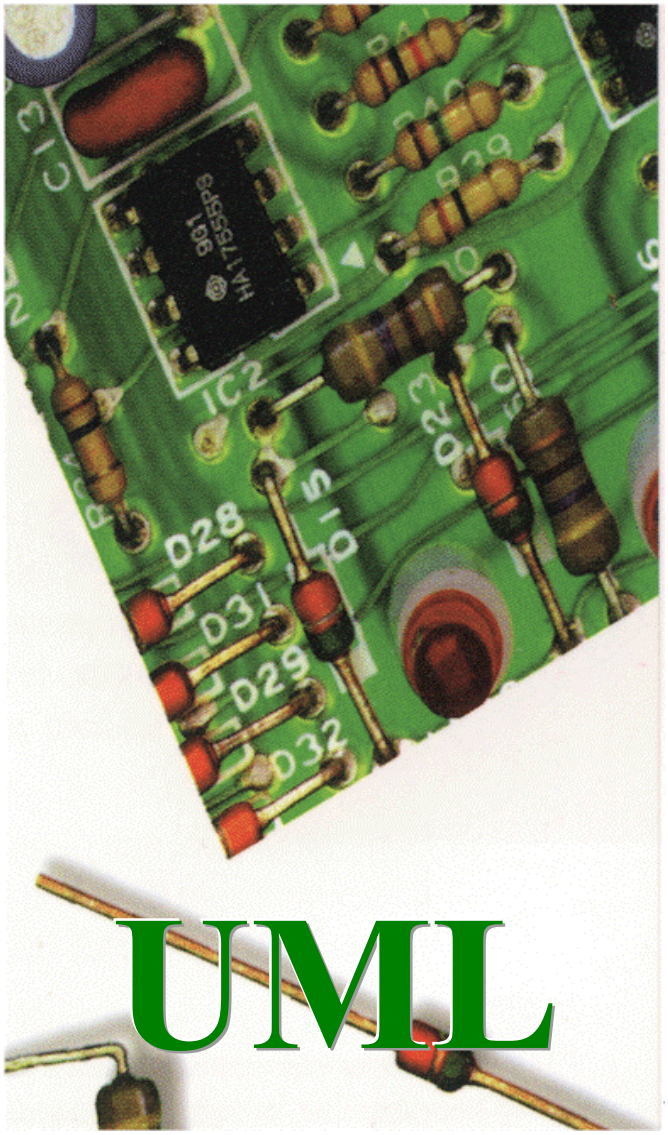
Este texto muestra las distintas técnicas que se necesitan para diseñar aplicaciones informáticas desde la perspectiva de la orientación a objetos, usando lo que se denomina UML (Lenguaje Unificado de Modelado).

Pretende adiestrar en las técnicas de análisis orientadas al objeto así como capacitar en los métodos, notación y símbolos de UML.

Los conceptos se llevan a la práctica con Visual Modeler, la herramienta de Microsoft para el modelado de objetos.

Va dirigido a personas con amplia experiencia en desarrollo de aplicaciones desde la perspectiva de la programación.



DISEÑO ORIENTADO A OBJETOS CON UML

RAÚL ALARCÓN

GRUPO EIDOS

ISBN 84-88457-03-0

Diseño orientado a objetos con UML

Raúl Alarcón

Responsable editorial
Paco Marín (fmarin@eidos.es)

Coordinación de la edición
Antonio Quirós (aquiros@eidos.es)

Autoedición
Magdalena Marín (mmarin@eidos.es)
Raúl Alarcon (ralarcon@eidos.es)

Grupo EIDOS
C/ Téllez 30 Oficina 2
28007-Madrid (España)
Tel: 91 5013234 Fax: 91 (34) 5017824
www.grupoeidos.com/www.eidos.es
www.LaLibreriaDigital.com

Índice

1

Introducción al modelado orientado a objetos

El desarrollo de proyectos software ha sufrido una evolución desde los primeros sistemas de calculo, implementados en grandes computadores simplemente ayudados mediante unas tarjetas perforadas donde los programadores escribían sus algoritmos de control, hasta la revolución de los sistemas de información e Internet. Han existido dos grandes cambios desde aquellos sistemas meramente algorítmicos donde todo el esfuerzo de desarrollo se centraba en la escritura de programas que realizaran algún tipo de calculo. El primero de ellos es la aparición del modelo relacional, un modelo con fuerte base matemática que supuso el desarrollo las bases de datos y propició la aparición de los grandes sistemas de información. El segundo cambio es sobre los lenguajes de programación, la aparición de los *Lenguajes Orientados a Objetos* (aunque los primero lenguajes con características de orientación a objetos aparecieron en la década de los setenta, por ejemplo *Simula 67*) supuso una revolución en la industria software. El problema entonces radicaba en poder sacarle partido a los lenguajes orientados a objetos por lo que aparecieron numerosas metodologías para el diseño orientado objetos, hubo un momento en el que se podía decir que el concepto de orientación a objetos estaba "de moda" y todo era orientado a objetos, cuando realmente lo que ocurría es que las grandes empresas que proporcionaban los compiladores y lenguajes de programación "lavaban la cara" a sus compiladores, sacaban nuevas versiones que adoptaran alguno de los conceptos de orientación a objetos y los vendían como orientados a objetos.

Para poner un poco de orden, sobre todo en lo que respecta a la modelización de sistemas software, aparece UML (Unified Modeling Languaje, *Lenguaje Unificado de Modelado*) que pretende unificar las tres metodologías más difundidas (OMT, Bootch y OOSE) e intentar que la industria software termine su maduración como *Ingeniería* . Y lo consigue en tal manera que lo que UML proporciona son las herramientas necesarias para poder obtener los *planos del software* equivalentes a los que se utilizan en la construcción, la mecánica o la industria aeroespacial. UML abarca todas las fases del

ciclo de vida de un proyecto, soporta diferentes maneras de visualización dependiendo de quién tenga que interpretar *los planos* y en que fase del proyecto se encuentre. Lo que describiéremos en este curso es una *introducción al diseño orientado a objetos* y que solución aporta UML, explicando sus características principales.

Modelado

Para producir software que cumpla su propósito hay que obtener los requisitos del sistema, esto se consigue conociendo de una forma disciplinada a los usuarios y haciéndolos participar de manera activa para que no queden "cabos sueltos". Para conseguir un software de calidad, que sea duradero y fácil de mantener hay que idear una sólida base arquitectónica que sea flexible al cambio. Para desarrollar software rápida y eficientemente, minimizando el trabajo de recodificación y evitando crear miles de líneas de código inútil hay que disponer, además de la gente y las herramientas necesarias, de un enfoque apropiado.

Para conseguir, que a la hora de desarrollar software de manera industrial se obtenga un producto de calidad, es completamente necesario seguir ciertas pautas y no abordar los problemas de manera somera, con el fin de obtener un modelo que represente lo suficientemente bien el problema que hemos de abordar. El modelado es la espina dorsal del desarrollo software de calidad. Se construyen modelos para poder comunicarnos con otros, para explicar el comportamiento del sistema a desarrollar, para comprender, nosotros mismos, mejor ese sistema, para controlar el riesgo y en definitiva para poder atacar problemas que sin el modelado su resolución seria imposible, tanto desde el punto de vista de los desarrolladores (no se pueden cumplir los plazos estimados, no se consigue ajustar los presupuestos...) como desde el punto de vista del cliente, el cual, si finalmente se le entrega el producto del desarrollo, se encontrará con infinidades de problemas, desde que no se cumplen las especificaciones hasta fallos que dejan inutilizado el sistema.

Cuando nos referimos al desarrollo software en el ámbito industrial, no se pretende que la capacidad de modelar se reduzca a empresas que disponen de gran numero de empleados o empresas que han de abordar proyectos eminentemente grandiosos, si no que nos referimos a la capacidad de obtener un producto comercial (sea cual sea su coste o tamaño) que cumpla lo que en la industria se suele denominar como ***calidad total***[1] y que además pueda reportar beneficios a corto o medio plazo, evitando, por ejemplo, implantaciones casi eternas debido a la falta de previsión o al haber abordado los problemas muy a la ligera.

Por todas estas razones es inevitable el uso de modelos. Pero, ¿qué es un modelo?. La respuesta es bien sencilla, ***un modelo es una simplificación de la realidad***. El modelo nos proporciona los planos de un sistema, desde los más generales, que proporcionan una visión general del sistema, hasta los más detallados. En un modelo se han de incluir los elementos que tengan más relevancia y omitir los que no son interesantes para el nivel de abstracción que se ha elegido. A través del modelado conseguimos cuatro objetivos:

- Los modelos nos ayudan a visualizar cómo es o queremos que sea un sistema.
- Los modelos nos permiten especificar la estructura o el comportamiento de un sistema.
- Los modelos nos proporcionan plantillas que nos guían en la construcción de un sistema.

[1] Calidad total se refiere a niveles de calidad tan altos en todas las fases de la vida de un proyecto (ya sea mecánico, industrial, civil...) que no haya que estar modificando en las fases siguientes debido a errores que se debían haber detectado previamente.

- Los modelos documentan las decisiones que hemos adoptado.

En la realidad, no siempre se hace un modelado formal, la probabilidad de que exista un modelado formal para abordar un sistema es inversamente proporcional a la complejidad del mismo, esto es, cuanto más fácil sea un problema, menos tiempo se pasa modelándolo y esto es porque cuando hay de aportar una solución a un problema complejo el uso del modelado nos ayuda a comprenderlo, mientras que cuando tenemos un problema fácil el uso del modelado que hacemos se reduce a representar mentalmente el problema o, como mucho, a escribir unos cuantos garabatos sobre un papel.

Principios básicos del modelado

Existen cuatro principios básicos, estos principios son fruto de la experiencia en todas las ramas de la ingeniería.

a) *La elección de qué modelos se creen influye directamente sobre cómo se acomete el problema*. Hay que seleccionar el modelo adecuado para cada momento y dependiendo de que modelo se elija se obtendrán diferentes beneficios y diferentes costes. En la industria software se ha comprobado que un modelado orientado a objetos proporciona unas arquitecturas más flexibles y readaptables que otros por ejemplo orientados a la funcionalidad o a los datos.

b) Todo modelo puede ser expresado a diferentes niveles de precisión. Esto es, es necesario poder seleccionar el nivel de detalle que se desea ya que en diferentes partes de un proyecto y en diferentes etapas se tendrán unas determinadas necesidades.

c) Los mejores modelos están ligados a la realidad. Lo principal es tener modelos que nos permitan representar la realidad lo más claramente posible, pero no sólo esto, tenemos que saber, exactamente cuando se apartan de la realidad para no caer en la ocultación de ningún detalle importante.

d) Un único modelo no es suficiente. Cualquier sistema que no sea trivial se afronta mejor desde pequeños modelos casi independientes, que los podamos construir y estudiar independientemente y que nos representen las partes más diferenciadas del sistema y sus interrelaciones.

Orientación a Objetos

La programación estructurada tradicional se basa fundamentalmente en la ecuación de Wirth:

Algoritmos + Estructuras de Datos = Programas

Esta ecuación significa que en la programación estructurada u orientada a procedimientos los datos y el código se trata por separado y lo único se realiza son funciones o procedimientos que tratan esos datos y los van pasando de unos a otros hasta que se obtiene el resultado que se desea.

La *Programación Orientada a Objetos*, *POO* (OOP, Object Oriented Programming, en ingles), es una técnica de programación cuyo soporte fundamental es el ***objeto***. Un objeto es una extensión de un *Tipo Abstracto de Datos (TAD),* concepto ampliamente utilizado desde la década de los setenta. Un TAD es un tipo definido por el usuario, que encapsula un conjunto de datos y las operaciones sobre estos datos.

A la hora de definir TAD's (u objetos) se usa un concepto que nos ayuda a representar la realidad mediante modelos informáticos, la ***abstracción,*** que es un proceso mental por el que se evitan los detalles para centrarse en las cosas más genéricas de manera que se facilite su comprensión. De hecho la abstracción no sólo se utiliza en la informática, un arquitecto al que le han encargado realizar los planos de un edificio no comenzará por diseñar los planos con máximo nivel de detalle, sino que comenzará a realzar ciertos esbozos en un papel para posteriormente ir refinando. Por supuesto que cuando está realizando los esbozos no se preocupa de por dónde van a ir las líneas eléctricas ni las tuberías de saneamiento, abstrae esos detalles para atacarlos posteriormente cuando tenga clara la estructura del edificio.

La diferencia entre el concepto de TAD y el de ***objeto*** radica en que además del proceso de abstracción que se utiliza para su definición, existen otros dos con los que se forma el núcleo principal de la programación orientada a objetos, estos son la *herencia* y el *polimorfismo*.

Ventajas de la orientación a objetos

Las ventajas más importantes de la programación orientada a objetos son las siguientes:

- *Mantenibilidad* (facilidad de mantenimiento). Los programas que se diseñan utilizando el concepto de orientación a objetos son más fáciles de leer y comprender y el control de la complejidad del programa se consigue gracias a la ocultación de la información que permite dejar visibles sólo los detalles más relevantes.

- *Modificabilidad* (facilidad para modificar los programas). Se pueden realizar añadidos o supresiones a programas simplemente añadiendo, suprimiendo o modificando objetos.

- *Resusabilidad*. Los objetos, si han sido correctamente diseñados, se pueden usar numerosas veces y en distintos proyectos.

- *Fiabilidad*. Los programas orientados a objetos suelen ser más fiables ya que se basan en el uso de objetos ya definidos que están ampliamente testados.

Estas ventajas son directas a los programadores. Estos, se podría decir, que son los ejecutores de un determinado proyecto software. Pero la orientación a objetos no sólo reporta beneficios a los programadores. En las etapas de análisis, previas a la codificación, el utilizar un modelado orientado a objetos reporta grandes beneficios ya estas mismas ventajas son aplicables a todas las fases del ciclo de vida de un proyecto software.

La tendencia actual es a tratar temas conceptuales de primer plano (o sea, en las fases de análisis) y no temas finales de implementación. Los fallos durante la etapa de implementación son más difíciles de corregir y más costosos que si se dan en las etapas previas. El modelado orientado a objetos tiende al refinamiento sucesivo de manera que se llega a la etapa de implementación con un diseño lo suficientemente explicito para que no existan casos inesperados y todo independientemente del lenguaje de programación (salvo en etapas muy próximas a la implementación donde no hay más remedio que contar con el soporte que se recibe del lenguaje elegido). El desarrollo orientado a objetos es más una manera de pensar y no una técnica de programación.

Conceptos básicos de la orientación a objeto

Como ya hemos dicho la orientación a objetos se basa en conceptos como clase, objeto, herencia y polimorfismo, pero también en otros muchos. En esta sección se intenta, sin entrar en detalles, realizar

una breve descripción de los conceptos más importantes que existen en el modelado orientado a objetos. Estos conceptos serán explicados y ampliados posteriormente desde la perspectiva de UML.

- **Clase:** Es una descripción de un conjunto de objetos similares. Por ejemplo la clase *Coches.* Una clase contiene los atributos y las operaciones sobre esos atributos que hacen que una clase tenga la entidad que se desea.

- **Objeto:** Un objeto es una cosa, generalmente extraída del vocabulario del espacio del problema o del espacio de la solución. Todo objeto tiene un nombre (se le puede identificar), un estado (generalmente hay algunos datos asociados a él) y un comportamiento (se le pueden hacer cosas a objeto y él puede hacer cosas a otros objetos). Un objeto de la clase *Coches* puede ser un *Ford Mustang*.

- **Atributo:** Es una característica concreta de una clase. Por ejemplo atributos de la clase *Coches* pueden ser el *Color*, el *Numero de Puertas...*

- **Método:** Es una operación concreta de una determinada clase. Por ejemplo de la clase *Coches* podríamos tener un método *arrancar()* que lo que hace es poner en marcha el coche.

- **Instancia:** Es una manifestación concreta de una clase (un objeto con valores concretos). También se le suele llamar *ocurrencia.* Por ejemplo una instancia de la clase *Coches* puede ser: Un Ford Mustang, de color Gris con 3 puertas

- **Herencia:** Es un mecanismo mediante el cual se puede crear una nueva clase partiendo de una existente, se dice entonces que la nueva clase hereda las características de la case existentes aunque se le puede añadir más capacidades (añadiendo datos o capacidades) o modificar las que tiene. Por ejemplo supongamos que tenemos la *VehiculosDeMotor*. En esta clase tenemos los siguientes atributos: *Cilindrada y Numero de Ruedas*, y el método *acelerar()*. Mediante el mecanismo de herencia podemos definir la clase *Coches* y la clase *Motos.* Estas dos clases heredan los atributos *Cilindrada* y *Numero de Ruedas* de la clase *VehículosDeMotor* pero a su vez tendrán atributos propios (como hemos dicho antes el *Numero de Puertas* es un atributo propio de la clase *Coches* que no tienen sentido en la clase *Motos*). Se puede decir que *Coches* extiende la clase *VehículosDeMotor*, o que *VehículosDeMotor* es una ***generalización*** de las clases *Coches* y *Motos.*

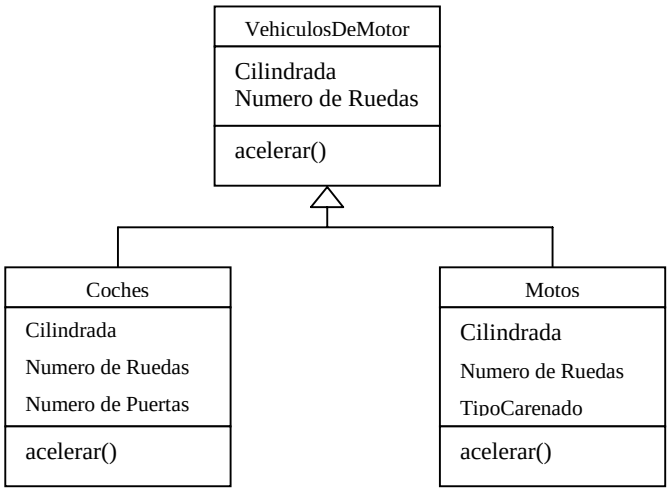


Figura 1. Ejemplo de Herencia

- **Polimorfismo:** Hace referencia a la posibilidad de que dos métodos implementen distintas acciones, aun teniendo el mismo nombre, dependiendo del objeto que lo ejecuta o de los parámetros que recibe. En el ejemplo anterior teníamos dos objetos que heredaban el método *acelerar()* de la clase *VehiculosDeMotor*. De hecho en clase *VehiculosDeMotor* al ser general no tiene sentido que tenga una implementación concreta de este método. Sin embargo, en las clases Coches y Motos si que hay una implementación clara y distinta del método *acelerar(),* lo podemos ver en el código fuente 1 y 2. De este modo podríamos tener un objeto *VehículosDeMotor,* llamado *vdm,* en el que residiera un objeto Coche. Si realizáramos la llamada *vdm.acelerar()* sabría exactamente que ha de ejecutar el método *Coches::acelerar().*

```
Coches::acelerar(){
Pisar más el pedal derecho
}
```

Código fuente 1. Posible implementación para coches

```
Motos::acelerar(){
Girar más el puño derecho
}
```

Código fuente 2. Implementación para motos

2

Introducción al lenguaje unificado de modelado, UML

UML es un lenguaje estándar que sirve para escribir los *planos del software*, puede utilizarse para visualizar, especificar, construir y documentar todos los artefactos que componen un sistema con gran cantidad de software. UML puede usarse para modelar desde sistemas de información hasta aplicaciones distribuidas basadas en Web, pasando por sistemas empotrados de tiempo real. UML es solamente un lenguaje por lo que es sólo una parte de un método de desarrollo software, es independiente del proceso aunque para que sea optimo debe usarse en un proceso dirigido por casos de uso, centrado en la arquitectura, iterativo e incremental.

UML es un lenguaje por que proporciona un vocabulario y las reglas para utilizarlo, además es un lenguaje de modelado lo que significa que el vocabulario y las reglas se utilizan para la representación conceptual y física del sistema.

UML es un lenguaje que nos ayuda a interpretar grandes sistemas mediante gráficos o mediante texto obteniendo modelos explícitos que ayudan a la comunicación durante el desarrollo ya que al ser estándar, los modelos podrán ser interpretados por personas que no participaron en su diseño (e incluso por herramientas) sin ninguna ambigüedad. En este contexto, UML sirve para *especificar*, modelos concretos, no ambiguos y completos.

Debido a su estandarización y su definición completa no ambigua, y aunque no sea un lenguaje de programación, UML se puede conectar de manera directa a lenguajes de programación como Java, C++ o Visual Basic, esta correspondencia permite lo que se denomina como ingeniería directa (obtener el código fuente partiendo de los modelos) pero además es posible reconstruir un modelo en UML partiendo de la implementación, o sea, la ingeniería inversa.

UML proporciona la capacidad de modelar actividades de planificación de proyectos y de sus versiones, expresar requisitos y las pruebas sobre el sistema, representar todos sus detalles así como la propia arquitectura. Mediante estas capacidades se obtiene una documentación que es valida durante todo el ciclo de vida de un proyecto.

Vista general de UML

Para conocer la estructura de UML, en la figura 3 vemos una vista general de todos sus componentes, esto hará que nos resulte más fácil la comprensión de cada uno de ellos.

El lenguaje UML se compone de tres elementos básicos, los bloques de construcción, las reglas y algunos mecanismos comunes. Estos elementos interaccionan entre sí para dar a UML el carácter de completitud y no-ambigüedad que antes comentábamos.

Los **bloques de construcción** se dividen en tres partes: **Elementos**, que son las abstracciones de primer nivel, **Relaciones**, que unen a los elementos entre sí, y los **Diagramas**, que son agrupaciones interesantes de elementos.

Existen cuatro tipos de elementos en UML, dependiendo del uso que se haga de ellos: *elementos estructurales, elementos de comportamiento, elementos de agrupación y elementos de anotación.*

Las relaciones, a su vez se dividen para abarcar las posibles interacciones entre elementos que se nos pueden presentar a la hora de modelar usando UML, estas son: *relaciones de dependencia, relaciones de asociación, relaciones de generalización y relaciones de realización.*

Se utilizan diferentes diagramas dependiendo de qué, nos interese representar en cada momento, para dar diferentes perspectivas de un mismo problema, para ajustar el nivel de detalle..., por esta razón UML soporta un gran numero de diagramas diferentes aunque, en la practica, sólo se utilicen un pequeño número de combinaciones.

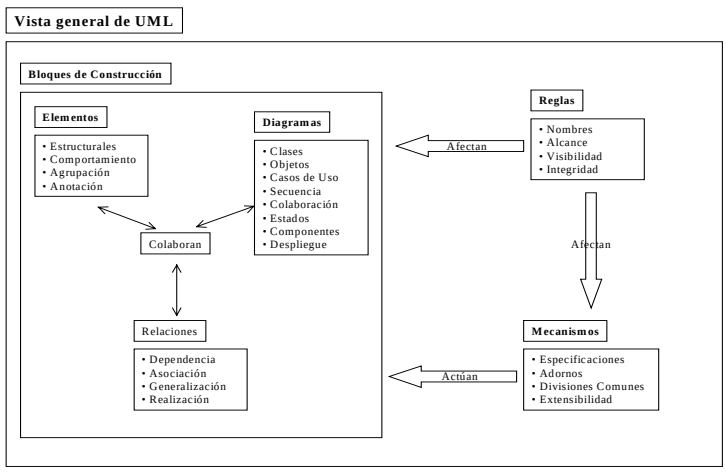


Figura 2. Vista general de los elementos de UML –

UML proporciona un conjunto de reglas que dictan las pautas a la hora de realizar asociaciones entre objetos para poder obtener modelos bien formados, estas son reglas semánticas que afectan a los **nombres**, al **alcance** de dichos nombres, a la **visibilidad** de estos nombres por otros, a la **integridad** de unos elementos con otros y a la **ejecución**, o sea la vista dinámica del sistema.

UML proporciona una serie de mecanismos comunes que sirven para que cada persona o entidad adapte el lenguaje a sus necesidades, pero dentro de un marco ordenado y siguiendo unas ciertas reglas para que en el trasfondo de la adaptación no se pierda la semántica propia de UML. Dentro de estos mecanismos están las **especificaciones**, que proporcionan la explicación textual de la sintaxis y semántica de los bloques de construcción. Otro mecanismo es el de los **adornos** que sirven para conferir a los modelos de más semántica, los adornos son elementos secundarios ya que proporcionan más nivel de detalle, que quizá en un primer momento no sea conveniente descubrir. Las **divisiones comunes** permiten que los modelos se dividan al menos en un par de formas diferentes para facilitar la comprensión desde distintos puntos de vista, en primer lugar tenemos la división entre clase y objeto (clase es una abstracción y objeto es una manifestación de esa abstracción), en segundo lugar tenemos la división interfaz / implementación donde la interfaz presenta un contrato (algo que se va a cumplir de una determinada manera) mientras que la implementación es la manera en que se cumple dicho contrato. Por ultimo, los **mecanismos de extensibilidad** que UML proporciona sirven para evitar posibles problemas que puedan surgir debido a la necesidad de poder representar ciertos matices, por esta razón UML incluye los *estereotipos*, para poder extender el vocabulario con nuevos bloques de construcción, los *valores etiquetados*, para extender las propiedades un bloque, y las *restricciones*, para extender la semántica.

De esta manera UML es un lenguaje estándar *"abierto-cerrado"* siendo posible extender el lenguaje de manera controlada.

Bloques de construcción de UML

A continuación se van a describir todos los elementos que componen los bloques estructurales de UML, así como su notación, para que nos sirva de introducción y se vaya generando un esquema conceptual sobre UML. En temas sucesivos se tratará con más profundidad cada uno de los bloques.

Elementos Estructurales

Los elementos estructurales en UML, es su mayoría, son las partes estáticas del modelo y representan cosas que son conceptuales o materiales.

Clases

Una clase es una descripción de un conjunto de objetos que comparten los mismos atributos, operaciones, relaciones y semántica. Una clase implementa una o más interfaces. Gráficamente se representa como un rectángulo que incluye su nombre, sus atributos y sus operaciones (figura 3).

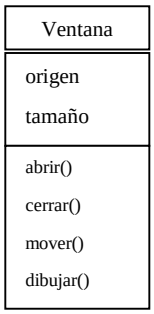


Figura 3. Clases

Interfaz

Una interfaz es una colección de operaciones que especifican un servicio de una determinada clase o componente. Una interfaz describe el comportamiento visible externamente de ese elemento, puede mostrar el comportamiento completo o sólo una parte del mismo. Una interfaz describe un conjunto de especificaciones de operaciones (o sea su signatura) pero nunca su implementación. Se representa con un circulo, como podemos ver en la figura 4, y rara vez se encuentra aislada sino que más bien conectada a la clase o componente que realiza.

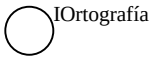


Figura 4. Interfaz

Colaboración

Define una interacción y es una sociedad de roles y otros elementos que colaboran para proporcionar un comportamiento cooperativo mayor que la suma de los comportamientos de sus elementos. Las colaboraciones tienen una dimensión tanto estructural como de comportamiento. Una misma clase puede participar en diferentes colaboraciones. Las colaboraciones representan la implementación de patrones que forman un sistema. Se representa mediante una elipse con borde discontinuo, como en la figura 5.

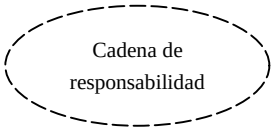


Figura 5. Colaboración

Casos de Uso

Un caso de uso es la descripción de un conjunto de acciones que un sistema ejecuta y que produce un determinado resultado que es de interés para un actor particular. Un caso de uso se utiliza para organizar los aspectos del comportamiento en un modelo. Un caso de uso es realizado por una colaboración. Se representa como en la figura 6, una elipse con borde continuo.

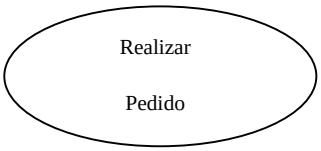


Figura 6. Casos de Uso

Clase Activa

Es una clase cuyos objetos tienen uno o más procesos o hilos de ejecución por lo y tanto pueden dar lugar a actividades de control. Una clase activa es igual que una clase, excepto que sus objetos representan elementos cuyo comportamiento es concurrente con otros elementos. Se representa igual que una clase (figura 3), pero con líneas más gruesas (figura 7).

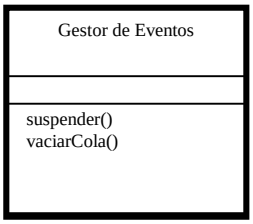


Figura 7. Clases Activas

Componentes

Un componente es una parte física y reemplazable de un sistema que conforma con un conjunto de interfaces y proporciona la implementación de dicho conjunto. Un componente representa típicamente el empaquetamiento físico de diferentes elementos lógicos, como clases, interfaces y colaboraciones.

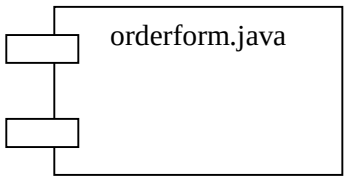


Figura 8. Componentes

Nodos

Un nodo es un elemento físico que existe en tiempo de ejecución y representa un recurso computacional que, por lo general, dispone de algo de memoria y, con frecuencia, de capacidad de procesamiento. Un conjunto de componentes puede residir en un nodo.

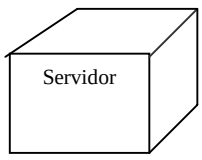


Figura 9. Nodos

Estos siete elementos vistos son los elementos estructurales básico que se pueden incluir en un modelo UML. Existen variaciones sobre estos elementos básicos, tales como actores, señales, utilidades (tipos de clases), procesos e hilos (tipos de clases activas) y aplicaciones, documentos, archivos, bibliotecas, páginas y tablas (tipos de componentes).

Elementos de comportamiento

Los elementos de comportamiento son las partes dinámicas de un modelo. Se podría decir que son los verbos de un modelo y representan el comportamiento en el tiempo y en el espacio. Los principales elementos son los dos que siguen.

Interacción

Es un comportamiento que comprende un conjunto de mensajes intercambiados entre un conjunto de objetos, dentro de un contexto particular para conseguir un propósito específico. Una interacción involucra otros muchos elementos, incluyendo mensajes, secuencias de acción (comportamiento invocado por un objeto) y enlaces (conexiones entre objetos). La representación de un mensaje es una flecha dirigida que normalmente con el nombre de la operación.

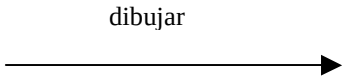


Figura 10. Mensajes

Maquinas de estados

Es un comportamiento que especifica las secuencias de estados por las que van pasando los objetos o las interacciones durante su vida en respuesta a eventos, junto con las respuestas a esos eventos. Una maquina de estados involucra otros elementos como son estados, transiciones (flujo de un estado a otro), eventos (que disparan una transición) y actividades (respuesta de una transición)

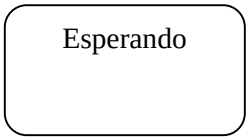


Figura 11. Estados

Elementos de agrupación

Forman la parte organizativa de los modelos UML. El principal elemento de agrupación es el **paquete**, que es un mecanismo de propósito general para organizar elementos en grupos. Los elementos estructurales, los elementos de comportamiento, incluso los propios elementos de agrupación se pueden incluir en un paquete.

Un paquete es puramente conceptual (sólo existe en tiempo de desarrollo). Gráficamente se representa como una carpeta conteniendo normalmente su nombre y, a veces, su contenido.

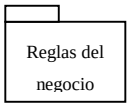


Figura 12. Paquetes

Elementos de anotación

Los elementos de anotación son las partes explicativas de los modelos UML. Son comentarios que se pueden aplicar para describir, clasificar y hacer observaciones sobre cualquier elemento de un modelo. El tipo principal de anotación es la **nota** que simplemente es un símbolo para mostrar restricciones y comentarios junto a un elemento o un conjunto de elementos.

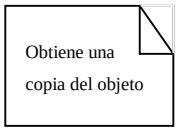


Figura 13. Notas

Relaciones

Existen cuatro tipos de relaciones entre los elementos de un modelo UML. *Dependencia*, *asociación*, *generalización* y *realización*, estas se describen a continuación:

Dependencia

Es una relación semántica entre dos elementos en la cual un cambio a un elemento (el elemento independiente) puede afectar a la semántica del otro elemento (elemento dependiente). Se representa como una línea discontinua (figura 14), posiblemente dirigida, que a veces incluye una etiqueta.

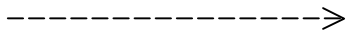


Figura 14. Dependencias

Asociación

Es una relación estructural que describe un conjunto de enlaces, los cuales son conexiones entre objetos. La agregación es un tipo especial de asociación y representa una relación estructural entre un todo y sus partes. La asociación se representa con una línea continua, posiblemente dirigida, que a veces incluye una etiqueta. A menudo se incluyen otros adornos para indicar la multiplicidad y roles de los objetos involucrados, como podemos ver en la figura 15.

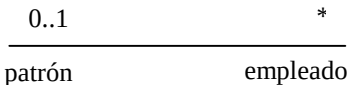


Figura 15. Asociaciones

Generalización

Es una relación de especialización / generalización en la cual los objetos del elemento especializado (el hijo) pueden sustituir a los objetos del elemento general (el padre). De esta forma, el hijo comparte la estructura y el comportamiento del padre. Gráficamente, la generalización se representa con una línea con punta de flecha vacía (figura 16).

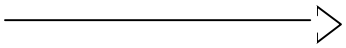


Figura 16. Generalización

Realización

Es una relación semántica entre clasificadores, donde un clasificador especifica un contrato que otro clasificador garantiza que cumplirá. Se pueden encontrar relaciones de realización en dos sitios: entre interfaces y las clases y componentes que las realizan, y entre los casos de uso y las colaboraciones que los realizan. La realización se representa como una mezcla entre la generalización (figura 16) y la dependencia (figura 14), esto es, una línea discontinua con una punta de flecha vacía (figura 17).

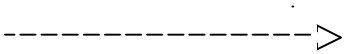


Figura 17. Realización.

Diagramas

Los diagramas se utilizan para representar diferentes perspectivas de un sistema de forma que un diagrama es una proyección del mismo. UML proporciona un amplio conjunto de diagramas que normalmente se usan en pequeños subconjuntos para poder representar las cinco vistas principales de la arquitectura de un sistema.

Diagramas de Clases

Muestran un conjunto de clases, interfaces y colaboraciones, así como sus relaciones. Estos diagramas son los más comunes en el modelado de sistemas orientados a objetos y cubren la vista de diseño estática o la vista de procesos estática (sí incluyen clases activas).

Diagramas de Objetos

Muestran un conjunto de objetos y sus relaciones, son como fotos instantáneas de los diagramas de clases y cubren la vista de diseño estática o la vista de procesos estática desde la perspectiva de casos reales o prototípicos.

Diagramas de Casos de Usos

Muestran un conjunto de casos de uso y actores (tipo especial de clases) y sus relaciones. Cubren la vista estática de los casos de uso y son especialmente importantes para el modelado y organización del comportamiento.

Diagramas de Secuencia y de Colaboración

Tanto los diagramas de secuencia como los diagramas de colaboración son un tipo de diagramas de interacción. Constan de un conjunto de objetos y sus relaciones, incluyendo los mensajes que se pueden enviar unos objetos a otros. Cubren la vista dinámica del sistema. Los diagramas de secuencia enfatizan el ordenamiento temporal de los mensajes mientras que los diagramas de colaboración muestran la organización estructural de los objetos que envían y reciben mensajes. Los diagramas de secuencia se pueden convertir en diagramas de colaboración sin perdida de información, lo mismo ocurren en sentido opuesto.

Diagramas de Estados

Muestran una maquina de estados compuesta por estados, transiciones, eventos y actividades. Estos diagramas cubren la vista dinámica de un sistema y son muy importantes a la hora de modelar el comportamiento de una interfaz, clase o colaboración.

Diagramas de Actividades

Son un tipo especial de diagramas de estados que se centra en mostrar el flujo de actividades dentro de un sistema. Los diagramas de actividades cubren la parte dinámica de un sistema y se utilizan para modelar el funcionamiento de un sistema resaltando el flujo de control entre objetos.

Diagramas de Componentes

Muestra la organización y las dependencias entre un conjunto de componentes. Cubren la vista de la implementación estática y se relacionan con los diagramas de clases ya que en un componente suele tener una o más clases, interfaces o colaboraciones

Diagramas de Despliegue

Representan la configuración de los nodos de procesamiento en tiempo de ejecución y los componentes que residen en ellos. Muestran la vista de despliegue estática de una arquitectura y se relacionan con los componentes ya que, por lo común, los nodos contienen uno o más componentes.

Arquitectura

El desarrollo de un sistema con gran cantidad de software requiere que este sea visto desde diferentes perspectivas. Diferentes usuarios (usuario final, analistas, desarrolladores, integradores, jefes de proyecto...) siguen diferentes actividades en diferentes momentos del ciclo de vida del proyecto, lo que da lugar a las diferentes vistas del proyecto, dependiendo de qué interese más en cada instante de tiempo.

La arquitectura es el conjunto de decisiones significativas sobre:

- La organización del sistema
- Selección de elementos estructurales y sus interfaces a través de los cuales se constituye el sistema.
- El Comportamiento, como se especifica las colaboraciones entre esos componentes.
- Composición de los elementos estructurales y de comportamiento en subsistemas progresivamente más grandes.
- El estilo arquitectónico que guía esta organización: elementos estáticos y dinámicos y sus interfaces, sus colaboraciones y su composición.

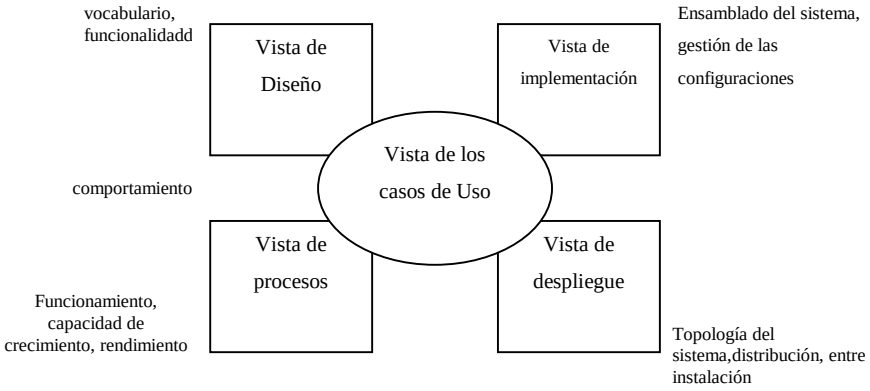


Figura 18. Modelado de la arquitectura de un sistema

La una arquitectura que no debe centrarse únicamente en la estructura y en el comportamiento, sino que abarque temas como el uso, funcionalidad, rendimiento, capacidad de adaptación, reutilización, capacidad para ser comprendida, restricciones, compromisos entre alternativas, así como aspectos estéticos. Para ello se sugiere una arquitectura que permita describir mejor los sistemas desde diferentes vistas, figura 18, donde cada una de ellas es una proyección de la organización y la estructura centrada en un aspecto particular del sistema.

La *vista de casos de uso* comprende la descripción del comportamiento del sistema tal y como es percibido por los usuarios finales, analistas y encargados de las pruebas y se utilizan los diagramas de casos de uso para capturar los aspectos estáticos mientras que los dinámicos son representados por diagramas de interacción, estados y actividades.

La *vista de diseño* comprende las clases, interfaces y colaboraciones que forman el vocabulario del problema y de la solución. Esta vista soporta principalmente los requisitos funcionales del sistema, o sea, los servicios que el sistema debe proporcionar. Los aspectos estáticos se representan mediante diagramas de clases y objetos y los aspectos dinámicos con diagramas de interacción, estados y actividades.

La *vista de procesos* comprende los hilos y procesos que forman mecanismos de sincronización y concurrencia del sistema cubriendo el funcionamiento, capacidad de crecimiento y el rendimiento del sistema. Con UML, los aspectos estáticos y dinámicos se representan igual que en la vista de diseño, pero con el énfasis que aportan las clases activas, las cuales representan los procesos y los hilos.

La *Vista de implementación* comprende los componentes y los archivos que un sistema utiliza para ensamblar y hacer disponible el sistema físico. Se ocupa principalmente de la gestión de configuraciones de las distintas versiones del sistema. Los aspectos estáticos se capturan con los diagramas de componentes y los aspectos dinámicos con los diagramas de interacción, estados y actividades.

La *vista de despliegue* de un sistema contiene los nodos que forman la topología hardware sobre la que se ejecuta el sistema. Se preocupa principalmente de la distribución, entrega e instalación de las partes que constituyen el sistema. Los aspectos estáticos de esta vista se representan mediante los diagramas de despliegue y los aspectos dinámicos con diagramas de interacción, estados y actividades.

Ciclo de Vida

Se entiende por *ciclo de vida de un proyecto software* a todas las etapas por las que pasa un proyecto, desde la concepción de la idea que hace surgir la necesidad de diseñar un sistema software, pasando por el análisis, desarrollo, implantación y mantenimiento del mismo y hasta que finalmente muere por ser sustituido por otro sistema.

Aunque UML es bastante independiente del proceso, para obtener el máximo rendimiento de UML se debería considerar un proceso que fuese:

- Dirigido por los *casos de uso*, o sea, que los casos de uso sean un artefacto básico para establecer el comportamiento del deseado del sistema, para validar la arquitectura, para las pruebas y para la comunicación entre las personas involucradas en el proyecto.

- Centrado en la *arquitectura* de modo que sea el artefacto básico para conceptuar, construir, gestionar y hacer evolucionar el sistema.

- Un proceso *iterativo*, que es aquel que involucra la gestión del flujo de ejecutables del sistema, e *incremental*, que es aquel donde cada nueva versión corrige defectos de la anterior e incorpora nueva funcionalidad. Un proceso iterativo e incremental se denomina *dirigido por el riesgo*, lo que significa que cada nueva versión se ataca y reducen los riesgos más significativos para el éxito del proyecto.

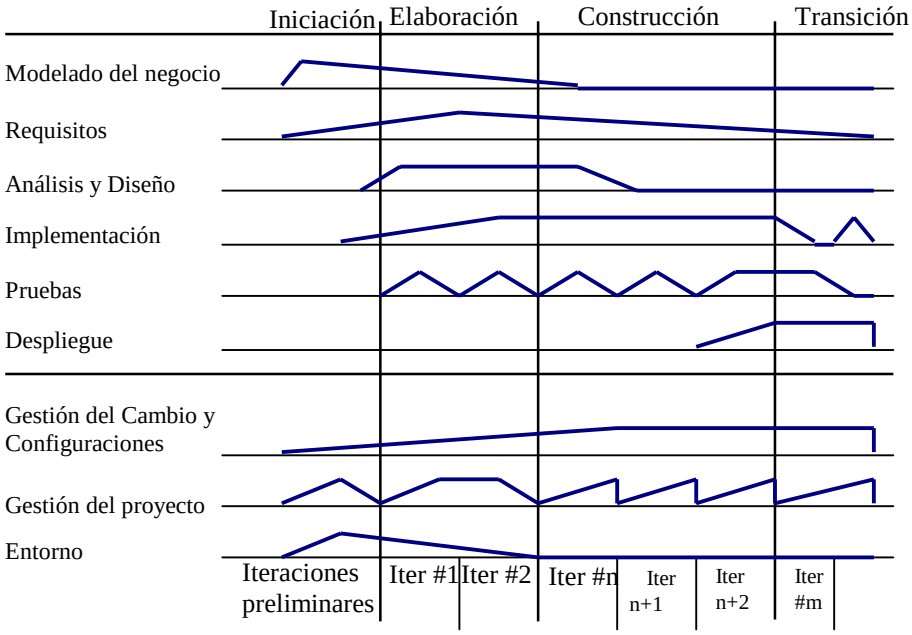


Figura 19. Ciclo de vida del desarrollo software

Este proceso, dirigido a los casos de uso, centrado en la arquitectura, iterativo e incremental pude descomponerse en fases, donde cada fase es el intervalo de tiempo entre dos hitos importantes del proceso, cuando se cumplen los objetivos bien definidos, se completan los artefactos y se toman decisiones sobre si pasar o no a la siguiente fase. En el ciclo de vida de un proyecto software existen cuatro fases, véanse en la figura 19. La **iniciación**, que es cuando la idea inicial está lo suficientemente fundada para poder garantizar la entrada en la fase de **elaboración**, esta fase es cuando se produce la definición de la arquitectura y la visión del producto. En esta fase se deben determinar los requisitos del sistema y las pruebas sobre el mismo. Posteriormente se pasa a la fase de **construcción**, que es cuando se pasa de la base arquitectónica ejecutable hasta su disponibilidad para los usuarios, en esta fase se reexaminan los requisitos y las pruebas que ha de soportar. La **transición**, cuarta fase del proceso, que es cuando el software se pone en mano de los usuarios.

Raramente el proceso del software termina en la etapa de transición, incluso durante esta fase el proyecto es continuamente reexaminado y mejorado erradicando errores y añadiendo nuevas funcionalidades no contempladas.

Un elemento que distingue a este proceso y afecta a las cuatro fases es una ***iteración,*** que es un conjunto de bien definido de actividades, con un plan y unos criterios de evaluación, que acaban en una versión del producto, bien interna o externa.

3

Modelado Estructural

A la hora de modelar un sistema es necesario identificar las cosas más importantes eligiendo un nivel de abstracción capaz de identificar todas las partes relevantes del sistema y sus interacciones. En este primer acercamiento no debemos contar con los detalles particulares de las cosas que hemos identificado como importantes, estos detalles atacarán posteriormente en cada una de las partes elegidas.

Por ejemplo, si estuviésemos trabajando en una tienda que vende ordenadores como proveedor final y nuestro jefe nos pidiera que montáramos un ordenado, directamente la división que tendríamos es la siguiente: necesitamos un monitor, un teclado, un ratón y una CPU, sin importarnos (inicialmente, claro) que la CPU este compuesta por varios elementos más. Una vez que tenemos claro que partes forman un ordenador nos daríamos cuenta que tanto el teclado como el ratón y el monitor son parte más o menos atómicas ya que, aunque estos tres objetos están compuestos por un montón de componentes electrónicos, la composición de estos no nos interesa para al nivel de abstracción que estamos trabajando (el de proveedor final). Al darnos cuenta de esto prepararíamos el monitor, el teclado y el ratón, pero en nuestro almacén tenemos monitores de 15 y 17 pulgadas, teclados ergonómicos y estándares y ratones de diferentes tamaños, colores, etc. Una vez que hemos determinado las propiedades de cada uno de ellos pasaríamos a preocuparnos por la CPU, ahora es cuando veríamos que para montar la CPU nos hacen falta una placa base, un microprocesador, una disquetera, un disco duro y un CD-ROM, cada uno de estos elementos con sus propiedades, disquetera de 1,44Mb, disco duro de 4Gb, microprocesador a 500Mhz y un CD-ROM 32x.

Una vez que tenemos todo el material en el banco de trabajo tendríamos que montar el ordenador sabiendo que cada una de las partes interactúa con el resto de alguna manera, en la CPU la disquetera, el CD-ROM y el disco duro van conectados al bus de datos, este además está conectado a la placa base y el micro tiene un lugar bien definido también en la placa base, después conectaríamos el teclado, el monitor y el ratón a la CPU y ¡ya esta!, nuestro ordenador está montado.

El modelado estructural es la parte de UML que se ocupa de identificar todas las partes importantes de un sistema así como sus interacciones. Para modelar las partes importantes del vocabulario del sistema se utilizan las clases, que nos permiten: identificación y representación de sus propiedades y sus operaciones mediante el mecanismo de abstracción. Las relaciones se utilizan para poder modelar las interacciones entre las clases.

Representación de las Clases en UML

Una clase es una descripción de un conjunto de objetos que comparten los mismos atributos, operaciones, relaciones y semántica. Hay que hacer una especial diferenciación para no confundir el concepto de clase con el de objeto. Un objeto es una instancia individual de una clase, así podríamos tener una clase que fuera los monitores y un objeto de la clase monitores que fuera un monitor marca "A" de 17 pulgadas, con la pantalla plana. En UML se usan un rectángulo para representar las clases, con distintos compartimentos para indicar sus atributos y sus operaciones.

Una clase es identifica por un nombre que la distingue del resto, el nombre es una cadena de texto. Ese nombre sólo se denomina nombre simple; un nombre de camino consta del nombre de la clase precedido del nombre del paquete en el que se encuentra incluida. En las figuras 20 y 21 podemos ver la estructura de una clase y distintos nombres de camino y nombres simples.

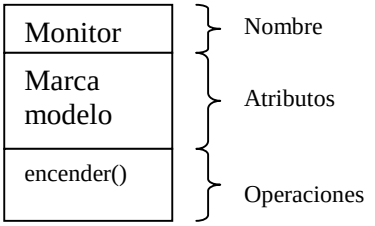


Figura 20. Representación de Clases

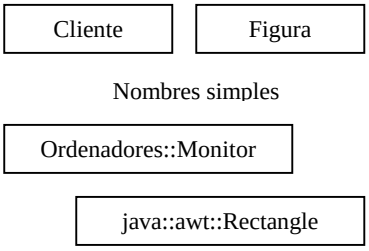


Figura 21. Tipos de nombres

Un atributo es una propiedad de una clase identificada con un nombre. Describe un rango de valores que pueden tomar las instancias de la propiedad. Los atributos representan propiedades comunes a todos los objetos de una determinada clase, por ejemplo todos los monitores tienen la propiedad dimensión, que se suele medir en pulgadas. Un cliente tiene las propiedades nombre, apellidos, dirección, teléfono... Los atributos son propiedades interesantes de las clases. Una instancia de una determinada clase tendrá valores concretos para sus atributos, mientras que las clases tienen rangos de valores que pueden admitir sus atributos.

El nombre de los atributos es un texto que normalmente se escribe en minúsculas si sólo es una palabra o la primera palabra del nombre del atributo en minúsculas y el resto de las palabras con la primera letra en mayúsculas, por ejemplo, nombrePropietario.

Una operación es una implementación de un servicio que puede ser requerido a cualquier objeto de la clase para que muestre su comportamiento. Una operación representa algo que el objeto puede hacer. Por ejemplo, un comportamiento esperado por cualquier usuario de un monitor es que este se pueda encender y apagar. El nombre de las operaciones sigue las mismas reglas de notación que los atributos pero suelen ser verbos cortos que indican una acción o comportamiento de la clase.

Como norma general, cuando se dibuja una clase no hay que mostrar todos sus atributos ni todas sus operaciones, sólo se deben mostrar los subconjuntos de estos más relevantes y una buena práctica es utilizar *estereotipos*[2] para organizar los atributos y las operaciones.

2 Estereotipos son bloques básicos de construcción a los que se les ha añadido etiquetas, iconos o texto explicativo para proporcionar más semántica a estos bloques, consiguiendo así unos diagramas más explicativos. Por ejemplo, dentro del bloque de las clases, se estereotipa el espacio reservado para las operaciones para poder indicar de que tipo son. Existe una lista de los estereotipos "estándar" de UML en el apéndice A.

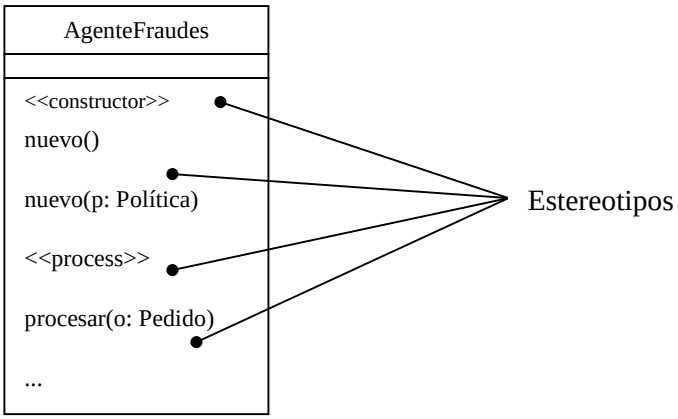


Figura 22. Estereotipos para características de las clases

Para modelar el vocabulario del sistema hay que identificar aquellas cosas que utilizan los usuarios o programadores para describir el problema, para conseguir esto se sugiere un análisis basado en casos de uso. Para cada clase hay que determinar un conjunto de responsabilidades y posteriormente determinar los atributos y las operaciones necesarias para llevar a cabo las responsabilidades de la clase.

A veces, las cosas que tenemos que modelar no tienen equivalente software, como por ejemplo, gente que emite facturas o robots que empaquetan automáticamente los pedidos, pueden formar parte del flujo de trabajo que se intenta modelar. Para modelar cosas que no son software hay que obtener abstracciones para poder representar esas cosas como clases y si lo que se está modelando es algún tipo de hardware que contiene software se tiene que considerar modelarlo como un tipo de Nodo, de manera que más tarde se pueda completar su estructura.

En ocasiones puede resultar necesario especificar más aun sobre las características de las operaciones o atributos como por ejemplo, tipos de datos que utilizan, visibilidad, características concretas del lenguaje que utilizan, excepciones a que puede producir... UML proporciona una notación concreta para estas características avanzadas pero se entrará en discusión sobre ellas ya que están fuera de los objetivos de esta introducción al modelado orientado a objetos con UML.

Responsabilidades

Una responsabilidad es un contrato u obligación de una clase, ósea, el fin para el que es creada. Cuando creamos una clase se está expresando que todos los objetos de la clase tienen el mismo tipo de estado y el mismo tipo de comportamiento. Los atributos y las operaciones son características de la clase que le permiten llevar a cabo sus responsabilidades. Al iniciar el modelado de clases es una buena practica por iniciar especificando las responsabilidades de cada clase. Una clase puede tener cualquier numero de responsabilidades pero en la práctica el numero se reduce a una o a unas pocas. Cuando se van refinando los modelos las responsabilidades se van convirtiendo en atributos y operaciones, se puede decir que las responsabilidades de una clase están en un nivel más alto de abstracción que los atributos y las operaciones. Para representar responsabilidades se utiliza un cuarto compartimiento en el bloque de construcción de la clase, en el cual se especifican mediante frases cortas de texto libre las responsabilidades que ha de cumplir la clase.

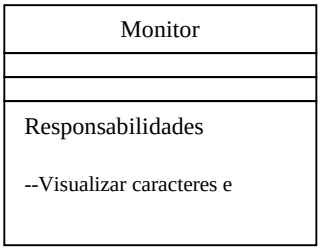


Figura 23. Responsabilidades

A la hora de determinar las responsabilidades, y su distribución, que tiene que cumplir un sistema en su conjunto, hemos de identificar un grupo de clases que colaboren entre sí para llevar a cabo algún tipo de comportamiento. Hay que determinar un conjunto de responsabilidades para cada una de esas clases teniendo cuidado de no dar demasiadas responsabilidades a una sola clase ni obtener clases con muy pocas responsabilidades. De esta manera, clases con muchas responsabilidades se dividen en varias abstracciones menores y las clases con responsabilidades triviales se introducen en otras mayores

Relaciones

Como ya hemos comentado en otras ocasiones, las relaciones son la manera de representar las interacciones entre las clases. Siguiendo con el ejemplo del montaje del ordenador, cada pieza interactúa con otra de una determinada manera y aunque por si solas no tienen sentido todas juntas forman un ordenador, esto es lo que se denomina una relación de asociación, pero además hay unas que no pueden funcionar si no están conectadas a otras como por ejemplo un teclado, el cual, sin estar conectado a una CPU es totalmente inútil, además si la CPU cambiase su conector de teclado este ya no se podría conectar a no ser que cambiase el también, esto se puede representar mediante una relación de dependencia. Es más, tenemos una disquetera de 1,44Mb, un disco duro, un CD-ROM. Para referirnos a todos estos tipos de discos podríamos generalizar diciendo que tenemos una serie de discos con ciertas propiedades comunes, como pueden ser la capacidad, la tasa de transferencia en lectura y escritura... esto es lo que se denomina una relación de generalización. La construcción de relaciones no difiere mucho de la distribución de responsabilidades entre las clases. Si se modela en exceso se obtendrán diagramas con un alto nivel de dificultad para poder leerlos debido principalmente al lío que se forma con las relaciones, por el contrario, si se modela insuficientemente se obtendrán diagramas carentes de semántica.

Para poder representar con UML cómo se conectan las cosas entre sí, ya sea lógica o físicamente, utilizamos las relaciones. Existen tres tipos de relaciones muy importantes: dependencias, generalizaciones y asociaciones. Una relación se define como una conexión entre elementos. A continuación describimos los tres tipos más importantes de relaciones:

Relación de Dependencia

Es una relación de uso entre dos elementos de manera que el un cambio en la especificación del elemento independiente puede afectar al otro elemento implicado en la relación. Determinamos el elemento dependiente aquel que necesita del otro elemento implicado en la relación (el independiente) para poder cumplir sus responsabilidades. Por ejemplo supongamos que tenemos una clase que representa un aparato reproductor Vídeo, con sus funciones y sus propiedades. Bien, para utilizar el método grabar*()* de la clase video, dependemos directamente de la clase Canal ya que grabaremos un canal u otro dependiendo de cual tenga seleccionado aparato de vídeo. A su vez la clase Televisión también depende de la clase Canal para poder visualizar un determinado canal por la Televisión.

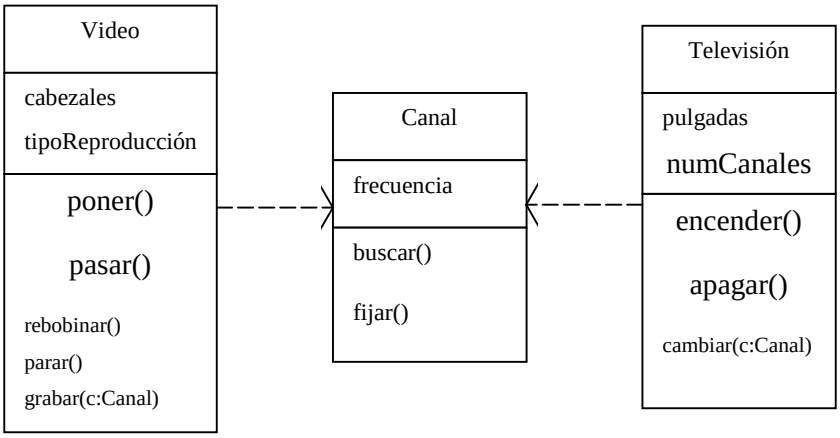


Figura 24. Ejemplo de Dependencias

En la figura 24 podemos observar un ejemplo de relación de dependencia. Si en algún momento la clase Canal cambiara (se modificara o añadiera nueva funcionalidad) las clases Video y Televisión (que dependen de ella) podrían verse afectadas por el cambio y dejar de funcionar. Como podemos observar en esta figura tanto al método grabar*()* de la clase video, como al método cambiar() de la clase Televisión se le ha añadido más semántica representando el parámetro *c* de tipo Canal. Este es un mecanismo común en UML de diseño avanzado de clases. Cuando queremos aportar más información sobre una clase y sus métodos existe una nomenclatura bien definida para determinar más los atributos y métodos de la clase indicando si son ocultos o públicos, sus tipos de datos, parámetros que utilizan y los tipos que retornan.

Normalmente, mediante una dependencia simple sin adornos suele bastar para representar la mayoría de las relaciones de uso que nos aparecen, sin embargo, existen casos avanzados en los que es conveniente dotar al diagrama de más semántica, para estos casos UML proporciona estereotipos concretos, estos se pueden consultar en el apéndice *A*.

Relación de Generalización

Es una relación entre un elemento general (llamado superclase o padre) y un caso más específico de ese elemento (llamado subclase o hijo). La generalización a veces es llamada relación *"es-un-tipo-de"* , ósea, un elemento (por ejemplo, una clase Rectángulo) es-un-tipo-de un elemento más general (por ejemplo, la clase figura). La generalización implica que los objetos hijo se pueden utilizar en cualquier lugar donde aparece el padre, pero no a la inversa. La clase hijo siempre hereda todos los atributos y métodos de sus clases padre y a menudo (no siempre) el hijo extiende los atributos y operaciones del padre. Una operación de un hijo puede tener la misma signatura que en el padre pero la operación puede ser redefinida por el hijo; esto es lo que se conoce como polimorfismo. La generalización se representa mediante una flecha dirigida con la punta hueca. Una clase puede tener ninguno, uno o varios padres. Una clase sin padres y uno o más hijos se denomina clase raíz o clase base. Una clase sin hijos se denomina clase hoja. Una clase con un único padre se dice que utiliza herencia simple y una clase con varios padres se dice que utiliza herencia múltiple. UML utiliza las relaciones de generalización para el modelado de clases e interfaces, pero también se utilizan para establecer generalizaciones entre otros elementos como por ejemplo los paquetes.

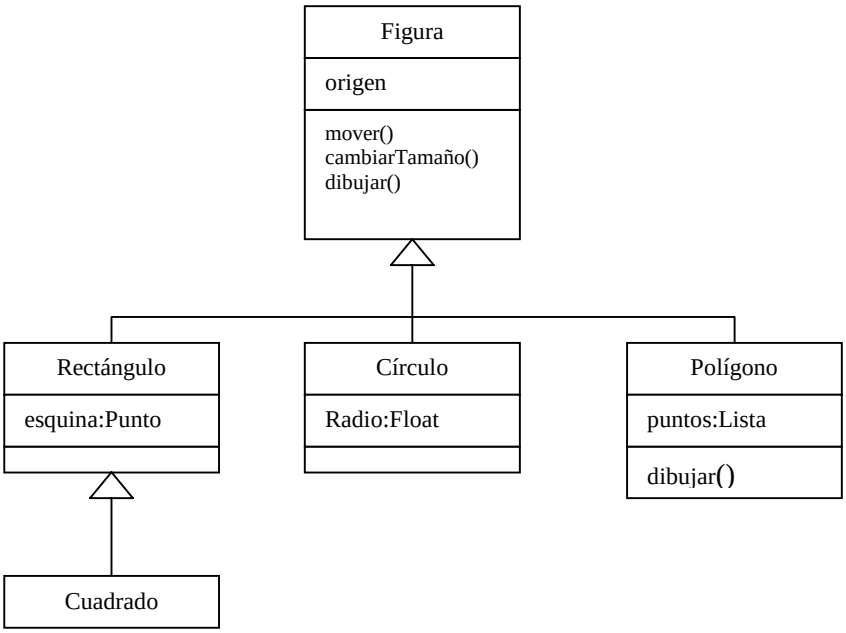


Figura 25. Ejemplo de Generalización

Mediante las relaciones de generalización podemos ver que un Cuadrado es-un-tipo-de Rectángulo que a su vez es-un-tipo-de figura. Con esta relación podemos representar la herencia, de este modo, la clase cuadrado, simplemente por herencia sabemos que tiene dos atributos, *esquina* (heredado de su padre Rectángulo) y *origen* (heredado del padre de Rectángulo, la clase figura, que se puede decir que es su abuelo). Lo mismo ocurre con las operaciones, la clase Cuadrado dispone de las operaciones mover(), cambiarTamaño() y dibujar(), heredadas todas desde figura.

Normalmente la herencia simple es suficiente para modelar los problemas a los que nos enfrentamos pero en ciertas ocasiones conviene modelar mediante herencia múltiple aunque vistos en esta situación se ha de ser extremadamente cuidadoso en no realizar herencia múltiple desde padres que solapen su estructura o comportamiento. La herencia múltiple se representa en UML simplemente haciendo llegar flechas (iguales que las de la generalización) de un determinado hijo a todos los padres de los que hereda. En el siguiente ejemplo podemos observar como se puede usar la herencia múltiple para representar especializaciones cuyos padres son inherentemente disjuntos pero existen hijos con propiedades de ambos. En el caso de los *Vehículos*, estos se pueden dividir dependiendo de por donde circulen, así tendremos Vehículos aéreos, terrestres y acuáticos. Esta división parece que nos cubre completamente todas las necesidades, y así es. Dentro de los vehículos terrestres tendremos especializaciones como coches, motos, bicicletas, etc. Dentro de los acuáticos tendremos, por ejemplo, barcos, submarinos, etc. Dentro de los aéreos tendremos por ejemplo, aviones, globos, zeppelines... En este momento tenemos una clasificación bastante clara de los vehículos, pero que ocurre con los vehículos que pueden circular tanto por tierra como por agua, ósea vehículos anfibios, como por ejemplo un tanque de combate preparado para tal efecto, en este caso podemos pensar en utilizar un mecanismo de herencia múltiple para representar que dicho tanque reúne capacidades, atributos y operaciones tanto de vehículo terrestre como acuático.

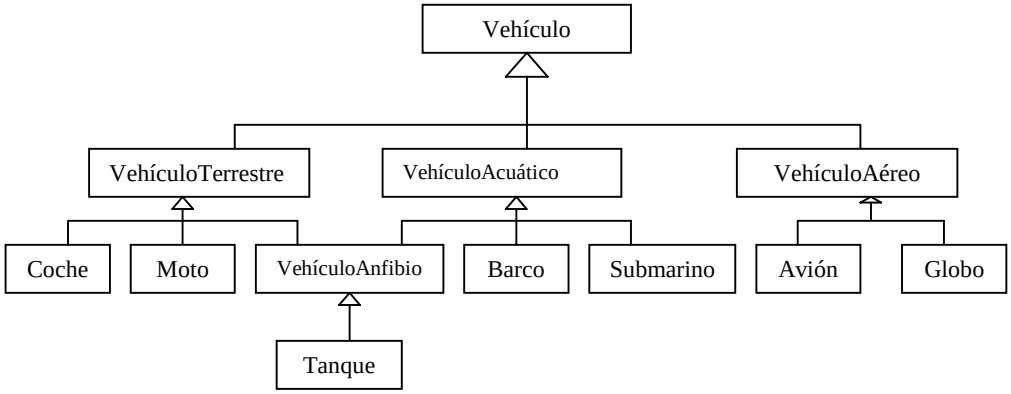


Figura 26. Ejemplo de Herencia Múltiple

Relación de Asociación

Una asociación es una relación estructural que especifica que los objetos de un elemento están conectados con los objetos de otro. Dada una asociación entre dos clases, se puede navegar desde un objeto de una de ellas hasta uno de los objetos de la otra, y viceversa. Es posible que la asociación se dé de manera recursiva en un objeto, esto significa que dado un objeto de la clase se puede conectar con otros objetos de la misma clase. También es posible, aunque menos frecuente, que se conecten más de dos clases, estas se suelen llamar asociaciones n-arias. Las relaciones de asociaciones se utilizan cuando se quieren representar relaciones estructurales.

A parte de la forma básica de representar las asociaciones, mediante una línea continua entre las clases involucradas en la relación, existen cuatro adornos que se aplican a las asociaciones para facilitar su comprensión:

- ***Nombre:*** Se utiliza para describir la naturaleza de la relación. Para que no exista ambigüedad en su significado se le puede dar una dirección al nombre por medio de una flecha que apunte en la dirección que se pretende que el nombre sea leído.

- ***Rol:*** Cuando una clase participa en una asociación esta tiene un rol especifico que juega en dicha asociación. El rol es la cara que dicha clase presenta a la clase que se encuentra en el otro extremo. Las clases pueden jugar el mismo o diferentes roles en otras asociaciones.

- ***Multiplicidad:*** En muchas situaciones del modelado es conveniente señalar cuantos objetos se pueden conectar a través de una instancia de la asociación. Este “cuantos” se denomina multiplicidad del rol en la asociación y se expresa como un rango de valores o un valor explicito. Cuando se indica multiplicidad en un extremo de una asociación se está indicando que, para cada objeto de la clase en el extremo opuesto debe haber tantos objetos en este extremo. Se puede indicar una multiplicidad de exactamente uno (1), cero o uno (0..1), muchos (0..*), uno o más (1..*) e incluso un numero exacto (por ejemplo, 5).

- ***Agregación:*** Una asociación normal entre dos clases representa una relación estructural entre iguales, es decir, ambas clases están conceptualmente al mismo nivel. A veces interesa representar relaciones del tipo “todo / parte”, en las cuales una cosa representa la cosa grande (el “todo”) que consta de elementos más pequeños (las “partes”). Este tipo de relación se denomina de agregación la cual representa una relación del tipo “tiene-un”.

Una agregación es sólo un tipo especial de asociación, esta se especifica añadiendo simplemente un rombo vacío en la parte del todo.

- ***Composición:*** Es una variación de la agregación simple que añade una semántica importante. La composición es una forma de agregación, con una fuerte relación de pertenencia y vidas coincidentes de la parte del todo. Las partes con una multiplicidad no fijada puede crearse después de la parte que representa el todo (la parte compuesta), una vez creadas pertenecen a ella de manera que viven y mueren con ella. Las partes pueden ser eliminadas antes que el todo sea destruido pero una vez que este se elimine todas sus partes serán destruidas. El todo, además, se ha de encargar de toda la gestión de sus partes, creación, mantenimiento, disposición... En la figura 29 vemos una relación de composición donde un marco pertenece a una ventana, pero ese marco no es compartido por ninguna otra ventana, esto contrasta con la agregación simple en la que una parte puede ser compartida por varios objetos agregados. Por ejemplo una pared puede estar compartida por varias habitaciones.

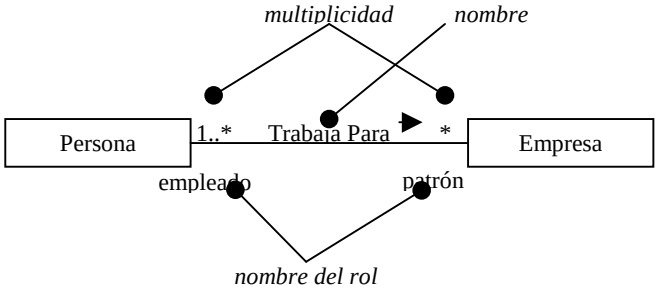


Figura 27. Ejemplo de asociación y sus partes

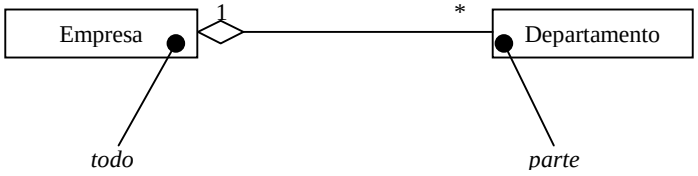


Figura 28. Ejemplo de agregación

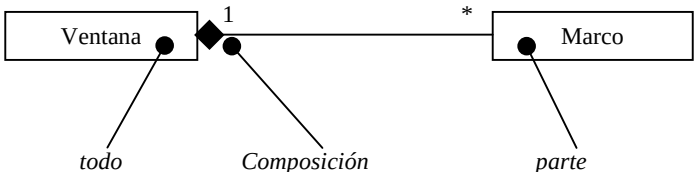


Figura 29. Ejemplo de Composición

Interfaces

Una interfaz es una colección de operaciones que sirven para especificar el servicio que una clase o componente da al resto de las partes involucradas en un sistema. Al declarar una interfaz, se puede enunciar el comportamiento deseado de una abstracción independientemente de su implementación. Los clientes trabajan con esa interfaz de manera independiente, así que si en un momento dado su implementación cambia, no seremos afectados, siempre y cuando se siga manteniendo su interfaz intacta cumpliendo las responsabilidades que tenia.

A la hora de construir sistemas software es importante tener una clara separación de intereses, de forma, que cuando el sistema evolucione, los cambios en una parte no se propaguen afectando al resto. Una forma importante de lograr este grado de separación es especificar unas líneas de separación claras en el sistema, estableciendo una frontera entre aquellas partes que pueden cambiar independientemente. Al elegir las interfaces apropiadas, se pueden utilizar componentes estándar y bibliotecas para implementar dichas interfaces sin tener que construirlas uno mismo.

UML utiliza las interfaces para modelar las líneas de separación del sistema. Muchos lenguajes de programación soportan el concepto de interfaces, como pueden ser Java, Visual Basic y el IDL de CORBA. Además, las interfaces no son sólo importantes para separar la especificación y la implementación de clases o componentes, sino que al pasar a sistemas más grandes, se pueden usar para especificar la vista externa de un paquete o subsistema.

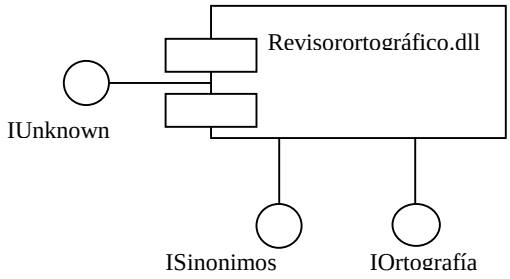


Figura 30. Interfaces

Al igual que una clase, una interfaz puede participar en relaciones de generalización, asociación y dependencia. Además una interfaz puede participar en relaciones de realización. Una realización es una relación semántica entre dos clasificadores en la que un clasificador especifica un contrato que el otro clasificador garantiza que va a llevar a cabo.

Una interfaz especifica un contrato para una clase o componente sin dictar su implementación. Una clase o componente puede realizar diversos componentes, al hacer eso se compromete ha cumplir fielmente esos contratos, lo que significa que proporciona un conjunto de métodos que implementan apropiadamente las operaciones definidas en el interfaz.

Existen dos formas de representar un interfaz en UML, la primera es mediante una piruleta conectada a un lado de una clase o componente. Esta forma es útil cuando queremos visualizar las líneas de separación del sistema ya que por limitaciones de estilo no se pueden representar las operaciones o las señales de la interfaz. La otra forma de representar una interfaz es mostrar una clase estereotipada que permite ver las operaciones y otras propiedades, y conectarla mediante una relación de realización con la componente o el clasificador que la contiene. Una realización se representa como una flecha de punta vacía con la línea discontinua.

La figura 31 muestra un ejemplo sobre las dos formas de representar un interfaz con UML.

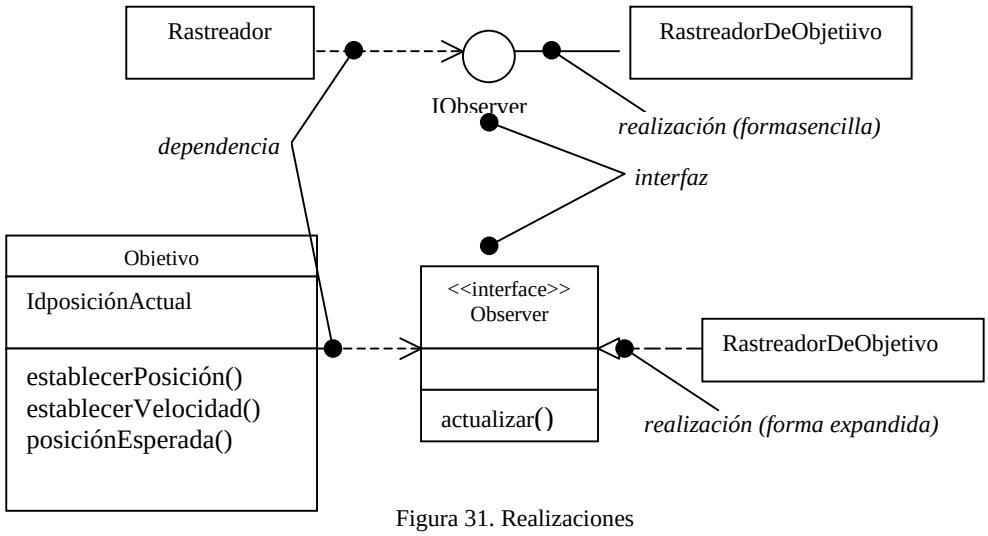


Figura 31. Realizaciones

Roles

Según hemos dicho, una clase puede implementar varios interfaces, por lo tanto una instancia de esa clase debe poder soportar todos esos interfaces, no obstante en determinados contextos, sólo un o más interfaces tendrán sentido. En este caso cada interfaz representa un rol que juega el objeto. Un rol denota un comportamiento de una entidad en un contexto particular, dicho de otra manera un rol es la cara que una abstracción presenta al mundo.

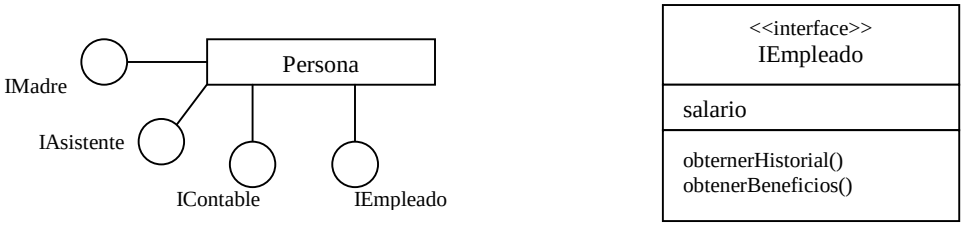


Figura 32. Roles

Para ilustrar este concepto supongamos la clase *Persona* y los roles que una persona puede desempeñar dependiendo del contexto en que se encuentre, como ejemplos de roles podemos tener: Madre, Asistente, Contable, Empleado, Directivo, Piloto, Cantante, etc. La cantidad de roles que pueda soportar una clase es en principio indefinido, solamente depende de ámbito de actuación del sistema que estemos modelando. Cada uno de los roles que se han definido tendrá una correspondencia con un interfaz concreto.

Como se puede observar en la figura 32, el interfaz IEmpleado tiene unas operaciones específicas para una persona que desempeñe el rol de Empleado. También podemos observar los diferentes interfaces que tiene la clase *Persona*, aunque lo usual es utilizar la notación en forma de piruleta para denotar líneas de separación del sistema que comúnmente serán necesario para componentes más que para clases y para estas utilizar la notación expandida de los interfaces con relaciones de realización.

Paquetes

Visualizar, especificar, construir y documentar grandes sistemas conlleva manejar una cantidad de clases, interfaces, componentes, nodos, diagramas y otros elementos que puede ser muy elevada. A medida que el sistema va creciendo hasta alcanzar un gran tamaño, se hace necesario organizar estos elementos en bloques mayores. En UML el paquete es un mecanismo de propósito general para organizar elementos de modelado en grupos.

La visibilidad de los elementos dentro de un paquete se puede controlar para que algunos elementos sean visibles fuera del paquete mientras que otros permanezcan ocultos. Los paquetes también se pueden emplear para presentar las diferentes vistas de la arquitectura de un sistema.

Todos los grandes sistemas se jerarquizan en niveles para facilitar su comprensión. Por ejemplo, cuando hablamos de un gran edificio podemos hablar de estructuras simples como paredes, techos y suelos, pero debido al nivel de complejidad que supone hablar de un gran edificio utilizamos abstracciones mayores como pueden ser zonas públicas, el área comercial y las oficinas. Al mismo tiempo, estas abstracciones pueden ser que se agrupen en otras mayores como la zona de alquileres y la zona de servicios del edificio, estas agrupaciones puede ser que no tengan nada que ver con la estructura final del edificio sino se usan simplemente para organizar los planos del mismo.

Términos y conceptos

En UML las abstracciones que organizan un modelo se llaman *paquetes*. Un paquete es un mecanismo de propósito general para organizar elementos en grupos. Los paquetes ayudan a organizar los elementos en los modelos con el fin de poder comprenderlos más fácilmente. Los paquetes también permiten controlar el acceso a sus contenidos para controlar las líneas de separación de un sistema. UML proporciona una representación gráfica de los paquetes como se muestra en la figura 33, esta notación permite visualizar grupos de elementos que se pueden manipular como un todo y en una forma permite controlar la visibilidad y el acceso a los elementos individuales.

Cada paquete ha de tener un nombre que lo distinga de otros paquetes, el nombre puede ser un nombre simple o un nombre de camino que indique el paquete donde está contenido. Al igual que las clases, un paquete, se puede dibujar adornado con valores etiquetados o con apartados adicionales para mostrar sus detalles.

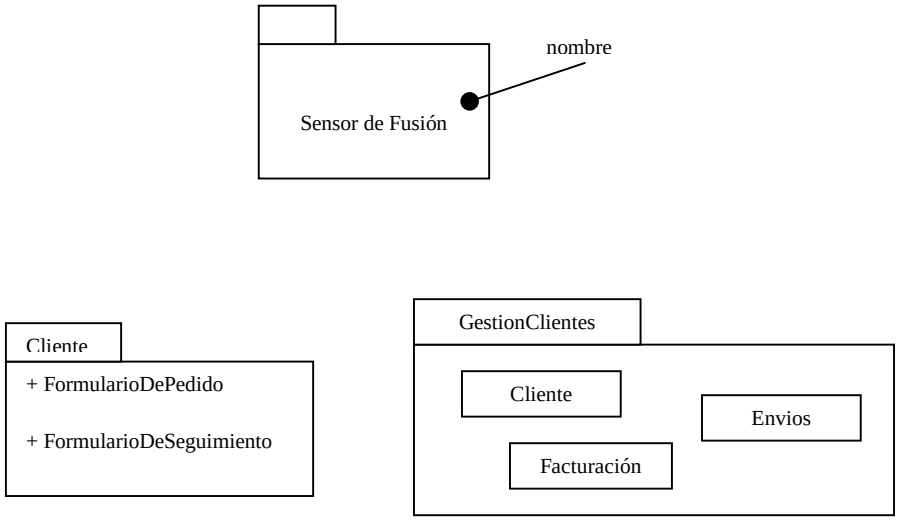


Figura 33. Distintos tipos de paquetes

Elementos Contenidos

Un paquete puede contener otros elementos, incluyendo clases, interfaces, componentes, nodos, colaboraciones, casos de uso, diagramas e incluso otros paquetes. La posesión es una relación compuesta, lo que significa que el elemento se declara en el paquete. Si el paquete se destruye, el elemento es destruido. Cada elemento pertenece exclusivamente a un paquete.

Un paquete forma un espacio de nombres, lo que quiere decir que los elementos de la misma categoría deben tener nombres únicos en el contexto del paquete contenedor. Por ejemplo no se pueden tener dos clases llamadas *Cola* dentro de un mismo paquete, pero si se puede tener la clase *Cola* dentro del paquete *P1* y otra clase (diferente) llamada *Cola* en el paquete *P2*. Las clases *P1::Cola* y *P2::Cola* son, de hecho, clases diferentes y se pueden distinguir por sus nombres de camino.

Diferentes tipos de elementos dentro del mismo paquete pueden tener el mismo nombre, por ejemplo, podemos tener dentro de un paquete la clase *Temporizador* y un componente llamado *Temporizador*. En la práctica, para evitar confusiones, es mejor asociar cada elemento a un nombre único para todas las categorías.

Los paquetes pueden contener a otros paquetes, esto significa que es posible descomponer los modelos jerárquicamente. Por ejemplo, se puede tener la clase *Cámara* dentro del paquete *Visión* y este a su vez dentro del paquete *Sensores*. El nombre completo de la case será *Sensores::Visión::Cámara*. En la práctica es mejor evitar paquetes muy anidados, aproximadamente, dos o tres niveles de anidamiento es el límite manejable. En vez de anidamiento se utilizarán relaciones de dependencia o generalización sobre los paquetes para poder utilizar elementos externos a un paquete que residan en otro. Como se muestra en la figura 34 es posible que se establezcan relaciones de tipo dependencia o generalización entre paquetes. Tendremos relaciones de dependencia cuando un o más elementos incluidos en un paquete tengan relaciones de dependencia con elementos que se encuentran en otro paquete. Para poder utilizar los elementos que se encuentran en otros paquetes se ha de *importar* el paquete que nos interesa, por eso cuando se habla de dependencias entre paquetes se utiliza un estereotipo que representa la importación. Cuando un paquete importa a otro sólo podrá acceder a los elementos que sean públicos en el paquete que está importando, estos elementos se dice que son *exportados* por el paquete que los contiene.

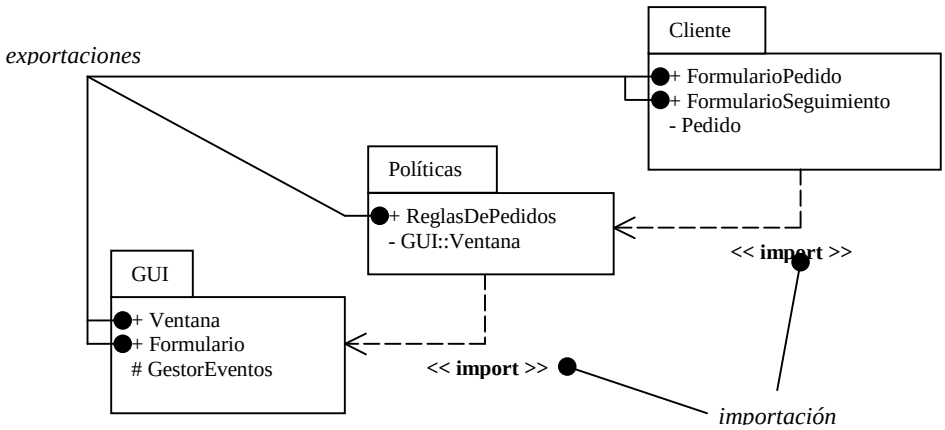


Figura 34. Importación y Exportación

Instancias

Los términos “instancia” y “objeto” son en gran parte sinónimos y, por ello, la mayoría de las veces pueden intercambiarse. Una instancia es una manifestación concreta de una abstracción a la que se puede aplicar un conjunto de operaciones y que puede tener un estado que almacena los efectos de la operación. Las instancias se utilizan para modelar cosas concretas o prototípicas del mundo real. Casi todos los bloques de construcción de UML participan de esta dicotomía clase / objeto. Por ejemplo, puede haber casos de uso e instancias de casos de uso, nodos e instancias de nodos, asociaciones e instancias de asociaciones, etc.

Una abstracción denota la esencia ideal de una cosa; una instancia denota una manifestación concreta. Esta separación de abstracción e instancia aparecerá en todo lo que se modele. Para una abstracción dada puede haber infinidad de instancias, para una instancia dada habrá una única abstracción que especifique las características comunes a todas las instancias. En UML se pueden representar abstracciones y sus instancias. Casi todos los bloques de construcción de UML pueden modelarse en términos de su esencia o en términos de sus instancias. La mayoría de las veces se trabajará con ellos como abstracciones. Cuando se desee modelar manifestaciones concretas o prototípicas se necesitará trabajar con sus instancias.

Una *instancia* es una manifestación concreta de una abstracción a la que se puede aplicar un conjunto de operaciones y posee un estado que almacena el efecto de las operaciones. Gráficamente, una instancia se representa subrayando su nombre.

Operaciones

Un objeto no sólo es algo que normalmente ocupa espacio en el mundo real, también es algo a lo que se le pueden hacer cosas. Las operaciones que se pueden ejecutar sobre un objeto se declaran en la abstracción del objeto (en la clase). Por ejemplo, si la clase *Transacción* define la operación *commit*, entonces, dada una instancia *t : Transacción*, se pueden escribir expresiones como *t.commit()*. La ejecución de esta expresión significa que sobre *t* (el objeto) opera *commit* (la operación).

Estado

Un objeto también tiene estado, en este sentido incluye todas las propiedades (normalmente estáticas) del objeto más los valores actuales (normalmente dinámicos) de estas propiedades. Estas propiedades incluyen los atributos del objeto, así como sus partes agregadas. El estado de un objeto, pues, dinámico, deforma que al visualizar su estado se está especificando su estado en un momento dado tiempo y del espacio. Es posible mostrar el cambio de estado de un objeto representándolo muchas veces en el mismo diagrama de interacción, pero con cada ocurrencia se está representando un estado diferente. Cuando se opera sobre un objeto normalmente cambia su estado, mientras que cuando se consulta un objeto su estado no cambia. En la figura 35 podemos observar dos instancias (dos objetos) donde se muestra por un lado sus atributos y por otro su estado de manera explicita.

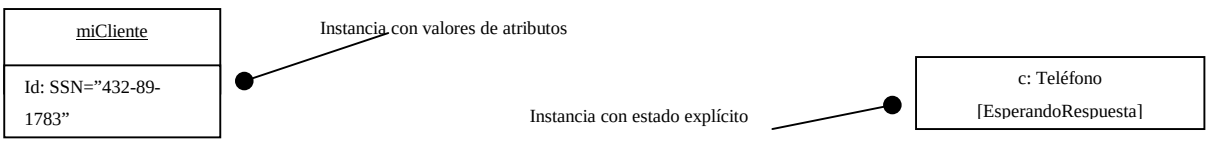


Figura 35. Estado de un objeto

Modelado de instancias concretas

Cuando se modelan instancias concretas, en realidad se están visualizando cosas que existen en el mundo real. Una de las cosas para las que se emplean los objetos es para poder modelar instancias concretas del mundo real, para modelar instancias concretas:

- Hay que identificar aquellas instancias necesarias y suficientes para visualizar, especificar, construir o documentar el problema.
- Hay que representar esos objetos en UML como instancias. Cuando sea posible hay que dar un nombre a cada objeto. Si no hay nombre significativo para el objeto se puede representar como un objeto anónimo.
- Hay que mostrar el estereotipo, los valores, etiquetados y los atributos (con sus valores) de cada instancia necesarios y suficientes para modelar el problema.
- Hay que representar las instancias y sus relaciones en un diagrama de objetos u otro diagrama apropiado al tipo de instancia (nodo, componente...)

Por ejemplo, la siguiente figura muestra un diagrama de objetos extraído de un sistema de validación de tarjetas de crédito. Hay un multiobjeto que contiene instancias anónimas de la clase *Transacción*, también hay dos objetos con nombre explicito (*agentePrincipal* y *actual*) ambos muestran su clase aunque de formas diferentes. La figura 36 también muestra explícitamente el estado actual del objeto *agentePrincipal.*

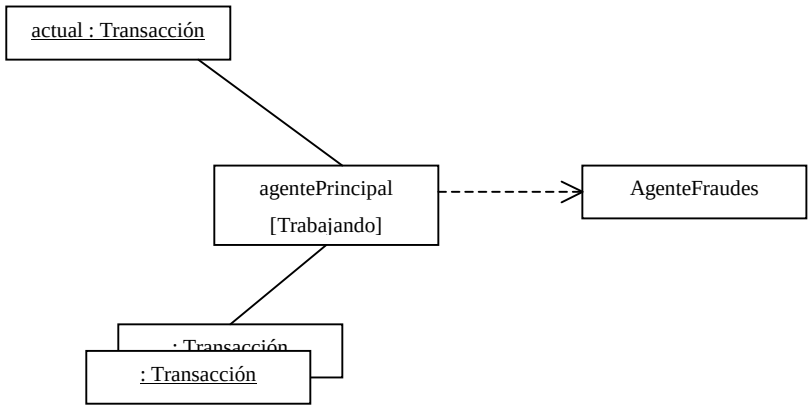


Figura 36 Modelado de Instancias concretas

Modelado de instancias prototípicas

La utilización más importante que se hace de las instancias es modelar las interacciones dinámicas entre objetos. Generalmente, cuando se modelan tales interacciones no se están modelando instancias concretas que existen en el mundo real, más bien se están modelando objetos conceptuales. Estos objetos son prototípicos y, por tanto, son roles con los que conforman las instancias concretas. Por ejemplo si se quiere modelar la forma en que reaccionan los objetos a los eventos del ratón en una aplicación con ventanas, se podría dibujar un diagrama de interacción con instancias prototípicas de ventanas, eventos y manejadores. Para modelar instancias prototípicas:

- Hay que representar esos objetos en UML como instancias.

- Hay que mostrar las propiedades de cada instancia necesarias y suficientes para modelar el problema.

- Hay que representar las instancias y sus relaciones en un diagrama de interacción o de actividades.

La figura 37 muestra un diagrama de interacción que ilustra un escenario parcial para el inicio de una llamada telefónica. Todos los objetos que se representan son prototípicos y son sustitutos de objetos que podrán existir en el mundo real.

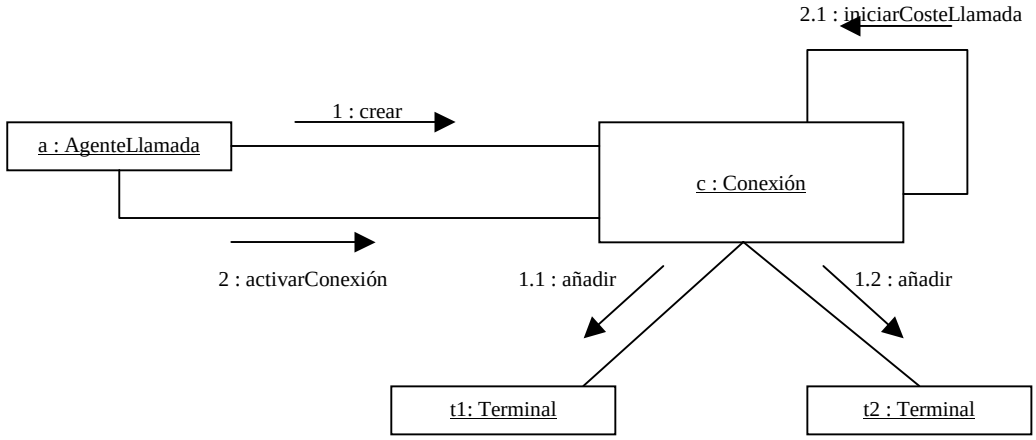


Figura 37. Modelado de instancias prototípicas

4

Diagramas de clases y de objetos

Los diagramas de clases son los más utilizados en el modelado de sistemas orientados a objetos. Un diagrama de clases muestra un conjunto de clases, interfaces y colaboraciones, así como sus relaciones.

Los diagramas de clases se utilizan para modelar la vista de diseño estática de un sistema. Principalmente, esto incluye modelar el vocabulario del sistema, modelar las colaboraciones o modelar esquemas. Los diagramas de clases también son la base para un par de diagramas relacionados, los diagramas de componentes y los diagramas de despliegue. Los diagramas de clases son importantes no sólo para visualizar, especificar y documentar modelos estructurales, sino que también para construir sistemas ejecutables aplicando ingeniería directa e inversa.

Por otro lado, los diagramas de objetos modelan las instancias de los elementos contenidos en los diagramas de clases. Un diagrama de objetos muestra un conjunto de objetos y sus relaciones en un momento concreto. Se utilizan para modelar la vista de diseño estática o la vista de procesos estática, esto conlleva el modelado de una instantánea del sistema en un momento concreto y la representación de un conjunto de objetos con su estado y con sus relaciones.

Diagramas de Clases

Un diagrama de clases es un diagrama que muestra un conjunto de clases, interfaces, colaboraciones y sus relaciones. Al igual que otros diagramas los diagramas de clases pueden contener notas y restricciones. También pueden contener paquetes o subsistemas, los cuales su usan para agrupar los elementos de un modelo en partes más grandes. A veces se colocarán instancias en los diagramas de clases, especialmente cuando se quiera mostrar el tipo (posiblemente dinámico) de una instancia.

Los diagramas de componentes y de despliegue son muy parecidos a los de clases, simplemente que muestran componentes y nodos respectivamente en vez de clases.

Usos comunes

Los diagramas de clases se utilizan para modelar la vista de diseño estática de un sistema. Esta vista soporta principalmente los requisitos funcionales de un sistema, los servicios que el sistema debe proporcionar a los usuarios finales.

Cuando se modela la vista de diseño estática de un sistema, normalmente se utilizarán los diagramas de clases de unas de estas tres formas:

1. **Para modelar el vocabulario de un sistema.**

   El modelado del vocabulario de un sistema implica tomar decisiones sobre qué abstracciones son parte del sistema en consideración y cuáles caen fuera de sus límites. Los diagramas de clases se utilizan para especificar estas abstracciones y sus responsabilidades.

2. **Para modelar colaboraciones simples.**

   Una colaboración es una sociedad de clases, interfaces y otros elementos que colaboran para proporcionar un comportamiento cooperativo mayor que la suma de todos sus elementos. Por ejemplo, cuando se modela la semántica de una transición en un sistema distribuido, no se puede observar simplemente a una clase aislada para comprender qué ocurre. Más bien, esta semántica se lleva a cabo por un conjunto de clases que colaboran entre sí. Los diagramas de clases se emplean para visualizar y especificar este conjunto de clases.

3. **Para modelar el esquema lógico de una base de datos**

   Se puede pensar en un esquema como en un plano para el diseño conceptual de una base de datos. En muchos dominios se necesitará almacenar información persistente en una base de datos relacional o en una base de datos orientada a objetos. Se pueden modelar esquemas para estas bases de datos mediante diagramas de clases.

Modelado de colaboraciones simples

Ninguna clase se encuentra aislada. En vez de ello, cada una trabaja en colaboración con otras para llevar a cabo alguna semántica mayor que la asociada a cada clase individual. Por tanto, aparte de capturar el vocabulario del sistema, también hay que prestar atención a la visualización, especificación, construcción y documentación de la forma en que estos elementos del vocabulario colaboran entre sí. Estas colaboraciones se representan con los diagramas de clases.

Cuando se crea un diagrama de clases, se está modelando una parte de los elementos y relaciones que configuran la vista de diseño del sistema. Por esta razón, cada diagrama de clases debe centrarse en una colaboración cada vez.

Para modelar una colaboración:

- Hay que identificar los mecanismos que se quiere modelar. Un mecanismo representa una función o comportamiento de parte del sistema que se está modelando que resulta de la interacción de una sociedad de clases, interfaces y otros elementos.

- Para cada mecanismo, hay que identificar las clases, interfaces y otras colaboraciones que participan en esta colaboración. Asimismo, hay que identificar las relaciones entre estos elementos.

- Hay que usar escenarios para recorrer la interacción entre estos elementos. Durante el recorrido, se descubrirán partes del modelo que faltaban y partes que eran semánticamente incorrectas.

- Hay que asegurarse de rellenar estos elementos con su contenido. Para las clases hay que comenzar obteniendo un reparto equilibrado de responsabilidades. Después, a lo largo del tiempo, hay que convertir las responsabilidades en atributos y operaciones concretos.

En la figura 38 se muestra un conjunto de clases extraídas de la implementación de un robot autónomo. La figura se centra en las clases implicadas en el mecanismo para mover el robot autónomo a través de una trayectoria. Hay muchas más clases implicadas en este sistema, pero el diagrama muestra sólo aquellas abstracciones implicadas directamente en mover el robot. Algunas de las clases que se presentan en este diagrama aparecerán en otros diagramas del sistema.

Aparece una clase abstracta (*Motor*) con dos hijos concretos, MotorDireccion y MotorPrincipal. Ambas clases heredan las cinco operaciones de clase padre. A su vez, las dos clases se muestran como partes de otra clase, *Conductor*. La clase agente trayectoria tiene una asociación uno a uno con *Conductor* y una asociación uno a muchos con SensorDeColision. No se muestran atributos ni operaciones para Agente trayectoria, aunque sí se indican sus responsabilidades.

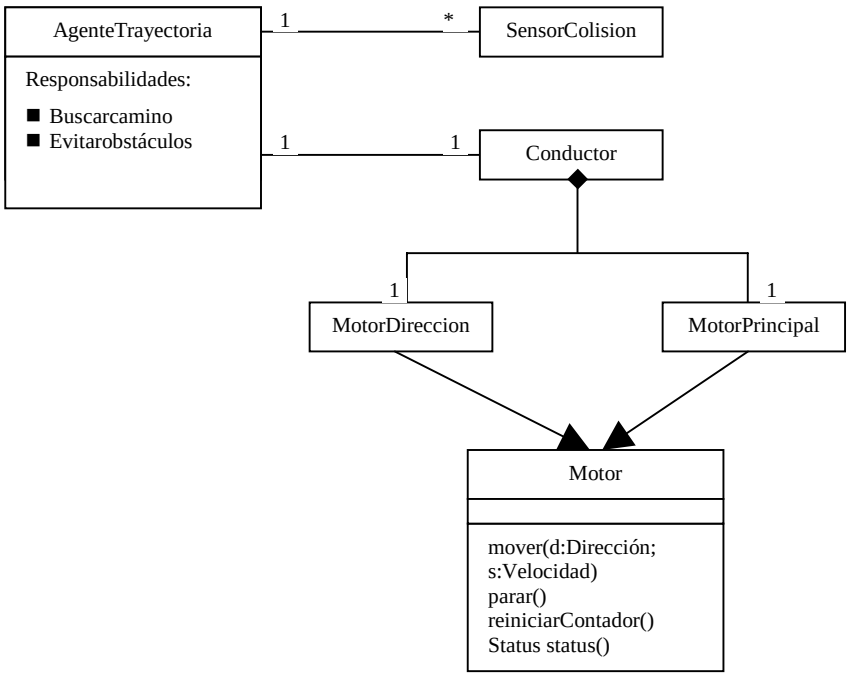


Figura 38. Modelado de colaboraciones simples

Modelado de un Esquema Lógico de Base de Datos

Muchos de los sistemas que se modelen tendrán objetos persistentes, lo que significa que estos objetos podrán ser almacenados en una base de datos con el fin de poderlos recuperar posteriormente. La mayoría de las veces se empleará una base de datos relacional, una base de datos orientada a objetos o

una base de datos híbrida objeto-relacional para el almacenamiento persistente. UML es apropiado para modelar esquemas lógicos de bases de datos, así como bases de datos físicas. Los diagramas de clases UML son un superconjunto de los diagramas entidad-relación, una herramienta de modelado para el diseño lógico de bases de datos utilizado con mucha frecuencia. Mientras que los diagramas E-R se centran sólo en los datos, los diagramas de clases van un paso más allá, permitiendo el modelado del comportamiento. En la base de datos física, estas operaciones lógicas normalmente se convierten en disparadores (*triggers)* o procedimientos almacenados. Para modelar un esquema de base de datos:

- Hay que identificar aquellas clases del modelo cuyo estado debe trascender en el tiempo de vida de las aplicaciones.

- Hay que crear un diagrama de clases que contenga las clases que hemos identificado y marcarlas como persistentes (con un valor etiquetado estándar), también se pueden definir un conjunto de valores etiquetados para cubrir los detalles específicos de las bases de datos.

- Hay que expandir los detalles estructurales de estas clases, esto significa que hay que especificar los detalles de sus atributos y centrar la atención en las asociaciones que estructuran estas clases y en sus cardinalidades.

- Hay que buscar patrones comunes que complican el diseño físico de las bases de datos, tales como asociaciones cíclicas, asociaciones uno a uno y asociaciones n-arias. Donde sea necesario deben crearse abstracciones intermedias para simplificar la estructura lógica.

- Hay que considerar también el comportamiento de las clases persistentes expandiendo las operaciones que sean importantes para el acceso a los datos y la integridad de los datos. En general, para proporcionar una mejor separación de los intereses, las reglas de negocio relativas a la manipulación de conjuntos de estos objetos deberían encapsularse en una capa por encima de estas clases persistentes.

- Donde sea posible, hay que usar herramientas que ayuden a transformar un diseño lógico en un diseño físico

La figura 39 muestra un conjunto de clases extraídas de un sistema de información de una universidad. Esta figura es la extensión de un diagrama anterior, y ahora se muestran las clases a un nivel suficientemente detallado para construir una base de datos física. Las clases que aparecen en el diagrama se han marcado como persistentes, indicando que sus instancias se han concebido para almacenarse en una base de datos.

Todos los atributos son de tipo primitivo, cuando se modela un esquema, generalmente una relación con cualquier tipo no primitivo se modela mediante agregaciones explícitas en vez de con atributos.

Diagramas de Objetos

En UML los diagramas de clases se utilizan para visualizar los aspectos estáticos de los bloques de construcción del sistema. Los diagramas de interacción se utilizan para ver los aspectos dinámicos del sistema y constan de instancias de estos bloques y mensajes enviados entre ellos. Un diagrama de objetos contiene un conjunto de instancias de los elementos encontrados en un diagrama de clases. Por lo tanto, un diagrama de objetos expresa la parte estática de una interacción, consistiendo en los objetos que colaboran pero sin ninguno de los mensajes enviados entre ellos. En ambos casos, un diagrama de objetos congela un instante en el tiempo, como se muestra en la figura 40.

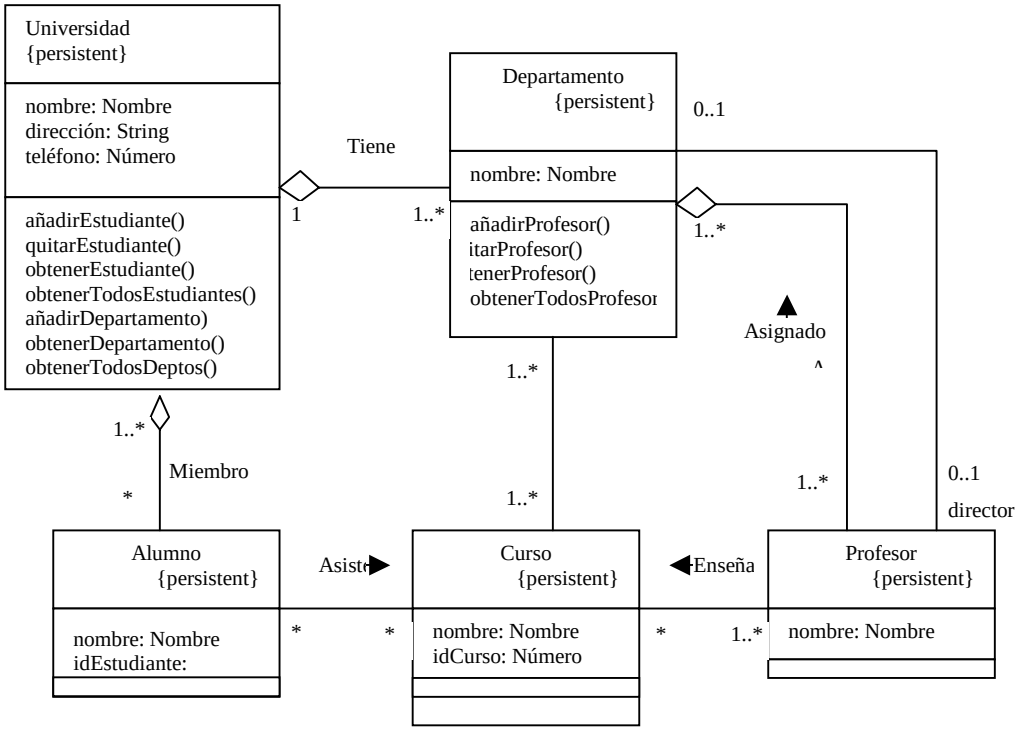


Figura 39. Modelado de un esquema

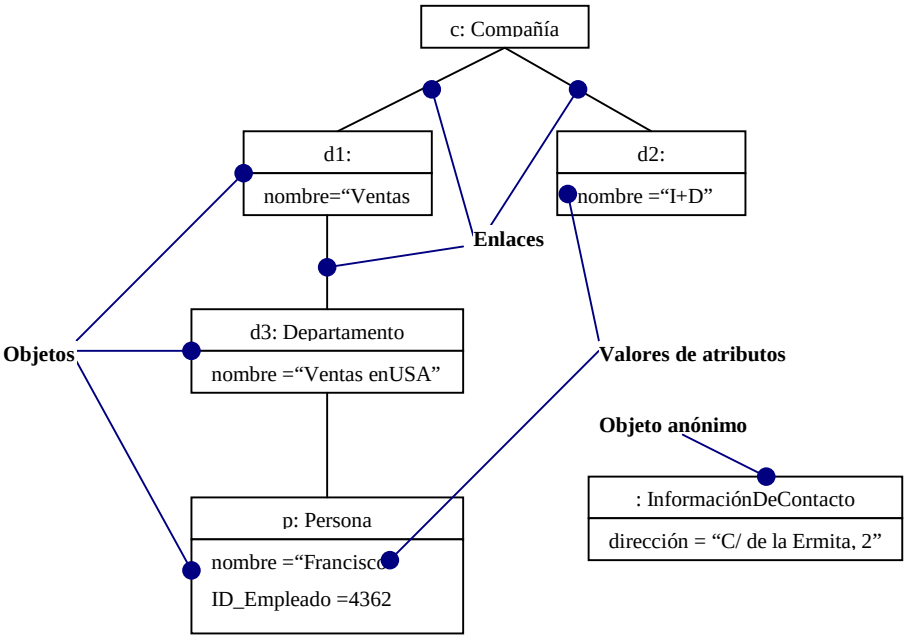


Figura 40. Un diagrama de Objetos

Un diagrama de objetos es simplemente un diagrama que representa un conjunto de objetos y sus relaciones en un momento concreto. Un diagrama de objetos contiene *objetos* y *enlaces*. Al igual que los demás diagramas, puede contener notas y restricciones. Los diagramas de objetos también pueden contener paquetes o subsistemas, los cuales se usan para agrupar los elementos de un modelo en partes

más grandes. A veces se colocarán clases en los diagramas de objetos, especialmente cuando se quiera mostrar que clase hay detrás de cada instancia. Resumiendo, un diagrama de objetos se puede tratar como una instancia de un diagrama de clases o la parte estática de un diagrama de interacción.

Usos comunes

Los diagramas de objetos se emplean para modelar la vista de diseño estática o la vista de procesos estática del sistema, igual que se hace con los diagramas de clases, pero desde la perspectiva de instancias reales o prototípicas. Esta vista sustenta principalmente los requisitos funcionales de un sistema (o sea, los servicios que debe proporcionar el sistema a los usuarios finales). Los diagramas de objetos permiten modelar estructuras de datos estáticas.

Al modelar la vista de diseño estática o la vista de procesos estática de un sistema, normalmente los diagramas de objetos se suelen utilizar para modelar estructuras de objetos.

El modelado de estructuras de objetos implica tomar una instantánea de los objetos de un sistema en un cierto momento. Un diagrama de objetos presenta una escena estática dentro de la historia representada por un diagrama de interacción. Los diagramas de objetos sirven para visualizar, especificar, construir y documentar la existencia de ciertas instancias en el sistema, junto a sus relaciones.

Modelado de estructuras de objetos

Cuando se construye un diagrama de clases, de componentes o de despliegue, lo que realmente se está haciendo es capturar como un grupo un conjunto de abstracciones que son interesantes y, en ese contexto, exhibir su semántica y las relaciones entre las abstracciones del grupo. Estos diagramas sólo muestran “lo que puede ser”. Si la clase *A* tiene una asociación uno a muchos con la clase *B*, entonces para una instancia de *A* podría haber cinco instancias de *B*. Además, en un momento dado, esa instancia de *A*, junto con las instancias de *B* relacionadas, tendrá ciertos valores para sus atributos y máquinas de estados.

Si se congela un sistema en ejecución o uno se imagina un instante concreto en un sistema modelado, aparecerá un conjunto de objetos, cada uno de ellos en un estado específico, y cada uno con relaciones particulares con otros objetos. Los diagramas de objetos se pueden utilizar para visualizar, especificar, construir y documentar la estructura de esos objetos. Los diagramas de objetos son especialmente útiles para modelar estructuras de datos complejas.

Cuando se modela la vista de diseño de un sistema, se puede utilizar un conjunto de diagramas de clases para especificar completamente la semántica de las abstracciones y sus relaciones. Con los diagramas de objetos, sin embargo, no se puede especificar completamente la estructura de objetos del sistema. Pueden existir una multitud de posibles instancias de una clase particular, y para un conjunto de clases con relaciones entre ellas, pueden existir muchas más configuraciones posibles de esos objetos. Por lo tanto, al utilizar diagramas de objetos sólo se pueden mostrar significativamente conjuntos interesantes de objetos concretos o prototípicos. Esto es lo que significa modelar una estructura de objetos (un diagrama de objetos muestra un conjunto de objetos relacionados entre sí en un momento dado).

Para modelar diagramas de objetos:

- Hay que identificar el mecanismo que se desea modelar. Un mecanismo representa alguna función o comportamiento de la parte del sistema que se está modelando, que resulta de la interacción de un conjunto de clases, interfaces y otros elementos.

- Para cada mecanismo, hay que identificar las clases, interfaces y otros elementos que participan en esta colaboración; identificar, también, las relaciones entre estos elementos.

- Hay que considerar un escenario en el que intervenga este mecanismo. También hay que congelar este escenario en un momento concreto, y representar cada objeto que participe en el mecanismo.

- Hay que mostrar el estado y valores de los atributos de cada un de esos objetos, si son necesarios para comprender el escenario.

- Analógicamente, hay que mostrar los enlaces de esos objetos, que representarían instancias de las asociaciones entre ellos.

La figura 42 representa un conjunto de objetos extraídos de la implementación de un robot autónomo, esta figura se centra en alguno de los objetos implicados en el mecanismo utilizado por el robot para calcular un modelo del mundo en el que se mueve. Hay muchos más objetos implicados en un sistema de ejecución, pero este diagrama se centra sólo en las abstracciones implicadas directamente en la creación de la vista del mundo.

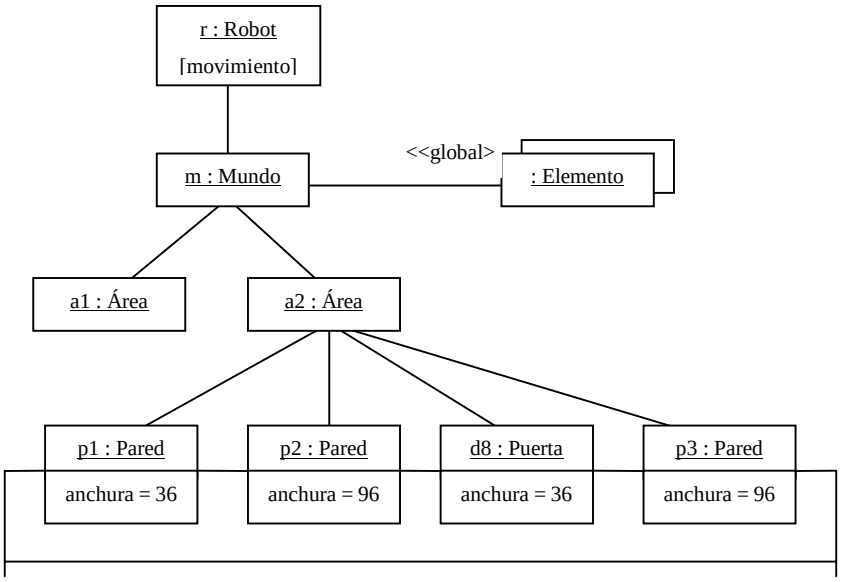


Figura 41. Modelado de Estructuras de Objetos.

Como indica la figura 41, un objeto representa el propio robot (*r*, una instancia del *Robot*), y *r* se encuentra actualmente en estado de *movimiento*. Este objeto tiene un enlace con *m*, una instancia de *Mundo*, que representa una abstracción del mundo del robot. Este objeto tiene un enlace con un multiobjeto que consiste en instancias de *Elemento*, que representa entidades que el robot ha identificado, pero aun no ha asignado en su vista del mundo. Estos elementos se marcan como estado global del robot.

En el instante representado, *m* está enlazado a dos instancias de *Área*. Una de ellas (*a2*) se muestra con sus propios enlaces a tres objetos *Pared* y un objeto *Puerta*. Cada una de estas paredes está etiquetada con su anchura actual, y cada una se muestra enlazada a sus paredes vecinas. Como sugiere este diagrama de objetos, el robot ha reconocido el área que lo contiene, que tiene paredes en tres lados y una puerta en el cuarto.

5

Modelado del comportamiento

Interacciones

En cualquier sistema, los objetos interactúan entre sí pasándose mensajes. Una interacción es un comportamiento que incluye un conjunto de mensajes intercambiados por un conjunto de objetos dentro de un contexto para lograr un propósito.

Las interacciones se utilizan para modelar los aspectos dinámicos de las colaboraciones, que representan sociedades de objetos que juegan roles específicos, y colaboran entre sí para llevar a cabo un comportamiento mayor que la suma de los comportamientos de sus elementos. Estos roles representan instancias prototípicas de clases, interfaces, componentes, nodos y casos de uso. Los aspectos dinámicos se visualizan, se especifican, se construyen y se documentan como flujos más complejos que impliquen bifurcaciones, iteraciones, recursión y concurrencia. Cada iteración puede modelarse de dos formas: bien destacando la ordenación temporal de los mensajes, bien destacando la secuencia de mensajes en el contexto de una organización estructural de objetos. Las interacciones bien estructuradas son como los algoritmos bien estructurados: eficientes, sencillos, adaptables y comprensibles.

Los sistemas con gran cantidad de software reaccionan dinámicamente a eventos o estímulos externos. Por ejemplo, el sistema de una compañía aérea puede manejar gran cantidad de información que normalmente se encuentra almacenada en un disco sin ser accedida, hasta que algún evento externo los ponga en acción, como, por ejemplo, una reserva, el movimiento de un avión o la programación de un vuelo. En los sistemas reactivos, como puede ser el sistema de un procesador de un horno microondas, la creación de objetos y el trabajo se llevan a cabo cuando se estimula el sistema con un evento, tal como la pulsación de un botón o el paso del tiempo.

En UML, los aspectos estáticos de un sistema se modelan mediante elementos tales como los diagramas de clases y los diagramas de objetos. Estos diagramas permiten especificar, construir y documentar los elementos del sistema, incluyendo clases, interfaces, componentes, nodos y casos de uso e instancias entre ellos, así como la forma en que estos elementos se relacionan entre sí.

En UML los aspectos dinámicos de un sistema se modelan mediante interacciones. Al igual que un diagrama de objetos, una interacción tiene una naturaleza estática, ya que establece el escenario para un comportamiento del sistema introduciendo todos los objetos que colaboran para realizar alguna acción. Pero, a diferencia de los diagramas de objetos, las interacciones incluyen mensajes enviados entre objetos. La mayoría de las veces, un mensaje implica la invocación de una operación o el envío de una señal. Un mensaje también puede incluir la creación o la destrucción de objetos.

Las interacciones se usan para modelar el flujo de control dentro de una operación, una clase, un componente, un caso de uso o el propio sistema. Con los diagramas de interacción, se puede razonar de dos formas sobre estos flujos. Una posibilidad es centrarse en como fluyen estos mensajes a lo largo del tiempo. La otra posibilidad es centrarse en las relaciones estructurales entre los objetos de una interacción y después considerar como se pasan los mensajes en el contexto de esa estructura

UML proporciona una representación gráfica para los mensajes, esta notación permite visualizar un mensaje de forma que podamos destacar todas sus partes más importantes: nombre, parámetros (si los tiene) y secuencia. Gráficamente un mensaje se representa como una línea dirigida y casi siempre incluye el nombre de su operación.

Una **interacción** es un comportamiento que comprende un conjunto de mensajes intercambiados entre un conjunto de objetos dentro de un contexto para lograr un propósito.

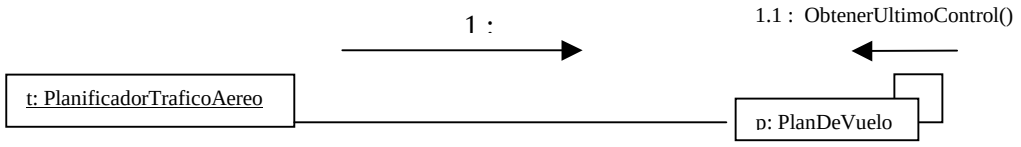


Figura 42. Mensajes, enlaces y secuenciamiento

Un **mensaje** es la especificación de una comunicación entre objetos que transmite información, con la expectativa de que se desencadenará una actividad.

Contexto

Una interacción puede aparecer siempre que unos objetos estén enlazados a otros. Las interacciones aparecerán en la colaboración de objetos existentes en el contexto de un sistema o subsistema. También aparecerán interacciones en el contexto de una operación y por último, aparecerán interacciones en el contexto de una clase.

Con mucha frecuencia, aparecerán interacciones en la colaboración de objetos existentes en el contexto de un sistema o subsistema. Por ejemplo, en un sistema de comercio electrónico sobre web, en el cliente habrá objetos que interactúen entre sí (por ejemplo, instancias de las clases *PedidoDeLibro* y *FormularioDePedido*). También habrá objetos en el cliente (de nuevo instancias de la clase *PedidoDeLibro*) que interactúen con objetos del servidor (por ejemplo instancias de *GestorDePedidos*). Por ello estas interacciones no sólo implican colaboraciones locales de objetos (como las que hay entorno a la clase *FormularioDePedido*), sino que también podrían atravesar muchos niveles conceptuales del sistema (como las interacciones que tienen que ver con *GestorDePedidos*)

También aparecerán interacciones entre objetos en la implementación de una operación. Los parámetros de una operación, sus variables locales y los objetos globales a la operación (pero visibles a ella) pueden interactuar entre sí para llevar a cabo un algoritmo de esa implementación de la operación. Por ejemplo, la invocación de la operación *moverAPosicion(p: Posición)* definida en la clase de un robot móvil implicará la interacción de un parámetro (*p*), un objeto global a la operación (por ejemplo, el objeto *posicionActual*) y posiblemente varios objetos locales (como por ejemplo las variables necesarias para calcular la posición)

Por último, aparecerán interacciones en el contexto de una clase. Las interacciones se pueden utilizar para visualizar, especificar, constituir y documentar la semántica de una clase, por ejemplo comprender el significado de la clase *AgenteTrazadoRayos*, podríamos crear interacciones que mostrasen como colaboran entre sí los atributos de la clase (y los objetos globales con los parámetros definidos en las operaciones de la clase).

Objetos y Roles

Los objetos que participan en una interacción son o bien elementos concretos o bien elementos prototípicos. Como elemento concreto, un objeto representa algo del mundo real. Por ejemplo, *p*, una instancia de la clase *Persona*, podría denotar a una persona particular. Alternativamente, como elemento prototípico, *p* podría representar cualquier instancia de la clase *Persona*.

En el contexto de una interacción podemos encontrar instancias de clases, componentes, nodos y casos de uso. Aunque las clases abstractas y las interfaces, por definición, no pueden tener instancias directas, podríamos encontrar instancias de esos elementos en una interacción. Tales instancias no representarán instancias directas de la clase abstracta o interfaz, pero podrían, respectivamente, instancias indirectas (o prototípicas) de cualquier clase hija concreta de la clase abstracta o de alguna clase concreta que realice el interfaz.

Un diagrama de objetos se puede ver como una representación del aspecto estático de una interacción, que configura el escenario para la interacción al especificar todos los objetos que colaboran entre sí. Una interacción va más allá al introducir una secuencia dinámica de mensajes que pueden fluir a través de los enlaces que conectan a esos objetos

Enlaces

Un enlace es una conexión semántica entre objetos. En general, un enlace es una instancia de una asociación. Como se muestra en la figura 43, siempre que una clase tenga una asociación con otra clase, podría existir un enlace entre las instancias de dos clases; siempre que haya un enlace entre dos objetos, un objeto puede mandar un mensaje al otro.

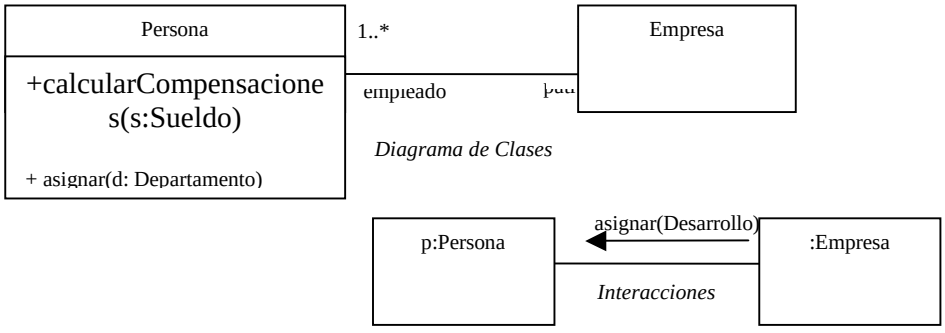


Figura 43. Visión Diagrama de Clases / Diagrama de Interacciones

Mensajes

Supóngase que se dispone de un conjunto de objetos y un conjunto de enlaces que conectan esos objetos. Si esto es todo lo que se tiene, entonces se está ante un modelo completamente estático que puede representarse mediante un diagrama de objetos. Los diagramas de objetos modelan el estado de una sociedad de objetos en un momento dado y son útiles cuando se quiere visualizar, especificar, construir o documentar una estructura de objetos estática.

Supóngase que se quiere modelar los cambios de estado en una sociedad de objetos a lo largo de un periodo de tiempo. Se puede pensar en esto como en el rodaje de una película sobre un conjunto de objetos, donde los fotogramas representan momentos sucesivos en la vida de los objetos. Si estos no son totalmente pasivos, se verá como se pasan mensajes de unos a otros, cómo se envían eventos y cómo invocan operaciones. Además, en cada fotograma, se puede mostrar explícitamente el estado actual y el rol de cada instancia individual.

Un mensaje es la especificación de una comunicación entre objetos que transmite información, con la expectativa de que se desencadenará alguna actividad. La recepción de una instancia de un mensaje puede considerarse la instancia de un evento.

Cuando se pasa un mensaje, la acción resultante es una instrucción ejecutable que constituye una abstracción de un procedimiento computacional. Una acción puede producir un cambio en el estado.

En UML se modelan varios tipos de acciones:

- *Llamada*. Invoca una operación sobre un objeto; un objeto puede enviarse un mensaje a sí mismo, lo que resulta en la invocación local de una operación.
- *Retorno*. Devuelve un valor al invocador.
- *Envío*. Envía una señal a un objeto.
- *Creación*. Crea un objeto.
- *Destrucción*. Destruye un objeto; un objeto puede "suicidarse" al destruirse a sí mismo.

En la figura 44 se muestra un ejemplo donde se utilizan diferentes tipos de mensajes.

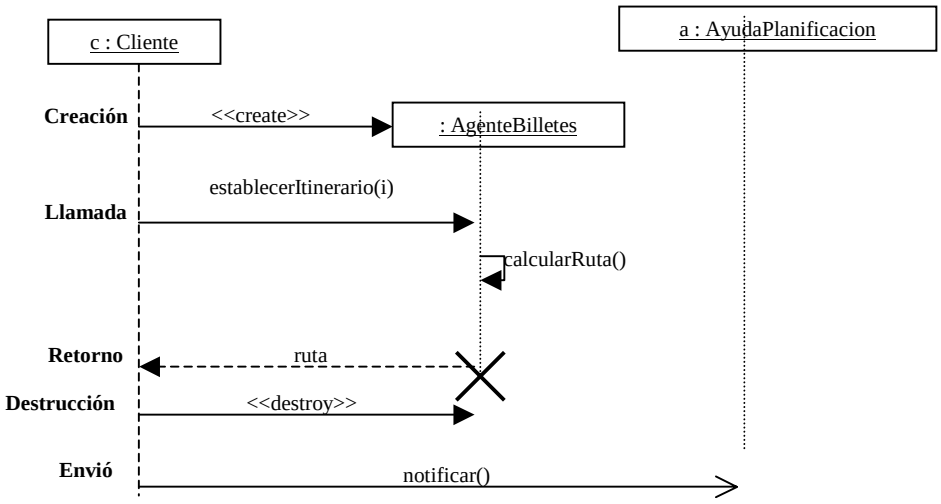


Figura 44. Mensajes

Los diagramas de secuencia y colaboración son prácticamente isomorfos, es decir, se puede partir de uno de ellos y transformarlo en el otro sin pérdida de información. No obstante, hay algunas diferencias visuales. Por una parte, los diagramas de secuencia permiten modelar la línea de vida de un objeto. La línea de vida de un objeto representa la existencia de un objeto durante un período de tiempo, posiblemente cubriendo la creación y la destrucción del objeto. Por otra parte, los diagramas de colaboración permiten modelar los enlaces estructurales que pueden existir entre los objetos de la interacción.

Modelado de un flujo de control

La mayoría de las veces, las interacciones se utilizan con el propósito de modelar el flujo de control que caracteriza el comportamiento de un sistema, incluyendo casos de uso, patrones, mecanismos y *frameworks*, o el comportamiento de una clase o una operación individual. Mientras que las clases, las interfaces, los componentes, los nodos y sus relaciones modelan los aspectos estáticos del sistema, las interacciones modelan los aspectos dinámicos.

Cuando se modela una interacción, lo que se hace esencialmente es construir una representación gráfica de las acciones que tienen lugar entre un conjunto de objetos. Para modelar un flujo de control:

- Hay que establecer el contexto de la interacción, o sea, si el sistema es global, si estamos ante una clase o ante una operación individual.

- Hay que establecer el escenario para la interacción, identificando qué objetos juegan un rol; se deben establecer sus propiedades iniciales, incluyendo los valores de sus atributos, estado y rol.

- Si el modelo destaca la organización estructural de esos objetos hay que identificar los enlaces que los conectan y que sean relevantes para los trayectos de comunicación que tiene lugar en la interacción. Si es necesario, hay que especificar la naturaleza de los enlaces con estereotipos estándar y las restricciones de UML.

- Hay que especificar los mensajes que pasan de un objeto a otro mediante una organización temporal. Si es necesario, hay que distinguir los diferentes tipos de mensajes; se deben incluir parámetros y valores de retorno para expresar los detalles necesarios de la interacción.

- Para expresar los detalles necesarios de la interacción, hay que adornar cada objeto con su estado y su rol siempre que sea preciso.

-

En la figura 45 se representa un conjunto de objetos que interactúan en el contexto de un mecanismo de publicación y suscripción. Incluye tres objetos *r* (un *ResponsablePrecioStock*), *s1* y *s2* (ambos instancias de *SuscritorPrecioStock*). Esta figura es un ejemplo de diagrama de secuencia, que resalta la ordenación temporal de los mensajes.

La figura 46 es semánticamente equivalente a la figura anterior, pero ha sido dibujada como un diagrama de colaboración, en el cual destaca la organización estructural de los objetos. Esta figura representa el mismo flujo de control, pero también muestra los enlaces entre esos objetos.

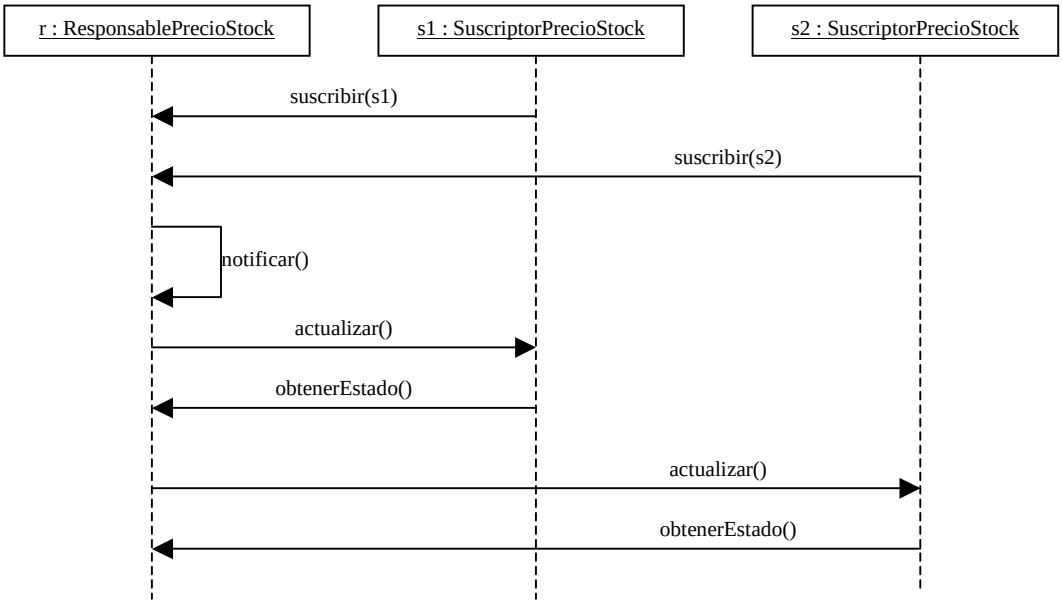


Figura 45. Flujo de Control en el Tiempo

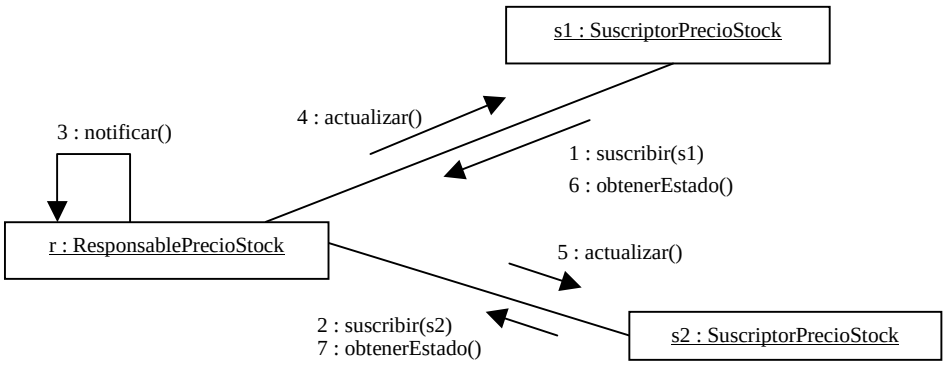


Figura 46. Flujo de Control por Organización

Casos de Uso

Ningún sistema se encuentra aislado. Cualquier sistema interesante interactúa con actores humanos o mecánicos que lo utilizan con algún objetivo y que esperan que el sistema funcione de forma predecible. Un caso de uso especifica el comportamiento de un sistema o una parte del mismo, y es una descripción de un conjunto de secuencias de acciones, incluyendo variantes, que ejecutan un sistema para producir un resultado observable de valor para un actor.

Los casos de uso se emplean para capturar el comportamiento deseado del sistema en desarrollo, sin tener que especificar como se implementa ese comportamiento. Los casos de uso proporcionan un medio para que los desarrolladores, los usuarios finales del sistema y los expertos del dominio lleguen a una comprensión común del sistema. Además, los casos de uso ayudan a validar la arquitectura y a verifica el sistema mientras evoluciona a lo largo del desarrollo. Conforme se desarrolla el sistema, los

casos de uso son realizados por colaboraciones, cuyos elementos cooperan para llevar a cabo cada caso de uso.

Un factor clave al definir casos de uso es que no se especifica como se implementan. Por ejemplo, puede especificarse cómo se debería comportar un cajero automático enunciando mediante casos de uso cómo interactúan los usuarios con el sistema; pero no se necesita saber nada del interior del cajero. Los casos de uso especifican un comportamiento deseado, no imponen cómo se llevará a cabo ese comportamiento. Lo más importante es que permiten que los usuarios finales y los expertos del dominio se comuniquen con los desarrolladores sin entrar en detalles. Estos detalles llegarán, pero los casos de uso permiten centrarse en cuestiones más importantes para el usuario final.

En UML todos los comportamientos se modelan como casos de uso que pueden especificarse independientemente de su realización. *Un caso de uso es una descripción de un conjunto de secuencias de acciones, incluyendo variantes, que ejecuta el sistema para producir un resultado observable.* En esta definición hay varias partes importantes.

Un caso de uso describe un conjunto de secuencias, donde cada secuencia representa una interacción de los elementos externos al sistema (sus actores) con el propio sistema (y con sus abstracciones clave). En realidad, estos comportamientos son funciones al nivel de sistema que se utilizan durante la captura de requisitos y el análisis para visualizar, especificar, construir y documentar el comportamiento esperado del sistema. ***Un caso de uso representa un requisito funcional del sistema***. Por ejemplo, un caso de uso fundamental en un banco es el procesamiento de prestamos.

Un caso de uso involucra la interacción de actores y el sistema. Un actor representa un conjunto coherente de roles que juegan los usuarios de los casos de uso al interactuar con estos. Los actores pueden ser personas o pueden ser sistemas mecánicos. Por ejemplo, en el modelado de un banco, el procesamiento de un préstamo implica, entre otras cosas, la interacción entre un cliente y un responsable de préstamos.

Un caso de uso puede tener variantes. En cualquier sistema interesante se pueden encontrar casos de uso que son versiones especializadas de otros casos de uso, casos de uso incluidos como parte de otros, y casos de uso que extienden el comportamiento de otros casos de usos básicos. Se puede factorizar el comportamiento común y reutilizable de un conjunto de casos de uso organizándolos según estos tres tipos de relaciones. Por ejemplo, cuando se modela un banco aparecen muchas variaciones del caso de uso básico de procesar un préstamo, tales como las diferencias entre procesar una gran hipoteca frente a un pequeño préstamo comercial. En cada caso, sin embargo, estos casos de uso comparten algo de comportamiento, como el caso de uso de aprobar el préstamo para ese cliente, un comportamiento que es parte del procesamiento de cualquier tipo de préstamo.

Un caso de uso realiza cierto trabajo cuyo efecto es tangible. Desde la perspectiva de un actor determinado, un caso de uso produce algo de valor para algún actor, como el cálculo de un resultado, la generación de un nuevo objeto, o el cambio del estado de otro objeto. Por ejemplo, en el modelado de un banco, el procesamiento de un préstamo produce un préstamo aceptado, que se concreta en una cantidad de dinero entregada al cliente.

Los casos de uso se pueden aplicar al sistema completo. También se pueden aplicar a partes del sistema, incluyendo subsistemas e incluso clases e interfaces individuales. En cada caso, estos casos de uso no sólo representan el comportamiento esperado de estos elementos, sino que también pueden utilizarse como la base para establecer casos de prueba para esos elementos mientras evolucionan durante el desarrollo del sistema.

La representación gráfica de un caso de uso es una elipse que encierra el nombre, que normalmente suele ser una expresión verbal que describe algún comportamiento del vocabulario del sistema que se

está modelando. Los actores se representan mediante un muñeco muy sencillo el cual suele tener asociado un nombre que explica el rol de ese actor frente al sistema.

Casos de uso y actores

Un actor representa un conjunto coherente de roles que los usuarios de los casos de uso juegan al interactuar con éstos. Normalmente, un actor representa un rol que es jugado por una persona, un dispositivo hardware o incluso otro sistema al interactuar con nuestro sistema. Por ejemplo, si una persona trabaja para un banco, podría ser un *ResponsablePrestamos*. Si tiene sus cuentas personales en ese banco, está jugando también el rol de *Cliente*. Una instancia de un actor, por lo tanto, representa una interacción individual con el sistema de una forma específica. Aunque se utiliza actores en los modelos, éstos no forman parte del sistema. Son externos a él.

Como se muestra en la figura 47, se pueden representar categorías de actores más generales (como *Cliente*) y especializarlos (como *ClienteComercial*) a través de relaciones de generalización.

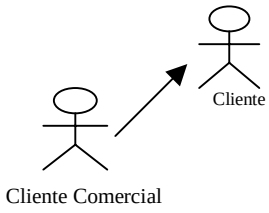


Figura 47. Actores

Los actores sólo se pueden conectar a los casos de uso a través de asociaciones. Una asociación entre un actor y un caso de uso indica que el actor y ese caso de uso se comunican entre sí, y cada uno puede enviar y recibir mensajes.

Casos de uso y flujo de eventos

Un caso de uso describe *qué* hace un sistema (o subsistema, una clase o interfaz), pero no específica *cómo* lo hace. Cuando se modela, es importante tener clara la separación de objetivos entre las vistas externa e interna.

El comportamiento de un caso de uso se puede especificar describiendo un flujo de eventos de forma textual, lo suficientemente claro para que alguien ajeno al sistema lo entienda fácilmente. Cuando se escribe este flujo de eventos se debe incluir cómo y cuándo empieza y acaba el caso de uso, cuándo interactúa con los actores y qué objetos se intercambian, el flujo básico y los flujos alternativos del comportamiento.

Por ejemplo, en el contexto de un cajero automático, se podría describir el caso de uso *ValidarUsuario* de la siguiente forma:

- *Flujo de eventos principal*: El caso de uso empieza cuando el sistema pide al *Cliente* un número de identificación personal (PIN, Personal Identification Number). El *Cliente* puede introducir un PIN a través del teclado. El *Cliente* acepta la entrada pulsando el botón Enter. El sistema comprueba este PIN para ver si es válido. Si el PIN es válido, el sistema acepta la entrada, y así acaba el caso de uso.

- *Flujo de eventos excepcional*: El *Cliente* puede cancelar una transacción en cualquier momento pulsando el botón cancelar, reiniciando de esta forma el caso de uso. No efectúa ningún cambio en la cuenta del *Cliente*.

- *Flujo de eventos excepcional*: El *Cliente* puede borrar un PIN en cualquier momento antes de introducirlo, y volver a teclear un nuevo PIN.

- *Flujo de eventos excepcional*: Si el *Cliente* introduce un PIN inválido, el caso de uso vuelve a empezar. Si esto ocurre tres veces en una sesión, el sistema cancela la transacción completa, impidiendo que el *Cliente* utilice el cajero durante 60 segundos.

El flujo de eventos de un caso de uso se puede especificar de muchas formas, incluyendo texto estructurado informal como en el ejemplo, texto estructurado formal (con pre y poscondiciones) y pseudocódigo.

Casos de uso y escenarios

Normalmente, primero se describe el flujo de eventos de un caso de uso mediante texto. Sin embargo, conforme se mejora la comprensión de los requisitos del sistema estos flujos se pueden especificar gráficamente mediante diagramas de interacción. Normalmente, se usa un diagrama de secuencia para especificar el flujo principal de un caso de uso, y se usan variaciones de ese diagrama para especificar los flujos excepcionales del caso de uso.

Conviene separar el flujo principal de los flujos alternativos, porque un caso de uso describe un conjunto de secuencias, no una única secuencia, y seria imposible expresar todos los detalles de un caso de uso no trivial en una única secuencia diferente.

Este caso de uso (*Contratar Empleado*), en realidad describe un conjunto de secuencias, donde cada secuencia representa un posible flujo a través de todas las variantes. Cada secuencia se denomina escenario. Un escenario es una secuencia específica de acciones que ilustra un comportamiento. Los escenarios son a los casos de uso lo que las instancias a las clases, es decir, un escenario es básicamente una instancia de un caso de uso.

Casos de uso y colaboraciones

Un caso de uso captura el comportamiento esperado del sistema (o subsistema, clase o interfaz) que está desarrollando, sin tener que especificar cómo se implementa ese comportamiento. Esta separación es importante porque el análisis de un sistema (que especifica un comportamiento) no debería estar influenciado, mientras sea posible, por cuestiones de implementación (que especifican cómo se lleva a cabo el comportamiento). No obstante, un caso de uso debe implementarse al fin y al cabo, y esto se hace creando una sociedad de clases y otros elementos que colaborarán para llevar a cabo el comportamiento del caso de uso. Esta sociedad de elementos, incluyendo tanto su estructura estática como la dinámica, se modela en UML como una ***colaboración***.

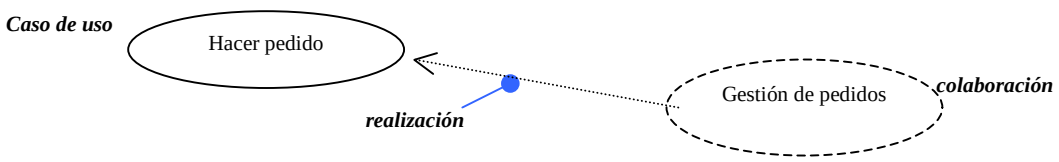


Figura 48. Casos de uso y colaboraciones

Como se muestra en la figura 48, la realización de un caso de uso puede especificarse explícitamente mediante una colaboración. Pero aunque la mayoría de las veces, un caso de uso es realizado exactamente por una colaboración, no será necesario mostrar explícitamente esta relación.

Modelado del Comportamiento de un Elemento

La mayoría de las veces, los casos de uso se utilizan para el modelado del comportamiento de un elemento, ya sea un sistema completo, un subsistema, o una clase. Cuando se modela el comportamiento de estos elementos, es importante centrarse en lo que hace el elemento, no en cómo lo hace.

Es importante aplicar de esta forma los casos de uso a los elementos por tres razones. Primera, el modelado del comportamiento de un elemento, mediante los casos de uso, permite a los expertos del domino especificar su vista externa del sistema a nivel suficiente para que los desarrolladores construyan su vista interna. Los casos de uso proporcionan un foro en el que pueden intercambiar opiniones los expertos del dominio, los usuarios finales y los desarrolladores. Segunda, los casos de uso proporcionan a los desarrolladores una forma de abordar y comprender un elemento. Un sistema, subsistema o una clase pueden ser complejos y estar repletos de operaciones y otras partes. Cuando se especifican los casos de uso de un elemento, se ayuda a que los usuarios de ese elemento lo puedan abordar directamente, de acuerdo con el modo en que ellos utilizarán el sistema. En ausencia de estos casos de uso, los usuarios tendrían que descubrir por su cuenta cómo usar el elemento. Los casos de uso permiten que el creador de un elemento comunique su intención sobre cómo se debería usar. Tercera, los casos de uso sirven de base para probar cada elemento según evolucionan durante el desarrollo. Al probar continuamente cada elemento frente a sus casos de uso, se está validando su implementación a lo largo del desarrollo. Estos casos de uso no sólo sirven como punto de partida para las pruebas de regresión, sino que cada vez que se añade un caso de uso a un elemento, hay que reconsiderar su implementación, para asegurarse de que ese elemento es flexible al cambio. Si no lo es, la arquitectura debe reorganizarse del modo adecuado. Para modelar el comportamiento de un elemento:

- Hay que identificar los actores que interactúan con el elemento. Los actores candidatos pueden incluir grupos que requieran un cierto comportamiento para ejecutar sus tareas o que se necesiten directa o indirectamente para ejecutar las funciones del elemento.

- Hay que organizar los actores identificando tanto los roles más generales como los más especializados.

- Hay que considerar las formas más importantes que tiene cada actor de interactuar con el elemento. También deben considerarse las interacciones que implican el cambio de estado del elemento o su entorno o que involucran una respuesta ante algún evento.

- Hay que considerar también las formas excepcionales en las que cada actor puede interactuar con el elemento.

- Hay que organizar estos comportamientos como casos de uso, utilizando reglas de inclusión y extensión para factorizar el comportamiento común y distinguir el comportamiento excepcional.

Por ejemplo, un sistema de venta interactúa con clientes, que efectuarán pedidos y querrán llevar un seguimiento. A su vez, el sistema enviará todos los pedidos y facturas al cliente. Como se muestra en la figura 49, el comportamiento de ese sistema se puede modelar declarando estos comportamientos

como casos de uso (*Hacer pedido, Seguir pedido, Enviar Pedido, Facturar al Cliente*). El comportamiento puede factorizarse (*Validar Cliente*) y también pueden distinguirse sus variantes (*Enviar pedido parcial*). Para cada caso de uso se incluirá una especificación del comportamiento, ya sea a través de texto, una máquina de estados o interacciones.

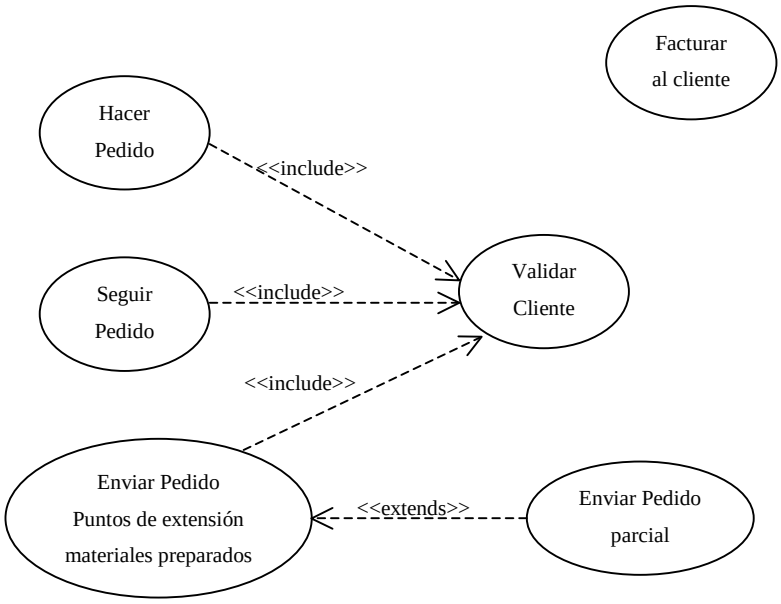


Figura 49. Modelado del Comportamiento de un elemento.

6

Diagramas para el modelado del comportamiento

En este tema vamos a estudiar que tipos de diagramas se utilizan en UML para modelar el comportamiento de un sistema, subsistema, clase o interfaz. De los cinco tipos de diagramas que se utilizan en UML con este fin, diagramas de casos de uso, de secuencia, de colaboración, de actividades y de estados, únicamente nos dedicaremos al estudio de los cuatro primeros, ya que para estudiar los diagramas de estado (o maquinas de estados) es necesario profundizar en el estudio de los elementos avanzados para el modelado de la parte dinámica de los sistemas (eventos y señales) lo que para nosotros está fuera del objetivo de este curso de Diseño Orientado a Objetos con UML, el cual se centra en servir de iniciación a las técnicas de modelado orientado a objetos aplicando el rigor y la metodología de UML.

Los diagramas de casos de uso se emplean para modelar la vista de casos de uso de un sistema, la mayoría de las veces esto implica modelar el contexto del sistema, subsistema o clase, o el modelado de los requisitos de comportamiento de esos elementos.

Los diagramas de interacción (incluye los diagramas de secuencia y de colaboración) se utilizan para modelar los aspectos dinámicos un sistema. La mayoría de las veces esto implica tener que modelar instancias concretas o prototípicas de clases, interfaces, componentes y nodos, junto con los mensajes enviados entre ellas, todo en el contexto de un escenario que ilustra un comportamiento.

Los diagramas de actividades se centran fundamentalmente en mostrar el flujo de control entre diferentes actividades. Se suelen utilizar para modelar los aspectos dinámicos de un sistema. La mayoría de las veces esto implica modelar pasos secuénciales (y posiblemente también concurrentes) de un proceso computacional. Mediante los diagramas de actividades también se puede modelar el flujo de un objeto conforme pasa de estado a estado en diferentes puntos del flujo de control.

Diagramas de Casos de Uso

Los diagramas de casos de uso se emplean para visualizar el comportamiento de un sistema, un subsistema o una clase, de forma que los usuarios puedan comprender cómo utilizar ese elemento y de forma que los desarrolladores puedan implementarlo. La figura 50 muestra un diagrama de casos de uso para modelar el comportamiento de un teléfono móvil.

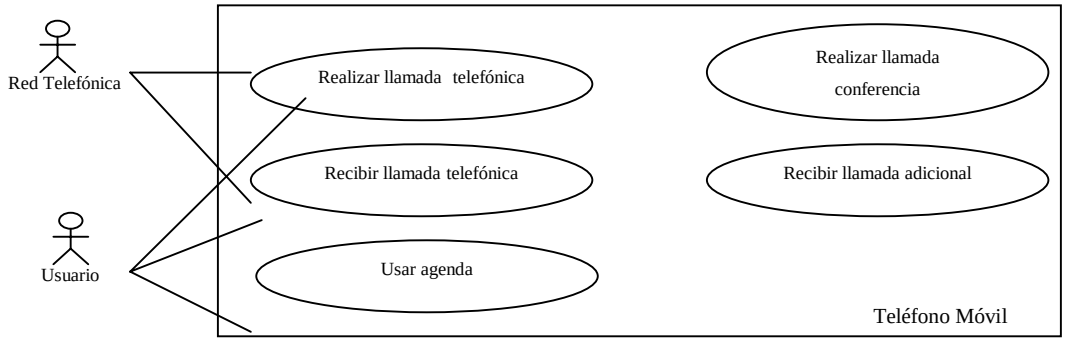


Figura 50. Ejemplo de Diagrama de Casos de Uso

Los diagramas de casos de uso muestran un conjunto de casos de uso, actores y sus relaciones, estas pueden ser relaciones de dependencia, generalización y asociación. También pueden contener paquetes, que se emplean para agrupar elementos del modelo en partes mayores.

Usos comunes

Los diagramas de casos de uso se emplean para modelar la vista de casos de uso estática de un sistema. Esta vista cubre principalmente el comportamiento del sistema (los servicios visibles externamente que proporciona el sistema en el contexto de su entorno). Cuando se modela la vista de casos de uso estática de un sistema, normalmente se emplearán los diagramas de casos de uso de una de las dos formas siguientes:

1. Para modelar el contexto de un sistema.

   Modelar el contexto de un sistema implica dibujar una línea alrededor de todo el sistema y asegurar qué actores quedan fuera del sistema e interactúan con él. Aquí, se emplearán los diagramas de casos de uso para especificar los actores y significado de sus roles.

2. Para modelar los requisitos de un sistema.

   El modelado de los requisitos de un sistema implica especificar qué debería hacer el sistema (desde un punto de vista externo), independientemente de cómo se haga. Aquí se emplearán los diagramas de casos de uso, para especificar el comportamiento deseado del sistema. De esta forma, un diagrama de casos de uso permite ver el sistema entero como una caja negra; se puede ver qué hay fuera del sistema y cómo reacciona a los elementos externos, pero no se puede ver cómo funciona por dentro.

Modelado del contexto de un sistema

Dado un sistema, algunos elementos se encuentran dentro de él y otros fuera. Por ejemplo, en un sistema de validación de tarjetas de crédito existen elementos como cuentas, transacciones, agentes de detección de fraudes, etc. que están dentro del sistema. También existen cosas como clientes de tarjetas de crédito y comercios que están fuera del sistema. Los elementos de un sistema son responsables de llevar a cabo el comportamiento que esperan los elementos externos. Todos estos elementos externos que interactúan con el sistema constituyen su contexto. Este contexto define el entorno en el que reside el sistema.

La decisión sobre qué incluir como actores es importante, porque al hacer eso se especifica un tipo de cosas que interactúan con el sistema. La decisión sobre que no incluir es igualmente importante, si no más, porque restringe el entorno para que sólo incluya aquellos actores necesarios en la vida del sistema. Para modelar el contexto de un sistema:

- Hay que identificar los actores en torno al sistema, considerando qué grupos requieren ayuda del sistema para llevar a cabo sus tareas; qué grupos son necesarios para ejecutar las funciones del sistema; qué grupos interactúan con el hardware externo o con otros sistemas software; y qué grupos realizan funciones secundarias de administración y mantenimiento.

- Hay que organizar los actores similares en jerarquías de generalización / especialización.

- Hay que proporcionar un estereotipo para cada uno de esos actores, si así se ayuda a entender el sistema.

- Hay que introducir esos actores en un diagrama de casos de uso y especificar las vías de comunicación de cada actor con cada uno de los casos de uso del sistema.

Por ejemplo, la figura 51 muestra el contexto de un sistema de validación de tarjetas de crédito, destacando los actores en torno al sistema. Se puede ver que existen *Clientes*, de los cuales hay dos categorías (*Cliente individual* y *Cliente corporativo*). Estos actores representan los roles que juegan las personas que interactúan con el sistema. En este contexto, también hay actores que representan a otras instituciones tales como *Comercio* (que es donde los *Clientes* realizan una transacción con tarjeta para comprar un articulo o servicio) y *Entidad Financiera* (que presta servicio como sucursal bancaria para la cuenta de la tarjeta de crédito). En el mundo real estos dos actores probablemente sean sistemas con gran cantidad de software.

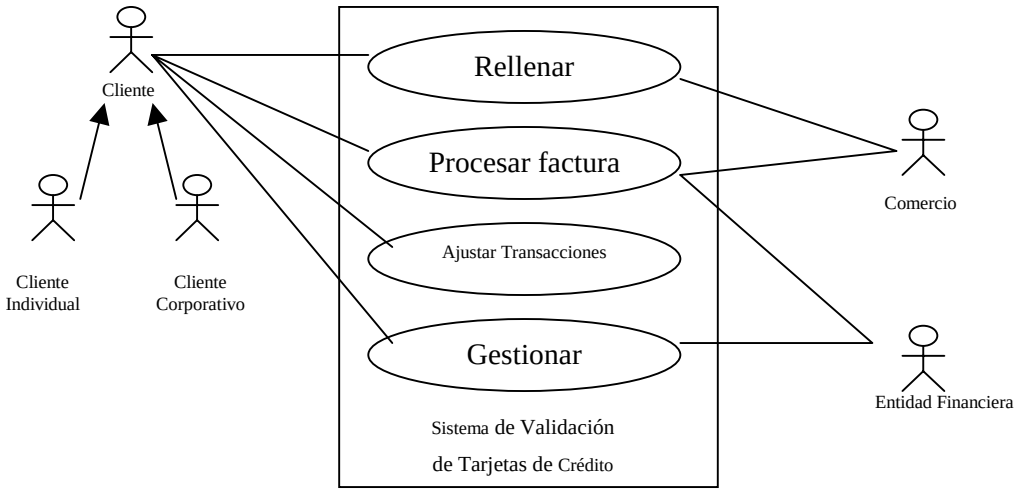


Figura 51. Modelado del Contexto de un Sistema

Modelado de los requisitos de un sistema

Un requisito es una característica de diseño, una propiedad o un comportamiento de un sistema. Cuando se enuncian los requisitos de un sistema se está estableciendo un contrato entre los elementos externos al sistema y el propio sistema, que establece lo que se espera que haga el sistema. La mayoría de las veces no importa cómo lo hace, sólo importa *que* lo hace. Un sistema con un comportamiento correcto llevará a cabo todos sus requisitos de manera fiel, predecible y fiable. Al construir un sistema, es importante que al comenzar exista un acuerdo sobre qué debería hacer el sistema, aunque, con toda seguridad, la comprensión de los requisitos evolucionará conforme se vaya implementando el sistema de manera iterativa e incremental. Análogamente, cuando se le proporciona un sistema a alguien para que los use, es esencial saber cómo se comporta para utilizarlo correctamente.

Los requisitos se pueden expresar de varias formas, desde texto sin estructurar hasta expresiones en lenguaje formal, y en cualquier otra forma intermedia. La mayoría de los requisitos funcionales de un sistema, si no todos, se pueden expresar con casos de uso, y los diagramas de casos de uso de UML son fundamentales para manejar esos requisitos. Para modelar los requisitos de un sistema:

- Hay que establecer el contexto del sistema, identificando los actores a su alrededor.
- Hay que considerar el comportamiento que cada actor espera del sistema o requiere que éste le proporcione.
- Hay que nombrar esos comportamientos comunes como casos de uso.
- Hay que factorizar el comportamiento común en nuevos casos de uso que puedan ser utilizados por otros; Hay que factorizar el comportamiento variante en otros casos de uso que extiendan los flujos principales.
- Hay que adornar esos casos de uso con notas que enuncien los requisitos no funcionales; puede que haya que asociar varias de esas notas al sistema global.

La figura 52 extiende la anterior (figura 51), aunque omite las relaciones entre los actores y los casos de uso, añade casos de uso adicionales que son invisibles para el cliente normal, aunque son comportamientos fundamentales del sistema. Este diagrama es valioso porque ofrece un punto de vista común para los usuarios finales, los expertos del dominio y los desarrolladores para visualizar, especificar, construir y documentar sus decisiones sobre los requisitos funcionales del sistema. Por ejemplo, *Detectar fraude de tarjeta* es un comportamiento importante tanto para el *Comercio* como para la *Entidad Financiera*. Análogamente, *Informe estado de Cuentas,* es otro comportamiento requerido del sistema por varias entidades.

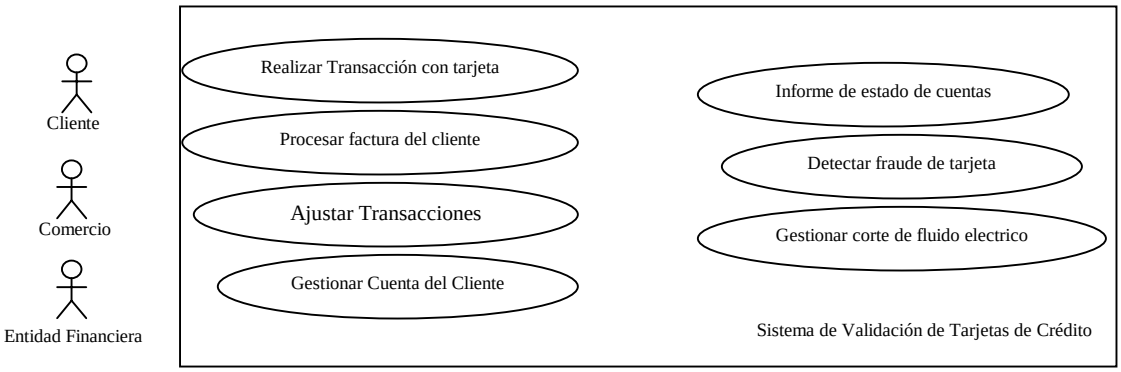


Figura 52. Modelado de los requisitos de un sistema

El requisito modelado por el caso de uso *Gestionar Corte de Fluido Eléctrico* es un poco diferente de los demás, porque representa un comportamiento secundario del sistema necesario para un funcionamiento fiable y continuo.

Diagramas de Interacción

Cuando se modelan sistemas con gran cantidad de software se tiene un problema importante: ¿Cómo modelar sus aspectos dinámicos?. Imaginemos, por un momento, cómo podría visualizarse un sistema en ejecución. Si se dispone de un depurador interactivo asociado al sistema, podría verse una sección de memoria y observar cómo cambia su contenido a lo largo del tiempo. Enfocando con más precisión, incluso se podría realizar un seguimiento de los objetos de interés. A lo largo del tiempo, se vería la creación de objetos, los cambios en el valor de sus atributos y la destrucción de algunos de ellos.

El valor de visualizar así los aspectos dinámicos de un sistema es bastante limitado, especialmente si se trata de un sistema distribuido con múltiples flujos de control concurrentes. También se podría intentar comprender el sistema circulatorio humano mirando la sangre que pasa a través de una arteria a lo largo del tiempo. Una forma mejor de modelar los aspectos dinámicos de un sistema es construir representaciones gráficas de escenarios que impliquen la interacción de ciertos objetos interesantes y los mensajes enviados entre ellos.

En UML, estas representaciones gráficas se modelan con los diagramas de interacción. Los diagramas de interacción se pueden construir de dos formas: destacando la ordenación temporal de los mensajes (diagramas de secuencia) y destacando la relación estructural de los objetos que interactúan (diagramas de colaboración), en cualquier caso, los dos tipos de diagramas son equivalentes. Los diagramas de interacción contienen objetos, enlaces y mensajes.

Diagramas de Secuencia

Un diagrama de secuencia destaca la ordenación temporal de los mensajes. Como se muestra en la figura 53, un diagrama de secuencia se forma colocando en primer lugar los objetos que participan en la interacción en la parte superior del diagrama, a lo largo del eje X. Normalmente, se coloca a la izquierda el objeto que inicia la interacción, y los objetos subordinados a la derecha. A continuación, se colocan los mensajes que estos objetos envían y reciben a lo largo del eje Y, en orden de sucesión en el tiempo, desde arriba hasta abajo. Esto ofrece al lector una señal visual clara del flujo de control a lo largo del tiempo.

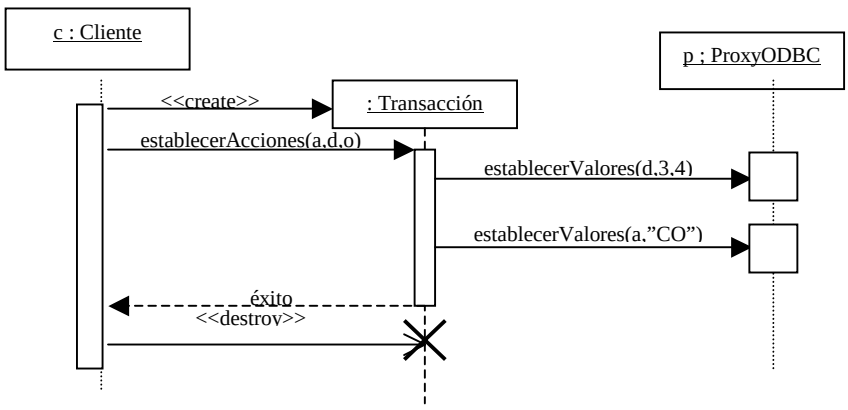


Figura 53. Diagramas de Secuencia

Los diagramas de secuencia tienen dos características que los distinguen de los diagramas de colaboración: En primer lugar, está la línea de vida de un objeto, es la línea vertical discontinua que representa la existencia de un objeto a lo largo de un periodo de tiempo. Pueden crearse objetos durante la interacción. Sus líneas de vida aparecen cuando reciben el mensaje estereotipado como <<*create*>>. Los objetos pueden destruirse durante la interacción. Sus líneas de vida acaban con la recepción del mensaje estereotipado como <<*destroy*>> (además se muestra la señal visual de una gran X que marca el fin su línea de vida). En segundo lugar está el foco de control que es un rectángulo estrecho situado sobre la línea de vida que representa el período de tiempo durante el cual un objeto ejecuta una acción, bien sea directamente o a través de un procedimiento subordinado.

Diagramas de Colaboración

Un diagrama de colaboración destaca la organización de los objetos que participan en una interacción. Como se muestra en la figura 54, un diagrama de colaboración se construye colocando en primer lugar los objetos que participan en la colaboración como nodos del grafo. A continuación se representan los enlaces que conectan esos objetos como arcos del grafo. Por último, estos enlaces se adornan con los mensajes que envían y reciben los objetos. Esto da al lector una señal visual cara del flujo de control en el contexto de la organización estructural de los objetos que colaboran.

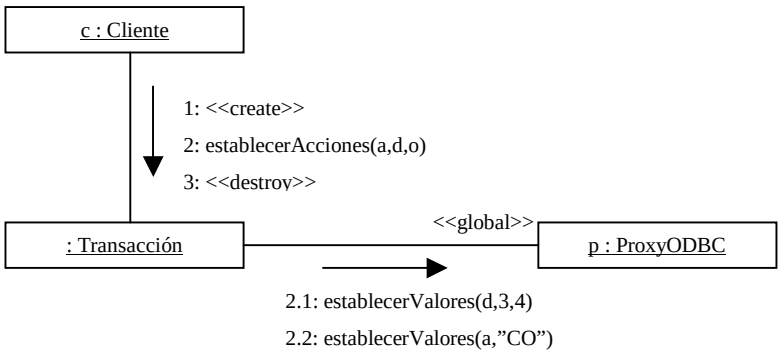


Figura 54. Diagrama de Colaboración

Los diagramas de colaboración tienen dos características que los distinguen de los diagramas de secuencia.

En primer lugar, el camino. Para indicar cómo se enlaza un objeto a otro, se puede asociar un estereotipo de camino al extremo más lejano de un enlace (como <<*local*>>, que indica que el objeto designado es local al emisor). Normalmente, sólo se necesita representar explícitamente el camino del enlace para los caminos *local, parameter, global* y *self* (pero no *association*).

En segundo lugar, está el número de secuencia. Para indicar la ordenación temporal de un mensaje, se precede de un número (comenzando con el mensaje número 1), que se incrementa secuencialmente por cada nuevo mensaje en el flujo de control (2, 3, etc.). Para representar el anidamiento, se utiliza la numeración decimal de Dewey (1 es el primer mensaje; 1.1 es el primer mensaje dentro del mensaje 1; 1.2 es el segundo mensaje dentro del mensaje 1; etc.). El anidamiento se puede representar a cualquier nivel de profundidad. Nótese también que, a través del mismo enlace, se pueden mostrar varios mensajes (posiblemente enviados desde distintas direcciones), y cada uno tendrá un número de secuencia único.

Modelado de flujos de control por ordenación temporal

Considérense los objetos existentes en el contexto de un sistema, un subsistema, una operación o una clase. Considérense también los objetos y roles que participan en un caso de uso o una colaboración. Para modelar un flujo de control que discurre entre esos objetos y roles se utiliza un diagrama de interacción; para destacar el paso de mensajes conforme se desarrollan en el tiempo se utiliza un diagrama de secuencia, un tipo de diagrama de interacción.

Para modelar un flujo de control por ordenación temporal:

- Hay que establecer el contexto de la interacción, bien sea un sistema, un subsistema, una operación, o una clase, o bien un escenario de un caso de uso o de una colaboración.

- Hay que establecer un escenario de la interacción, identificando qué objetos juegan un rol en ella. Los objetos deben organizarse en el diagrama de secuencia de izquierda a derecha, colocando los objetos más importantes a la izquierda y sus objetos vecinos a la derecha.

- Hay que establecer la línea de vida de cada objeto. En la mayoría de los casos los objetos persistirán la interacción completa. Para aquellos objetos creados y destruidos durante la interacción, hay que establecer sus líneas de vida, según sea apropiado, e indicar explícitamente su creación y destrucción con mensajes estereotipados apropiadamente.

- A partir del mensaje que inicia la interacción, hay que ir colocando los mensajes subsiguientes de arriba abajo entre las líneas de vida, mostrando las propiedades de cada mensaje (tales como sus parámetros), según sea necesario para explicar la semántica de interacción.

- Si es necesario visualizar el anidamiento de mensajes o el intervalo de tiempo en el que tiene lugar la computación, hay que adornar la línea de vida de cada objeto con su foco de control.

- Si es necesario especificar restricciones de tiempo o espacio, hay que adornar cada mensaje con una marca de tiempo y asociar las restricciones apropiadas.

- Si es necesario especificar este flujo de control más formalmente, hay que asociar pre y poscondiciones a cada mensaje.

Un único diagrama de secuencia sólo puede mostrar un flujo de control (aunque es posible mostrar variaciones sencillas utilizando la notación de UML para iteración y la bifurcación). Normalmente, se realizarán varios diagramas de interacción, algunos de los cuales serán los principales y los demás mostrarán caminos alternativos o condiciones excepcionales. Se pueden utilizar paquetes para organizar estas colecciones de diagramas de secuencia, dando a cada diagrama un nombre adecuado para distinguirlo del resto.

Por ejemplo, la figura 55 representa un diagrama de secuencia que especifica el flujo de control para iniciar una simple llamada telefónica entre dos partes. A este nivel de abstracción existen cuatro objetos involucrados: dos *Interlocutores* (*s* y *r*), una *Centralita* de teléfonos sin nombre, y *c*, la materialización de la *Conversación* entre ambas partes. La secuencia comienza cuando un *Interlocutor (s)* emite una señal (*descolgarAuricular*) al objeto *Centralita*. A su vez, la *Centralita* llama a *darTonoDeLlamada* sobre este *Interlocutor*, y el *Interlocutor* itera sobre el mensaje *marcarDígito*. Nótese que este mensaje tiene una marca temporal (*marcando*) que se utiliza en una restricción de tiempo (su *tiempoDeEjecución* debe ser menor de 30 segundos). Este diagrama no indica qué ocurre si se viola la restricción temporal. Para ello podría ejecutarse una bifurcación o un diagrama de secuencia totalmente separado. El objeto *Centralita* se llama a sí mismo con el mensaje *enrutarLlamada*. A continuación crea un objeto *Conversación (c)*, al cual delega el resto del trabajo. Aunque no se representa esta interacción, *c* tendrá la responsabilidad adicional de formar parte del sistema de

contabilidad de la centralita (lo cual se expresaría en otro diagrama de interacción). El objeto *Conversación (c)* llama al *Interlocutor (r)*, el cual envía asincrónicamente el mensaje *descolgarAuricular*. Entonces, el objeto *Conversación* indica a la *Centralita* que debe *conectar* la llamada, y luego indica a los dos objetos *Interlocutor* que pueden *conectar*, tras lo cual pueden intercambiar información, como se indica en la nota adjunta.

Un diagrama de interacción puede comenzar a acabar en cualquier punto de una secuencia.

Una traza completa del flujo de control sería increíblemente compleja, de forma que es razonable dividir partes de un flujo mayor en diagramas separados.

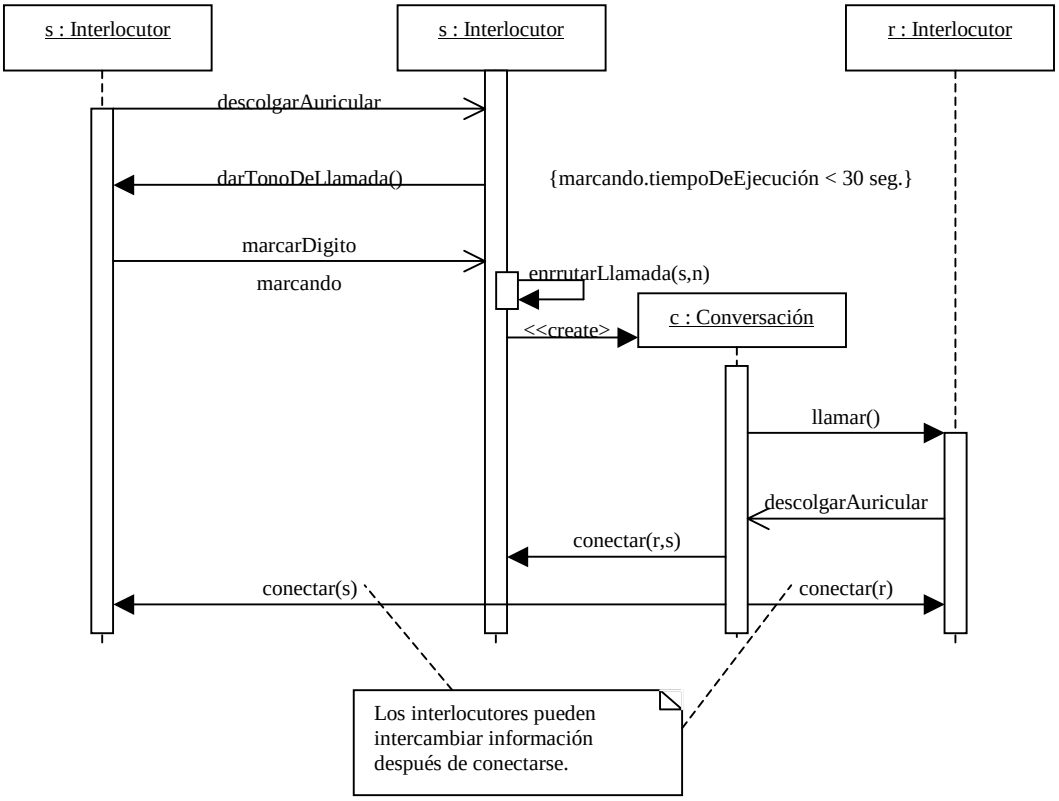


Figura55. Modelado de flujos de control por ordenación temporal

Modelado de flujos de control por organización

Considérense los objetos existentes en el contexto de un sistema, un subsistema, una operación o una clase. Considérense también los objetos y roles que participan en un caso de uso o una colaboración. Para modelar un flujo de control que discurre entre esos objetos y roles se utiliza un diagrama de interacción; para mostrar el paso de mensajes en el contexto de esa estructura se utiliza un diagrama de colaboración, un tipo de diagrama de iteración..

Para modelar un flujo de control por organización:

- Hay que establecer el contexto de la interacción, bien sea un sistema, un subsistema, una operación, o una clase, o bien un escenario de un caso de uso o de una colaboración.

- Hay que establecer un escenario de la interacción, identificando qué objetos juegan un rol en ella. Los objetos deben organizarse en el diagrama de colaboración como los nodos del grafo, colocando los objetos más importantes en el centro y sus objetos vecinos hacia el exterior.

- Hay que establecer las propiedades iniciales de cada uno de estos objetos. Si los valores de los atributos, los valores etiquetados, el estado o el rol de algún objeto cambia de forma significativa durante la interacción, hay que colocar un objeto duplicado en el diagrama, actualizarlo con los nuevos valores y conectarlo con un mensaje esteriotipado como *become* o *copy* (con un número de secuencia apropiado).

- Hay que especificar los enlaces entre esos objetos, junto a los mensajes que pueden pasar.

    1. Colocar los enlaces de asociaciones en primer lugar; éstos son los más importantes, porque representan conexiones estructurales.

    2. Colocar los demás enlaces a continuación, y adornarlos con los estereotipos de camino adecuados (como *global* y *local*) para especificar explícitamente cómo se conectan estos objetos entre sí.

- Comenzando por el mensaje que inicia la interacción, hay que asociar cada mensaje subsiguiente al enlace apropiado, estableciendo su número de secuencia. Los anidamientos se representan con la numeración decimal de Dewey.

- Si es necesario especificar restricciones de tiempo o espacio, hay que adornar cada mensaje con una marca de tiempo y asociar las restricciones apropiadas.

- Si es necesario especificar este flujo de control más formalmente, hay que asociar pre y poscondiciones a cada mensaje.

Al igual que los diagramas de secuencia, un único diagrama de colaboración sólo puede mostrar un flujo de control (aunque se pueden representar variaciones sencillas utilizando la notación UML para la iteración y la bifurcación). Normalmente se realizarán varios diagramas de interacción, algunos de los cuales serán principales y otros mostrarán caminos alternativos o condiciones excepcionales. Los paquetes se pueden utilizar para organizar estas colecciones de diagramas de colaboración, dando a cada diagrama un nombre para distinguirlo del resto.

Por ejemplo, la figura 56 muestra un diagrama de colaboración que especifica el flujo de control para matricular un nuevo estudiante en una universidad, destacando las relaciones estructurales entre los objetos. Se ven cinco objetos: un *EncargadoMatriculas (r)*, un *Estudiante (s)*, dos objetos *Curso (c1* y *c2)*, y un objeto *Universidad* sin nombre. El flujo de control está explícitamente numerado. La acción comienza cuando el *EncargadoMatriculas* crea un objeto *Estudiante*, añade el estudiante a la universidad (mensaje *añadirEstudiante*), y a continuación dice al objeto *Estudiante* que se matricule.

El objeto Estudiante invoca a *obtenerPlanEstudios* sobre sí mismo, de donde presumiblemente obtiene los objetos *Curso* en los que se debe matricular. Después, el objeto *Estudiante* se añade a sí mismo a cada objeto *Curso*. El flujo acaba con *s* representado de nuevo, mostrando que ha actualizado el valor de su atributo *matriculado*.

Nótese que este diagrama muestra un enlace entre el objeto *Universidad* y los dos objetos *Curso*, más otro enlace entre el objeto *Universidad* y el objeto *Estudiante*, aunque no se representan mensajes a través de estos caminos. Estos enlaces ayudan a entender cómo el objeto *Estudiante* puede ver a los dos objetos *Curso* a los cuales se añade. *S, c1* y *c2* están enlazados a la *Universidad* a través de una asociación, así que *s* puede encontrar a *c1* y a *c2* en su llamada a *obtenerPlanEstudios* (la cual podría devolver una colección de *Curso* objetos) indirectamente, a través del objeto *Universidad*.

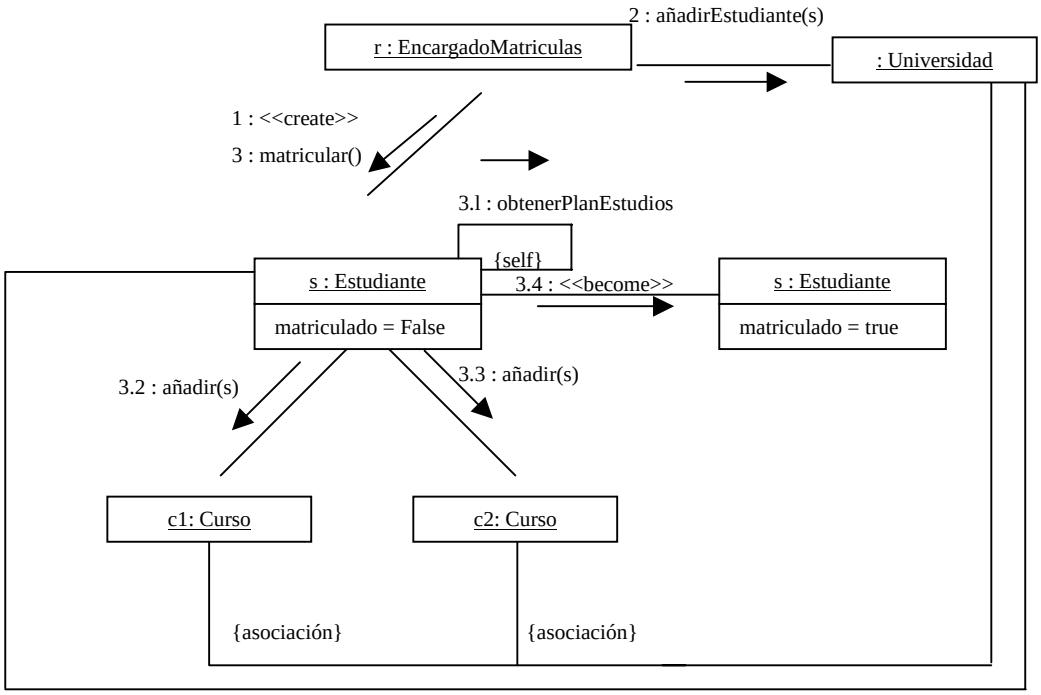


Figura 56. Modelado de flujos de control por organización.

Diagramas de actividades

En la industria de la construcción se utilizan frecuentemente técnicas como los diagramas de Gantt y los diagramas Pert para visualizar, especificar, construir y documentar el flujo de trabajo de un proyecto.

Cuando se modelan sistemas con gran cantidad de software aparece un problema similar. ¿Cuál es la mejor forma de modelar un flujo de trabajo o una operación, que son ambos aspectos de dinámica del sistema? La respuesta es que existen dos elecciones básicas, similares al uso de diagramas de Gantt y diagramas Pert.

Por un lado, se pueden construir representaciones gráficas de escenarios que involucren la interacción de ciertos objetos interesantes y los mensajes que se pueden enviar entre ellos. En UML se pueden modelar estas representaciones de dos formas: destacando la ordenación temporal de los mensajes (con diagramas de secuencia) o destacando las relaciones estructurales entre los objetos que interactúan (con diagramas de colaboración). Los diagramas de interacción son similares a los diagramas de Gantt, los cuales se centran en los objetos (recursos) que juegan alguna actividad a lo largo del tiempo.

Por otro lado, estos aspectos dinámicos se pueden modelar con diagramas de actividades, que se centran en las actividades que tienen lugar entre los objetos, como se muestra en la siguiente figura. En este sentido, los diagramas de actividades son similares a los diagramas Pert. Un diagrama de interacción muestra objetos que pasan mensajes; un diagrama de actividades muestra las operaciones que se pasan entre los objetos. La diferencia semántica es sutil, pero tiene como resultado una forma muy diferente de mirar el mundo.

Así pues, un *diagrama de actividades* muestra el flujo de actividades. Una *actividad* es una ejecución no atómica en curso, dentro de una máquina de estados. Las actividades producen finalmente alguna *acción*, que está compuesta de computaciones atómicas ejecutables que producen un cambio en el

estado del sistema o la devolución de un valor. Las acciones incluyen llamadas a otras operaciones, envío de señales, creación o destrucción de objetos o simples cálculos, como la evaluación de una expresión. Gráficamente, un diagrama de actividades es una colección de nodos y arcos.

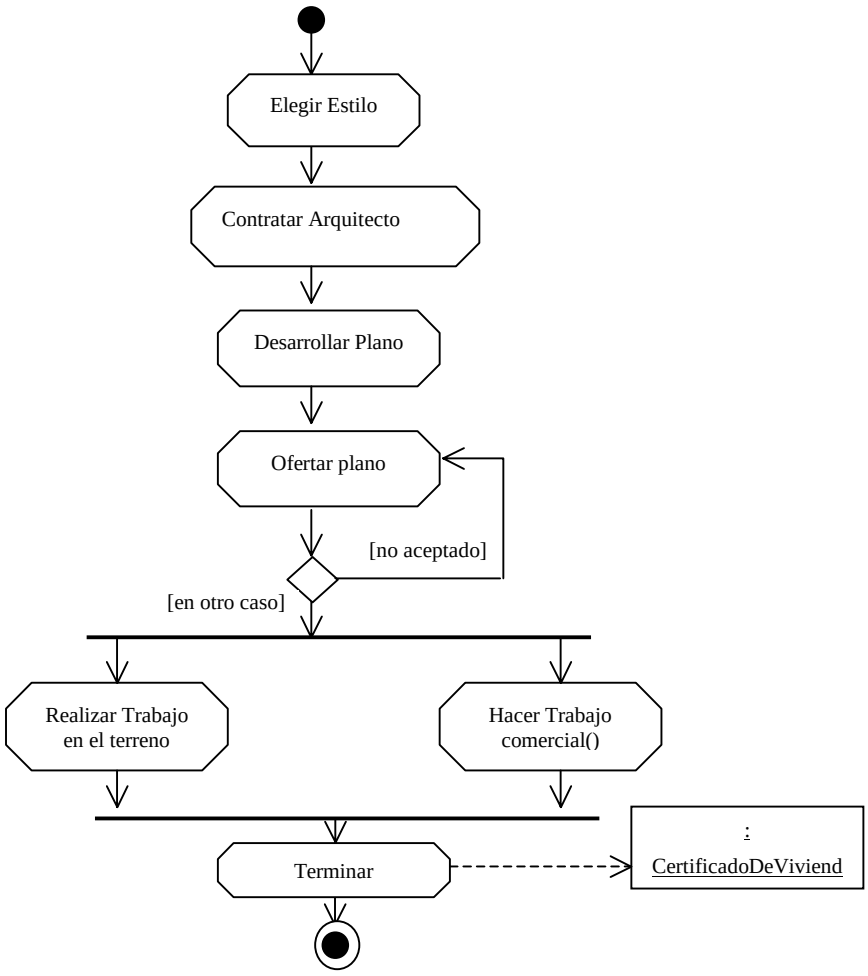


Figura57. Diagramas de Actividades

Normalmente, los diagramas de actividades contienen, estados de actividad y estados de acción, transiciones y objetos.

Estados de la acción y estados de la actividad

En el flujo de control modelado por un diagrama de actividades suceden cosas. Por ejemplo, se podría evaluar una expresión que estableciera el valor de un atributo o que devolviera algún valor. También se podría invocar una operación sobre un objeto, enviar una señal a un objeto o incluso crear o destruir un objeto. Estas computaciones ejecutables y atómicas se llaman estados de acción, porque son estados del sistema, y cada una representa la ejecución de una acción. Como se muestra en la figura 58, un estado de acción se representa con una figura en forma de píldora (un símbolo con líneas horizontales arriba y abajo y lados convexos). Dentro de esa figura se puede escribir cualquier expresión.

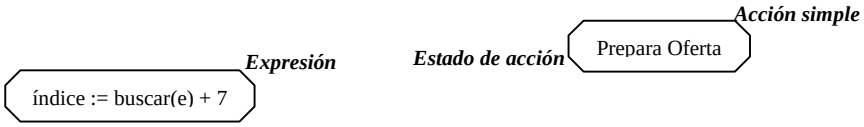


Figura 58. Estados de Acción

Los estado de acción no se pueden descomponer. Además, los estados de acción son atómicos, lo que significa que pueden ocurrir eventos, pero no se interrumpe la ejecución del estado de acción. Por último, se considera generalmente que la ejecución de un estado de acción conlleva un tiempo insignificante.

En contraposición, los estados de actividad pueden descomponerse aún más, representando su actividad con otros diagramas de actividades. Además, los estados de actividad no son atómicos, es decir, pueden ser interrumpidos y, en general, se considera que invierten algún tiempo en completarse. Un estado de acción se puede ver como un caso especial de un estado de actividad. Un estado de acción es un estado de actividad que no se puede descomponer más. Análogamente, un estado de actividad puede ser visto como un elemento compuesto, cuyo flujo de control se compone de otro estado de actividad y estados de acción. Si se entra en los detalles de un estado de actividad se encontrará otro diagrama de actividades. Como se muestra en la figura 59, no hay distinción en cuanto a la notación de los estados de actividad y los estados de acción, excepto que un estado de actividad puede tener partes adicionales, como acciones de entrada y salida (*entry/exit)* (acciones relacionadas con la entrada y la salida del estado, respectivamente) y especificaciones de submáquinas.

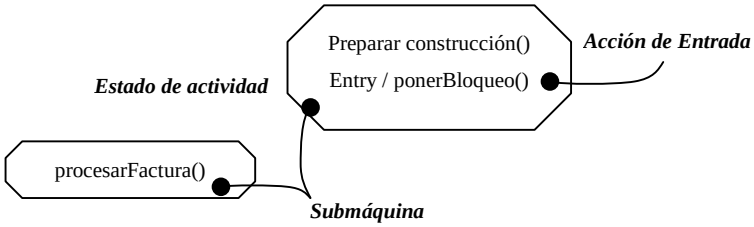


Figura 59. Estados de Actividad

Transiciones

Cuando se completa la acción o la actividad de un estado, el flujo de control pasa inmediatamente al siguiente estado de acción o estado de actividad. Este flujo se especifica con transiciones que muestran el camino de un estado de actividad o estado de acción al siguiente. En UML, una transición se representa como una línea dirigida, como se muestra en la figura 60.

En realidad, un flujo de control tiene que empezar y parar en algún sitio (a menos, por supuesto, que sea un flujo infinito, en cuyo caso tendrá un principio pero no un final). Por lo tanto, como se aprecia en la figura, se puede especificar un estado inicial (un círculo relleno) y un estado final (un círculo relleno dentro de una circunferencia).

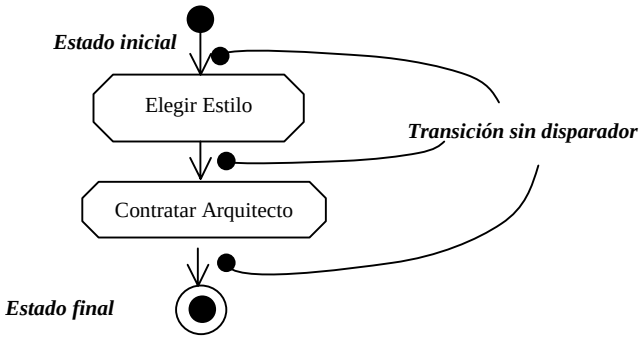


Figura 60. Transacciones sin disparadores

Bifurcación

Las transiciones secuenciales son frecuentes, pero no son el único camino que se necesita para modelar un flujo de control. Como en los diagramas de flujo, se puede incluir una bifurcación, que especifica caminos alternativos, elegidos según el valor de alguna expresión booleana. Como se muestra en la figura 61, una bifurcación se representa con un rombo. Una bifurcación puede tener una transición de entrada y dos o más de salida. En cada transición de salida se coloca una expresión booleana, que se evalúa solo una vez al entrar en la bifurcación. Las guardas de las transiciones de salida no deben solaparse (de otro modo el flujo de control seria ambiguo), pero deberán cubrir todas las posibilidades, de otra manera el flujo de control se vería interrumpido.

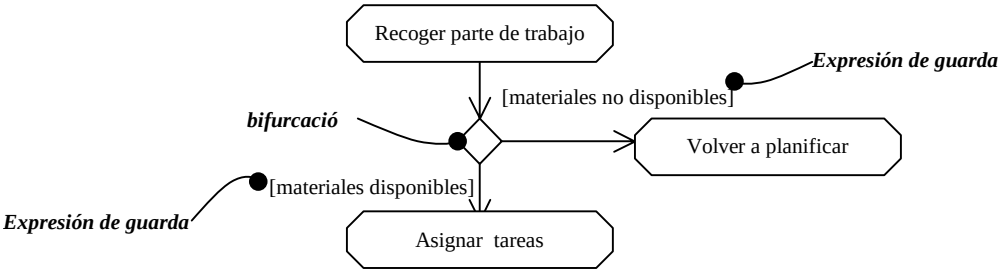


Figura 61. Bifurcación

Por comodidad, se puede utilizar la palabra clave *else* para marcar una transición de salida, la cual representa el camino elegido si ninguna de las otras expresiones de guarda toman el valor *verdadero*.

Se puede lograr el efecto de la iteración utilizando un estado de acción que establezca el valor de la variable de control de una iteración, otro estado de acción que incremente el valor de la variable y una bifurcación que evalúe si se ha terminado la iteración.

División y Unión

Las transiciones secuenciales y las bifurcaciones son los caminos más utilizados en los diagramas de actividades. Sin embargo, también es posible encontrar flujos concurrentes, especialmente cuando se modelan flujos de trabajo de procesos de negocio. En UML se utiliza una barra de sincronización para especificar la división y unión de estos flujos de control paralelos. Una barra de sincronización se representa como una línea horizontal o vertical ancha.

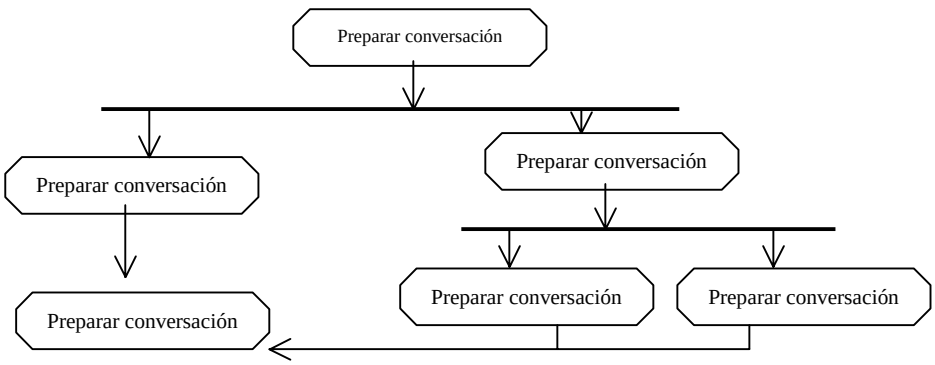


Figura 62. Divisiones y uniones

Por ejemplo, considérese los flujos de control implicados en el manejo de un dispositivo electrónico que imite la voz y los gestos humanos. Como se muestra en la figura 63 , una división representa la separación de un flujo de control sencillo en dos o más flujos de control concurrentes. Una división puede tener una transición de entrada y dos o más transiciones de salida, cada una de las cuales representa un flujo de control independiente. Después de la división, las actividades asociadas a cada uno de estos caminos continúan en paralelo. Conceptualmente, las actividades de cada uno de estos flujos son verdaderamente concurrentes, aunque en un sistema en ejecución, estos flujos pueden ser realmente concurrentes (en el caso de un sistema instalado en varios nodos) o secuenciales y entrelazados (en el caso de un sistema instalado en un único nodo), dando la ilusión de concurrencia real. También en la figura 62, una unión representa la sincronización de dos o más flujos de control concurrentes. Una unión puede tener dos o más transiciones de entrada y una transición de salida. Antes de llegar a la unión, las actividades asociadas con cada uno de los caminos continúa en paralelo. En la unión, los flujos concurrentes se sincronizan, es decir, cada uno se espera hasta que los demás flujos de entrada han alcanzado la unión., a partir de ahí se continúa el flujo de control que sale de la unión.

Calles (Swimlanes[3])

Una cosa especialmente útil cuando se modelan flujos de trabajo de procesos de organizaciones, es dividir los estados de actividad de un diagrama de actividades en grupos, donde cada uno representa la parte de la organización responsable de esas actividades. Cada calle tiene un nombre único dentro del diagrama. Una calle realmente no tiene una semántica profunda, excepto que puede representar alguna entidad del mundo real. Cada calle representa una responsabilidad de alto nivel de una parte de actividad global de un diagrama de actividades, y cada calle puede ser implementada en última instancia por una o más clases. En un diagrama de actividades organizado en calles, cada actividad pertenece a única calle, pero las transiciones pueden cruzar las calles. En UML cada grupo se denomina calle porque, visualmente, cada grupo se separa de sus vecinos por una línea vertical continua. Una calle especifica un lugar para las actividades. En la figura de la pagina siguiente se puede observar un ejemplo de calles.

Usos comunes

Los diagramas de actividades se utilizan para modelar los aspectos dinámicos de un sistema. Estos aspectos dinámicos pueden involucrar la actividad de cualquier tipo de abstracción en cualquier vista de la arquitectura de un sistema, incluyendo clases (las cuales pueden ser activas), interfaces, componentes y nodos.

Cuando se utiliza un diagrama de actividades para modelar algún aspecto dinámico de un sistema, se puede hacer en el contexto de casi cualquier elemento de modelado. Sin embargo, normalmente se usan los diagramas de actividades en el contexto del sistema global, un subsistema, una operación o una clase. También se pueden asociar diagramas de actividades a un caso de uso (para modelar un escenario) y a las colaboraciones (para modelar los aspectos dinámicos de una sociedad de objetos).

Cuando se modelan los aspectos dinámicos de un sistema, normalmente se utilizan los diagramas de actividades de dos formas:

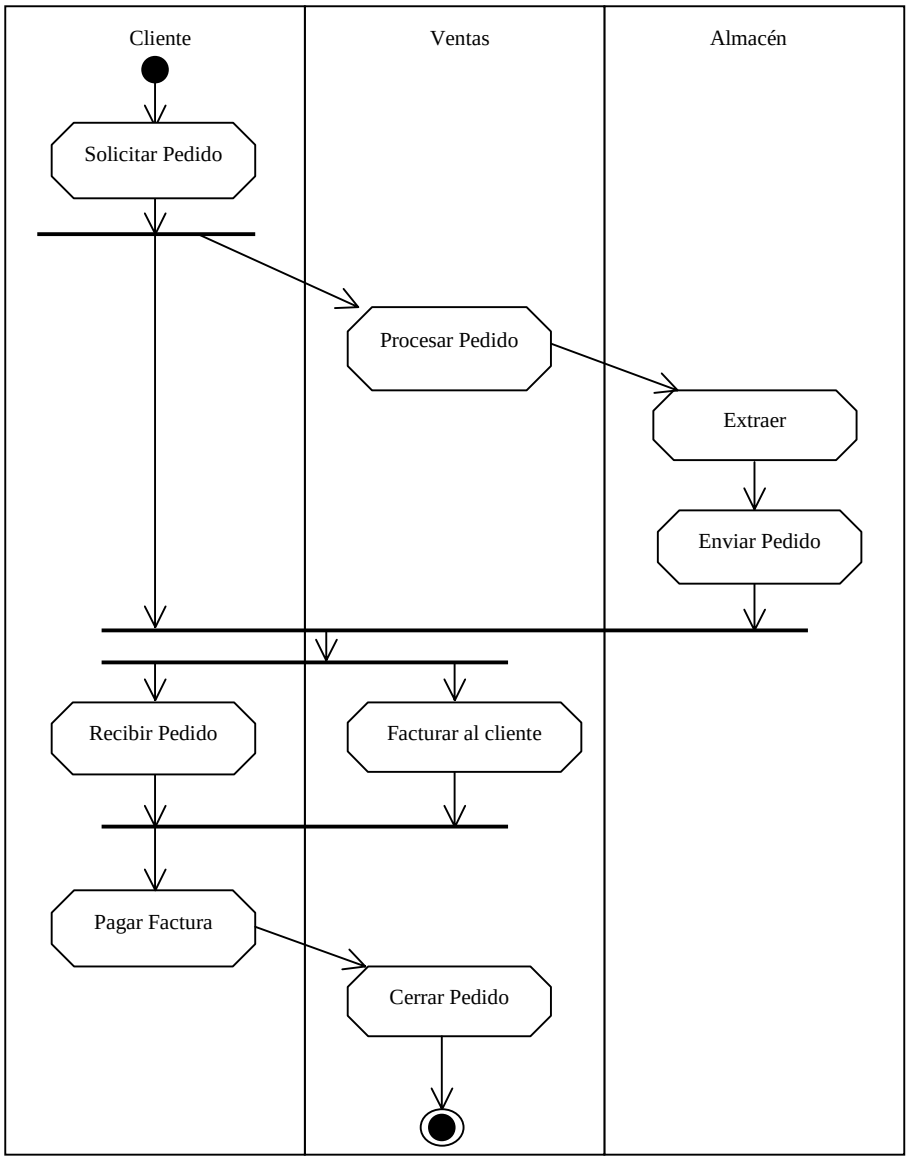
1. Para modelar un flujo de trabajo.

3 El termino Swimlanes se utiliza en el mismo sentido que las calles de una piscina de competición o las calles de una piscina.

Para ello se hace hincapié en las actividades, tal y como son vistas por los actores que colaboran con el sistema. A menudo, en el entorno de los sistemas con gran cantidad de software, existen flujos de trabajo y se utilizan para visualizar, especificar, construir y documentar procesos de negocio que implican al sistema que se está desarrollando. En este uso de los diagramas de actividades, es particularmente importante el modelado de los flujos de objetos.

2. Para modelar una operación.

Para ello se utilizan los diagramas de actividades como diagramas de flujo, para mostrar los detalles de una computación. En este uso de los diagramas de actividades, es particularmente importante el modelado de la bifurcación, la división y la unión. El contexto de un diagrama de actividades utilizado con esta finalidad incluye los parámetros de la operación, así como sus objetos locales.



7

Visual Modeler

Visual Modeler es una herramienta C.A.S.E. (Computer Aided Software Engineering) basada en el lenguaje de modelado UML que forma parte de paquete de desarrollo Visual Studio de Microsoft ™. Visual Modeler cubre las necesidades que se originan al modelar la ***vista de diseño estática*** de un sistema y dentro de la arquitectura del sistema se ocupa de las vistas de diseño y de despliegue, para ello proporciona la posibilidad de definir tres tipos de diagramas: diagramas de clases, diagramas de componentes y diagramas de despliegue.

Visual Modeler es una herramienta C.A.S.E. con capacidades de ingeniería directa (obtención de código útil a partir de modelos) y de ingeniería inversa (obtención de modelos a partir de código). Visual Modeler trabaja principalmente con dos lenguajes de implementación, Microsoft Visual Basic y Microsoft Visual C++, cuando tratamos de realizar una obtención de código a partir de un modelo. A la hora de realizar inversa sólo podemos obtener modelos a partir de código desarrollado con Microsoft Visual Basic (siempre y cuando tengamos el *Add-in* necesario integrado en el editor de Visual Basic). Visual Modeler también está integrado con Microsoft Visual Source Safe, con lo que podremos mantener diferentes versiones de un mismo modelo con las facilidades que Visual Source Safe nos proporciona.

Visual Modeler representa un subconjunto del lenguaje de definición de modelos que es UML, en este apéndice estudiaremos las características de UML que son representables con Visual Modeler así como las capacidades de ingeniería directa e inversa que proporciona desde el punto de vista de Microsoft Visual Basic.

Conceptos

Visual Modeler está orientado al diseño de aplicaciones en un entorno cliente / servidor en tres capas (Three Tiered System). Esto no significa que Visual Modeler no se pueda utilizar para diseñar aplicaciones que no requieran la división lógica y física de sus componentes en diferentes capas, para ello, simplemente es necesario seguir los siguientes pasos: En el menú de la aplicación seleccionar **Tools/Options...**, en este instante nos aparece un formulario que contiene las opciones de usuario sobre Visual Modeler distribuidos en diferentes solapas. Seleccionar la solapa que contiene las opciones para los diagramas (**Diagram**), en la parte de visualización (**Display**) hay una entrada que nos permite seleccionar o no que el tipo de diagrama sea en tres capas (**3 Tier Diagram**).

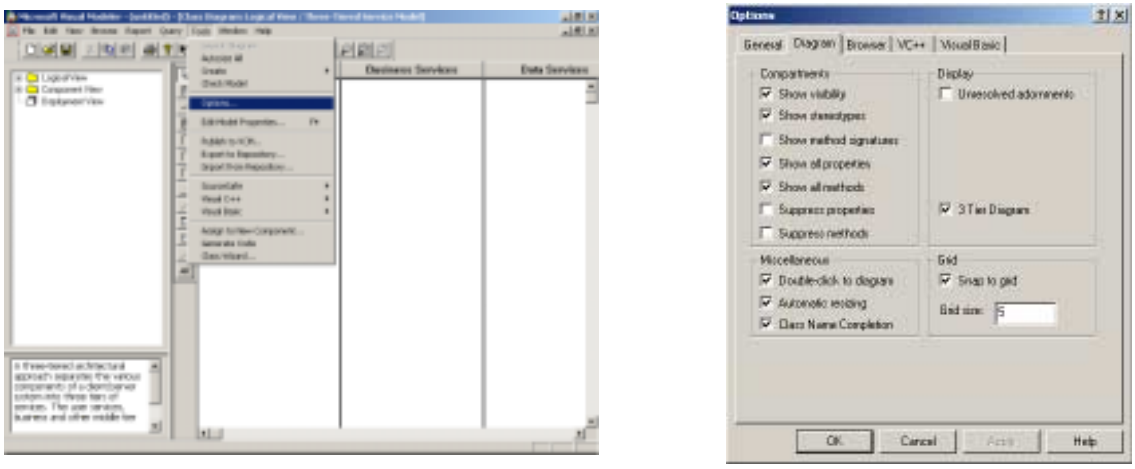


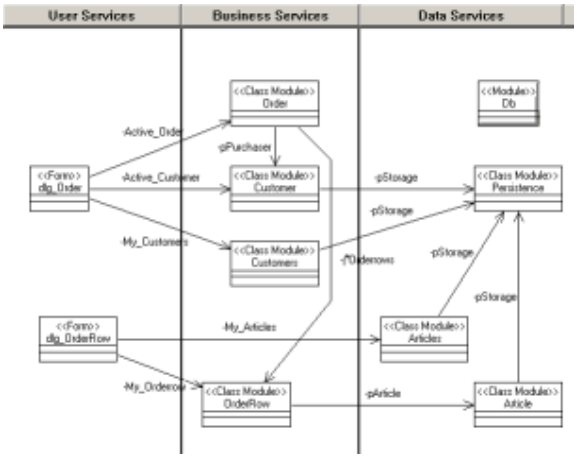
Figura 64. Cambiar el tipo de diagrama.

Sistemas cliente servidor en tres capas

Un sistema cliente / servidor en tres capas representa un sistema distribuido en el que se han separado los distintos servicios que componen el sistema. La división que normalmente se sigue para estos sistemas es la definición de tres capas lógicas (que posteriormente se convertirán en diferentes capas físicas) de la siguiente manera: En la capa más inferior se encuentra la **capa de datos** en esta capa se encuentran las diferentes bases de datos de las que la aplicación obtendrá y añadirá datos, en esta capa también se encuentran los procedimientos almacenados que nos ayudan a simplificar los accesos, modificaciones e inserciones sobre los datos. La capa de datos también aparecía en los sistemas cliente / servidor clásicos (o en dos capas). La siguiente capa que se define en un sistema cliente / servidor en tres capas es la capa que contiene la **lógica de negocios**, en esta capa se definen los componentes (entendiéndose por componente un conjunto de clases que hacen algo) que contiene la definición de las operaciones que son necesarias para que nuestro sistema haga su trabajo, en estos componentes residen las operaciones que operan sobre los datos de la capa inferior. En estos componentes están las reglas que dicen cómo hay que utilizar los datos y mantienen la integridad de los mismos, además la capa de negocios oculta una posible distribución física de los datos. Por último está la **capa de presentación**, está capa se encarga únicamente de presentar los datos a los usuarios y de establecer la interfaz para que exista una comunicación entre los usuarios y el sistema. Esta capa carece de procesamiento y se limita únicamente a mostrar los datos a los usuarios y comprobar que las peticiones de los usuarios son, por lo menos, semánticamente, correctas y evitar que se hagan peticiones a la capa de negocios que a priori son inviables por que no cumplen algún requisito previo.

En este sentido, un sistema cliente servidor en tres capas establece, al menos, tres capas donde los usuarios no tienen constancia sobre como se almacenan ni donde residen los datos, estos sólo se

comunican con la capa de presentación, esta se ocupa de procesar las peticiones de los usuarios y transmitirlas a los componentes de la capa de negocios, en esta capa se procesará la petición modificando, pidiendo o consultando los datos necesarios, estos son provistos por la capa de datos que además de proveer los datos necesarios a la capa superior se encarga de almacenarlos y mantenerlos correctamente.



Visual Modeler, por defecto, proporciona una configuración para definir modelos de sistemas cliente / servidor en tres capas. Visual Modeler diferencia entre **User Services**, **Business Services** y **Data Services**, pero con una consideración, las tres capas que Visual Modeler diferencia no se corresponden con las tres capas lógicas de un sistema cliente servidor en tres capas ya que los servicios de negocios (Business Services) y los servicios de datos (Data Services) representan la *capa de negocio* de un sistema cliente servidor en tres capas, esto es, los objetos de negocio y los objetos que acceden a los datos. En el contexto de Visual Modeler la capa de servicio de datos son los componentes que acceden explícitamente a los datos y no las bases de datos y los procedimientos almacenados que son los que componen la capa de datos de un sistema cliente servidor en tres capas.

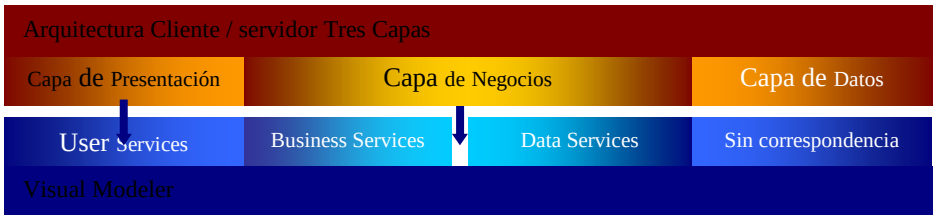


Figura 65. Correspondencia entre las capas de Visual Modeler y los sistemas en tres capas

Vistas de Visual Modeler

Como ya hemos dicho Visual Modeler es una herramienta C.A.S.E. que se puede utilizar para modelar la vista de diseño estática de los sistemas software y que además proporciona capacidades de ingeniería directa e inversa. Modelar la vista de diseño estática supone que se disponen de las herramientas necesarias para poder representar la parte estructural y arquitectónica de los sistemas, todo ello basado en el lenguaje de modelado orientado a objetos que es UML.

Visual Modeler proporciona tres vistas: La vista lógica, la vista de componentes y la vista de despliegue. En cada una de las vistas se usan unos tipos de diagramas diferentes.

Vista Lógica (*Logical View*)

Esta vista nos sirve para soportar los requisitos funcionales del sistema, o sea, los servicios que el sistema debe proporcionar y comprende las clases, interfaces y colaboraciones (conjuntos de clases) que forman el vocabulario del problema y de la solución.

Dentro de la vista lógica se pueden definir diferentes *paquetes lógicos* que sirven para dividir la vista lógica en diferentes subvistas o subsistemas. Por ejemplo cuando estamos definiendo un sistema en

tres capas la vista lógica se divide en tres partes: los servicios de usuario – *User Services* –, los servicios de negocios – *Business Services* – y los servicios de datos – *Data Services* –.

Cuando utilizamos el ***browser*** (figura 66), podemos observar que cuando estemos ante un diagrama para el modelado de un sistema cliente / servidor en tres capas, la vista lógica (Logical View) contiene cuatro entradas: Un diagrama que representa toda la vista lógica, y en este caso el diagrama de tres capas (Three Tiered Service Model), y tres paquetes lógicos, representados por carpetas, cada uno de los cuales contienen los diagramas que están asociados a cada capa en el sistema cliente / servidor en tres capas.

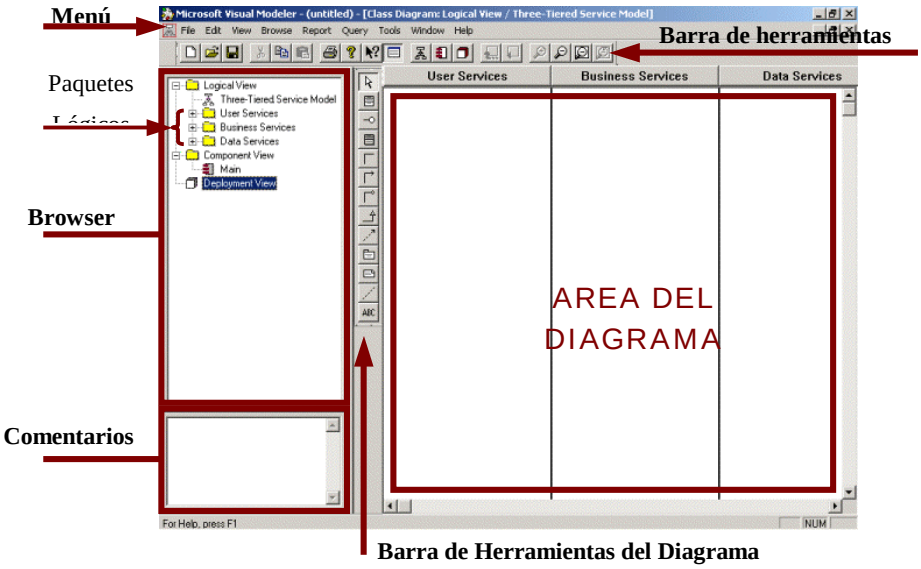


Figura 66. Visual Modeler

En la vista lógica de un sistema existirán uno o varios ***diagramas de clases***, estos contienen clases e interfaces que son objeto de nuestro sistema así como las relaciones que existen entre ellas.

Vista de Componentes (*Component View*)

En la vista de componentes de un sistema se representa la organización y las dependencias que existen entre los componentes que se van a utilizar para dar solución a nuestro problema. Un componente es un modulo software de nuestro sistema o un subsistema por si mismo (código fuente, código binario o código ejecutable), contiene la implementación de una o varias clases y puede tener varias interfaces. Las interfaces representan la parte del componente que es publica y puede ser utilizada por otros componentes.

En los diagramas de componentes pueden aparecer paquetes que representan un conjunto de componentes relacionados lógicamente y suelen coincidir con los paquetes lógicos. Estos paquetes permiten una división física de nuestro sistema.

En la figura 67 se muestra una organización física de los distintos componentes de un sistema ficticio. Se hacen tres divisiones principales: un paquete donde residen los componentes que tienen que ver con los servicios de usuarios, un paquete donde residen los componentes que tienen que ver con los servicios de negocios y un paquete que tienen que ver con los componentes que acceden a los datos. Este último paquete contiene dos paquetes más que diferencian los componentes que acceden a datos almacenados en SQL - Server y con los que acceden a los datos almacenados en Oracle.

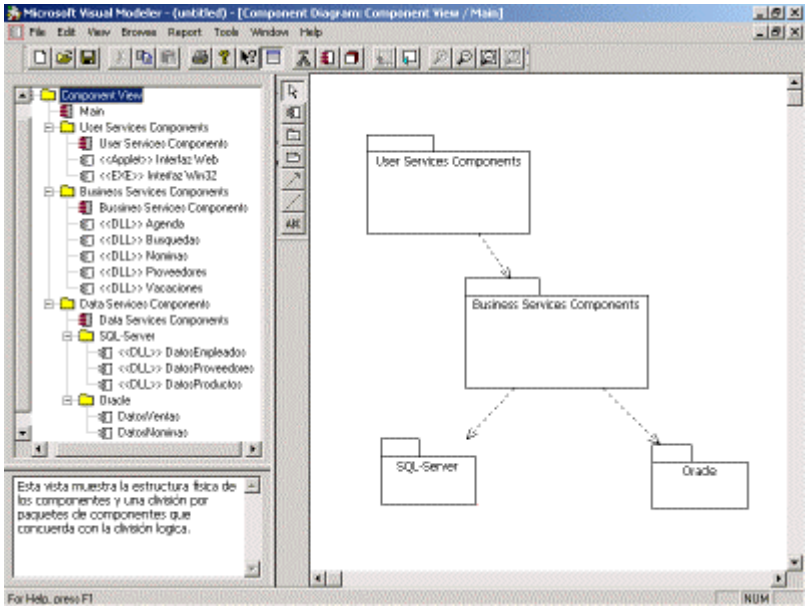


Figura 67. Ejemplos de Paquetes de Componentes

En la figura 68 se muestra un diagrama de componentes donde se aprecian las dependencias de los componentes realmente y no estructurados por paquetes, aun así, este diagrama se representa los mismos componentes que el anterior.

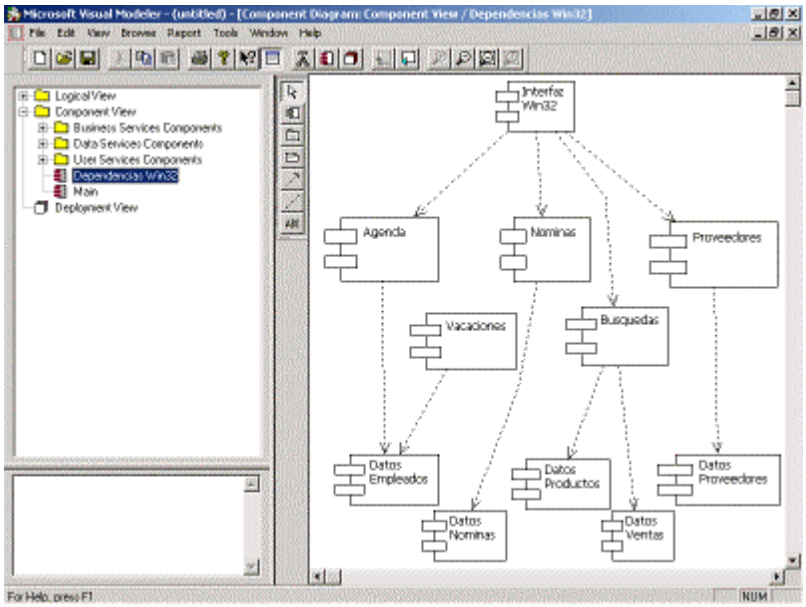


Figura 68. Diagrama de Componentes

Vista de Despliegue (Deployment View)

La vista de despliegue de un sistema representa la configuración de los nodos de procesamiento en tiempo de ejecución y los componentes que residen en ellos. Muestran una vista estática de una arquitectura y se relacionan con los componentes ya que por lo común los nodos tienen uno o varios componentes.

En la figura 69 podemos observar un diagrama de despliegue que muestra una posible configuración de los nodos, los cuales muestran los componentes que contienen, para nuestro sistema ficticio.

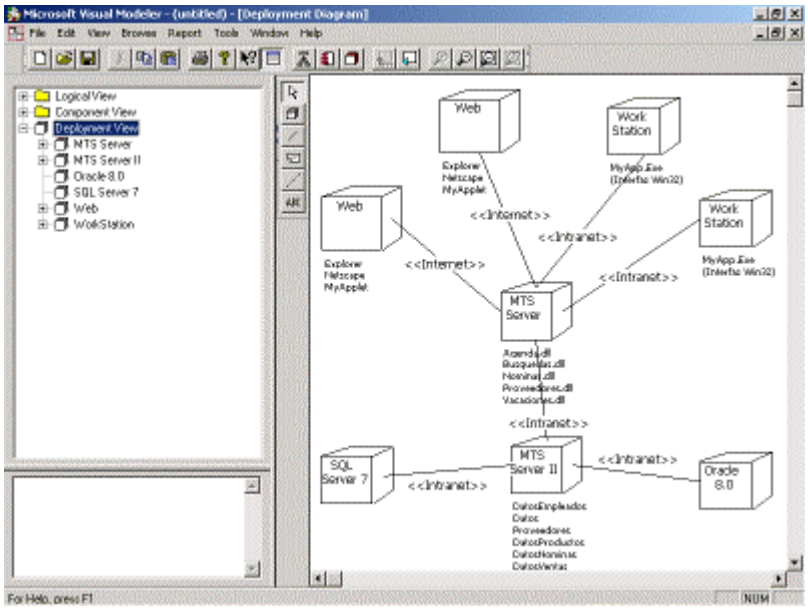


Figura 69. Diagrama de Despliegue

Barra de Herramientas de Visual Modeler

La barra de herramientas de Visual Modeler contiene una serie de accesos directos al conjunto de operaciones más utilizadas. A continuación vamos a describir cada uno de los accesos directos que componen la barra de herramientas.

Sobre Archivo

Crear un Nuevo Modelo (Create New Model): Este acceso directo nos permite crear un nuevo modelo en blanco.

Abrir un Modelo Existente (Open Existing Model): Nos permite abrir la caja de dialogo donde podemos seleccionar abrir un modelo que ya tengamos guardado en disco.

Guardar el Modelo o el Log (Save Model or Log): Guarda los cambios que se hayan producido en el modelo y el fichero *Log* si así está configurado.

Sobre Edición

Cortar Selección (Cut): Corta la selección actual del diagrama actual y la pone en el portapapeles.

Copiar Selección (Copy): Copia la selección actual del diagrama actual al portapapeles.

 **Pegar (Paste):** Pega el contenido del portapapeles al diagrama actual.

Sobre Utilidades

 **Imprimir Diagramas (Print Diagrams):** Muestra la caja de dialogo de impresión de diagramas.

 **Mostrar la Ayuda (Help Topics):** Muestra la ayuda de Visual Modeler.

 **Obtener ayuda dependiendo del contexto (Context Sensitive Help):** Obtiene ayuda de dependiendo del elemento que se selecciona después de pulsar sobre el acceso directo.

 **Ver Documentación (View Documentation):** Teniendo seleccionado este acceso icono nos permite visualizar en la parte inferior izquierda (debajo del Browser) una ventana que contiene la documentación sobre el elemento / diagrama que se tiene seleccionado en ese momento.

Sobre Diagramas

 **Buscar Diagrama de Clases (Browse Class Diagram):** Muestra una caja de dialogo donde podemos seleccionar los diagramas de clases que tengamos definidos en los distintos paquetes.

 **Buscar Diagrama de Componentes (Browse Component Diagram):** Muestra una caja de dialogo donde podemos seleccionar los diagramas de componentes que tengamos definidos en los distintos paquetes.

 **Buscar Diagrama de Despliegue (Browse Deployment Diagram):** Muestra el diagrama de despliegue que está asociado al modelo.

 **Buscar el Diagrama Padre (Browse Parent):** Muestra el diagrama que muestra el paquete lógico o el paquete de componente que contiene el diagrama que actualmente se está viendo.

 **Buscar el Diagrama Previo (Browse Previous Diagram):** Muestra el diagrama que estábamos visualizando inmediatamente antes que el actual.

Sobre Visualización

 **Aumentar Zoom (Zoom In):** Aumenta el zoom en el diagrama que se está visualizando actualmente.

 **Disminuir Zoom (Zoom Out):** Disminuye el zoom en el diagrama que se está visualizando actualmente.

 **Ajustar a Ventana (Fit in Window):** Ajusta la visualización para que se puedan ver todos los elementos del diagrama actual dentro del área visible del diagrama.

 **Deshacer Ajustar (Undo Fit in Window):** Ajusta el zoom a la visualización previa después de haber ajustado a ventana.

Barra de Herramientas para los Diagramas

Cada tipo de diagramas de Visual Modeler puede utilizar diferentes tipos de elementos, a continuación vamos a mostrar cada uno de los elementos que se utilizan en los diferentes diagramas así como su utilización.

Elementos comunes a todos los diagramas

Los siguientes elementos son utilizados en todos los tipos de diagramas que se usan en Visual Modeler, estos son: cajas de texto, notas, conexión de notas a elementos y la herramienta de selección. Los tres primeros elementos se utilizan para añadir notas aclaratorias o bloques de texto libre añadiendo así explicaciones que forman parte de los diagramas. Con estos elementos podemos aportar más información sobre que representa un determinado diagrama o notas aclaratorias sobre los elementos que componen el diagrama.

Caja de Texto (Text Box): Con este elemento podemos añadir una caja de texto al diagrama. Las cajas de texto nos pueden valer para mostrar el nombre que queremos asignar al diagrama o añadir texto explicativo sobre el diagrama. El tipo, tamaño, formato y color de la letra de la caja de texto se puede modificar utilizando los menús de la aplicación de la siguiente manera: Seleccionamos la caja de texto, una vez seleccionada pulsamos la siguiente secuencia de menús → ***Edit / Diagram Object Properties / Font...*** A continuación nos aparecerá una caja de dialogo donde podemos especificar todas las propiedades de la letra para nuestra caja de texto.

Nota (Note): Este elemento nos sirve para añadir una nota explicativa sobre algún elemento que aparece en el diagrama. Se pueden definir tantas notas como sea necesario para que nuestro diagrama no carezca de información. El formato del texto de la nota se puede modificar igual que para las cajas de texto.

Conexión de Notas a Elementos (Anchor Note to Item): Conecta una nota con el elemento al que afecta.

Herramienta de Selección (Selection Tool): Nos sirve para realizar una selección simple sobre un elemento o seleccionar múltiples elementos de un mismo diagrama. Para realizar una selección simple sólo hay que pulsar sobre el elemento que queremos que quede seleccionado. Para realizar una selección múltiple pulsamos en un punto del diagrama, sin soltar el botón del ratón procedemos moviendo el ratón, en ese instante aparece un rectángulo con línea discontinua, todos los elementos que se encuentren completamente incluidos dentro del rectángulo quedarán seleccionados. Con uno o varios elementos seleccionados podemos arrastrarlos para situarlos donde queramos.

Elementos para los Diagramas de Clases

Los diagramas de clases muestran un conjunto de clases e interfaces así como sus relaciones. Estos diagramas son los más comunes en el modelado de sistemas orientados a objetos y cubre la vista de diseño estática. Dentro de los diagramas de clases podemos utilizar diferentes elementos, a continuación pasamos a describir cada uno de los elementos y su utilización.

Clases (Class)

Una clase es una descripción de un conjunto de objetos que comparten los mismos atributos, operaciones, relaciones y semántica. Para añadir una clase en un diagrama de clases hay dos opciones: Seleccionar de la barra de herramientas el icono de C*lase* y pulsar sobre el diagrama de clases en el cual queremos añadir una nueva clase; la otra opción es desde el menú de Visual Modeler seleccionar ***Tools / Create / Class*** y posteriormente pulsar sobre el diagrama que de clases en el cual queremos añadir una nueva clase.

Tras crear una nueva clase estamos en disposición de establecer sus propiedades principales, estas son: su nombre, sus atributos y sus operaciones.

Especificación del Nombre:

Automáticamente, tras crear la nueva clase estamos en disposición de determinar su nombre. Otra opción es liberar el cursor de la definición de la clase (automáticamente obtiene un nombre compuesto por la cadena "*NewClass*" seguido de un número que la identifique de manera única), posteriormente podemos obtener la especificación de la nueva clase pulsando con el botón derecho sobre la clase, nos aparece un menú contextual donde podemos seleccionar ***Open Specification...*** o haciendo un doble clic sobre la clase**,** en este instante nos aparece un nuevo formulario donde podemos modificar todas las propiedades de la clase (nombre, atributos, métodos y relaciones).

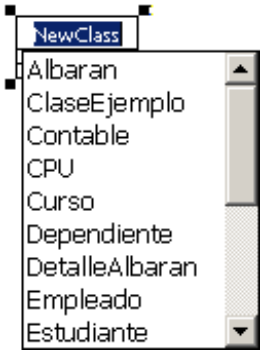


Figura 70. Dar nombre a una clase inmediatamente después de crearla.

En la figura de la izquierda podemos observar el formulario de especificación de una clase donde podemos especificar todas sus propiedades, además nos permite añadir documentación sobre la clase, en este espacio es donde se pueden definir inicialmente las responsabilidades de la clase. También podemos definir estereotipos para la clase dependiendo del uso que vayamos a realizar de ella. Este formulario contiene diferentes pestañas, en cada una de ellas podemos especificar: sus métodos, sus propiedades, sus relaciones y los componentes que la van a contener.

Especificación de los Métodos:

Para especificar los métodos que va a tener una clase existen dos vías: la primera es desde el diagrama, pulsando sobre la clase para obtener su menú contextual y pulsando la entrada ***New Method***, desde ahí podemos incluir nuevos métodos para la clase.

Otra opción es añadir nuevos métodos desde el formulario de especificación de la clase. Cuando añadimos métodos desde el formulario de especificación deberemos pulsar sobre la pestaña de métodos (*Methods*), ahí nos aparece una lista que contiene los métodos con sus propiedades, para añadir uno nuevo haremos un clic con el botón izquierdo, nos aparecerá un menú contextual donde podemos seleccionar insertar un nuevo método (*Insert*). Una vez creado, podemos acceder a la especificación de un método de la misma manera, sacando su menú contextual, pero seleccionando la entrada de especificación (*Specification...)*. En ese instante nos aparece un nuevo formulario donde podemos determinar todas las propiedades de nuestro método.

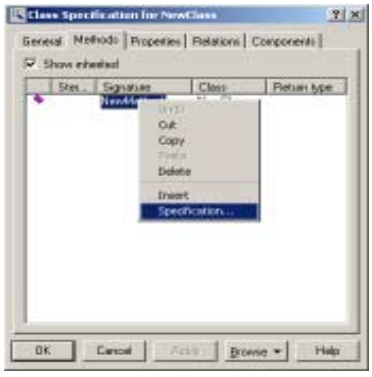
En las siguientes figuras se detalla paso a paso como añadir un nuevo método a una clase:

Paso 1

Aquí se muestra como, desde el formulario de especificación de nuestra clase (en este caso *NewClass*), se puede insertar un nuevo método. Hemos de obtener el menú contextual y posteriormente pulsar sobre la entrada de insertar, única entrada que estará habilitada si no existen métodos en nuestra clase.

Paso 2

Con las acciones realizadas en el *Paso 1* debemos conseguir un nuevo método, por defecto, la visibilidad del método se establece a pública y no tiene tipo de retorno (*Return Type*).

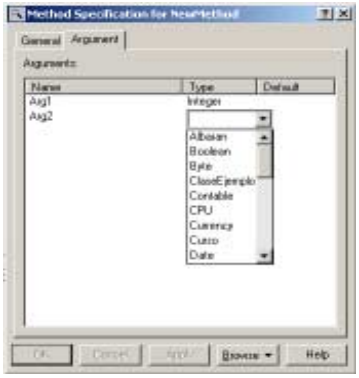


Paso 3

Una vez que hemos conseguido nuestro nuevo método podemos acceder a la especificación del método para modificar su visibilidad, su nombre, el tipo de datos que retorna y lo más importante, especificar los argumentos de nuestro nuevo método.

Paso 4

Podemos observar el formulario de especificación de las propiedades de nuestro método donde modificar su nombre, el tipo de datos que va a retornar (si es que retorna alguno), el estereotipo, el control de exportación (visibilidad) donde podemos determinar como van a ver nuestro método clases externas. También podemos añadir la documentación necesaria sobre nuestro método, esta documentación aparecerá posteriormente en el ***Object Browser*** del editor de Visual Studio. Este formulario también contiene una pestaña donde vamos a poder determinar los argumentos de nuestro método.



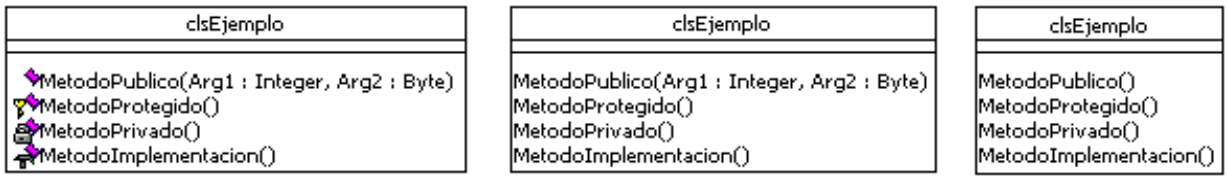
Paso 5

Aquí podemos ver los argumentos, además podremos determinar tanto el tipo de datos del argumento como el valor por defecto que va a tomar. Para añadir nuevos argumentos simplemente obtendremos el menú contextual pulsando con el botón derecho del ratón sobre la lista de argumentos y pulsando la entrada de insertar (*Insert*). Podemos determinar el tipo del argumento haciendo doble clic sobre el tipo obteniendo la lista de todos los tipos disponibles.

Paso 6

En la figura de la izquierda podemos observar como queda la lista de métodos de nuestra clase una vez que hemos añadido diferentes métodos, en la figura también se puede observar como se representan los diferentes tipos de visibilidad, entre ellos y por orden aparecen: pública, protegida, privada e implementación.

En la figura 71 podemos observar como quedará nuestra clase con los nuevos métodos dentro de un diagrama. Dependiendo del nivel de detalle que queramos mostrar obtendremos una u otra visualización.



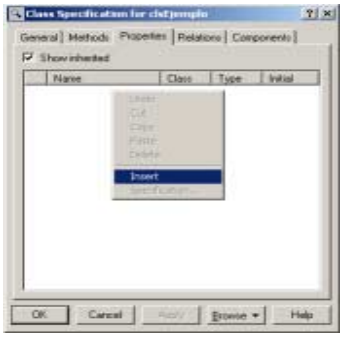
Mostrando visibilidad y argumentos | Mostrando visibilidad | Mostrando sólo nombres

Figura 71. Clase

Especificación de los Atributos:

La especificación de los atributos en una clase se realiza de forma análoga a la especificación de sus métodos. Existen las dos vías comunes, directamente desde el diagrama accediendo al menú contextual de la clase y seleccionando la entrada ***New Property***, la segunda manera es desde el formulario de especificación de la clase. De la segunda manera podremos acceder al formulario de especificación de las propiedades del atributo que vayamos a definir.

Las siguientes figuras muestran la definición de un atributo desde el formulario de especificación de la clase.



Paso 1

Desde el formulario de especificación de la clase accedemos a la pestaña de propiedades (*Properties*), en la lista de propiedades se mostrarán todas las que la clase posee y podremos añadir nuevas obteniendo el menú contextual y pulsando sobre insertar (*Insert*).

Paso 2

Cuando pulsamos para insertar una nueva propiedad estamos en disposición de introducir directamente su nombre.

Paso 3

Si queremos especificar el resto de sus propiedades (visibilidad, tipo, valor inicial) tendremos que pasar al formulario de especificación de la propiedad.

Paso 4

En este paso podemos ver el formulario de especificación de la propiedad. En este formulario podemos determinar el nombre de la propiedad, el tipo, el valor inicial, el control de exportación y la documentación sobre la propiedad. Esta documentación, al igual que en los métodos, puede ser visualizada posteriormente desde el Object Browser del editor de Visual Studio

Por último sólo nos queda por especificar las relaciones de la clase y el componente que la va a contener, si bien, estas dos partes las veremos posteriormente. La determinación de las relaciones se hace mejor desde los diagramas de clases cuando se ven realmente las relaciones que las clases tienen con el resto de las clases implicadas en un diagrama. La determinación de que componente va a incluir la clase se hace mejor cuando se tienen definidos los componentes que el modelo va ha contener.

Interfaces

Una interfaz es una colección de operaciones que especifican el servicio de una determinada clase o componente. Una interfaz describe el comportamiento visible externamente de ese elemento, puede mostrar el comportamiento completo o sólo una parte del mismo. Una interfaz describe un conjunto de especificaciones de operaciones (o sea, su signatura) pero nunca su implementación.

Desde Visual Modeler hay dos posibles representaciones para una interfaz (o sea, dos vistas diferentes del mismo elemento). La primera es cuando incluimos una interfaz en un diagrama de clases. En este caso la interfaz nos aparece como una clase con el estereotipo **<<*interfaz*>>**, normalmente una interfaz estará asociada a una clase mediante una relación. En UML la relación que se establece entre una interfaz y la clase que implementa el comportamiento que la interfaz describe se representa mediante una relación de realización. En Visual Modeler no existen relaciones de realización y la relación que se suele establecer entre la clase que implementa la interfaz y la propia interfaz es una relación de generalización, ya que las interfaces en Visual Modeler se entienden más bien como una clase virtual o una super-clase que establece el comportamiento de las clases que heredan de la interfaz y en cierto modo es correcto ya que las clases que implementan las interfaces son una especialización de estas que además contienen la implementación.

En la figura 72 podemos observar una interfaz relacionada con la clase que contiene su implementación. En la parte derecha de la figura podemos ver la especificación de la clase que implementa la interfaz y como ha heredado todos los métodos públicos de la interfaz.

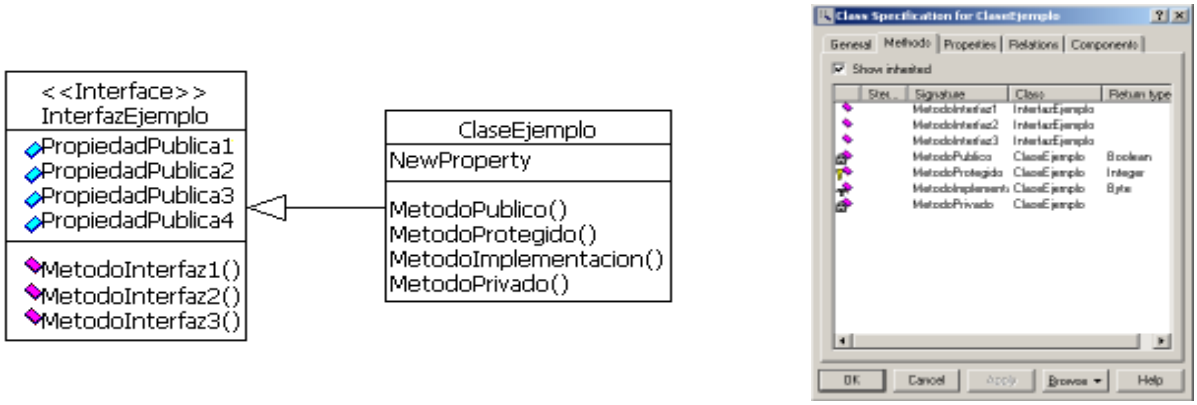


Figura 72. Ejemplo de Interfaz, clase y relación de generalización.

La otra manera que Visual Modeler tiene de representar una interfaz es cuando un componente contiene la implementación de la interfaz (o sea, debe contener la clase que implementa la interfaz). En este caso se ve la interfaz como una piruleta que está enlazada al componente que contiene su interfaz. En la figura 73 podemos ver un componente que contiene la clase y muestra la interfaz que tiene, además se ve la especificación del componente donde se ve que realmente ese componente realiza la clase y la interfaz.

Figura 73. Visualización del componente y de su especificación

Clases de Utilidades (Class Utility)

En Visual Modeler se pueden definir clases de utilidades. Una clase de utilidades es exactamente igual que una clase, se define igual, contiene métodos y atributos, es realizada por algún componente, etc. Lo único que diferencia una clase de utilidades de una clase normal es que todos los métodos y atributos que se definen como públicos en una clase de utilidades son visibles por todas las demás clases que existan en el modelo. Una clase de utilidades reúne la definición de los métodos y atributos globales al sistema que se está modelando. Cuando se aplica ingeniería directa para obtener código en Visual Basic, una clase de utilidades se convierte en un modulo con extensión ".bas".

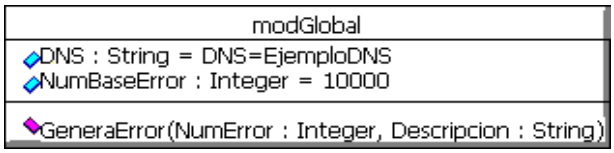


Figura 74. Ejemplo de Clase de Utilidades

Asociaciones (Asociation)

Una asociación en Visual Modeler, al igual que en UML, representa una relación estructural que describe un conjunto de enlaces, los cuales, son conexiones entre objetos. Como en UML, en las asociaciones de clases con Visual Modeler se pueden especificar adornos que determinen los roles que jugarán los objetos en cada extremo de la asociación, su multiplicidad y el nombre de la asociación. La asociación es una de las dos relaciones más importantes que se pueden definir en Visual Modeler, la otra es la generalización, y es así por que además de que la agregación y la asociación unidireccional son tipos de relaciones de asociación, dependiendo del modo que configuremos, adornemos y definamos la asociación obtendremos diferentes variaciones sobre el código que se puede generar en ingeniería directa.

Para definir una asociación entre dos clases simplemente tenemos que seleccionar la herramienta de asociación en la barra de herramientas del diagrama, primero haremos en clic en una de las dos clases entre las que vamos a establecer la asociación y posteriormente extenderemos la línea que representa la asociación hasta poder hacer un segundo clic en la otra clase que va a formar parte de la asociación.

Una vez que tenemos representada la asociación en el diagrama estamos en disposición de añadir los adornos que consideremos oportunos.

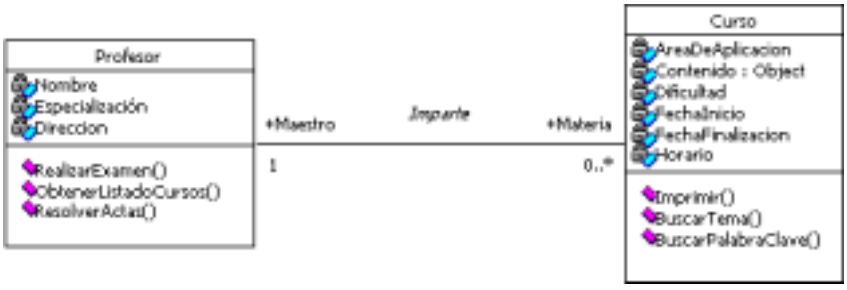
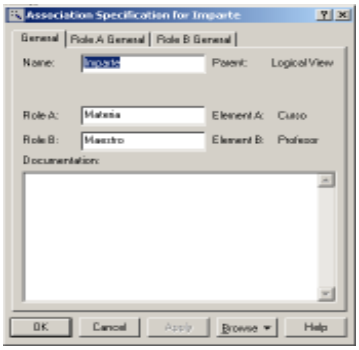
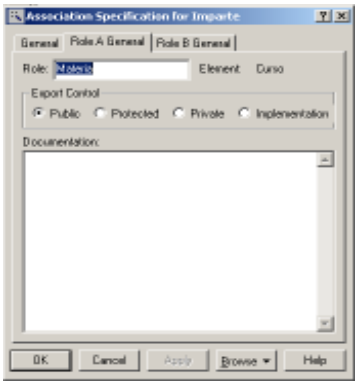


Figura 75. Asociación entre Clases

La figura 75 muestra la asociación entre dos clases de un diagrama. De la asociación y sus adornos podemos extraer la siguiente información: Un profesor *imparte* varios cursos (o ninguno). Además un determinado curso sólo puede ser impartido por un único profesor. El rol que juega el profesor en la asociación es el de *Maestro* y el rol que juegan los cursos es el de *Materia*.





Una vez que se ha definido la asociación entre dos clases podemos establecer las propiedades de esta haciendo un doble clic sobre la línea que representa la asociación. En ese instante nos aparecerá un formulario donde podemos especificar todas sus propiedades. Como vemos en la figura de la izquierda en el formulario podremos establecer el nombre de la asociación, los roles que van a jugar los objetos de las clases involucradas y la documentación que sea importante para explicar la asociación. El formulario de especificación de la asociación contiene tres pestañas: General, Rol A General y Rol B General. En cada una de las pestañas de los roles podremos definir como se van a exportar los objetos involucrados en la asociación. Podemos definir si van a ser objetos públicos, privados o protegidos, y añadir mas documentación al rol si es necesario. Es importante determinar el nombre de los roles y su visibilidad ya que dependiendo de estos parámetros obtendremos variaciones sobre la generación del código a la hora de realizar ingeniería directa. Si no especificamos los nombres de los roles cuando generemos el código, Visual Modeler asignará nombres automáticamente a las variables que utilizará para representar la asociación. Dependiendo del tipo de visibilidad que asignemos a los roles creará variables publicas o privadas dentro de la clase.

Para definir la multiplicidad de la asociación tendremos que obtener el menú contextual de la línea que representa la asociación desde el diagrama de clases. Una vez obtenido seleccionamos la entrada de multiplicidad (*Multiplicity*) y posteriormente seleccionamos la que deseamos 1 ó 0..*. La multiplicidad uno (1) indica que sólo una instancia de la clase donde se ha asignado esta multiplicidad puede asociarse con cada uno de las instancias de la clase que está en el otro lado de la asociación. La multiplicidad varios (0..*) significa que varias instancias (cero o muchas) de la clase donde se ha asignado esta multiplicidad pueden asociarse con cada una de las instancias de la clase que está en el otro lado de la asociación.

Asociación unidireccional (Unidirectional Association)

Una asociación unidireccional es un tipo de asociación donde se establece la relación estructural de manera que los objetos de la clase a la que llega la relación no pueden “viajar” a los objetos de la clase

de la que parte la asociación. Por lo demás la relación es exactamente igual que una asociación normal.

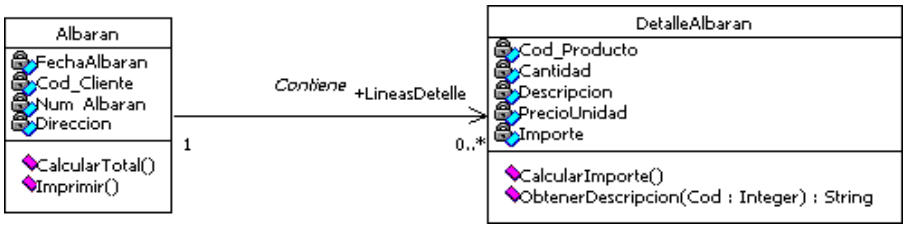


Figura 76. Ejemplo de asociación unidireccional

Agregación (Agregation)

La agregación es una relación de asociación a la que se le confiere más semántica. Una relación de agregación indica que una determinada clase (el todo) está compuesta agregando varias clases (las partes). Se suele utilizar para indicar que una clase tiene un atributo que no es un tipo simple, en este caso el atributo que no es una clase simple se suele decir que "forma-parte-de" la clase que la contiene. En la figura 77 podemos ver un ejemplo de agregación, el diagrama presenta una clase (Ordenador) que está formada por varias clases (Pantalla, Ratón, CPU y Teclado). De esta manera se puede decir que una *Pantalla* forma-parte-de un *Ordenador* o que un *Ordenador* esta-formado-por una *Pantalla*, un *Ratón*, una *CPU* y un *Teclado*.

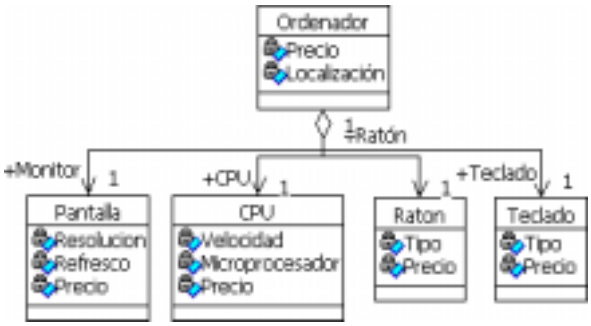


Figura 77. Ejemplo de Agregación

Generalización (Generalization)

Al igual que en UML, la generalización es una relación de especialización / generalización en la cual los objetos del elemento especializado (el hijo) pueden sustituir a los objetos del elemento general (el padre). De esta forma, el hijo comparte la estructura y el comportamiento del padre. La generalización es otra de las relaciones más importantes que se utilizan en Visual Modeler ya que es un tipo de relación que se suele utilizar normalmente a la hora de modelar un problema. Una relación de generalización también se suele llamar como relación "es-un-tipo-de" ya que los hijos en una generalización se puede decir que son un tipo de la clase padre. Se pueden construir árboles de generalización. Cuando se establece una relación de generalización los hijos heredan las propiedades y los métodos públicos o protegidos del padre. Cuando un hijo hereda de varios padres se dice que existe herencia múltiple, aunque la herencia múltiple no suele ser tan común como la herencia simple. En la figura 78 se muestra un ejemplo de relaciones de generalización.

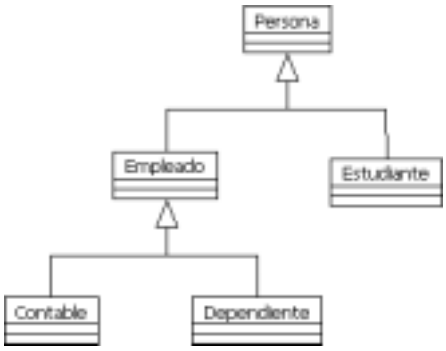


Figura 78. Ejemplo de Generalización.

Dependencia (Dependency)

Una relación de dependencia es una relación de semántica entre dos elementos en el cual un cambio en un elemento (elemento independiente) puede afectar a la semántica del otro elemento (elemento dependiente). Establece una relación de uso entre los elementos. En Visual Modeler el uso de relaciones de dependencia se limita a aportar más semántica a los diagramas ya que esta relación no tiene efecto cuando se realiza ingeniería directa, o sea, no aporta información a la hora de generar código, aunque si que aporta una información muy importante al lector de los diagramas ya que de un solo vistazo puede localizar las dependencias que se dan en un determinado diagrama.

Para Visual Modeler, únicamente en los diagramas de clases, esta relación puede significar instanciación (*Dependency or Instantiates*).

Esto significa que en los diagramas de clases, cuando aparezca una relación de dependencia, significa que la dependencia que existe en los elementos es por que el elemento independiente (al que llega la flecha) es instanciado por el elemento dependiente.

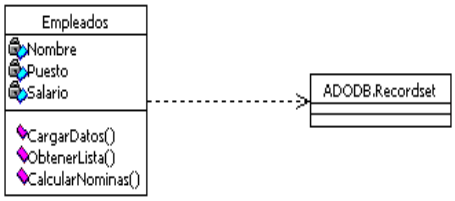


Figura 79. Ejemplo de Dependencia

En la figura 79 podemos observar como la clase *Empleados* utiliza la clase *Recordset* (de la librería ADODB), estos significa que para realizar su trabajo la clase Empleados instanciará objetos de la clase*Recordset*.

Cuando definimos una relación de dependencia podemos abrir su formulario de especificación para determinar sus propiedades, entre ellas podremos definir el nombre que queremos para la dependencia y las cardinalidades de las clases que están implicadas en la dependencia. Las cardinalidades nos especifican un rango de instancias, la cardinalidad especificada en el objeto dependiente nos especifica cuantos objetos dependen de los objetos de la clase independiente y la cardinalidad especificada en el objeto independiente indica de cuantos objetos dependen cada uno de los objetos de la clase dependiente. También podemos aportar la documentación que consideremos necesaria en el apartado dedicado para ello.

Paquetes Lógicos (Package)

Un paquete es un elemento de propósito general con fines organizativos. En la vista lógica de un sistema los paquetes nos sirven para dividir el sistema que estamos modelando en diferentes partes lógicas. Por ejemplo, en los diagramas de tres capas, por defecto, existen tres paquetes: User Services, Business Services y Data Services. En el caso de los diagramas en tres capas, estos paquetes se presentan de una manera un poco especial (en un diagrama de tres capas – Three Tiered Diagram). Cuando en la vista lógica añadimos un nuevo paquete, en este, estamos en disposición de agregar los diagramas que sean necesarios para modelar la parte que pretendemos separar del sistema global. Dentro de un paquete se pueden añadir las clases que nos sean necesarias e incluso podemos utilizar clases que estén contenidas en otros paquetes.

En la figura 80 podemos observar un ejemplo de paquetes lógicos, el diagrama muestra que un las clases contenidas en el paquete *ControlAcceso* dependen de las clases contenidas en el paquete *Personas*.

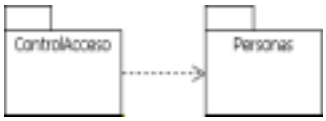


Figura 80. Ejemplo de Paquetes

Elementos para los Diagramas de Componentes

Los diagramas de componentes muestran la organización y las dependencias entre un conjunto de componentes. No hemos de olvidar que un componente es una parte física y reemplazable de un sistema que conforma un conjunto de interfaces y proporciona la implementación de dicho conjunto. Un componente representa típicamente el empaquetamiento físico de diferentes elementos lógicos como clases e interfaces.

Los diagramas de componentes cubren la vista de implementación estática y se relacionan con los diagramas de clases ya que un componente suele tener una o más clases e interfaces. A continuación pasamos a describir los elementos que pueden aparecer en un diagrama de componentes.

Componente (Component)

Como ya hemos dicho, un componente, representa el empaquetamiento físico de diferentes partes lógicas (clases e interfaces) que supone una parte física y reemplazable del sistema. Un componente

contiene la implementación de esa parte física y reemplazable. Cuando incluimos un componente en diagrama de componentes tenemos que darle un nombre. Hemos de poner especial cuidado al dar los nombres a los componentes ya que Visual Modeler no realiza control sobre los nombres y puede ser que posteriormente, a la hora de realizar ingeniería directa, tengamos problemas.

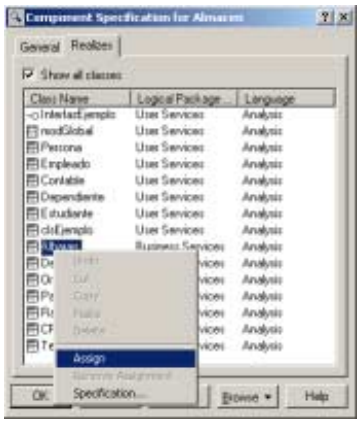
Figura 81. Ejemplo de Componente

Cuando introducimos componentes en nuestro modelo, hemos de determinar que clases o interfaces van a ser implementados en estos componentes, para ello hemos de abrir el formulario de especificación del componente y asignar las clases o interfaces que van ha ser implementados por él. En la especificación del componente también podemos determinar dos características muy importantes para el componente, estas son el lenguaje de implementación y el estereotipo. Cuando definimos el estereotipo estamos definiendo que tipo de componente va a ser, entre ellos podremos elegir que sea un componente ActiveX, un Applet, una Aplicación, una DLL o un EXE. Dependiendo de que seleccionemos obtendremos variaciones sobre el código.

Los siguientes pasos nos muestran la creación normal de un nuevo componente, en este caso se trata de un componente que se va a ocupar del control de almacén de una tienda de informática.

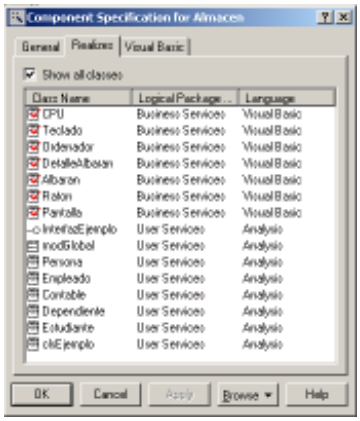
Paso 1

Creamos el nuevo componente, abrimos su formulario de especificación y le asignamos el nombre, lenguaje y estereotipo. En este caso nuestro componente se llama *Almacén*, el lenguaje de implementación va a ser *Visual Basic* y va a ser un componente *ActiveX* (COM DLL o COM EXE). También podemos añadir la documentación que consideremos necesaria para nuestro componente.



Paso 2

Pasamos a la pestaña de realización (*Realizes*) donde podemos determinar todas las clases e interfaces que van a implementarse en este componente. Para asignar una clase a este componente tenemos que seleccionar la clase, obtener su menú contextual y pulsar sobre la entrada de asignación (*Assing*).



Paso 3

Una vez que hemos seleccionada todas las clases que van ha ser implementadas en nuestro componente estas aparecen chequeadas con una marca roja. Posteriormente podremos generar el código para este componente y Visual Modeler establecerá un proyecto (en este caso Visual Basic) donde se van a generar los esqueletos de las clases que hemos seleccionado.

Dependencia (Dependency)

Con Visual Modeler, en los diagramas de componentes, podemos establecer las dependencias que hay entre los componentes. Existirán dependencias entre componentes siempre que clases o interfaces de un componente utilice clases o interfaces de otro. Para representar las dependencias entre los componentes utilizamos la relación de dependencia.

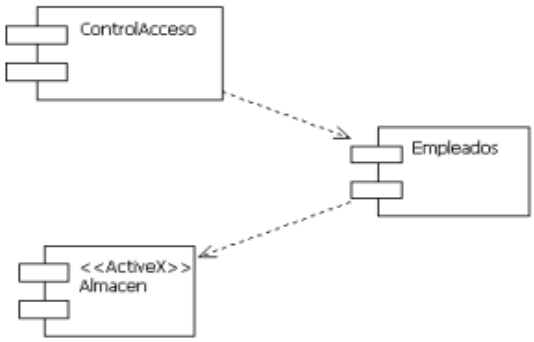


Figura 82. Dependencias entre Componentes

En la figura 82 podemos observar diferentes dependencias entre componentes. El componente que se encarga del control de acceso, *ControlAcceso*, utiliza clases que residen en el componente de *Empleados*, este componente a su vez utiliza clases del componente *Almacén*.

Paquetes de Componentes (Package)

En los diagramas de componentes también pueden aparecer paquetes que engloben varios componentes. En este caso, los paquetes nos determinan una separación física del sistema. Los paquetes de componentes suelen coincidir con los paquetes lógicos. Por ejemplo, en un sistema cliente servidor en tres capas existe una división lógica en Servicios de Usuario, Servicios de Negocios y Servicios de Datos. Esta división lógica suele transformarse en una división física cuando las clases de la capa de usuario se implementan en diferentes componentes y todos estos componentes son los que se despliegan en los clientes de la aplicación. Gráficamente, que todos los componentes de la capa de usuario se desplieguen juntos se puede representar incluyendo todos los componentes en un paquete. Lo mismo ocurre con las otras dos capas. Así en el diagrama de componentes de más alto nivel se

pueden ver tres paquetes cada uno de los cuales tiene dependencia del de la capa inferior. Esto se muestra en la siguiente figura 83.

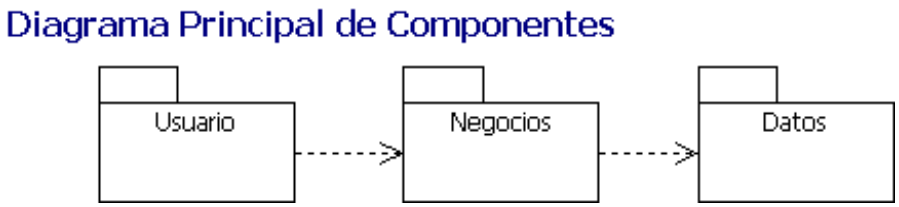
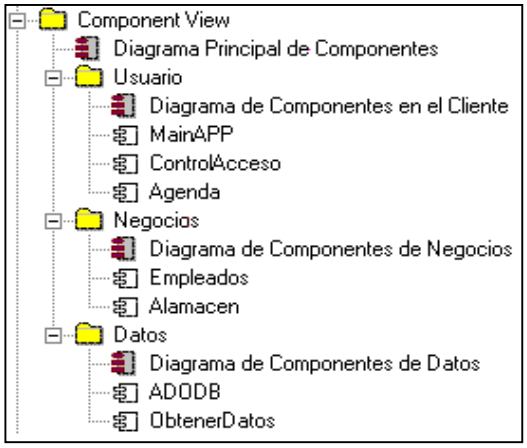


Figura 83. Paquetes de Componentes



La figura de la izquierda presenta el explorador de Visual Modeler en la vista de componentes. Podemos observar que se compone de tres paquetes (Usuario, Negocios y Datos). Cada paquete contiene una serie de componentes que dan la implementación a cada una de las capas lógicas. La división de los componentes físicamente en tres paquetes corresponde, además de la división lógica, a una distribución física de cada grupo de componentes ya que lo normal, en un sistema cliente / servidor en tres capas es que la capa de interfaz de usuario resida en una localización distinta a los componentes de negocio, que normalmente residen en un servidor con gran capacidad y que soporta a muchos clientes distintos. Lo mismo suele ocurrir con los componentes de la capa de datos, que puede que estén en el mismo servidor que los componentes de negocios o en otros (para dar una mayor escalabilidad – o sea, que más usuarios puedan acceder al sistema).

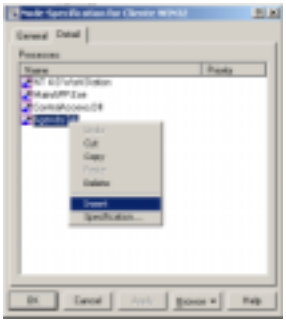
Elementos para los Diagramas de Despliegue

Los diagramas de despliegue representan la configuración de los nodos de procesamiento en tiempo de ejecución y los componentes que residen en ellos. Estos diagramas muestran una vista de despliegue estática de una arquitectura y se relacionan con los componentes ya que, por lo común, los nodos contienen uno o más componentes. A continuación pasamos a describir los elementos que suelen aparecer en los diagramas de componentes.

Nodo (Node)

Un nodo es un elemento físico que existe en tiempo de ejecución y representa un recurso computacional que, por lo general, dispone de algo de memoria y, con frecuencia, de capacidad de procesamiento. En un nodo suelen residir un conjunto de componentes.

Figura 84. Nodo



Cuando añadimos un nuevo nodo a un diagrama de despliegue estamos en disposición de especificar su configuración indicando que componentes van a residir en él y de documentarlo con la información que consideremos necesaria. Desde su formulario de especificación podemos determinar todas sus propiedades. En la parte derecha e izquierda se muestran unas figuras con el formulario de especificación del nodo de la figura 84. En una de las pestañas está la información general, su nombre y la documentación. En la otra pestaña se muestran los detalles del nodo, esto es, los procesos o componentes que va a ejecutarse en él. Para añadir un proceso debemos obtener el menú contextual en la lista de procesos e insertar uno nuevo (*Insert*). También podemos documentar el proceso accediendo al su formulario de especificación haciendo doble clic sobre el proceso que deseemos o desde el menú contextual pulsando la entrada de especificación (*Specification...*).

Conexión (Conection)

Representa algo de hardware existente para conectar dos nodos. Puede ser una conexión directa a través de cable RS232 o indirecta a través de conexiones vía satélite. Las conexiones suelen ser bidireccionales. Mediante conexiones y nodos podemos presentar diagramas que muestren el despliegue del sistema que estamos modelando, o sea, la configuración hardware que vamos a necesitar para desplegar el sistema. En el diagrama de la figura 85 podemos ver un diagrama de despliegue que se corresponde con la vista de componentes que hemos mostrado anteriormente, este diagrama presenta un posible despliegue para el sistema.

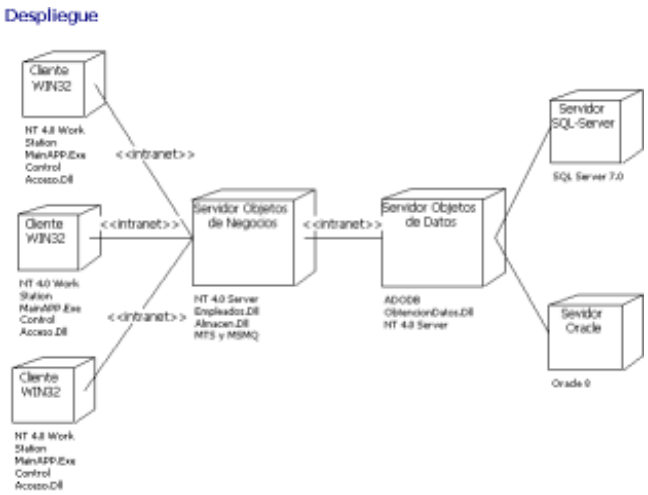


Figura 85. Ejemplo de Diagrama de Despliegue.

8

Ingeniería directa e inversa con Visual Modeler

Como ya hemos explicado Visual Modeler en conjunción con Visual Basic soporta tanto un proceso de ingeniería directa (obtención de código Visual Basic a partir de un modelo) como un proceso de ingeniería inversa (obtención de un modelo a partir de código Visual Basic) Mediante el uso de estas capacidades de Visual Modeler podemos reducir el tiempo de desarrollo, en el caso en el que realicemos un proceso de ingeniería directa, obteniendo código útil a partir de modelos bien formados. También es posible utilizar Visual Modeler para documentar proyectos de Visual Basic obteniendo realizando un proceso de ingeniería inversa.

La manera más rentable de utilizar Visual Modeler en conjunción con Visual Basic es utilizar un proceso de ida y vuelta, o sea, construir el modelo y generar el código, posteriormente trabajaremos sobre el código obtenido modificando algunos aspectos de las clases o de los métodos y atributos de las clases para finalmente realizar un proceso de ingeniería inversa para actualizar el modelo original.

En este capitulo estudiaremos los detalles más importantes de los procesos de ingeniería directa e inversa así de cómo configurar nuestros modelos para obtener el código más rentable de cara al desarrollo. También veremos que es necesario para poder conectar Visual Modeler con Visual Basic y así trabajar con estas dos herramientas para poder implementar sistemas útiles con el menor esfuerzo posible.

Visual Modeler Add-in para Visual Basic

Lo primero que tenemos que hacer para poder trabajar con Visual Modeler y Visual Basic de manera cooperativa es incluir las herramientas necesarias al entorno de desarrollo de Visual Basic. De esta manera, desde el editor de Visual Basic, en el *Administrador de Complementos* tenemos que seleccionar los complementos *Visual Modeler Add-In* y *Visual Modeler Menus Add-In* de manera que queden cargados y se carguen al iniciar Visual Basic. Estos componentes nos permitirán establecer la conexión entre Visual Modeler y Visual Basic para que podamos obtener código partiendo de los modelos y modelos partiendo de código. En las figuras 86 y 87 siguiente podemos observar como cargar los componentes necesarios.

Desde la barra de menús de Visual Basic seleccionamos ***Complementos / Administrador de Complementos***. En ese instante nos aparecerá el formulario de control sobre los complementos.

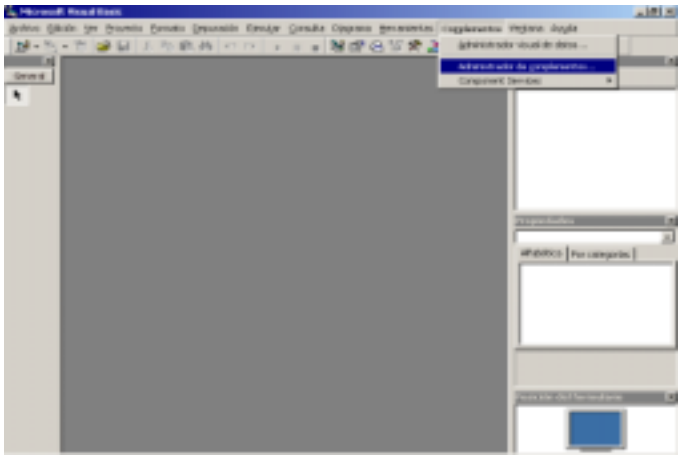


Figura 86. Selección del Administrador de Complementos de

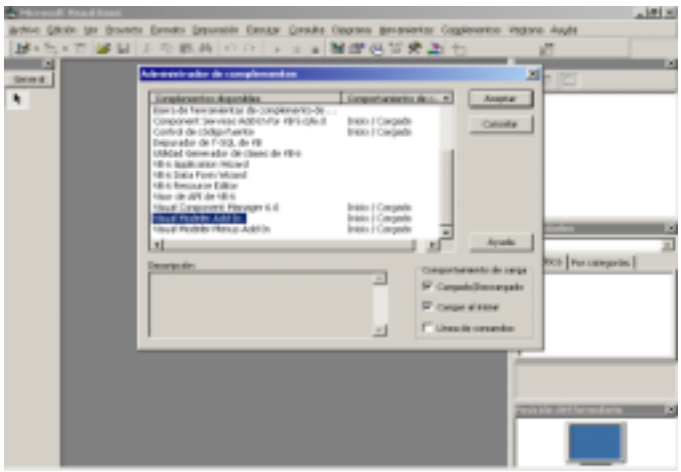


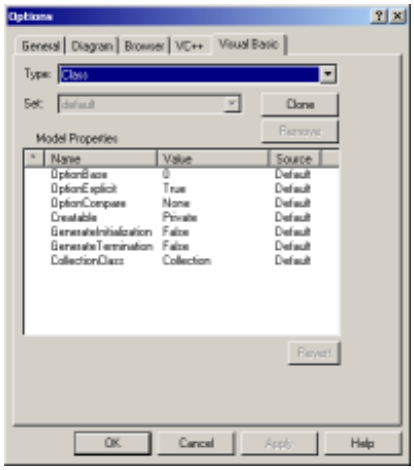
Figura 87. Complementos de Visual Modeler

Como podemos ver en la figura 87 hemos seleccionado los dos complementos de Visual Modeler para que queden cargados y se inicien automáticamente cuando se vuelva a arrancar Visual Basic la próxima vez.

Generación de Código desde Visual Modeler

A la hora de realizar un proceso de ingeniería directa debemos ser conscientes de las diferentes opciones que podemos configurar desde Visual Modeler. Existen diferentes opciones que se pueden configurar por defecto de manera que todos los elementos que creemos en Visual Modeler tengan esas opciones. También es posible modificar las opciones de generación de código para un elemento concreto, esto se puede conseguir una vez el elemento esté asignado a un componente y se haya seleccionado un lenguaje distinto del de análisis podemos modificar estas opciones desde el formulario de especificación del elemento (una vez que el elemento esté asignado a un componente nos aparecerá una nueva pestaña en el formulario que contiene las opciones para la generación de código).

A continuación vamos a describir cada una de las opciones posibles para la generación de código que podemos configurar por defecto. Para obtener la lista de opciones posibles para cada elemento debemos acceder al formulario de opciones para Visual Modeler, esto lo podemos conseguir desde el menú pulsando sobre ***Tools / Options...*** Una vez que obtenemos el formulario de opciones nos situamos en la pestaña del lenguaje que deseemos, Visual Basic o Visual C++, en este tema explicaremos todas las opciones de Visual Basic.



En la figura de la izquierda podemos ver el formulario de opciones en la pestaña de Visual Basic. En esta pestaña tenemos varios elementos interesantes:

- *Tipo de Elemento (Type):* Nos muestra una lista de los elementos disponibles sobre los que podemos seleccionar opciones de generación de código.
- *Propiedades del Modelo (Model Properties):* Esta es la lista de las opciones que podemos configurar por defecto para el elemento seleccionado.

Propiedades para las Clases

OptionBase

Esta propiedad indica que índice será tomado como inicial para los arrays de Visual Basic, por defecto se establece a cero, o sea, el índice del primer elemento de un array será el cero. El valor para esta propiedad debe ser cero o uno, si se especifica otro valor distinto Visual Modeler no generará el estamento *Option Base* a la hora de crear las clases en Visual Basic. Esta propiedad no tiene efecto sobre los arrays basados en tipos definidos por el usuario.

OptionExplicit

Esta opción es utilizada en Visual Basic para forzar la declaración de variables. Puede contener dos valores “True” o “False”. Si se selecciona el valor “True” se indica que se requiere la declaración de

variables para el modulo donde reside la declaración. Si se establece el valor a "False" se podrán utilizar variables sin que se hayan declarado.

OptionCompare

El estamento *Option Compare* opción es usado en Visual Basic para determinar el método usado por defecto a la hora de comparar tipos de datos string. Esta opción puede contener los siguientes valores: *None,* Visual Modeler no genera el estamento *Option Compare* cuando crea módulos de clase; *Text,* Visual Modeler genera el estamento *Option Compare Text* cuando crea módulos de clase; *Binary,* Visual Modeler genera el estamento *Option Compare Binary* cuando crea módulos de clase; *Database,* en este caso generará el estamento *Option Compare Database* cuando cree módulos de clase.

Creatable

Con esta propiedad se puede determinar como se expone la clase que se crea, o sea, el ámbito de la clase. Se puede elegir entre diferentes opciones, entre ellas: *Private*, valor por defecto, indica que la clase sólo va a ser visible dentro del proyecto que la contenga; *PublicNoCreatable*, este valor indica que la clase será vista por módulos que estén fuera del proyecto pero sólo se podrá instanciar dentro del proyecto que la contenga. Otros valores que puede contener son: *GlobalMultiUse, GlobalSingleUse, MultiUse y SingleUse*. Para encontrar más información sobre estos tipos de instanciación refiérase a la documentación de Visual Basic.

GenerateInitialization y GenerateTermination

Estas propiedades indican si cuando se genere el código de la clase se van ha incluir los métodos de iniciación y terminación del elemento. Si se establecen a "True", cuando se genere el esqueleto de la clase se crearán los métodos ***elemento*_Initialize** y ***elemento*_Terminate** (por ejemplo, de una clase se generarán los métodos Class_Initialize y Class_Terminate). Si se establecen a "False" no se generarán estos métodos.

CollectionClass

Esta propiedad es una de las más potentes que tienen las clases a la hora de generar código. Cuando en Visual Modeler tenemos dos clases relacionadas mediante una asociación, puede ser que un objeto de una de las clases se relacione con más de un objeto de la clase asociada, o sea, la multiplicidad de una de las clases de la asociación es distinta de uno (o de las dos clases). En este caso, cuando se genere código, la clase que en el lado opuesto de la que tiene multiplicidad distinta de uno va ha tener que contener una variable de Visual Basic para tener las instancias que se relacionan con ella.

Con la propiedad *CollectionClass* podemos determinar la clase que va a tener la colección de instancias. Por defecto contiene el valor **Collection**, eso quiere decir que la clase que va a contener las instancias será una variable de la clase Collection de Visual Basic (clase que puede mantener una colección de objetos de cualquier tipo). En la figura 88 se muestran dos clases relacionadas mediante una asociación unidireccional.

Para que la clase *Albarán* pueda implementar la relación que existe con *DetalleAlbarán*, ha de poder mantener una serie de instancias de esta clase ya que la multiplicidad es distinta de uno. El nombre del rol que tiene la clase *DetalleAlbarán* en la relación es *LineasDetalle*. Con esta configuración si la propiedad CollectionClass contiene su valor por defecto, o sea, Collection, cuando se genere el modulo de la clase *Albarán*, este contendrá una variable llamada mLineasDetalle de tipo Collection.

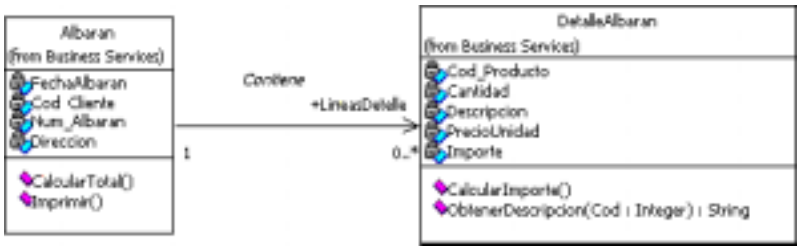


Figura 88. Asociación de dos Clases

El código fuente 3 muestra el código que se genera con esta opción.

```
Option Base 0
Option Explicit


'##ModelId=3884B20D0040
Private mFechaAlbaran As Date


'##ModelId=3884B2120173
Private mCod_Cliente As Long


##ModelId=3884B21D001A
Private mNum_Albaran As Long


##ModelId=3884B22401DD
Private mDireccion As String


##ModelId=3884B2EF0370
Private mLineasDetelle As Collection
```

Código fuente 3. Ejemplo de código con la propiedad CollectionClass establecida por defecto

Por el contrario, si seleccionamos otro valor para la propiedad CollectionClass, Visual Modeler generará un nuevo módulo con el nombre que contenga la propiedad CollectionClass donde se cree el código necesario para crear una nueva clase para mantener la colección de objetos *DetalleAlbarán*, con todos los métodos para añadir, insertar y acceder a los objetos, incluso generará los métodos necesarios para poder iterar con un bucle *For...Each* sobre una colección de objetos *DetalleAlbarán*. En el siguiente bloque de código se observa el código que se genera cuando se selecciona otra clase distinta de la que tiene por defecto para la propiedad CollectionClass para la clase *DetallesAlbarán* (en este caso se ha establecido la propiedad CollectionClass con un valor que contiene la cadena "**ColDetallesAlbaran**").

```
##ModelId=3884B22401DD
Private mDireccion As String

##ModelId=3884B2EF0370
Private mLineasDetelle As Collection
```

Código fuente 4. Código que se genera cuando se cambia el valor de la propiedad CollectionClass

El uso normal de esta propiedad es que por defecto contenga el valor *Collection* y para cada clase que valla a necesitar generar una colección modificar su valor dando un nombre que tenga sentido, para ello se puede acceder a la especificación de la clase y modificar el valor de la propiedad *CollectionClass* desde la pestaña de Visual Basic. Otra opción posible es modificar el valor de la propiedad CollectionClass cuando se ejecute el asistente para la generación de código (esta opción la analizaremos más adelante).

Propiedades para la relación de Generalización

ImplementsDelegation

Cuando establecemos una relación de generalización entre dos clases podemos establecer un control sobre como se va a convertir esa relación a la hora de generar código. Mediante la propiedad *ImplementsDelegation* podemos controlar en que casos se va a generar código que delegue la implementación de los métodos heredados de una superclase (clase de la que parte la relación de generalización). El valor por defecto es “True”, eso significa que Visual Modeler genera la declaración de objetos necesaria y pasa la ejecución de los métodos de la superclase a la subclase.

Por ejemplo si tenemos dos clases A y B, la clase B es una clase especializada de la clase A (B hereda de A) y la clase A contiene un método público llamado *Método()*, como aparece en la figura 89, Visual Modeler generará el código fuente 5.

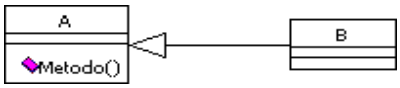


Figura 89. Ejemplo de Generalización

```
Option Base 0
Option Explicit
Implements A


‘local superclass object (generated)
‘##ModelID=38879EF0005C
Private mAObject As New A


‘##ModelID=38879EF00282
Private Sub A_Metodo()
      mAObject.Metodo
End Sub
```

Código fuente 5. Código generado por Visual Modeler

Propiedades para los Métodos

OperationName

Esta propiedad establece como se van a crear los nombres de los procedimientos o funciones para nuestras clases. Por defecto esta propiedad contiene el valor *$operation* esto significa que cuando se genere el método su nombre va a ser el mismo que el que contenga en la clase. Otra variables que podemos utilizar para generar el nombre del método son *$class* y *$package* de manera que si establecemos el valor de esta propiedad a *"$class_$operation"* cuando se genere el código se cambiará el nombre del método por el compuesto por el nombre de la clase, un guión bajo y el nombre de la clase. El valor de esta propiedad puede ser cambiado en el formulario de especificación del método y en el asistente de generación de código.

LibrayName

Esta propiedad indica el nombre de la librería donde va a residir el método declarado. Por defecto no contiene valor. El valor de esta propiedad puede ser cambiado en el formulario de especificación del método y en el asistente de generación de código.

AliasName

Define un nombre de alias valido. Suele ser útil cuando el nombre del método coincide con alguna palabra reservada de Visual Basic o cuando el nombre del método coincida con alguna variable global, con el nombre de alguna propiedad, una constante o con el nombre de otro método dentro del mismo alcance. El valor de esta propiedad puede ser cambiado en el formulario de especificación del método y en el asistente de generación de código.

IsStatic

Determina si el procedimiento o función va a ser generado utilizando la cláusula *Static* de Visual Basic. Por defecto el valor de la variable es "False". Si el valor se establece a "True" cuando se genere el método se utilizará la cláusula *Static*, o sea, los valores de las variables no se destruyen entre las sucesivas llamadas.

EntryCode y ExitCode

En estas propiedades podemos introducir código genérico para todos nuestros métodos de manera que cuando se genere el código del método se introduzcan las líneas de código que hemos establecido en esta propiedad. Por defecto, estas propiedades contienen el código necesario para el manejo de errores, si se quiere modificar el código de entrada o salida del método podemos utilizar las siguientes variables: *$class*, *$package*, *$operation* y *$type*. Esta propiedad puede ser modificada en el formulario de especificación del método o en el asistente de generación de código.

Propiedades para la especificación del Modulo

Module Specification

Aquí podemos determinar el archivo de proyecto Visual Basic (.vbp) que va a contener el código de definición. Por defecto, esta propiedad no contiene valor, eso significa que cuando se genere código Visual Modeler lo hará dentro del proyecto de Visual Basic que actualmente esté abierto siempre y cuando el nombre del proyecto sea el mismo que el del componente donde se ha asignado el elemento o si ya se ha realizado anteriormente algún proceso de ingeniería directa o inversa. En otro caso Visual Modeler creará un nuevo proyecto de Visual Basic para generar el código del elemento. Si en la propiedad se introduce una ruta de archivo de proyecto Visual Basic que sea válido, el código se generará en ese proyecto. Esta propiedad puede ser modificada en el formulario de especificación del método o en el asistente de generación de código.

Propiedades para la definición de una Propiedad o Rol

IsConst

Indica si la propiedad será generada como una constante. Por defecto contiene el valor “False”. Si se establece a “True” sólo tendrá efecto sobre propiedades que tengan un valor inicial establecido. Esta propiedad puede ser modificada en el formulario de especificación del método o en el asistente de generación de código.

New

Establece si cuando se genere la declaración de la variable para contener la propiedad en el modulo se va a incluir la cláusula *New* de Visual Basic. Esta propiedad puede ser modificada en el formulario de especificación del método o en el asistente de generación de código.

WithEvents

Establece si cuando se genere la declaración de la variable para contener la propiedad en el modulo se va a incluir la cláusula *WithEvents* de Visual Basic. Esta propiedad puede ser modificada en el formulario de especificación del método o en el asistente de generación de código.

Subscript

Indica los limites del array que Visual Modeler va a utilizar cuando genere el código. Por defecto no contiene valor, esto indica que no se va a generar un array. Si se utiliza un literal como por ejemplo "(5)", Visual Modeler utiliza ese literal para generar los límites del array. Si no se especifican más que los paréntesis Visual Modeler genera un array dinámico. Esta propiedad puede ser modificada en el formulario de especificación del método o en el asistente de generación de código.

NameIfUnlabeled

Especifica un valor para el nombre de la propiedad o rol si no está especificado. Por defecto contiene el valor *the$supplier*, que indica que se utilizará el literal "*the*" mas el nombre de la clase para nombrar la propiedad o rol. Se puede cambiar el valor de esta propiedad si se desea y se pueden utilizar las variables: *$class, $package, $relationship y $supplier*. Esta propiedad puede ser modificada en el formulario de especificación del método o en el asistente de generación de código.

GenerateDataMember

Indica si Visual Modeler generará variables en el ámbito del modulo para contener la propiedad o rol cuando se genere el código del elemento. Por defecto contiene valor "True", esto es, se generan una variable para contener la propiedad o rol. Si se establece a "False" después de generar el código para el elemento se tendrá que escribir el código necesario para mantener la propiedad o rol. Esta propiedad puede ser modificada en el formulario de especificación del método o en el asistente de generación de código.

GenerateGet/Let/SetOperation

Esta propiedad establece si se creará un método *Property Get/Let/Set* cuando se genere el miembro para mantener la propiedad o rol. Por defecto la propiedad está establecida a "False". Si se establece a "True" la primera vez que se genera el código para el elemento se crea el método *Property Get/Let/Set* y automáticamente se establece de nuevo a "False" (la siguiente vez ya no habrá que generar el método). Para más información sobre la manera de nombrar estos métodos véase la ayuda de Visual Modeler. Esta propiedad puede ser modificada en el formulario de especificación del método o en el asistente de generación de código

Asistente para la Generación de Código Visual Basic

Cuando hemos definido un modelo completamente, tenemos todos las clases e interfaces que nos van a hacer falta y hemos establecido los componentes que vamos a necesitar estableciendo el lenguaje a Visual Basic y asignando en cada caso qué clases van a realizar cada componente, estamos en disposición de generar el esqueleto del código para implementar nuestro sistema.

Tenemos que tener en cuenta que cada componente se va a convertir en un proyecto de Visual Basic. Para iniciar la el asistente para la generación de código lo más usual es obtener el menú contextual del componente del cual vamos a obtener el esqueleto del código y pulsar la entrada de generación del código (*Generate Code*). A continuación vamos a ver cada paso que hemos de seguir para obtener el código de nuestros componentes.

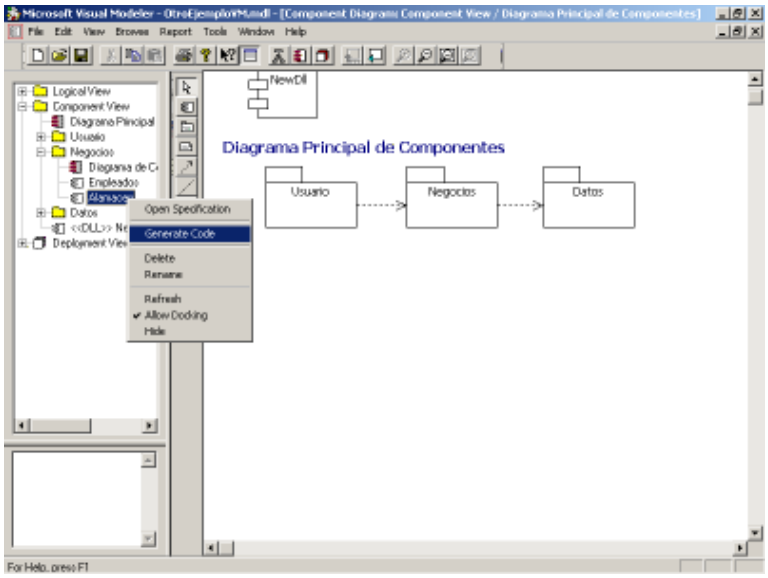
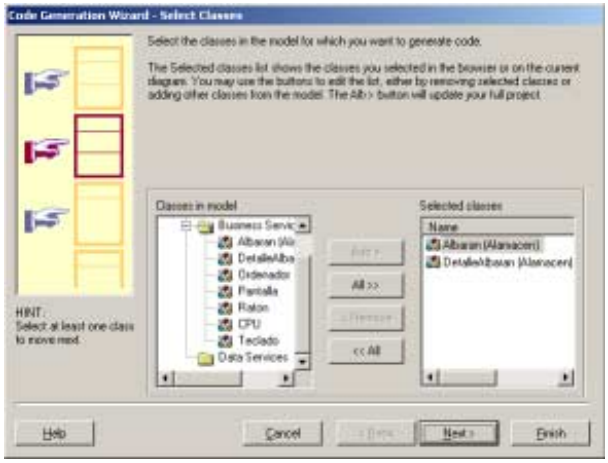
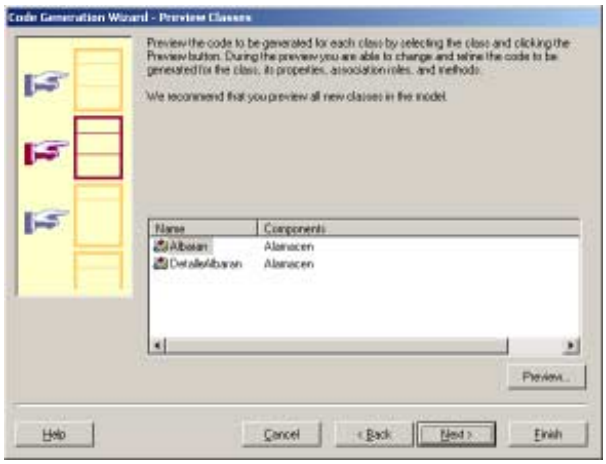


Figura 90. Inicio del Asistente para la Generación de Código.



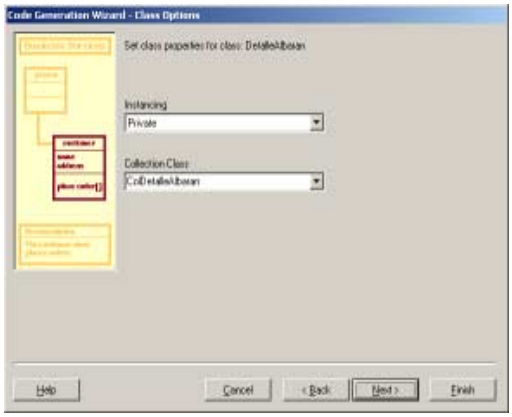
Paso 1

En este paso del asistente nos aparecen las clases que nuestro componente implementa por defecto. Aquí estamos en disposición de añadir más clases o quitar algunas de las que están incluidas.



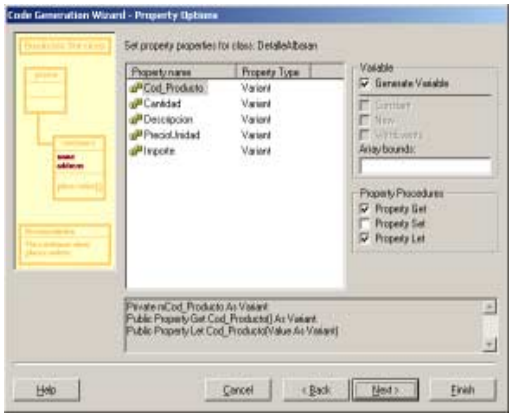
Paso 2

En este paso podemos ver la lista de clases o interfaces que hemos seleccionado para realizar la generación de código. Aquí podemos ver todas las características de cada clase seleccionando clase por clase y pulsando el botón de previsualización (*Preview*). El paso de previsualización es necesario sobre todo para clases que se generen por primera vez.



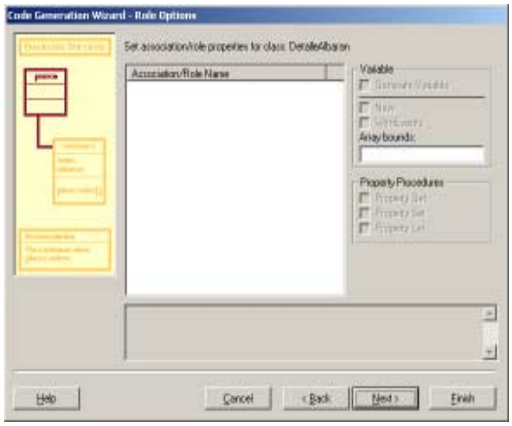
Paso 3 (Paso 1 de la previsualización)

En este paso podemos modificar la propiedad del modulo para la instanciación (*Instancing*, esta propiedad se corresponde con la propiedad *Creatable* de las clases). También podemos modificar la propiedad *CollectionClass*, si ahora seleccionamos otro nombre distinto de Collection generará el código necesario para crear el modulo de clase para la colección de nuestros objetos.



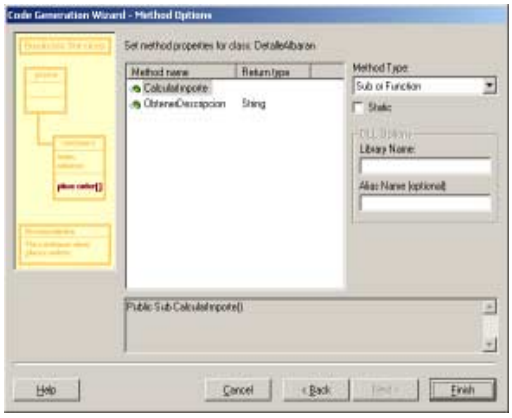
Paso 4 (Paso 2 de la previsualización)

Aquí paso podemos determinar la configuración de las propiedades, si se va a generar variable o no, si se generan métodos *Property* y ver el aspecto del código que se va a generar por cada propiedad de la clase. Aquí no podremos modificar valores para los tipos de datos, así que si alguna de las propiedades no tiene un tipo de datos correcto hemos de parar la generación.



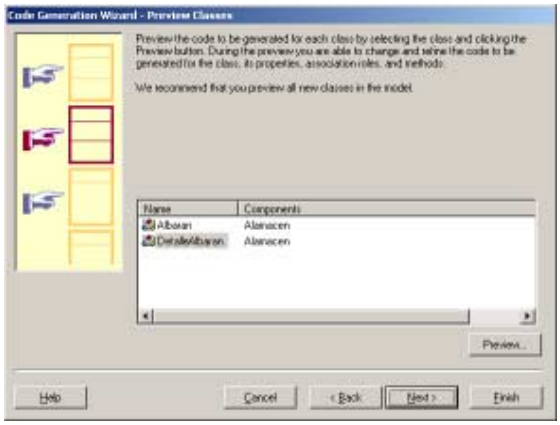
Paso 5 (Paso 3 de la previsualización)

En este paso, al igual que en el paso anterior podemos determinar la configuración de las propiedades para cada uno de los roles que juega nuestra clase en distintas relaciones de asociación. En este caso no tenemos ningún rol que configurar.



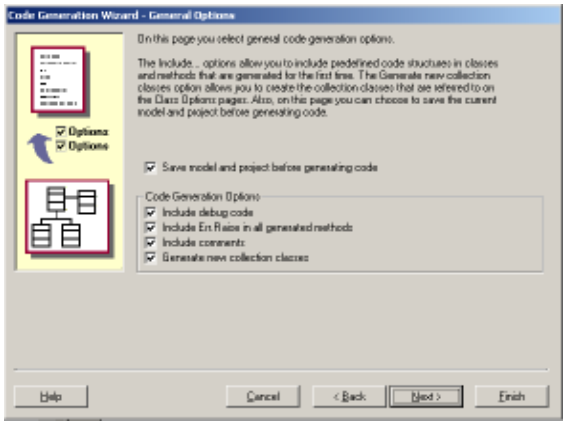
Paso 6 (Paso 4 de la previsualización)

En este instante estamos en disposición de establecer la configuración para los métodos de nuestra clase. Podemos cambiar el tipo de método que se va a generar, la determinación de si va a ser estático o no y en algunos casos el nombre de la librería donde se implementa y el alias para el nombre. Este es el último paso de la previsualización.



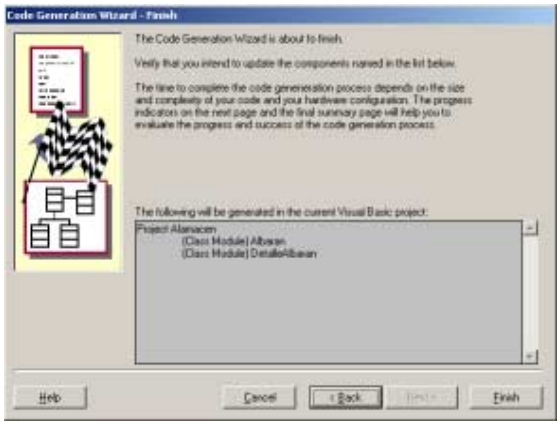
Paso 7

Repetiremos el paso de la previsualización para cada clase que lo requiera (en general las que no se hayan generado nunca), una vez terminada la configuración de cada clase pasaremos al siguiente paso del asistente.



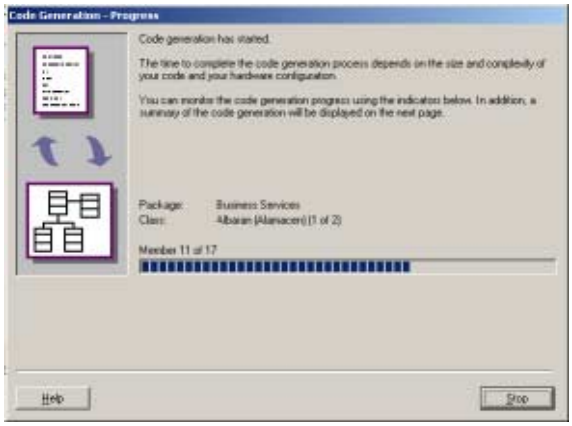
Paso 8

Aquí podemos seleccionar algunas opciones que pueden ser de utilidad. Podemos incluir o no: código para la depuración, generación de errores, comentarios. Otro aspecto importante es que podemos determinar si se van a generar las nuevas clases de colección o no. Si no seleccionamos esta opción tendremos que implementarlas nosotros mismos.



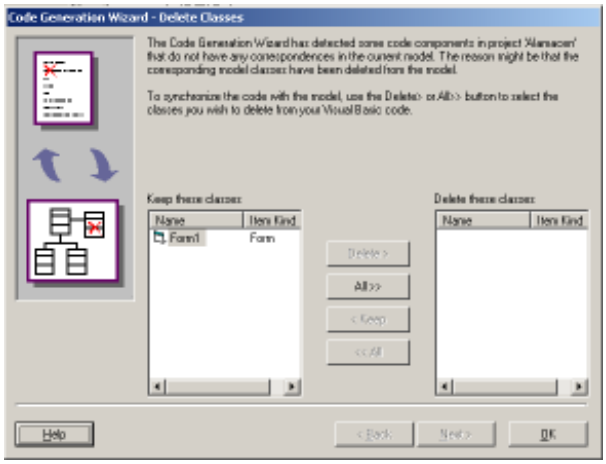
Paso 9

Este es el paso previo al inicio de la generación de código, aquí podemos observar las clases que vamos a crear y aun estamos a tiempo de volver hacia atrás para modificar alguno de los parámetros que tengamos definidos para cada clase. Ahora estamos en posición de finalizar la generación del código.



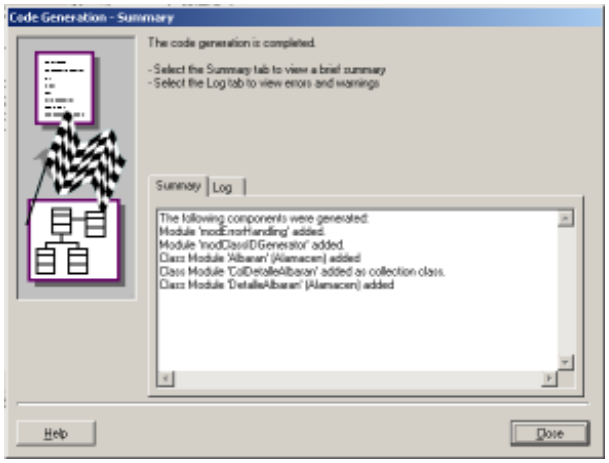
Paso 10

Aquí podemos observar el progreso de nuestra generación del esqueleto de las clases que se realizan en el componente que hemos seleccionado. Cuando termine el asistente nos habrá abierto un nuevo proyecto de Visual Basic con el nombre del componente y en el estarán las clases que dicho componente realiza.



Paso 11

En este paso se muestran los elementos que se han agregado al proyecto creado por Visual Modeler pero que no estaban en el componente. En este caso se muestra el formulario *Form1* que Visual Basic siempre añade al crear un nuevo proyecto. Si lo seleccionamos Visual Modeler lo eliminará del proyecto.



Paso 12

Por último, el asistente nos muestra un sumario donde podemos ver todos los elementos que ha creado y tenemos la posibilidad de inspeccionar un fichero Log donde se han registrado todas las acciones ordenadas por tiempo que Visual Modeler ha generado.

En la figura 91 podemos observar el aspecto del proyecto Visual Basic que Visual Modeler ha generado para nuestro componente. Como podemos ver en la figura, se han creado módulos específicos para la gestión de errores así como el modulo que ha creado para implementar la colección de objetos *DetallesAlbarán*

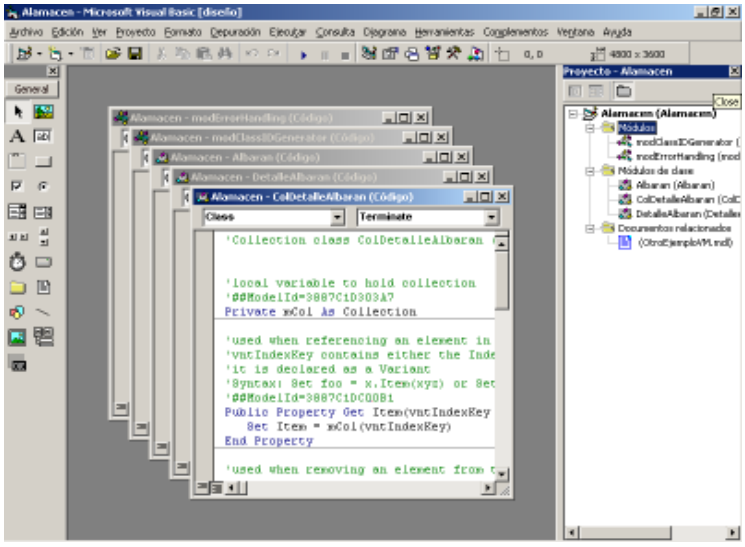
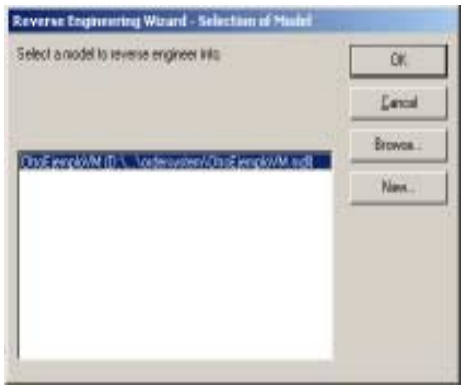


Figura 91. Proyecto de Visual Basic después de ejecutar el Asistente para generación de código

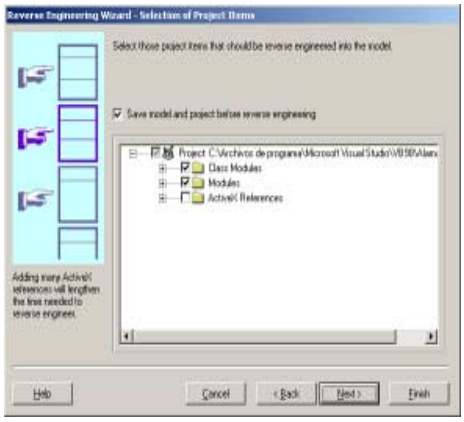
Asistente para ingeniería inversa

Después de haber ejecutado el asistente para la generación del código estamos en disposición de implementar nuestras clases y modificar algunos aspectos que consideramos que son necesarios. Una vez que hemos cambiado y añadido el código necesario para implementar nuestro sistema podemos sincronizar nuestro proyecto Visual Basic con el modelo de Visual Modeler del cual hemos partido, para ello utilizaremos el asistente para ingeniería inversa. Podemos lanzar el asistente para ingeniería inversa tanto desde Visual Modeler (menú ***Tools / Visual Basic / Reverse Engineering Wizard...***) como desde Visual Basic (menú **Complementos / Visual Modeler / Reverse Engineering Wizard...**). Una vez que hemos lanzado el asistente para ingeniería inversa, podemos ir paso a paso para seleccionar que elementos son los que queremos que se haga ingeniería inversa. A continuación vemos cada uno de estos pasos.



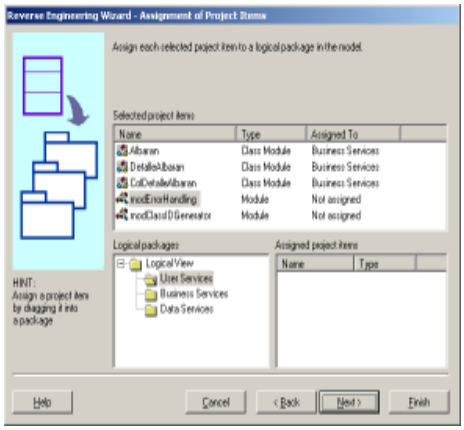
Paso 1

Lo primero que tenemos que hacer es seleccionar el modelo sobre el que queremos realizar la ingeniería inversa. Por defecto nos aparece el modelo del que proviene el código de este proyecto, aunque si el proyecto no hubiera sido creado por el asistente para generación de código de Visual Modeler podríamos crear un nuevo modelo.



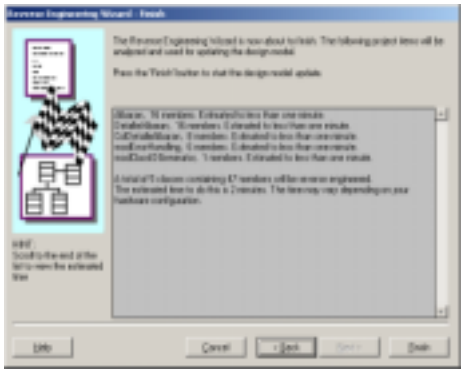
Paso 2

Una vez seleccionado el modelo sobre el que vamos a trabajar podemos seleccionar de que módulos del proyecto vamos a realizar ingeniería inversa. En este punto también podemos seleccionar librerías ActiveX que use nuestro proyecto aunque no las hayamos implementado nosotros.



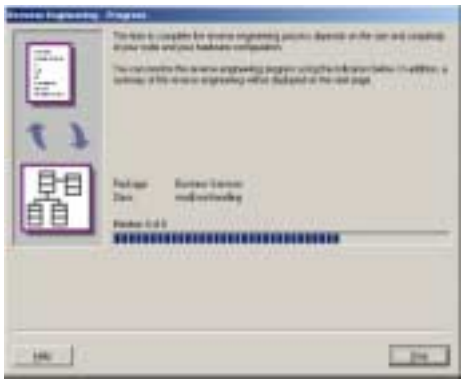
Paso 3

En este paso debemos determinar que módulos van a cada uno de los paquetes lógicos. Para ello, los módulos que no estén asignados los arrastraremos hasta el paquete lógico al que consideramos que deben ir.



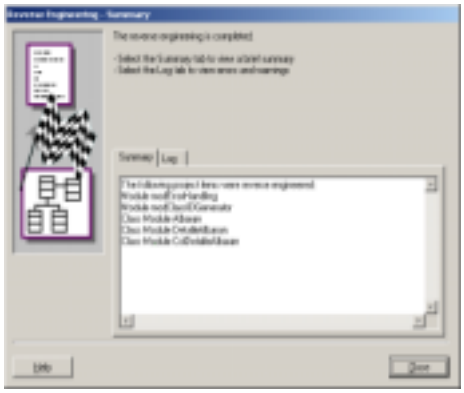
Paso 4

En este paso el asistente nos muestra un pequeño informe sobre los módulos que van a ser procesados y una estimación sobre el tiempo que se va a tardar en realizar el proceso de ingeniería inversa.



Paso 5

En este paso podemos ver el asistente de ingeniería inversa procesando los módulos que hemos seleccionado.



Paso 6

Por último podemos ver el sumario y el fichero Log sobre las acciones que Visual Modeler ha realizado con el modelo.

En la figura 92 podemos ver como queda nuestro modelo después de haber realizado el proceso de ingeniería inversa.