



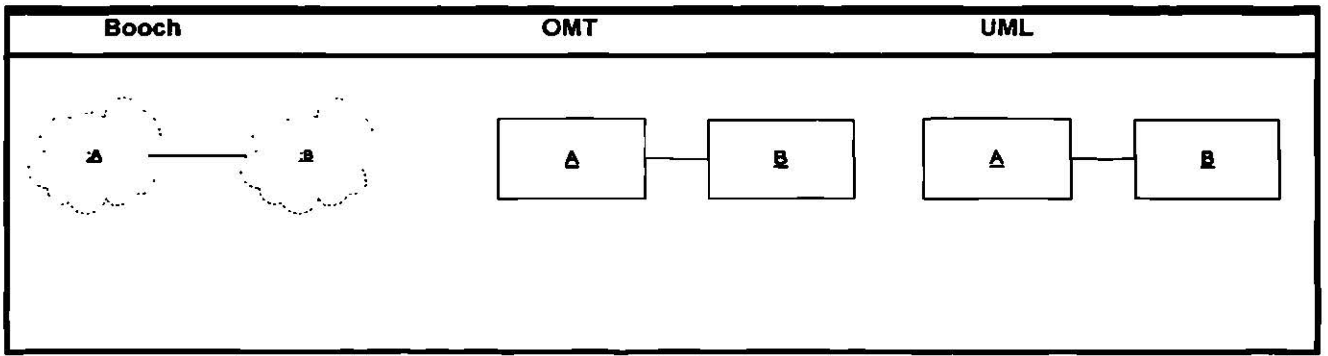
Objetos

En Booch, los objetos son representados por nubes. En OMT, como en UML, los objetos son representados por rectángulos. En UML, el nombre del objeto esta subrayado.



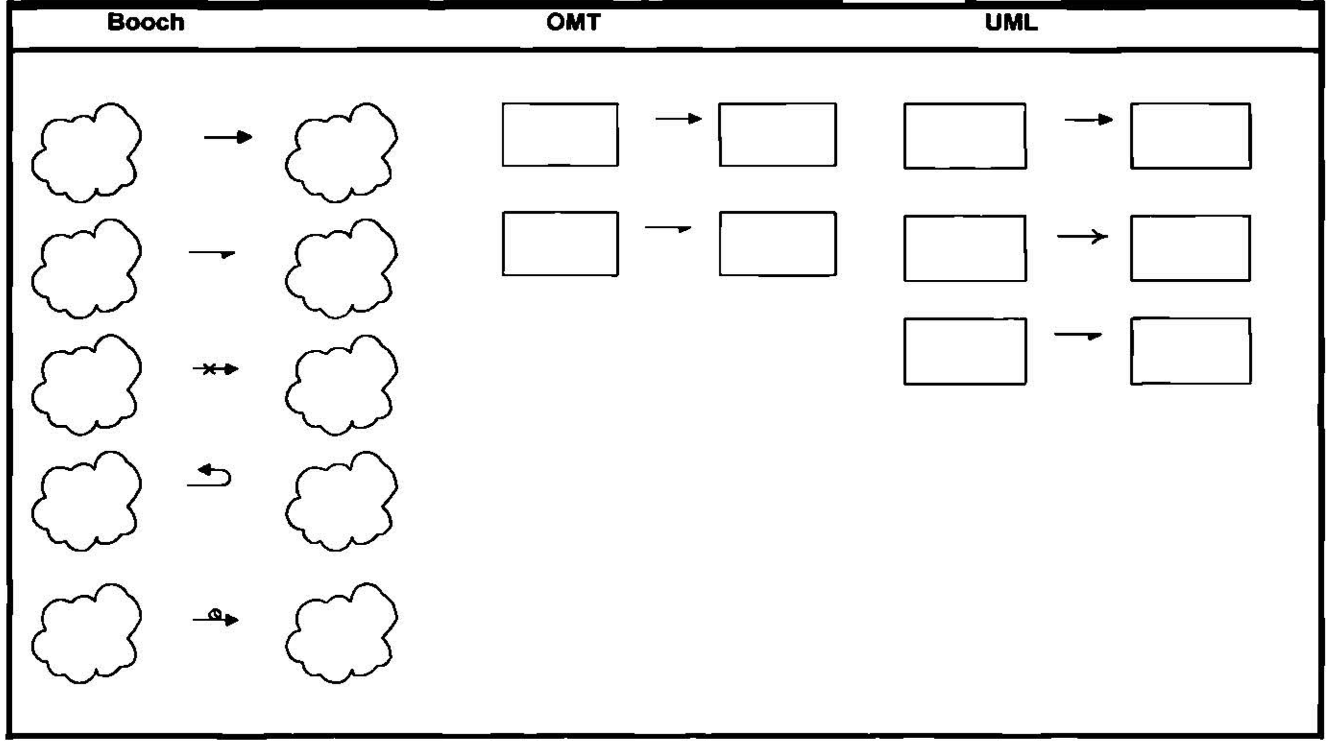
Links

En las tres notaciones, los vínculos son representados por una línea continua dibujada entre los objetos.



Mensajes

En las tres notaciones, los mensajes son representados usando flechas colocadas cerca de los vínculos. El tipo de control de flujo esta representado por el uso de una particular cabeza de la flecha.



Etiquetas de Mensajes

En las tres notaciones, las etiquetas de mensajes son representadas usando una expresión situada de frente al nombre del mensaje.

Booch	OMT	UML
Notación decimal	Notación de punto decimal	Notación de punto decimal modificado. Nombre del "thread". Paso con un thread. *[iteración]

Flujo de Datos

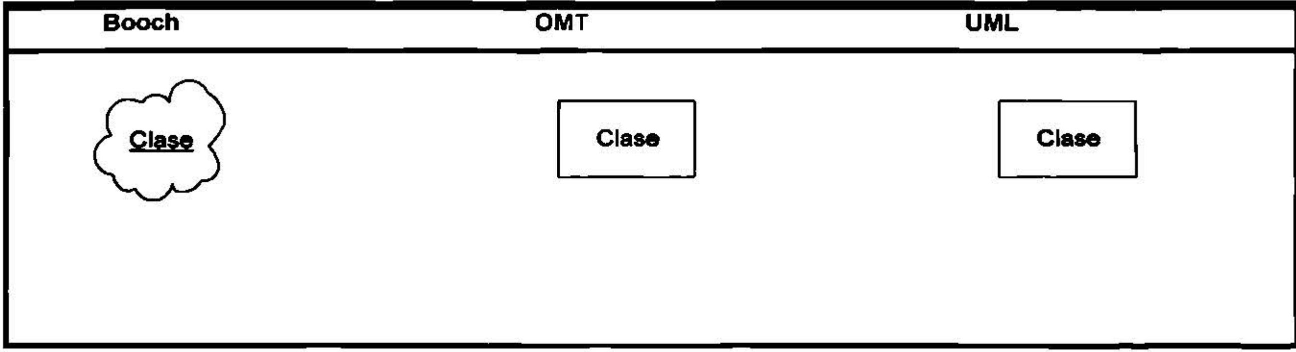
Booch y UML, permiten la representación de flujo de datos tan bien como el control de flujo (los mensajes) usando pequeños círculos conectados a una flecha apuntando en la dirección del flujo de los datos. Esta notación es opcional, ya que llega a ser redundante - el flujo de los datos puede ser representado en una etiqueta de mensajes.

Booch	OMT	UML
	Representado por una etiqueta de mensaje	

Clases

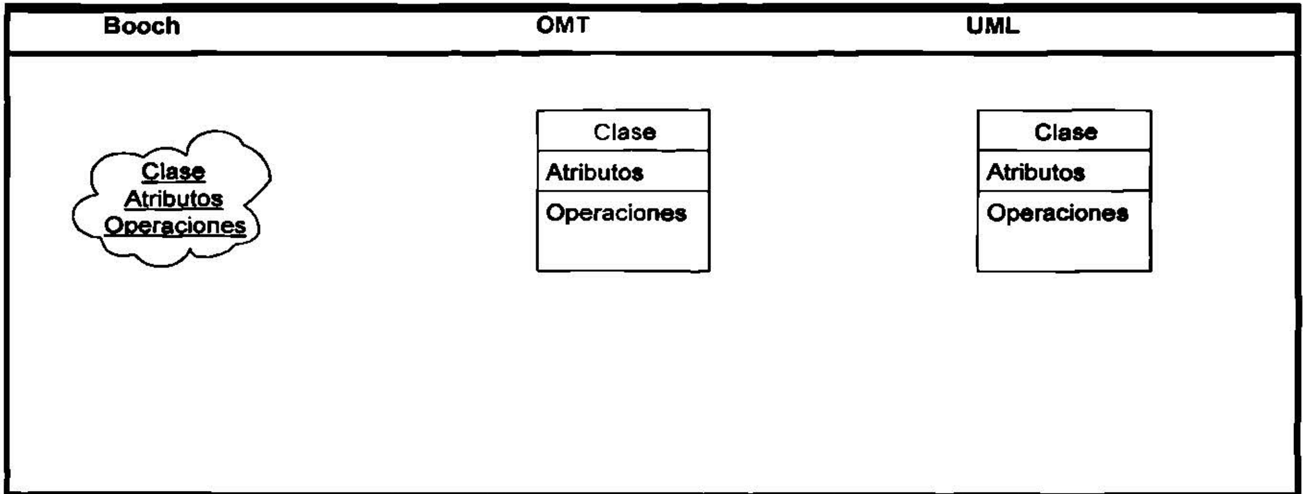
En Booch, las clases son representadas usando nubes punteadas. En OMT, como en UML, las clases son representadas usando rectángulos.

Clases Simples



Atributos y Operaciones

En las tres notaciones, los atributos y las operaciones son representadas con un icono de la clase. Algunos atributos y operaciones pueden ser ocultados para darle claridad a los diagramas. OMT y UML usan cajas para distinguir los atributos de las operaciones.



Visibilidad

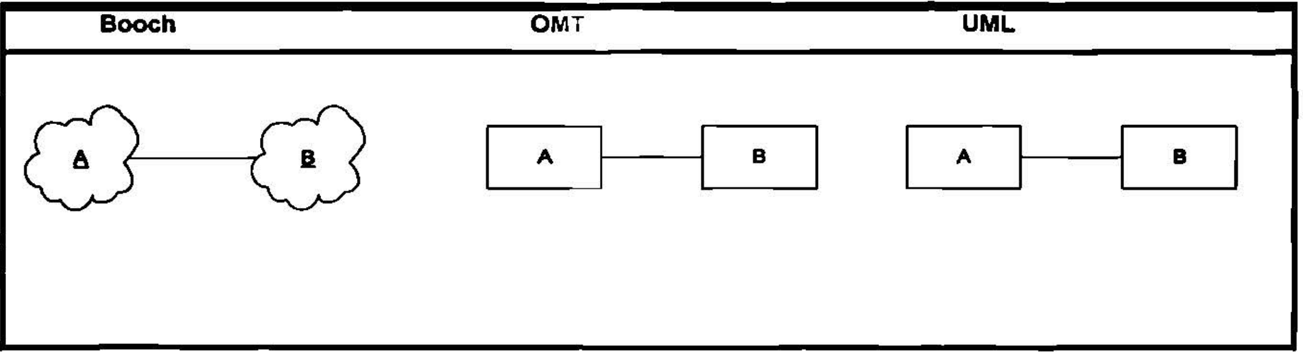
Las tres notaciones proponen niveles públicos, protegidos y privados con las mismas semánticas como el lenguaje de C++. Booch da un nivel de "implementación" adicional.

Booch	OMT	UML
nothing public	+ publico	**nothing** no especificado
| protegido	# protegido	+ publico
| | privado	- privado	# protegido
| | | implementación		- privado

RELACIONES

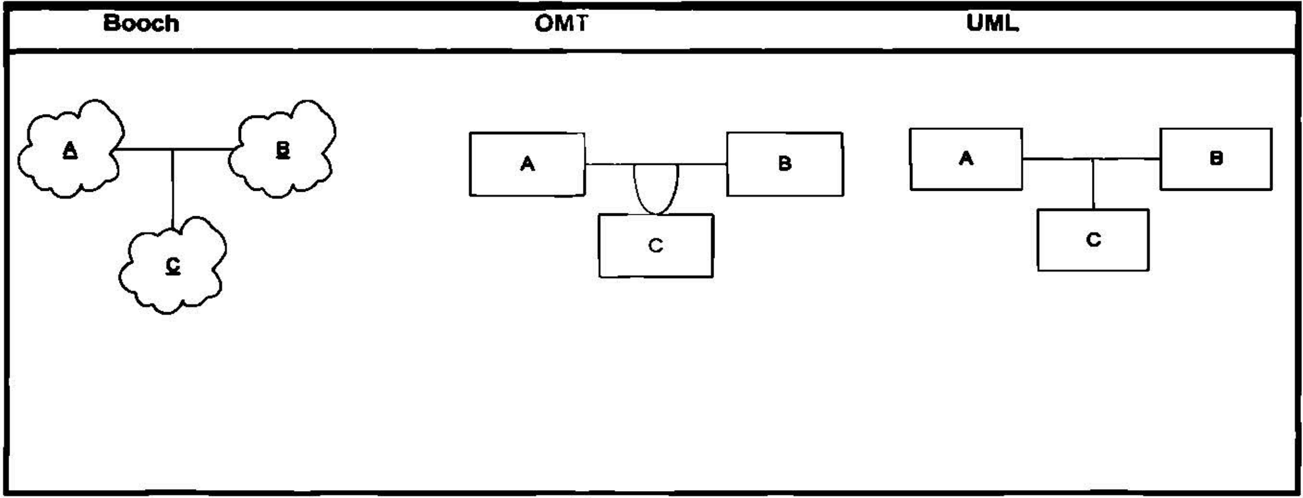
Asociaciones

En las tres notaciones, una asociación es representada por una línea continua dibujada entre las clases que participan.



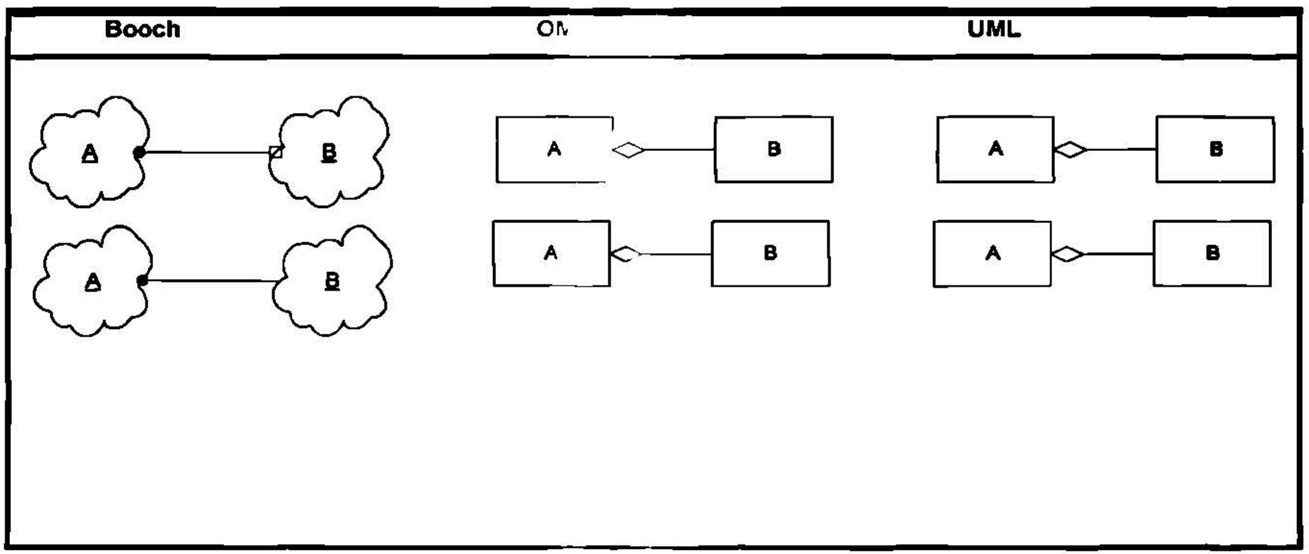
Asociaciones de Clases

En las tres notaciones, una asociación es representada usando una clase conectada a una asociación.



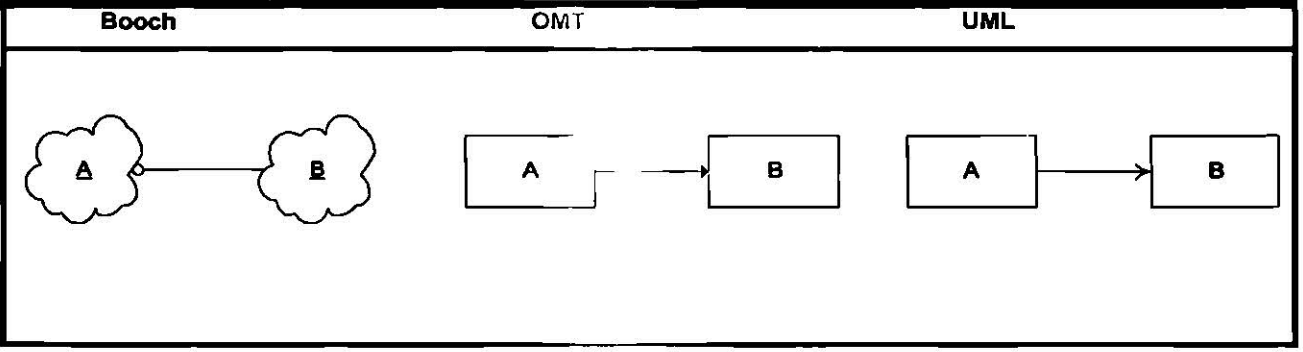
Agregación

No hay una estricta equivalencia entre Booch, OMT y UML. En lo que a agregaciones concierne, Booch se acerca más a la fase de diseño, OMT a la fase de análisis y UML cubre a las dos. La siguiente tabla representa dos de las más comunes situaciones: agregaciones por referencia y agregación por valor ("composición" en UML). Es importante notar que en el caso de OMT, la representación gráfica no distingue el tipo de agregación.



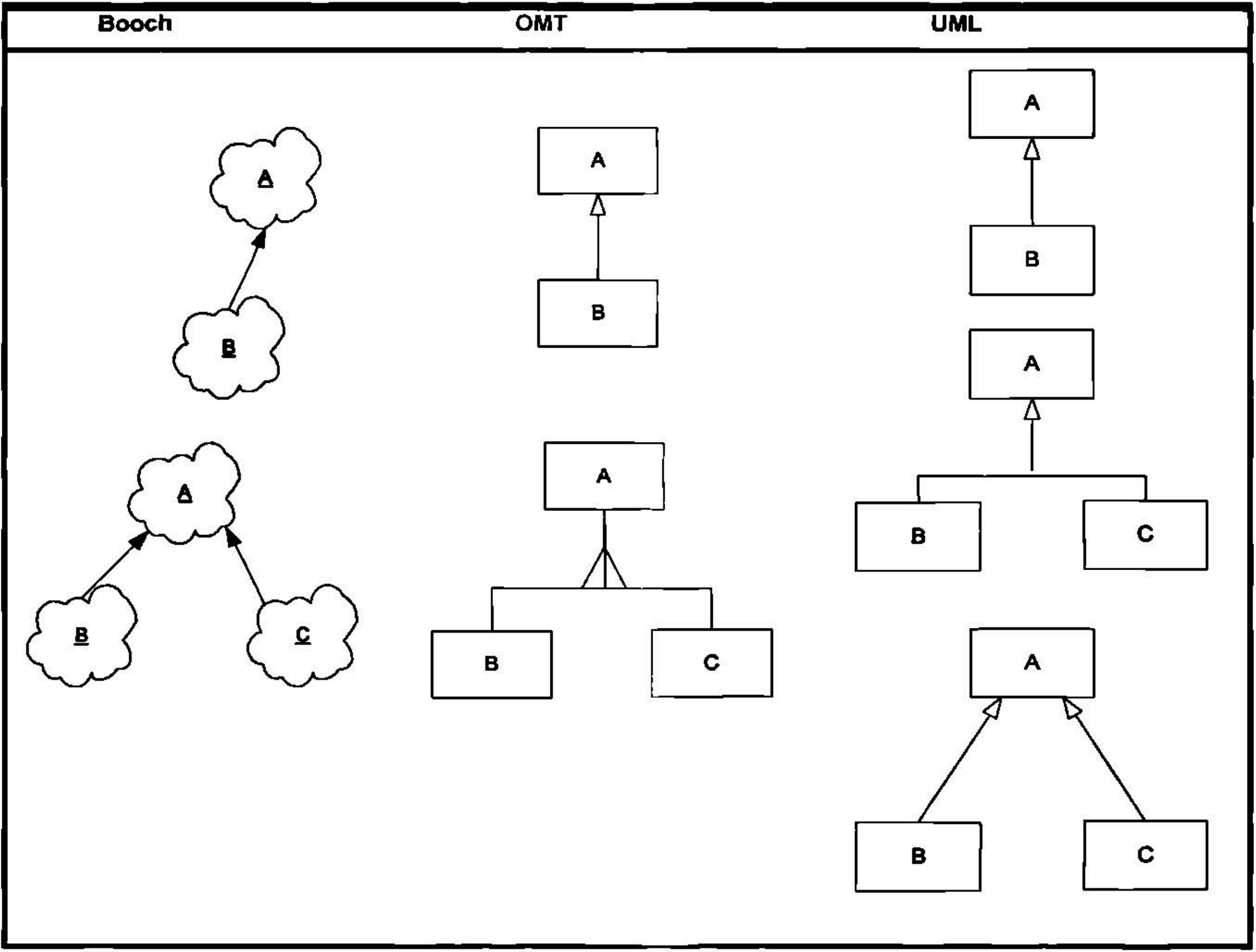
Dependencia

Booch representa una dependencia usando una asociación etiquetada con un pequeño círculo colocado en el lado del cliente. OMT representa una dependencia usando una flecha punteada con la punta rellena. UML usa una flecha punteada con una punta abierta.



Herencia

En las tres notaciones, le herencia es representada usando una flecha que apunta desde la subclase a la superclase.



Generación de Codigo C++

El siguiente código es generado automáticamente por el modelo de UML de Rational Rose. Este es sólo un ejemplo, por lo tanto no muestra todas las capacidades de Rational Rose.

Clase vacía

Modelo	Código
A	(ver código abajo)

```
#indef A_h
#define A_h 1
class A
{
public:
     //## Constructors (generated)
    A();
    A(const A& right);

    //## Destructor (generated)
    ~A();

    //## Assignment Operations (generated)
    const A& opertator = (const A& right);

    //## Equality Operations (generated)
    int operator == (const A& right) const;
    int operator ¡= (const A& right) const;
};
#endif
```

Clase con atributos y operaciones

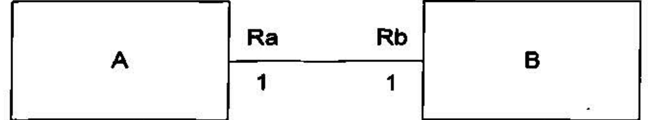
<table>
<tr><th>Modelo</th><th>Código</th></tr>
<tr><td>A
A1 : String
A2 : String
Op1()
Op2()</td><td>

```
Class A
{
public:
      ...
      //## Others Operations (specified)
      void Op1();
      void Op2();

      const String get_A1() const;
      void set_A1(const String value);
      const String get_A2() const;
      void set_A2(const String value);

private:
      String A1;
      String A2;
};

inline const String A::get_A1() const
{
      return A1;
}

inline void  A::set_A1(const String value)
{
      A1 = value;
}

inline const String A::get_A2() const
{
      return A2;
}

inline void  A::set_A2(const String value)
{
      A2 = value;
}
```

</td></tr>
</table>

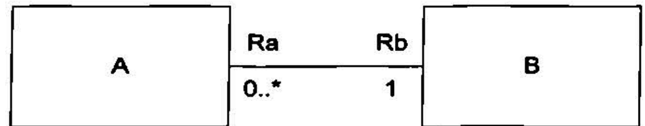
Asociaciones

Asociación 1 a 1



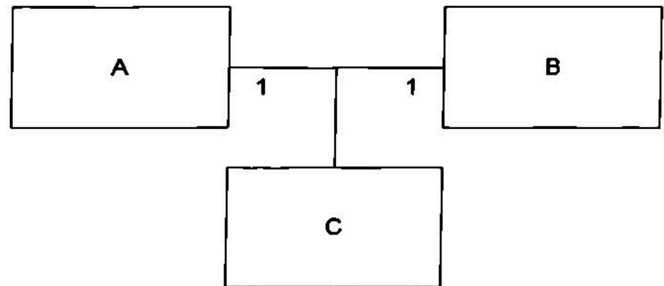

```
Class A
{
      ...

      const B* get_Rb() const;
      void set_Rb(B* const value);

private:
      B* Rb;
};

inline const B* A::get_Rb() const
{
      return Rb;
}

inline void A::set_Rb(B* const value)
{
      Rb = value;
}



Class B
{
      ...

      const A* get_Ra() const;
      void set_Ra(A* const value);

private:
      A* Ra;
};

inline const A* B::get_Ra() const
{
      return Ra;
}

inline void B::set_Ra(A* const value)
{
      Ra = value;
}
```

Asociación N a 1




```
Class B
{
      ...

      const UnboundedSetByReference<A> get_Ra() const;
      void set_Ra(const UnboundedSetByReference<A> value);

private:
      UnboundedSetByReference<A> Ra;
};

inline const UnboundedSetByReference<A> B::get_Ra() const
{
      return Ra;
}

inline void B::set_Ra(const UnboundedSetByReference<A> value)
{
      Ra = value;
}
```

Asociación de Clases

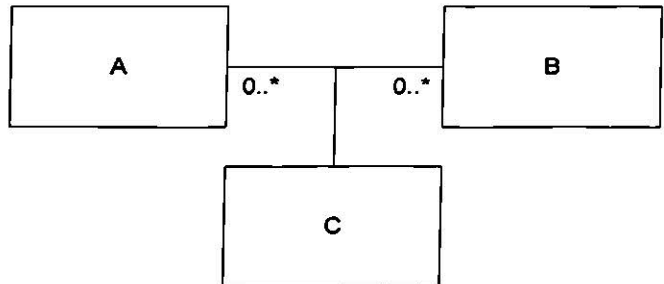



```
class A; class B

class C;
{
      ...
      const B* get_the_B() const;
      void set_the_B(B* const value);

      const A* get_the_A() const;
      void set_the_A(A* const value);

private:
      A* the_A;
      B* the_B;
};
```

```
#include "C.h"

class A;
{
      ...
      const C* get_the_C() const;
      void set_the_C(C* const value);

private:
      C* the_C;
};
```

```
#include "C.h"

class B;
{
      ...
      const C* get_the_C() const;
      void set_the_C(C* const value);

private:
      C* the_C;
};
```

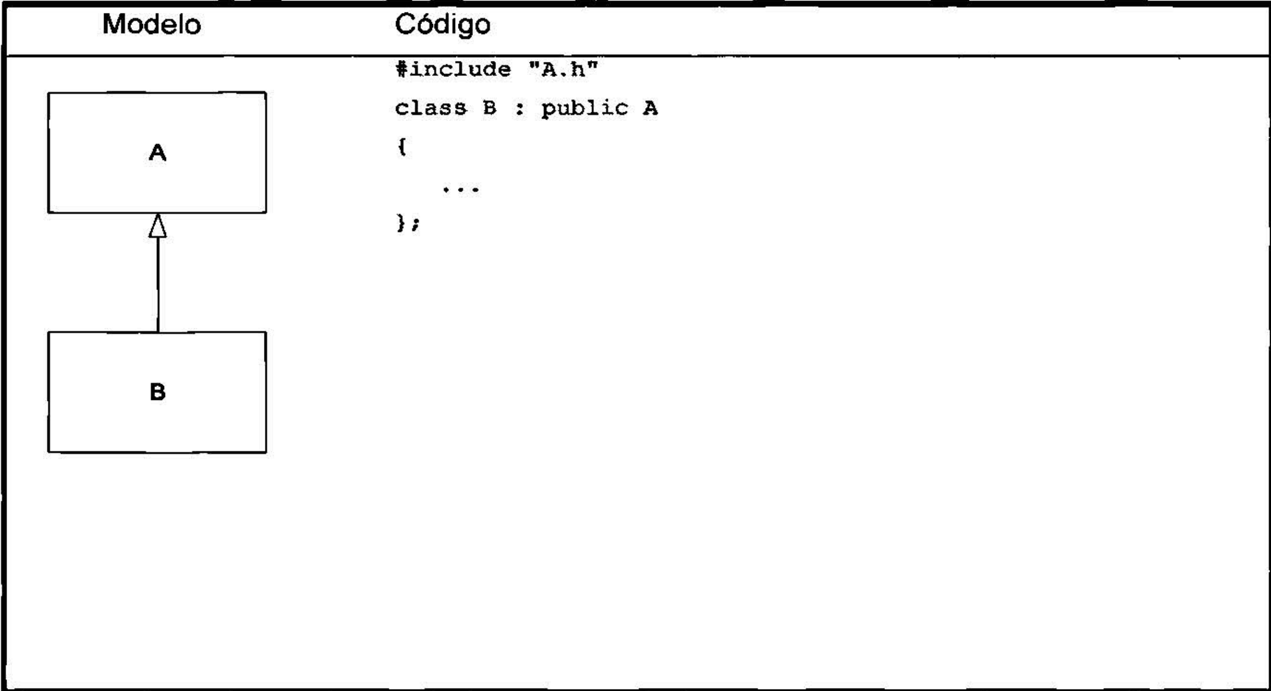
Asociación de clases N a N



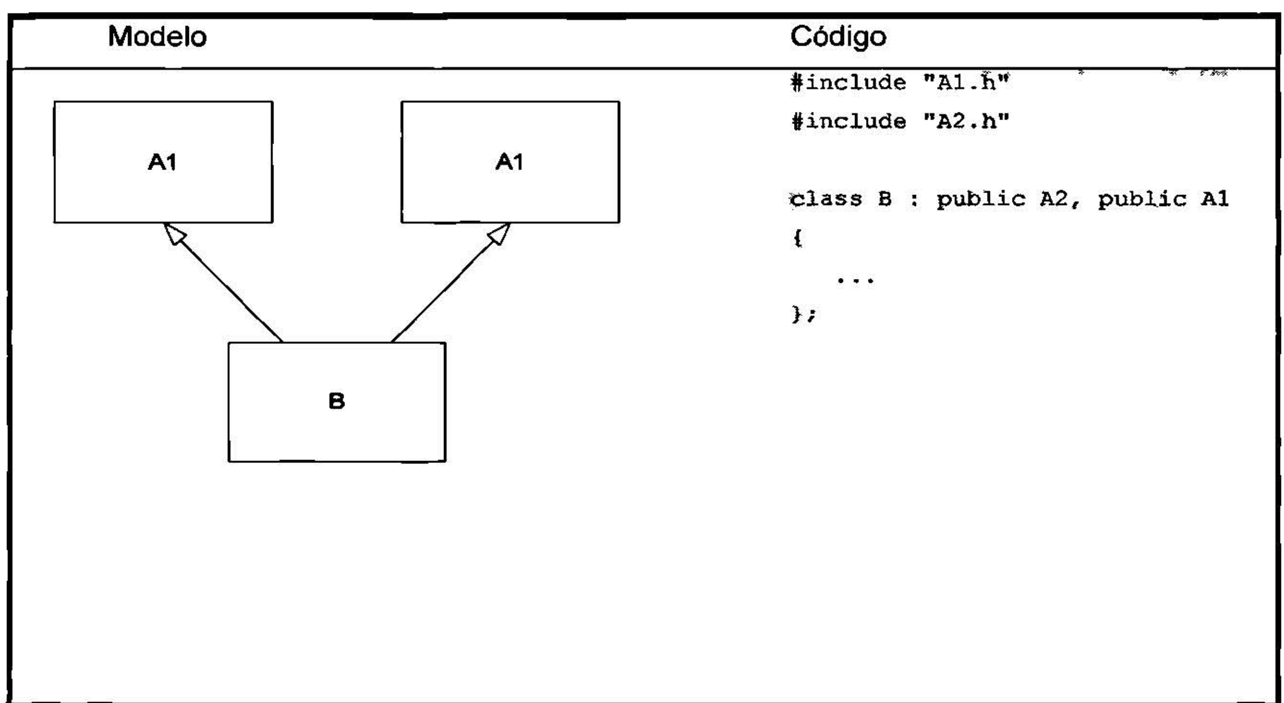

```
Class B
{
      ...

      const UnboundedSetByReference<A> get_the_C() const;
      void set_the_C(const UnboundedSetByReference<A> value);

private:
      UnboundedSetByReference<A> the_C;
};
```

Herencia

Herencia Simple



Herencia Múltiple



Generación de Codigo Java

El siguiente código es generado automáticamente por el modelo de UML de Rational Rose. Este es sólo un ejemplo, por lo tanto no muestra todas las capacidades de Rational Rose.

Clase vacía

Modelo	Código
A	(see code below)

```
public final class A {

    public A() {
        super();
        ...
    }

    protected void finalize() throws Throwable {
        super.finalize();
        ...
    }
    ...
};
```

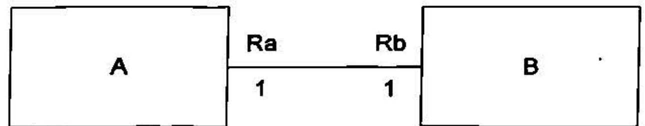
Clase con atributos y operaciones

Modelo	Código
A A1 : String A2 : String Op1() Op2()	(see code below)

```
public final class A {

    private String m_A1;
    private String m_A2;

    public void Op1() {
    ...
    }

    public void Op2() {
    ...
    }
    ...
};
```

Asociaciones

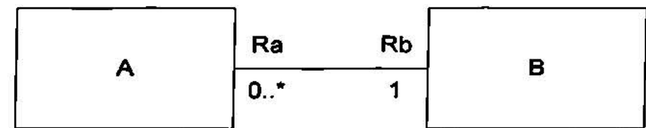
Asociación 1 a 1



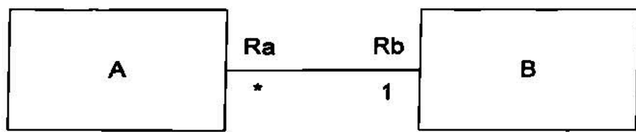

```
public class A {
{
     public B m_Rb;
     ...
}
```

```
public class B {
{
     public A m_Ra;
     ...
}
```

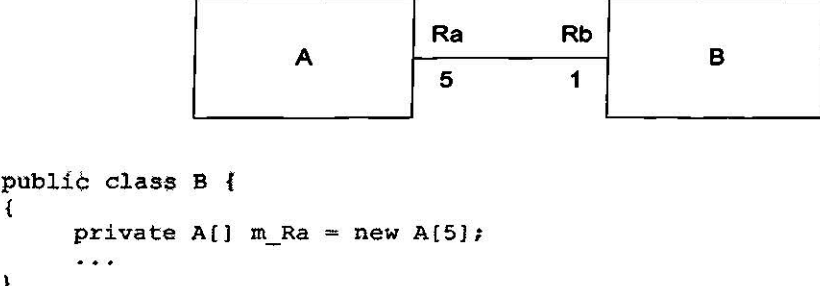
Asociación 1 a N




```
public class B {
{
     public Vector m_Ra = new Vector();
     ...
}
```



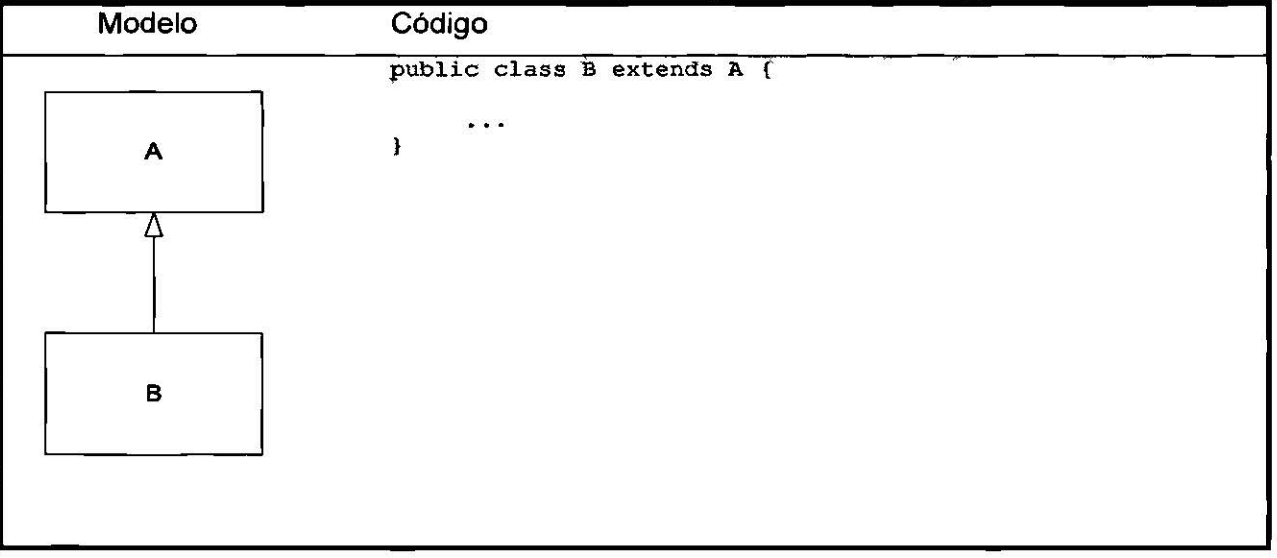

```
public class B {
{
     private Vector m_Ra = new Vector();
     ...
}
```




```
public class B {
{
     private A[] m_Ra = new A[5];
     ...
}
```

Herencia

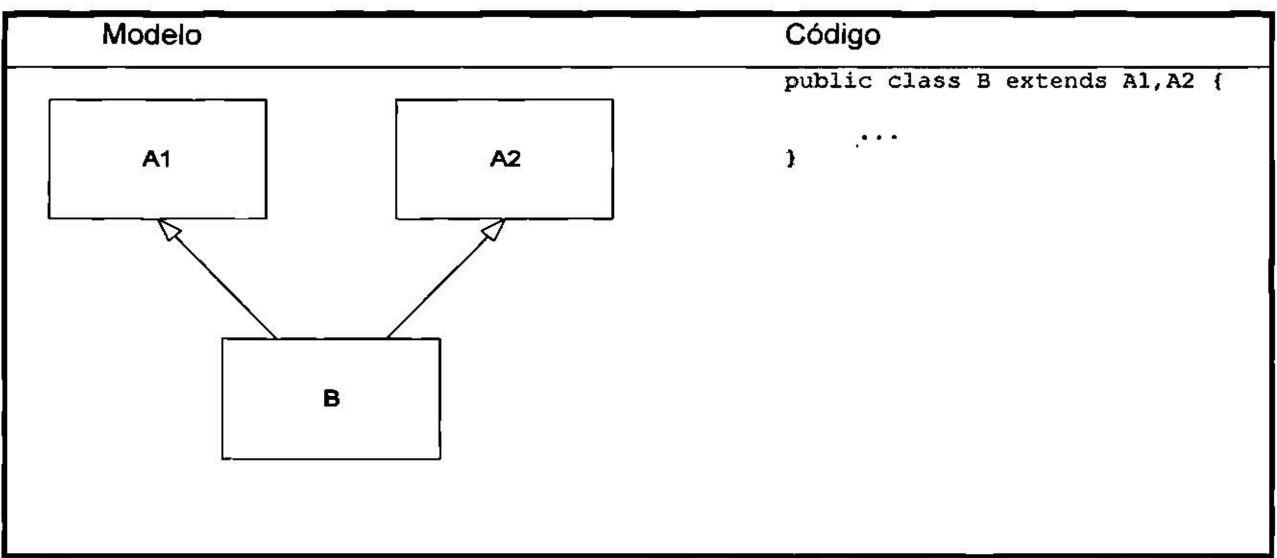
Herencia Simple



Modelo	Código

A ◁— B	`public class B extends A {`
	 `...`
	 `}`

Herencia Múltiple



Generación de Codigo VBasic

El siguiente código es generado automáticamente por el modelo de UML de Rational Rose. Este es sólo un ejemplo, por lo tanto no muestra todas las capacidades de Rational Rose.

Clase vacía

Modelo	Código

A	`Option Base 0`
	
	`Public Sub Class_Initialize()`
	`End Sub`
	
	`Private Sub Class_Terminate()`
	`End Sub`

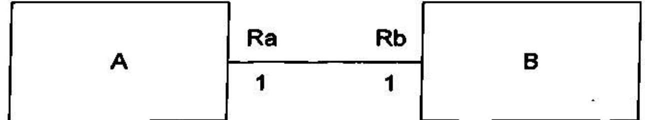
Clase con atributos y operaciones

Modelo	Código

A
A1 : String
A2 : String
Op1()
Op2()
	`Option Base 0`
	
	`Public Sub Class_Initialize()`
	`End Sub`
	
	`Private Sub Class_Terminate()`
	`End Sub`
	
	`Public Sub Op1()`
	`     On Error GoTo Op1Err`
	`     ...`
	`     Exit Sub`
	
	`Op1Err:`
	`     Call RaiseError(RutinaError, "A:Op1 Method")`
	
	`End Sub`
	
	`Public Property Get Op2() As Boolean`
	`On Error GoTo Op1Err`
	`     ...`
	`     Exit Property`
	
	`Op2Err:`
	`     Call RaiseError(RutinaError, "A:Op2 Method")`
	
	`End Property`

Asociaciones

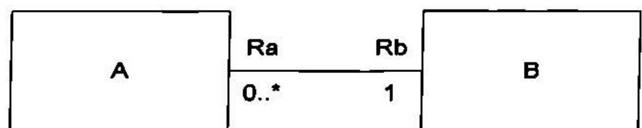
Asociación 1 a 1




```
Option Base 0

Public Rb As B

Public Sub Class_Initialize()
End Sub

Private Sub Class_Terminate()
End Sub
```

```
Option Base 0

Public Ra As A

Public Sub Class_Initialize()
End Sub

Private Sub Class_Terminate()
End Sub
```

Asociación N a 1




```
Option Base 0

Public Rb As B

Public Sub Class_Initialize()
End Sub

Private Sub Class_Terminate()
End Sub
```

```
Option Base 0

Public Ra As Collection

Public Sub Class_Initialize()
End Sub

Private Sub Class_Terminate()
End Sub
```

Herencia

Herencia Simple

Modelo	Código

A
Op1()
△
B
Op2()
	`Option Base 0`
	`Implements A`
	
	`'Objeto de superclase (generado)`
	`Private mAObject As New A`
	
	`Public Sub Class_Initialize()`
	`End Sub`
	
	`Private Sub Class_Terminate()`
	`End Sub`
	
	`Public Sub Op2()`
	`End Sub`
	
	`Private Sub A_Op1()`
	`    mAObject.Op1`
	`End Sub`

Bibliografía

UML Distilled

Applying the Standard Object Modeling Language

Martin Fowler with Kendall Scott

Ed. Addison-Wesley

Instant UML

Pierre-Alain Muller

Ed. Wrox

The Unified Software Development Process

Ivar Jacobson, James Rumbaugh y Grady Booch

Ed. Addison-Wesley

Internet

www.rationalrose.com

www.wrox.com

Libros recomendados

The Unified Modeling Language Reference Manual
Ivar Jacobson, James Rumbaugh y Grady Booch
Ed. Addison-Wesley

The Unified Modeling Language User Guide
Ivar Jacobson, James Rumbaugh y Grady Booch
Ed. Addison-Wesley

Visual Modeling with Rational Rose and UML
Terri Quatroni
Ed. Addison-Wesley

Beginnig Java 2
Ivar Horton
Ed. Wrox

Java al Descubierto
Jamie Jaworski
Ed. Prentice Hall