FACULTAD DE INGENIERÍA UNAM
DIVISIÓN DE EDUCACIÓN CONTINUA

MATERIAL DIDACTICO DEL CURSO

PROGRAMACIÓN ORIENTADA A OBJETOS

ING. EDGAR E. GARCIA CANO CASTILLO
8 al 12 de octubre de 2007

CC62

Palacio de Minería, Calle de Tacuba # 5-Entrepiso, Centro Histórico, Delegación Cuauhtémoc, Código Postal 06000,
México D.F. Tel/FAX 5510-4673, Teléfonos 5623-2947 y 5623-2960

Universidad Nacional Autónoma de México

Facultad de Ingeniería

División de Educación Continua

Curso de Metodología Orientada a Objetos y UML

Duración: 20 hrs.

Instructor:

Ing. Edgar Eduardo García Cano Castillo.

Índice

1. Introducción a Análisis y Diseño Orientado a Objetos.

A medida que se acercaban los años 80, la metodología orientada a objetos comenzaba a madurar como un enfoque de desarrollo de software. Empezamos a crear diseños de aplicaciones de todo tipo utilizando la forma de pensar orientada a los objetos e implementamos (codificamos) programas utilizando lenguajes y técnicas orientadas a los objetos.

La metodología orientada a objetos presenta características que lo hacen idóneo para el análisis, diseño y programación de sistemas; sin embargo, el análisis de requisitos, que es la relación entre la asignación de software al nivel del sistema y el diseño del software, se quedó atrás por lo que empezaron a surgir diferentes métodos de análisis y diseño orientado a objetos, entre los que destacan los métodos *Booch*, *OOSE* (Object Oriented Software Engineering), *Fusion* y *OMT* (Object Modeling Technique). Para poner fin a la "guerra de métodos" que se presentó en ese momento, se creó el *Lenguaje Unificado de Modelado* (UML).

Los métodos orientados a objetos centran su atención en objetos y clases:

- Un objeto es "una cosa".
- Una clase es "una plantilla o definición de cosas".

Los métodos orientados a objetos crean modelos con los cuales:

- Se pueden crear clases y objetos.
- Definen su estructura, comportamiento y propósito.
- Definen la relación entre clases.
- Definen la relación entre objetos.

La Promesa de la Orientación a Objetos:

- Los requerimientos y la arquitectura pueden ser capturados en una manera repetida y razonable.
- Los objetos modelan el comportamiento del mundo real, y son fáciles de entender.
- Los componentes pueden ser reutilizados a través de interfaces bien definidas.
- El cambio de requerimientos es fácil de soportar.
- Los costos de mantenimiento son reducidos considerablemente.

A fin de cuentas el Análisis y Diseño Orientado a Objetos (ADOO) es una técnica para modelar sistemas, que se encarga de describir y modelar el sistema como si fuera un conjunto de objetos relacionados que interactúan entre sí.

2. Objetos

La metodología orientada a objetos ha derivado de las metodologías anteriores a éste. Así como los métodos de diseño estructurado realizados guían a los desarrolladores que tratan de construir sistemas complejos utilizando algoritmos como sus bloques fundamentales de construcción, similarmente los métodos de diseño orientado a objetos han evolucionado para ayudar a los desarrolladores a explotar el poder de los lenguajes de programación basados en objetos y orientados a objetos, utilizando las clases y objetos como bloques de construcción básicos.

Actualmente el modelo de objetos ha sido influenciado por un número de factores no sólo de la *Programación Orientada a Objetos*, (*POO, Object Oriented Programming*, *OOP* por sus siglas en inglés). Además, el modelo de objetos ha probado ser un concepto uniforme en las ciencias de la computación, aplicable no sólo a los lenguajes de programación sino también al diseño de interfaces de usuario, bases de datos y arquitectura de computadoras por completo. La razón de ello es, simplemente, que una orientación a objetos nos ayuda a hacer frente a la inherente complejidad de muchos tipos de sistemas.

2.1 Definición

Se define a un objeto como "*una entidad tangible que muestra alguna conducta bien definida*". Un objeto "*es cualquier cosa, real o abstracta, acerca de la cual almacenamos datos y los métodos que controlan dichos datos*".

Los objetos tienen una cierta "integridad" la cual no deberá ser violada. En particular, un objeto puede solamente cambiar estado, conducta, ser manipulado o estar en relación con otros objetos de manera apropiada a este objeto.

Para identificar un objeto:

- Necesita existir independientemente.
- Debe tener atributos y operaciones.
- Debe ser relevante para el dominio del problema.

Un objeto debe tener una integridad, que es necesaria para que podamos interactuar con él sin confundirlo con otro, y para que puedan existir varios objetos de la misma clase.

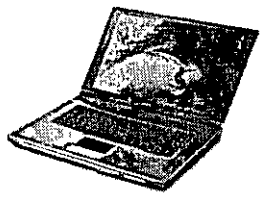
En las siguientes figuras se muestran ejemplos de algunos objetos.

Bicicleta

Antena

Taza

Cuenta de Banco



Computadora

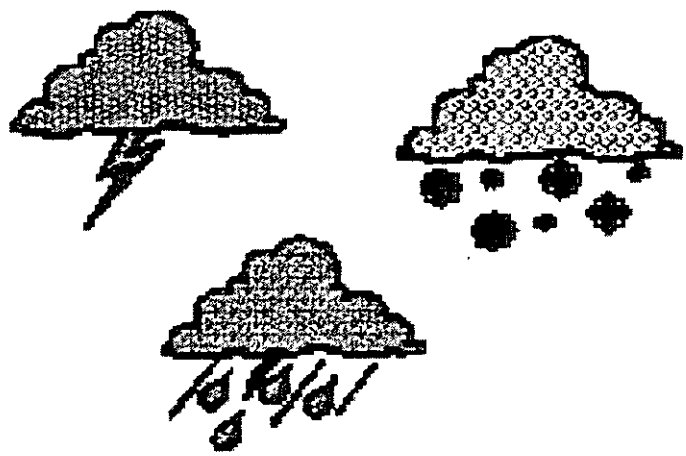
Tenis

Figuro 1. Objetos.

En la figura 2 se pueden obsevar los atributos y métodos de las nubes.

Atributos de las nubes: Tamaño, sombra, contenido de agua.

Operaciones de las nubes: Llover, nevar, tronar.



Figuro 2. Atributos y métodos de las nubes

Caso de estudio: Ejercicio 1

De acuerdo con el Apéndice A, revisar la siguiente información acerca del caso de estudio.

Tomando en consideración el dominio del problema de CirectClothing, se tienen los siguientes sustantivos. ¿Cuáles pueden ser catalogados como un objeto del sistema?

- *catálogo*
- *orden por correo*
- *ropa*
- *artículos*
- *artículos en descuento*
- *artículos mensuales*
- *artículos normales*
- *orden*
- *cliente*
- *pago*
- *almacén*
- *tarjeta de crédito*
- *orden por fax*
- *orden en línea*
- *inventario*
- *sistema*
- *Internet*
- *negocio*
- *año*
- *mes*
- *forma para orden*
- *cheque*
- *secretario(a)*

2.2 Abstracción.

La abstracción consiste en ignorar detalles y encontrar las principales características, de ésta manera se simplifica cómo los usuarios interactúan con los objetos. A través de esto nos enfocamos en los aspectos esenciales o distintivos de algo, ignorando detalles irrelevantes.

Hay dos tipos de abstracción:

- Abstracción funcional: Es lo que hace, acciones y eventos.

- Abstracción de datos: Son las características y las propiedades.

Finalmente los objetos que encontramos al analizar el dominio del problema y que tienen la misma estructura y comportamiento, forman parte de una clase. Así que lo que un objeto puede saber (estado) o hacer (comportamiento), está determinado por la clase a la que pertenece.

Caso de estudio: Ejercicio 2

Encontrar todos los atributos y métodos de los objetos encontrados en el Ejercicio 1

2.3 Encapsulación.

El encapsulamiento es guardar o poner algo dentro de un objeto, de esta manera se separan se separan los aspectos externos de un objeto, de los detalles internos de implementación que se ocultan en los objetos. La ventaja de la encapsulación consiste en que oculta los detalles internos, si la clase cambiara sólo afecta la implementación oculta, de tal forma que no se afecte toda la clase.

La encapsulación está formada por:

- *Interfaz pública*: Son las operaciones que podemos hacer con un objeto para interactuar con él.
- *Implementación*: Son las operaciones internas, qué se puede hacer o el propósito del objeto.
- *Información interna*: Son los datos, atributos escondidos y que son necesarios para complementar la función.

Los atributos y operaciones son miembros del objeto y pueden ser de tipo público o privado. Si el miembro es público, forma parte de la interfaz pública, de lo contrario forma parte de la implementación. En Sistemas Orientados a Objetos todos los atributos son privados y pueden ser accesados con operaciones públicas.

Caso de estudio: Ejercicio 3

Agregar los métodos encapsulados que deben llevar los objetos a los se le agregaron atributos y métodos en el ejercicio 2.

3. Clases

3.1 Definición.

Una *clase* es una plantilla para objetos múltiples con características similares. Las clases comprenden todas esas características de un conjunto particular de objetos. Cuando se escribe un programa en lenguaje orientado a objetos, no se definen objetos verdaderos sino se definen clases de objetos.

Una *instancia* de una clase es otro término para un objeto real. Si la clase es la representación general de un objeto, una instancia es su representación concreta. A menudo se utiliza indistintamente la palabra objeto o instancia para referirse, precisamente, a un objeto.

En los lenguajes orientados a objetos, cada clase está compuesta de dos cualidades: *atributos* (estado) y *métodos* (comportamiento o conducta). Los atributos son las características individuales que diferencian a un objeto de otro (ambos de la misma clase) y determinan la apariencia, estado u otras cualidades de ese objeto. Los atributos de un objeto incluyen información sobre su estado.

Los métodos de una clase determinan el comportamiento o conducta que requiere esa clase para que sus instancias puedan cambiar su estado interno o cuando dichas instancias son llamadas para realizar algo por otra clase o instancia. El comportamiento es la única manera en que las instancias pueden hacerse algo a sí mismas o tener que hacerles algo. Los atributos se encuentran en la parte interna mientras que los métodos se encuentran en la parte externa del objeto (**figura 3**).

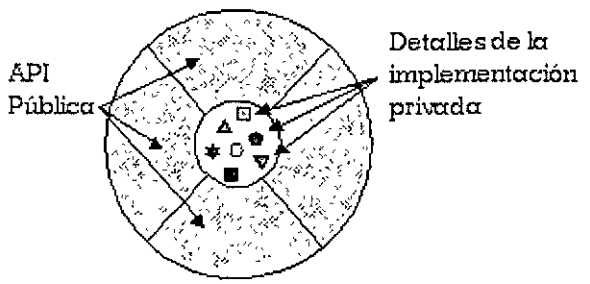


Figura 3. Representación visual de un objeto como componente de software.

Para definir el comportamiento de un objeto, se crean métodos, los cuales tienen una apariencia y un comportamiento igual al de las funciones en otros lenguajes de programación, los lenguajes estructurados, pero se definen dentro de una clase. Los métodos no siempre afectan a un solo objeto; los objetos también se comunican entre sí mediante el uso de métodos. Una clase u objeto puede llamar métodos en otra clase u objeto para avisar sobre los cambios en el ambiente o para solicitarle a ese objeto que cambie su estado.

Cualquier cosa que un objeto no sabe, o no puede hacer, es excluida del objeto. Además, como se puede observar de los diagramas, las variables del objeto se localizan en el centro o núcleo del objeto. Los métodos rodean y esconden el núcleo del objeto de otros objetos en el programa (encapsulamiento).

Esta imagen conceptual de un objeto —un núcleo de variables empaquetadas en una membrana protectora de métodos— es una representación ideal de un objeto y es el ideal por el que los diseñadores de sistemas orientados a objetos luchan. Sin embargo, no lo es todo: a menudo, por razones de eficiencia o la puesta en práctica, un objeto puede querer exponer algunas de sus variables o esconder algunos de sus métodos.

Caso de estudio: Ejercicio 4

Para cada objeto identificado en el Ejercicio 3, identificar las clases correspondientes a cada objeto.

3.2 Generalización

La generalización se encarga de identificar y definir los atributos y operaciones comunes en una colección de objetos, de ésta manera se reduce la redundancia en el proceso de desarrollo y también ayuda a la reutilización.

Caso de estudio: Ejercicio 5

Cuando se evaluaron los objetos cliente, representante de ventas y secretario(a), se encontraron características similares, encontrar ¿Cuáles serían las clases generales a estas?

3.3 Herencia

Otro concepto muy importante en la metodología orientada a objetos es el de *herencia*. La herencia es un mecanismo poderoso con el cual se puede definir una clase en términos de otra clase; lo que significa que cuando se escribe una clase, sólo se tiene que especificar la diferencia de esa clase con otra, con lo cual, la herencia dará acceso automático a la información contenida en esa otra clase.

Con la herencia, todas las clases están arregladas dentro de una jerarquía estricta. Cada clase tiene una superclase (la clase superior en la jerarquía) y puede tener una o más subclases (las clases que se encuentran debajo de esa clase en la jerarquía). Se dice que las clases inferiores en la jerarquía, las clases hijas, heredan de las clases más altas, las clases padres.

Las subclases heredan todos los métodos y variables de las superclases. Es decir, en alguna clase, si la superclase define un comportamiento que la clase hija necesita, no se tendrá que redefinir o copiar ese código de la clase padre (**figura 4**).

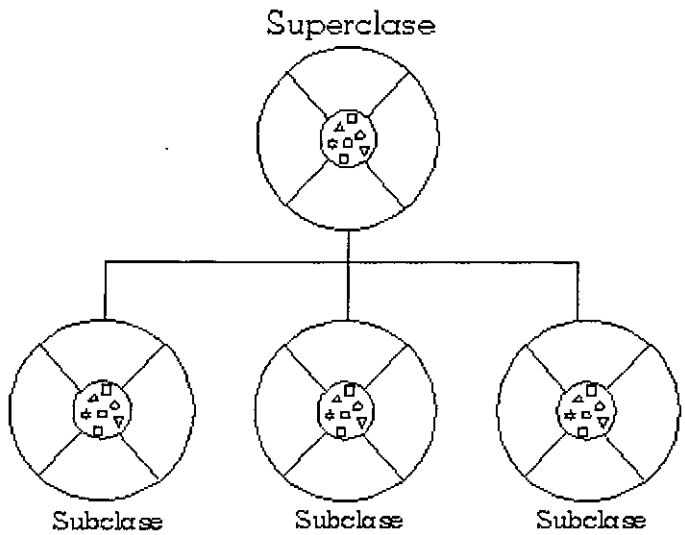


Figura 4. Clases, sub y super.

De esta manera, se puede pensar en una jerarquía de clase como la definición de conceptos demasiado abstractos en lo alto de la jerarquía y esas ideas se convierten en algo más concreto conforme se desciende por la cadena de la superclase.

Sin embargo, las clases hijas no están limitadas al estado y conducta provistos por sus superclases; pueden agregar variables y métodos además de los que ya heredan de sus clases padres. Las clases hijas pueden, también, sobrescribir los métodos que heredan por implementaciones especializadas para esos métodos. De igual manera, no hay limitación a un sólo nivel de herencia por lo que se tiene un árbol de herencia en el que se puede heredar varios niveles hacia abajo y mientras más niveles descienda una clase, más especializada será su conducta.

La herencia presenta los siguientes beneficios:

- Las subclases proveen conductas especializadas sobre la base de elementos comunes provistos por la superclase. A través del uso de herencia, los programadores pueden reutilizar el código de la superclase muchas veces.

- Los programadores pueden implementar superclases llamadas clases abstractas que definen conductas "genéricas". Las superclases abstractas definen, y pueden implementar parcialmente, la conducta pero gran parte de la clase no está definida ni implementada. Otros programadores concluirán esos detalles con subclases especializadas.

3.4 Especialización

La especialización se centra en crear nuevas clases más específicas que heredan de una clase mayor. La especialización es una herencia, la cual adiciona y modifica métodos para resolver un problema específico.

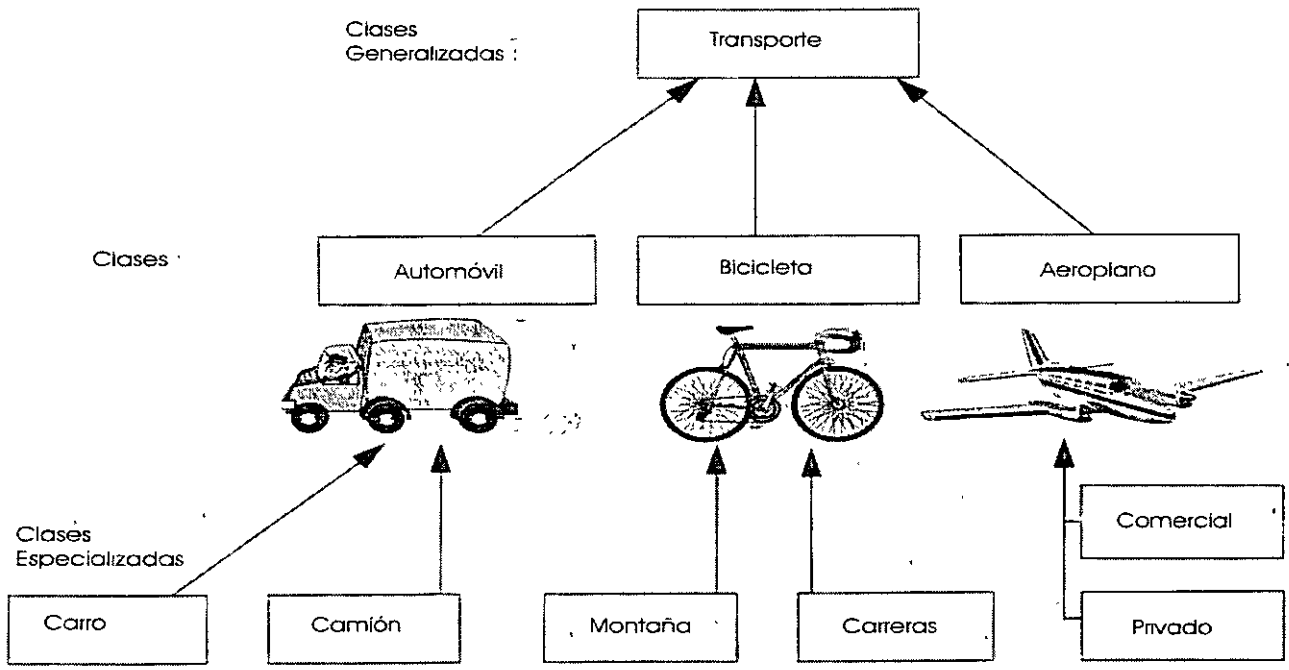


Figura 5. Generalización y Especialización

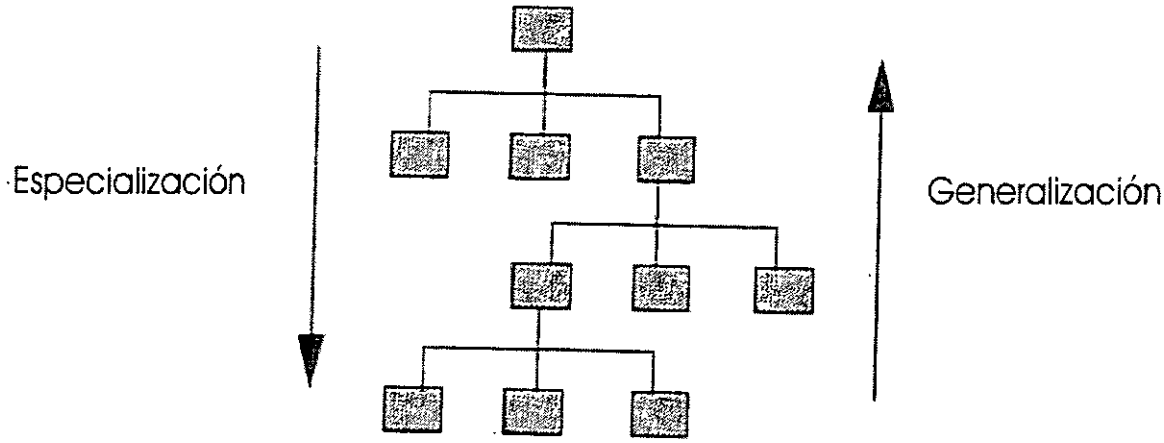


Figura 6. Generalización y Especialización

Caso de Estudio: Ejercicio 6

¿Qué otras clases se pueden encontrar en el sistema de DirectClothing, de acuerdo con los conceptos vistos?

3.5 Polimorfismo

El polimorfismo (varias formas) permite implementar una operación de herencia en una subclase (operaciones heredadas de una superclase). El polimorfismo significa que la operación existe en muchas clases, la operación tiene el mismo significado, pero cada clase personaliza la operación.

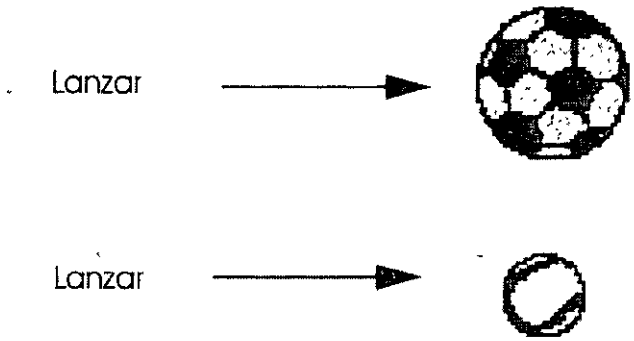


Figura 7. Polimorfismo

Caso de Estudio: Ejercicio 7

¿Qué métodos polimórficos se pueden encontrar para las clases que representan el sistema?

3.6 Clases Abstractas

Una clase abstracta, es una clase que tiene una operación que no está definida (las operaciones están especificadas, pero no tienen código), se sabe que va a hacer, pero no cómo lo va ha hacer, por éste motivo se le llama clase abstracta. Una subclase proveniente de una clase abstracta necesita implementar las operaciones abstractas, de lo contrario será una clase abstracta.

Las clases abstractas pueden tener una, todas o ninguna operación definida y se pueden tomar como plantilla para crear otras, pero no se puede crear una instancia de ellas.

Caso de Estudio: Ejercicio 8

¿Qué clases pueden se abstractas en el sistema de DirectClothing?

4. Interacción entre Objetos

Los objetos interactúan entre sí, esto lo hace por medio del envío de mensajes. Un ejemplo sobre esto es una llamada entre dos personas.

La manera de enviar mensajes entre objetos, depende de la naturaleza de las mismas y se aplican el siguiente proceso:

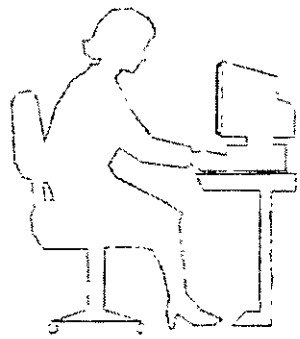
- Un objeto se encarga de enviar un mensaje a otro objeto.
- El objeto receptor del mensaje, puede mandar a su vez más mensajes, reaccionando de acuerdo con el mensaje que está recibiendo y que es manejado por la interfaz pública de éste objeto.

Un ejemplo de lo anterior es cuando una persona intercala con un radio, la persona maneja el radio con su interfaz pública que es su sintonizador.

Todos los objetos interactúan a través de la asociación, agregación o composición, descritos en los siguientes apartados.

4.1 Asociación

La asociación se produce entre dos objetos independientes que colaboran para realizar un objetivo, un objeto se beneficia de los servicios del otro; un ejemplo es cuando una persona "USA" una computadora. Esta es una relación débil.



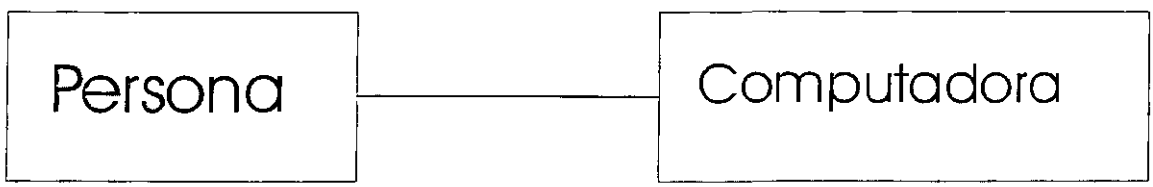


Figura 8. Asociación

4.2 Agregación

La agregación se produce entre dos objetos independientes, cuando uno de ellos utiliza al otro. Un ejemplo es cuando una computadora "TIENE UN" mouse. Esta es una relación regular.

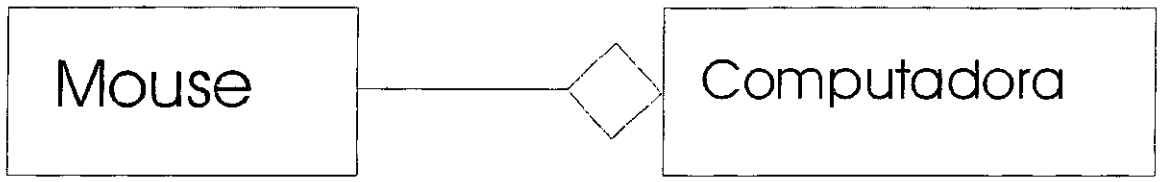


Figura 9. Agregación

4.3 Composición

La composición de dos objetos se produce cuando un esta compuesto por otro. Un ejemplo es cuando una computadora "ESTÁ COMPUESTA" o "SIEMPRE TIENE" un cpu. Esta es una relación fuerte.

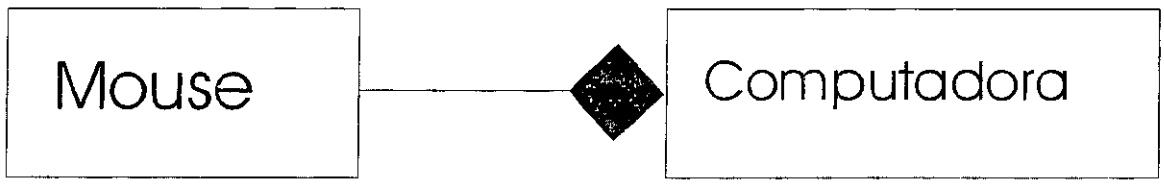


Figura 10. Composición

El tiempo de vida de los objetos depende del tipo de relación que estos presenten. Cuando dos objetos están asociados o agregados, su tiempo se traslapa. Un aeroplano se compone de varias piezas, que juntas hacen que el aeroplano pueda volar, sin embargo el tiempo de vida del aeroplano termina cuando el tiempo de vida de las alas se termina, Pero a pesar de esto, los objetos que componen al aeroplano son creados en diferente tiempo y pueden morir de igual manera en otro tiempo.

En una relación de composición, el tiempo de vida entre dos objetos es el mismo. Si un objeto muere, el otro también. Son creados al mismo tiempo y mueren igual.

Caso de estudio: Ejercicio 9

> *Hacer las relaciones debidas entre las clases, de acuerdo con los conceptos de Asociación, Agregación y Composición.*

5. Diseño Orientado a Objetos usando UML

En todas las disciplinas de la Ingeniería se hace evidente la importancia de los modelos ya que describen el aspecto y la conducta de "*algo*". Ese "*algo*" puede existir, estar en un estado de desarrollo o estar, todavía, en un estado de planeación. Es en este momento cuando los diseñadores del modelo deben investigar los requerimientos del producto terminado y dichos requerimientos pueden incluir áreas tales como funcionalidad, desempeño y confiabilidad. Además, a menudo, el modelo es dividido en un número de vistas, cada una de las cuales describe un aspecto específico del producto o sistema en construcción.

El modelado sirve no solamente para los grandes sistemas, aun en aplicaciones de pequeño tamaño se obtienen beneficios de modelado, sin embargo es un hecho que entre más grande y más complejo es el sistema, más importante es el papel de que juega el modelado por una simple razón: "El hombre hace modelos de sistemas complejos porque no puede entenderlos en su totalidad".

5.1 Definición

UML es una técnica para la especificación sistemas en todas sus fases. Nació en 1994 cubriendo los aspectos principales de todos los métodos de diseño antecesores y, precisamente, los padres de UML son Grady Booch, autor del método Booch; James Rumbaugh, autor del método OMT e Ivar Jacobson, autor de los métodos OOSE y Objectory. La versión 1.0 de UML fue liberada en Enero de 1997 y ha sido utilizado con éxito en sistemas construidos para toda clase de industrias alrededor del mundo: hospitales, bancos, comunicaciones, aeronáutica, finanzas, etc.

El UML es una notación gráfica y textual, rica y expresiva, que permite textos si las gráficas pudiesen hacerse difíciles de entender. No tiene un proceso, no se aplica como una receta de cocina. Cada diagrama de UML brinda una vista distinta del sistema en cuestión. El UML no tiene un ciclo de vida. Se utiliza en un ciclo iterativo e incremental. Asimismo, las siguientes ventajas son de importancia:

- Mejores tiempos totales de desarrollo (de 50 % o más).
- Modelar sistemas (y no sólo de software) utilizando conceptos orientados a objetos.
- Establecer conceptos y artefactos ejecutables.
- Encaminar el desarrollo del escalamiento en sistemas complejos de misión crítica.
- Crear un lenguaje de modelado utilizado tanto por humanos como por máquinas.
- Mejor soporte a la planeación y al control de proyectos.
- Alta reutilización y minimización de costos.

5.2 Vistas de UML

Las vistas muestran diferentes aspectos del sistema modelado. Una vista no es una gráfica, pero sí una abstracción que consiste en un número de diagramas y todos esos diagramas juntos muestran una "fotografía" completa del sistema. Las vistas también ligan el lenguaje de modelado a los métodos o procesos elegidos para el desarrollo. Las diferentes vistas que UML tiene son:

- Vista Use-Case: Una vista que muestra la funcionalidad del sistema como la perciben los actores externos.
- Vista Lógica: Muestra cómo se diseña la funcionalidad dentro del sistema, en términos de la estructura estática y la conducta dinámica del sistema.
- Vista de Componentes: Muestra la organización de los componentes de código.
- Vista Concurrente: Muestra la concurrencia en el sistema, dirigiendo los problemas con la comunicación y sincronización que están presentes en un sistema concurrente.
- Vista de Distribución: muestra la distribución del sistema en la arquitectura física con computadoras y dispositivos llamados *nodos*.

5.3 Diagramas de UML

Los diagramas son las gráficas que describen el contenido de una vista. UML tiene nueve tipos de diagramas que son utilizados en combinación para proveer todas las vistas de un sistema: diagramas de caso de uso, de clases, de objetos, de estados, de secuencia, de colaboración, de actividad, de componentes y de distribución. Una posible organización de estos diagramas es la que sigue:

- Diagrama de Casos de Uso
- Diagrama de Clases
- Diagrama de Objetos
- Diagramas de Comportamiento
    - Diagrama de Estados
    - Diagrama de Actividad
- Diagramas de Interacción
    - Diagrama de Secuencia
    - Diagrama de Colaboración
- Diagramas de implementación
    - Diagrama de Componentes

- Diagrama de Despliegue

Asimismo, los diagramas pueden clasificarse en dos grandes rubros:

Modelo estático (estructural)

- Diagrama de despliegue
- Diagrama de componentes
- Diagrama de clases
- Diagrama de objetos

Modelo dinámico (comportamiento)

- Diagrama de estados
- Diagrama de actividades
- Diagrama de secuencia
- Diagrama de colaboración
- Diagrama de casos de uso

Los conceptos utilizados en los diagramas son los elementos de modelo que representan conceptos comunes orientados a objetos, tales como clases, objetos y mensajes, y las relaciones entre estos conceptos incluyendo la asociación, dependencia y generalización. Un elemento de modelo es utilizado en varios diagramas diferentes, pero siempre tiene el mismo significado y simbología.

5.4 Fases del desarrollo de un sistema

Las fases del desarrollo de sistemas que soporta UML son: *Análisis de requerimientos*, *Análisis*, *Diseño*, *Programación* y *Pruebas*.

Análisis de Requerimientos

UML tiene casos de uso (use-cases) para capturar los requerimientos del cliente. A través del modelado de casos de uso, los actores externos que tienen interés en el sistema son modelados con la funcionalidad que ellos requieren del sistema (los casos de uso). Los actores y los casos de uso son modelados con relaciones y tienen asociaciones entre ellos o éstas son divididas en jerarquías. Los actores y casos de uso son descritos en un diagrama use-case. Cada use-case es descrito en texto y especifica los requerimientos del cliente: lo que él (o ella) espera del sistema sin considerar la funcionalidad que se implementará. Un análisis de requerimientos puede ser realizado también para procesos de negocios, no solamente para sistemas de software.

Análisis

La fase de análisis abarca las abstracciones primarias (clases y objetos) y mecanismos que están presentes en el dominio del problema. Las clases que se modelan son identificadas, con sus relaciones y descritas en un diagrama de clases. Las colaboraciones entre las clases para ejecutar los casos de uso también se consideran en esta fase a través de los modelos dinámicos en UML. Es importante notar que sólo se consideran clases que están en el dominio del problema (conceptos del mundo real) y todavía no se consideran clases que definen detalles y soluciones en el sistema de software, tales como clases para interfaces de usuario, bases de datos, comunicaciones, concurrencia, etc.

Diseño

En la fase de diseño, el resultado del análisis es expandido a una solución técnica. Se agregan nuevas clases que proveen de la infraestructura técnica: interfaces de usuario, manejo de bases de datos para almacenar objetos en una base de datos, comunicaciones con otros sistemas, etc. Las clases de dominio del problema del análisis son agregadas en esta fase. El diseño resulta en especificaciones detalladas para la fase de programación.

Programación o Desarrollo

En esta fase las clases del diseño son convertidas a código en un lenguaje de programación orientado a objetos. Cuando se crean los modelos de análisis y diseño en UML, lo más aconsejable es trasladar mentalmente esos modelos a código.

Pruebas

Normalmente, un sistema es tratado en pruebas de unidades, pruebas de integración, pruebas de sistema, pruebas de aceptación, etc. Las pruebas de unidades se realizan a clases individuales o a un grupo de clases y son típicamente ejecutadas por el programador. Las pruebas de integración integran componentes y clases en orden para verificar que se ejecutan como se especificó. Las pruebas de sistema ven al sistema como una "caja negra" y validan que el sistema tenga la funcionalidad final que le usuario final espera. Las pruebas de aceptación conducidas por el cliente verifican que el sistema satisface los requerimientos y son similares a las pruebas de sistema.

5.5 Diagramas de Caso de Uso

Sistema

Un sistema en un diagrama de caso de uso es descrito como una caja; el nombre del sistema aparece arriba o dentro de la caja. Ésta también contiene los símbolos para los casos de uso del sistema.

Actores

Un actor es alguien o algo que interactúa con el sistema; es quien utiliza el sistema. Por la frase "interactúa con el sistema" se debe entender que el actor envía a o recibe del sistema unos mensajes o intercambia información con el sistema. En pocas palabras, el actor lleva a cabo los casos de uso. Un actor puede ser una persona u otro sistema que se comunica con el sistema a modelar.

Un actor es un tipo (o sea, una clase), no es una instancia y representa a un rol. Gráficamente se representa con la figura de "stickman" (**figura 11**).

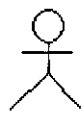


Figura 11. Stickman (El hombre de palitos).

Encontrando actores en un diagrama de casos de uso

Es posible obtener a los actores de un diagrama de casos de uso a través de las siguientes preguntas:

- ¿Quién utilizará la funcionalidad principal del sistema (actores primarios)?
- ¿Quién necesitará soporte del sistema para realizar sus actividades diarias?
- ¿Quién necesitará mantener, administrar y trabajar el sistema (actores secundarios)?
- ¿Qué dispositivos de hardware necesitará manejar el sistema?
- ¿Con qué otros sistemas necesitará interactuar el sistema a desarrollar?
- ¿Quién o qué tiene interés en los resultados (los valores) que el sistema producirá?

Casos de uso

Un caso de uso (**figura 12**) representa la funcionalidad completa tal y como la percibe un actor. Un caso de uso en UML es definido como un conjunto de secuencias de acciones que un sistema ejecuta y que permite un resultado observable de valores para un actor en particular. Gráficamente se representan con una elipse y tiene las siguientes características:

- Un caso de uso siempre es iniciado por un actor.
- Un caso de uso provee valores a un actor.
- Un caso de uso es completo.

Figura 12. Caso de Uso.

Encontrando casos de uso

El proceso para encontrar casos de uso inicia encontrando al actor o actores previamente definidos. Por cada actor identificado, hay que realizar las siguientes preguntas:

- ¿Qué funciones del sistema requiere el actor? ¿Qué necesita hacer el actor?
- ¿El actor necesita leer, crear, destruir, modificar o almacenar algún tipo de información en el sistema?
- ¿El actor debe ser notificado de eventos en el sistema o viceversa? ¿Qué representan esos eventos en términos de funcionalidad?
- ¿El trabajo diario del actor podría ser simplificado o hecho más eficientemente a través de nuevas funciones en el sistema? (Comúnmente, acciones actuales del actor que no estén automatizadas)

Otras preguntas que nos ayudan a encontrar casos de uso pero que no involucran actores son:

- ¿Qué entradas/salidas necesita el sistema? ¿De dónde vienen esas entradas o hacia dónde van las salidas?
- ¿Cuáles son los mayores problemas de la implementación actual del sistema?

Caso de estudio: Ejercicio 10

Realizar el diagrama de Casos de uso para el sistema de Training Systems de apéndice A.

5.6 Diagrama de Clases

Un diagrama de clases sirve para visualizar las relaciones entre las clases que involucran el sistema, las cuales pueden ser asociativas, de herencia, de uso y de contención.

Un diagrama de clases esta compuesto por los siguientes elementos:

- Clase: atributos, métodos y visibilidad.
- Relaciones: Herencia, Composición, Agregación, Asociación y Uso.

Clase

Es la unidad básica que encapsula toda la información de un Objeto (un objeto es una instancia de una clase). A través de ella podemos modelar el entorno en estudio (una Casa, un Auto, una Cuenta Corriente, etc.).

En UML, una clase (**figura 13**) es representada por un rectángulo (figura 6) que posee tres divisiones:

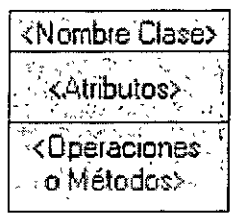


Figura 13. Clase.

En donde:

- **Superior**: Contiene el nombre de la Clase
- **Intermedio**: Contiene los atributos (o variables de instancia) que caracterizan a la Clase (pueden ser private, protected o public).
- **Inferior**: Contiene los métodos u operaciones, los cuales son la forma como interactúa el objeto con su entorno (dependiendo de la visibilidad: private, protected o public).

Para el ejemplo (**figura 14**), una Cuenta Corriente que posee como característica:

- Balance

Puede realizar las operaciones de:

- Depositar
- Girar
- y Balance

El diseño asociado es:

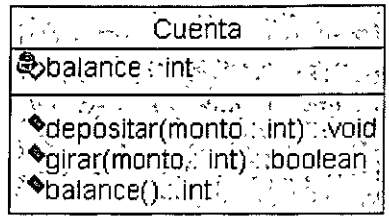


Figura 14. Clase Cuenta.

Atributos

Los atributos o características de una Clase pueden ser de tres tipos, los que definen el grado de comunicación y visibilidad de ellos con el entorno, estos son:

- **public** (+, ): Indica que el atributo será visible tanto dentro como fuera de la clase, es decir, es accesible desde todos lados.

- **private** (-, ): Indica que el atributo sólo será accesible desde dentro de la clase (sólo sus métodos lo pueden acceder).

- **protected** (#, ): Indica que el atributo no será accesible desde fuera de la clase, pero si podrá ser accedido por métodos de la clase además de las subclases que se deriven (ver herencia).

Métodos

Los métodos u operaciones de una clase son la forma en como ésta interactúa con su entorno, éstos pueden tener las características:

- **public** (+, ): Indica que el método será visible tanto dentro como fuera de la clase, es decir, es accesible desde todos lados.

- **private** (-, ): Indica que el método sólo será accesible desde dentro de la clase (sólo otros métodos de la clase lo pueden acceder).

- **protected** (#, ): Indica que el método no será accesible desde fuera de la clase, pero si podrá ser accedido por métodos de la clase además de métodos de las subclases que se deriven (ver herencia).

Relaciones entre Clases

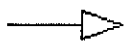
Ahora ya definido el concepto de Clase, es necesario explicar como se pueden interrelacionar dos o más clases (cada uno con características y objetivos diferentes). Antes es necesario explicar el concepto de cardinalidad de relaciones. En UML, la cardinalidad de las relaciones indica el grado y nivel de dependencia, se anotan en cada extremo de la relación y éstas pueden ser:

a. **uno o muchos**: 1..* (1..n)

b. **0 o muchos**: 0..* (0..n)

c. **número fijo**: m (m denota el número).

Herencia (Especialización/Generalización)



Indica que una subclase hereda los métodos y atributos especificados por una Super Clase, por ende la Subclase además de poseer sus propios métodos y atributos, poseerá las características y atributos visibles de la Super Clase (public y protected), ejemplo (**figura 15**):

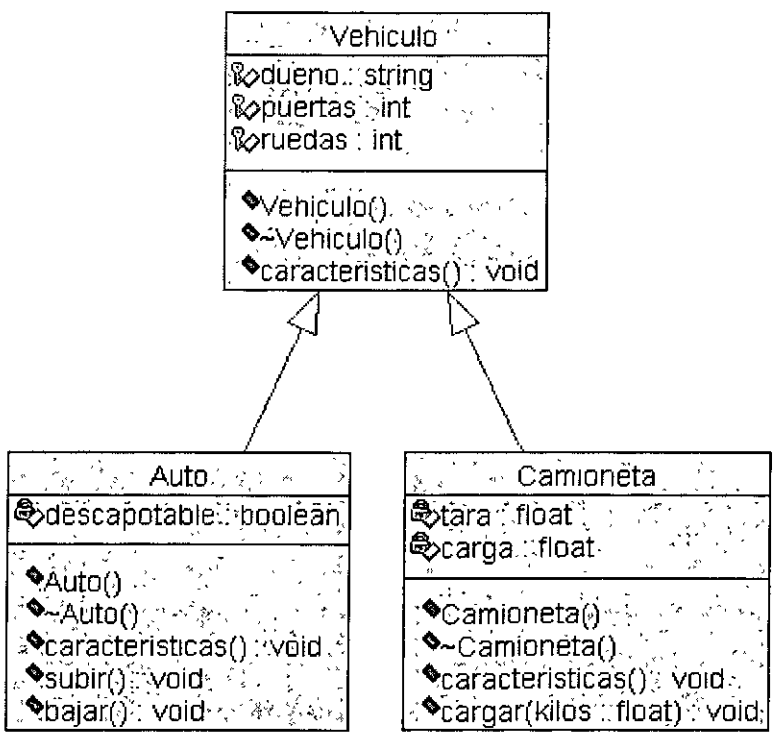


Figura 15. Ejemplo de herencia.

En la figura se especifica que Auto y Camión heredan de Vehículo, es decir, Auto posee las Características de Vehículo (Precio, VelMax, etc) además posee algo particular que es Descapotable, en cambio Camión también hereda las características de Vehiculo (Precio, VelMax, etc) pero posee como particularidad propia Acoplado, Tara y Carga.

Cabe destacar que fuera de este entorno, lo único "visible" es el método Características aplicable a instancias de Vehículo, Auto y Camión, pues tiene definición publica, en cambio atributos como Descapotable no son visibles por ser privados.

Agregación

Para modelar objetos complejos, no bastan los tipos de datos básicos que proveen los lenguajes: enteros, reales y secuencias de caracteres. Cuando se requiere componer objetos que son instancias de clases definidas por el desarrollador de la aplicación, tenemos dos posibilidades:

- **Por Valor**: Es un tipo de relación estática, en donde el tiempo de vida del objeto incluido esta condicionado por el tiempo de vida del que lo incluye. Este tipo de relación es comúnmente llamada **Composición** (el Objeto base se construye a partir del objeto incluido, es decir, es "parte/todo").

- **Por Referencia**: Es un tipo de relación dinámica, en donde el tiempo de vida del objeto incluido es independiente del que lo incluye. Este tipo de relación es comúnmente llamada **Agregación** (el objeto base utiliza al incluido para su funcionamiento).

  Un ejemplo (**figura 16**) es el siguiente:

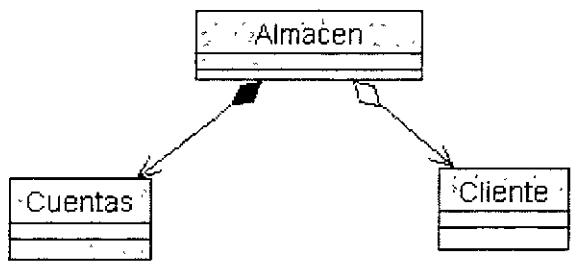


Figura 16. Relaciones por valor y por referencia.

En donde se destaca que:

- Un Almacén posee Clientes y Cuentas (los rombos van en el objeto que posee las referencias).

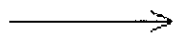
- Cuando se destruye el Objeto Almacén también son destruidos los objetos Cuenta asociados, en cambio no son afectados los objetos Cliente asociados.

- La composición (por Valor) se destaca por un rombo relleno.

- La agregación (por Referencia) se destaca por un rombo transparente.

La flecha en este tipo de relación indica la navegabilidad del objeto referenciado. Cuando no existe este tipo de particularidad la flecha se elimina.

Asociación



La relación entre clases conocida como Asociación, permite asociar objetos que colaboran entre si. Cabe destacar que no es una relación fuerte, es decir, el tiempo de vida de un objeto no depende del otro. Ejemplo (**figura 17**):

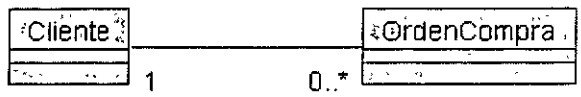
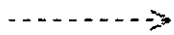


Figura 17. Asociación.

Un cliente puede tener asociadas muchas Ordenes de Compra, en cambio una orden de compra solo puede tener asociado un cliente.

Dependencia o Instanciación (uso)



Representa un tipo de relación muy particular, en la que una clase es instanciada (su instanciación es dependiente de otro objeto/clase). Se denota por una flecha punteada. El uso más particular de este tipo de relación es para denotar la dependencia que tiene una clase de otra, como por ejemplo (**figura 18**) una aplicación grafica que instancia una ventana (la creación del Objeto Ventana esta condicionado a la instanciación proveniente desde el objeto Aplicacion):

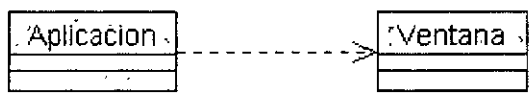


Figura 18. Dependencia.

Cabe destacar que el objeto creado (en este caso la Ventana gráfica) no se almacena dentro del objeto que lo crea (en este caso la Aplicación).

Clase Abstracta

Una clase abstracta (**figura 19**) se denota con el nombre de la clase y de los métodos con letra "itálica". Esto indica que la clase definida no puede ser instanciada pues posee métodos abstractos (aún no han sido definidos, es decir, sin implementación). La única forma de utilizarla es definiendo subclases, que implementan los métodos abstractos definidos.

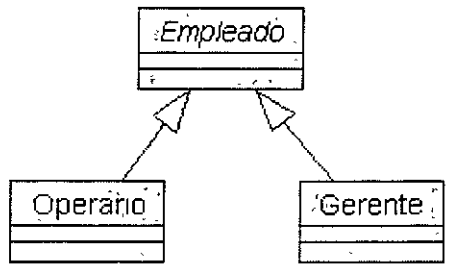


Figura 19. Clase abstracta.

Clase parametrizada

Una clase parametrizada (**figura 20**) se denota con un subcuadro en el extremo superior de la clase, en donde se especifican los parámetros que deben ser pasados a la clase para que esta pueda ser instanciada. El ejemplo más típico es el caso de un Diccionario en donde una llave o palabra tiene asociado un significado, pero en este caso las llaves y elementos pueden ser genéricos. La calidad de genérico puede venir dada de una plantilla (como en el caso de C++) o bien de alguna estructura predefinida (especialización a través de clases). En el ejemplo no se especificaron los atributos del Diccionario, pues ellos dependerán exclusivamente de la implementación que se le quiera dar.

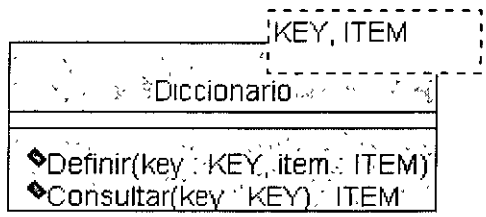


Figura 20. Clase parametrizada.

Caso de estudio: Ejercicio 11

Realizar el diagrama de clases para el sistema de Training Systems del apéndice A.

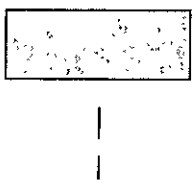
5.4 Diagramas de Secuencia

El diagrama de secuencia representa la forma en como un Cliente (Actor) u Objetos (Clases) se comunican entre si en petición a un evento. Esto implica recorrer toda la secuencia de llamadas, de donde se obtienen las responsabilidades claramente. Dicho diagrama puede ser obtenido de dos partes, desde el Diagrama Estático de Clases o el de Casos de Uso (son diferentes).

Los componentes de un diagrama de interacción son:

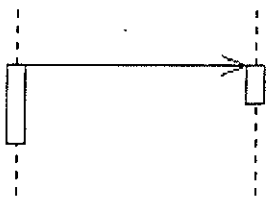
- Un Objeto o Actor.
- Mensaje de un objeto a otro objeto.
- Mensaje de un objeto a si mismo.

Objeto/Actor



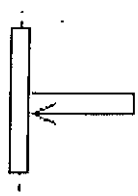
El rectángulo representa una instancia de un Objeto en particular, y la línea punteada representa las llamadas a métodos del objeto.

Mensaje a Otro Objeto



Se representa por una flecha entre un objeto y otro, representa la llamada de un método (operación) de un objeto en particular.

Mensaje al Mismo Objeto



No solo llamadas a métodos de objetos externos pueden realizarse, también es posible visualizar llamadas a métodos desde el mismo objeto en estudio.

En el presente ejemplo, tenemos el diagrama de interacción proveniente del siguiente modelo estático (**figura 21**):

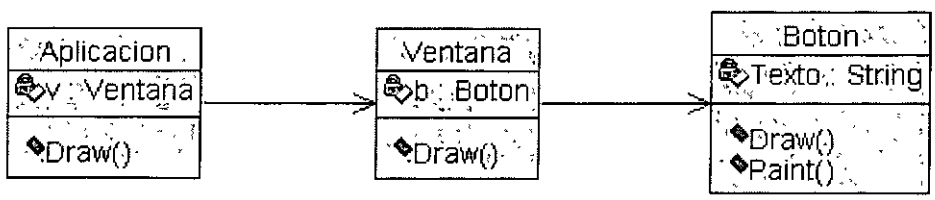


Figura 21. Diagrama estático.

Aquí se representa una aplicación que posee una Ventana gráfica, y ésta a su vez posee internamente un botón.

Entonces el diagrama de secuencia (**figura 22**) para dicho modelo es:

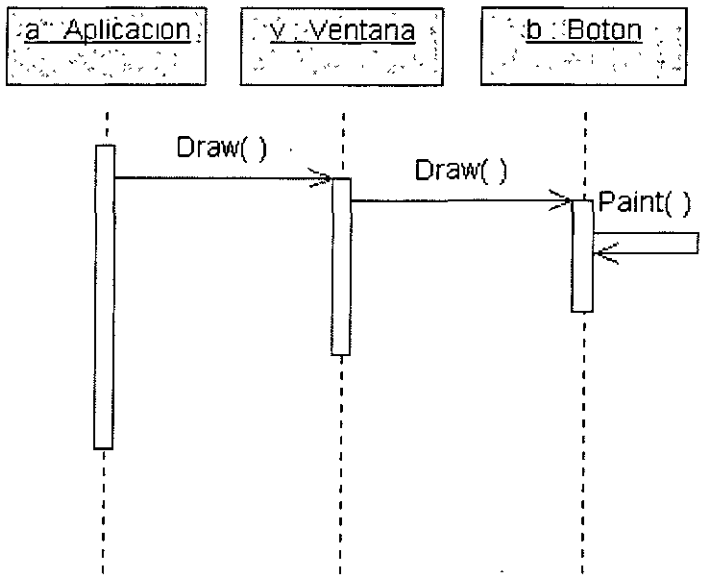


Figura 22. Diagrama de secuencia.

En donde se hacen notar las sucesivas llamadas a Draw() (entre objetos) y la llamada a Paint() por el objeto Botón.

En este diagrama se observa que sólo se consideran algunos objetos y es importante aclarar que estos no serán todos los objetos a considerar dentro del sistema, ya que todavía es posible agregar nuevos objetos que no se habían considerado en el dominio del análisis así como los objetos técnicos, como se mencionó anteriormente. Los objetos considerados se representan en rectángulos con el nombre subrayado y cada uno cuenta con su línea de vida vertical que muestra la vida del objeto.

Nótese que hasta el momento no se ha hecho énfasis en el orden que se pretende para estos diagramas. No obstante, siempre es necesario tomar en cuenta procesos completos para cada diagrama. Para completar una secuencia es perentorio regresar al punto de partida. Es decir, cada vez habrá que asegurar que el camino de retorno para toda secuencia sea ininterrumpido hasta el objeto que la haya iniciado.

Casos de Estudio: Ejercicio 12

Realizar los diagramas de secuencia del sistema de Training System del apéndice A.

5.5 Diagramas de Actividad

Un diagrama de actividades (**figura 23**) puede considerarse como un caso especial de un diagrama de estados en el cual casi todos los estados son estados acción (identifican una acción que se ejecuta al estar en él) y casi todas las transiciones evolucionan al término de dicha acción (ejecutada en el estado anterior). Un diagrama de actividades puede dar detalle a un caso de uso, un objeto o un mensaje en un objeto. Permiten representar transiciones internas al margen de las transiciones o eventos externos. En la figura 5.7 se presenta un ejemplo de diagrama de actividades para un mensaje de un objeto.

La interpretación de un diagrama de actividades depende de la perspectiva considerada: en un diagrama conceptual, la actividad es alguna tarea que debe ser realizada; en un diagrama de especificación o de implementación, la actividad es un método de una clase. Generalmente se suelen utilizar para modelar los pasos de un algoritmo.

Un estado de acción representa un estado con acción interna, con por lo menos una transición que identifica la culminación de la acción (por medio de un evento implícito). No deben tener transiciones internas ni transiciones basadas en eventos, ya que si fuera este el caso, se representaría con un diagrama de estados. Los estados de acción se representan por un rectángulo con bordes redondeados, y permiten modelar un paso dentro de un algoritmo. Las flechas dirigidas entre estados de acción representan transiciones con evento implícito que, en el caso de decisiones, pueden tener una condición o guarda asociada (que al igual que en los diagramas de estado evalúa a verdadero o a falso).

Las decisiones se representan mediante una transición múltiple que sale de un estado y donde cada camino tiene una etiqueta distinta. Se representa mediante un rombo al cual llega la transición del estado origen y del cual salen las múltiples transiciones de los estados destino. En un diagrama de actividades también pueden existir barras de sincronización, a las que se encuentran asociadas varios caminos salientes. Cada camino saliente se dirige a una actividad, realizándose dichas actividades en paralelo. Esto quiere decir que el orden en que se realicen dichas actividades es irrelevante, siendo válido cualquier orden entre ellas.

Otro elemento característico de los diagramas de actividades es el carril (swim lane). Este permite separar los distintos sectores, rubros o procesos por los que atravesará una parte del sistema durante su accionar. Pueden existir tantos carriles como sea necesario, pero la conformación de un diagrama lógico y comprensible se complica a medida que se añaden los mismos.

Dado que el diagrama de actividades permite expresar el orden en que se realizan las cosas, resulta adecuado para el modelado de organizaciones y el de programas concurrentes (permiten representa gráficamente los hilos de ejecución). Como la mayoría de las técnicas de modelado de comportamiento, los diagramas de actividades tienen sus puntos fuertes y sus puntos débiles, de forma que es necesario utilizarlos en combinación con otras técnicas. Su principal aportación al modelado del comportamiento es que soportan el comportamiento paralelo, lo que resulta adecuado para el modelado de flujo de trabajo y programación multihilos. Por contra, su principal desventaja es que no muestran de una forma clara los enlaces existentes entre las acciones y los objetos, siendo mucho más apropiado para ello los diagramas de interacción.

En general resulta adecuado utilizar diagramas de actividades para:

- Análisis de casos de uso: Durante el análisis de los casos de uso no estamos interesados en asociar acciones a objetos, sino en entender qué acciones se necesitan llevar a cabo y cuales son las dependencias en el comportamiento.

- Comprensión del flujo de trabajo a lo largo de diferentes casos de uso.

- Modelado de aplicaciones multihilos.

Por contra, resultan en general del todo inadecuados a la hora de mostrar la colaboración entre objetos y la evolución del comportamiento de los objetos durante su tiempo de vida.

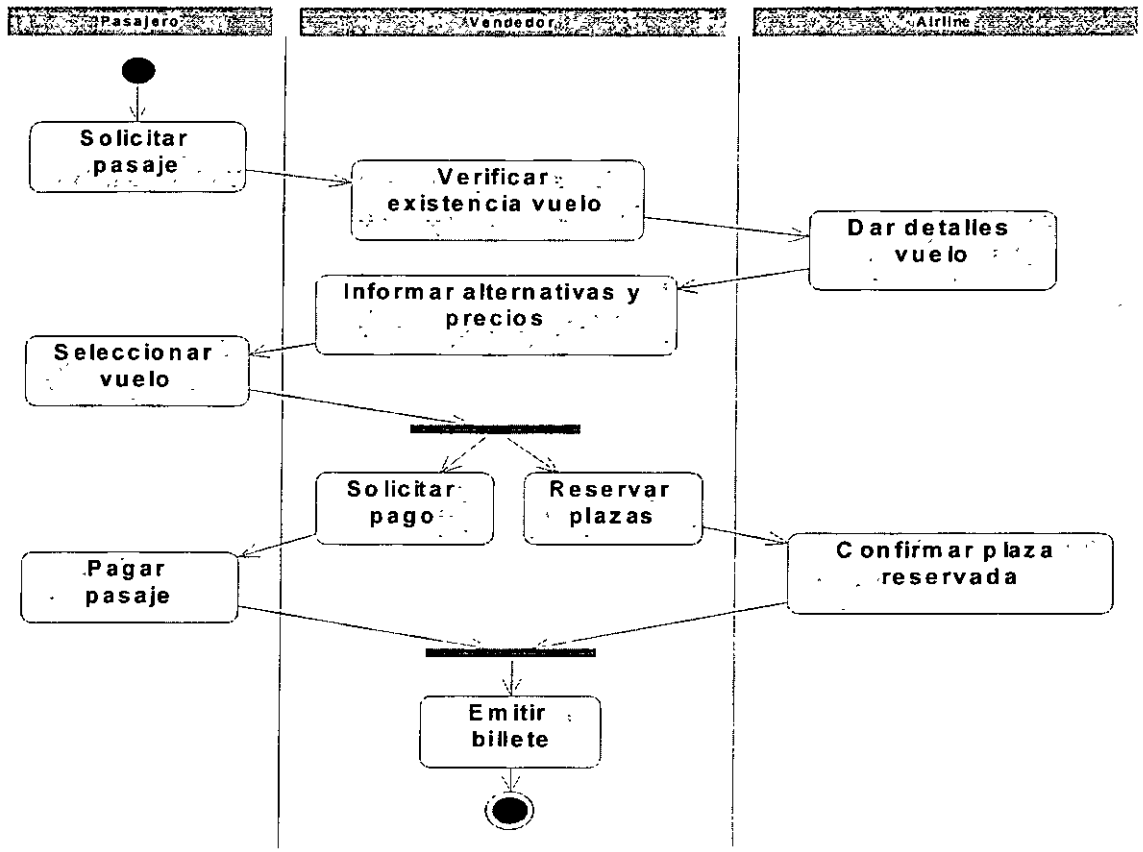


Figura 23. Diagrama de actividades con carriles.

Caso de estudio: Ejercicio 13

Realizar los diagramas de actividades del sistema de Training Systems del apéndice B.

Apéndice A

Caso de Estudio: DirectClothing Inc

DirectClothing Inc. Vende ropa a través de catálogos. El negocio está creciendo un 30% cada año y ellos necesitan un nuevo sistema de pedidos. Tú has sido contratado por DirectClothing para diseñar el nuevo sistema.

Un catálogo de ropa es creado cada mes y mandado a los subscriptores. El catálogo tiene una sección de productos en oferta, los especiales del mes y precios normales. Cada pieza de ropa puede cambiar de precio dependiendo el mes, por tal motivo cuando un cliente ordena por catálogo, se debe saber cuál es el catálogo del cual están ordenando y asignar el precio correctamente. Si el catálogo tiene una vigencia de seis meses, después de eso los productos se venderán en el precio del catálogo actual.

Para hacer una orden, los clientes pueden llamar a un representante de ventas, enviar una orden por correo o por fax.

DirectClothing se está expandiendo, y desea que los clientes puedan realizar su pedido a través de Internet. Los artículos que se presenten en la página tendrán el precio del catálogo actual.

Antes de que la orden sea registrada en el sistema, el departamento de inventario debe de checar cada artículo para saber si hay en existencia y así, se asigne a la orden. En dado caso de que no se tenga el artículo en existencia, éste quedará como pedido hasta que se tenga el artículo en existencia. Una vez que se tengan los artículos en existencia, se debe verificar el pago, después de hacer esto, la orden se manda al departamento almacén para que se empaquete el pedido.

DirectClothing acepta cheques y todas las tarjetas de crédito.

Apéndice B

Caso de Estudio: Training System Inc.

Traininig Systems se encarga de organizar cursos y necesita un sistema que se encargue de los horarios de los cursos, salones de clases, instructores y el registro de estudiantes.

Training Systems tiene 20 instructores, 15 contratados y 5 permanentes: Cada instructor tiene un conjuntos de cursos asignados. Los instructores no pueden enseñar dos semanas consecutivas para que no se traslapen los horarios. Los instructores pueden tomar cursos durante su semana libre mientras que el curso no esté lleno.

Training Systems cuenta con cinco salones de clases con varis configuraciones, tres con Unix y dos con Windows. De los tres salones con Unix, uno es para 15 estudiantes y los otros dos para 12. De los salones con Windows uno es para 15 estudiantes y el otro para 12.

Training Systems enseña Solaris y Java. Sólo se puede dar uno de esos cursos en la misma semana. El curso de Java sólo puede tener 10 estudiantes y los de Solaris sólo 8. Una clase puede ser cancelada si hay menos del 40% de alumnos registrados una semana antes de que se de el curso.

Training Systems realiza con un mes de anterioridad sus catálogos, antes de mostrarlos al público.

Training Systems tiene un programa de clientes frecuentes, en el cual ofrece un 10% de descuento por cada tres cursos tomados. Tiene un programa para empresas, donde ofrece un 10% de descuento si envían a 3 alumnos al mismo curso. También ofrecen 20% de descuento a las empresas y precompran 10 cursos en el mismo año.

Bibliografía

http://www.fi-b.unam.mx/pp/profesores/carlos/aydoo/intro.html

http://www.monografias.com/trabajos5/inso/inso.shtml

http://www.dcc.uchile.cl/~psalinas/uml/casosuso.html

http://www.dcc.uchile.cl/~psalinas/uml/modelo.html

http://www.dcc.uchile.cl/~cc61j/dinamica/sld005.htm

http://www.omg.org/uml/

http://www.celigent.com/uml/

http://www.cetus-links.org/oo_uml.html

http://www.enteract.com/~bradapp/docs/patterns-intro.html

http://www.monografias.com/trabajos22/desarrollo-software/desarrollo-software.shtml

G. Booch, J. Rumbaugh, I. Jacobson. "The Unified Modeling Language Reference Manual". Addison Wesley.

G. Booch, J. Rumbaugh, I. Jacobson. "The Unified Modeling Language User Guide". Addison Wesley

Craig Larman. UML y Patrones, Introducción al análisis y diseño orientado a objetos. Prentice Hall. Primera versión en Español, 1999.

Pierre-Alain Muller, "Instant UML"

Martin Fowler, "UML Destilled" ("UML Gota a Gota")

Terry Quatrani, "Visual Modeling ...", un caso de estudio

Manual de Sun Microsystems: Migrating to OO Programing with Java Technology