Universidad Católica
Nstra. Sra. de la Asunción
Facultad de Ciencias y Tecnología
Ingeniería Informática

Trabajo Práctico

Tema:
UML

Materia:
Teoría y Aplicación de la Informática 2
10° semestre

Profesor:
Juan De Urraza

Integrantes:
Lucía Coronel Antonelli
Ma. Carolina Willigs

Octubre 2003

1- INTRODUCCIÓN

La Programación Orientada a objeto trajo consigo la aparición de una gran diversidad de métodos y técnicas las cuales por un lado comparten aspectos en común y por otro utilizan distintas notaciones. Esta situación dio origen a una serie de inconvenientes especialmente para el aprendizaje, la aplicación, la construcción y el uso de herramientas de este tipo en la construcción del un nuevo SW de calidad. Pero lo que resultaba peor aun era que en ninguno de los estándares el código podría ser reutilizado.

UML busca combinar y "unificar" las notaciones provenientes de los distintos mecanismos para:

- Modelado Orientado a Objetos
- Modelado de Datos
- Modelado de Componentes
- Modelado de Flujos de Trabajo (Workflows)

2- DEFINICIONES DE UML

UML (Unified Modelling Language o Lenguaje Unificadode Modelado) es un lenguaje gráfico que permite visualizar, especificar, construir y documentarlos artefactos (piezas de información utilizadas o producidas por un proceso de desarrollo) de un sistema software.

UML (*Unified Modeling Language*) es un lenguaje para especificar, construir, visualizar y documentar los sistemas software. Con UML se tiene un lenguaje visual para el modelado, pero no un lenguaje visual de programación, es decir, a partir de él no se deriva el código en algún tipo de lenguaje de programación. Algunos elementos del software (bucles, saltos) quedan mucho mejor expresados con el propio código fuente, pero la arquitectura y estructura del sistema se puede expresar y comprender perfectamente a partir del lenguaje UML. Con esta metodología se cubrirán las etapas de desarrollo software de análisis, diseño, programación y testado.

UML es un lenguaje para especificar, visualizar, construir y documentar los ``artefactos" *software* (desde las fases iniciales hasta la implementación del sistema), así como el modelado de flujo de trabajo y otros sistemas no software.

La definición del UML se basa en los siguientes documentos:

- UML Semantics: Define la semántica y sintaxis del UML. Para ello utiliza tres visiones consistentes:
- *Abstract syntax*: Para presentar el meta-modelo UML, sus conceptos (meta-clases), relaciones, y restricciones se utilizan diagramas de clases.

- *Well-formedness rules*: Definen las reglas y restricciones que rigen en los modelos válidos. Las reglas se expresan en lenguaje natural y en OCL (Object Constraint Language). OCL es un lenguaje de especificación que utiliza lógica para especificar propiedades invariantes de sistemas que se componen de conjuntos y relaciones entre conjuntos.
- *Semantics*: Las semánticas de manejo del modelo se describen en lenguaje natural.
  Estas tres visiones constituyen la definición del UML.
  Un *meta-modelo* es un lenguaje para la especificación de un modelo, el modelo propósito. Es decir, es un modelo para elementos de modelado. El propósito del meta-modelo UML es el proporcionar una declaración común y única de la sintaxis y semánticas de los elementos del UML. UML Notation Guide: Define la notación y proporciona ejemplos. La notación será la sintaxis gráfica para expresar la semántica descrita por el meta-modelo UML. La notación gráfica y la sintaxis textual son las partes mas visibles, siendo utilizadas por personas y herramientas en el modelado de sistemas. También contiene las semánticas UML, sin embargo, sus definiciones se encuentran en *UML Semantics*.
- UML Extension for the Objectory Process for Software Engineering y UML Extension for Business Modeling: Extensiones específicas de UML, *Objectory Process* y *Business Engineering*. El UML es ampliamente aplicable sin extensiones, de forma que las compañías y proyectos sólo deberían definir extensiones cuando es necesario el introducir nueva notación y terminología. Para reducir la confusión se definen:
- *UML Variant*: Un lenguaje con una semántica bien definida que se construye encima del meta-modelo UML. Este lenguaje puede ser una especialización del meta-modelo UML, sin cambios en las semánticas del UML o con redefinición de algunos de sus elementos.
- *UML Extension*: Un conjunto predefinido de estereotipos, valores etiquetados, restricciones e iconos de notación que colectivamente extienden y adaptan el UML a un dominio o un proceso específico, por ejemplo, *Objectory Process Extension*.
- Object Constraint Language Specification (OCL): El UML incorpora el lenguaje de restricción de objetos a fin de superar las deficiencias que poseen los elementos UML para definirse a sí mismos. Podría haberse utilizado el lenguaje natural, pero su obvia ambiguedad provocó la necesidad de la incorporación del OCL. Se puede decir que se trata de un lenguaje de expresión, porque se utiliza únicamente para expresar un valor, sin provocar cambios en el modelo; de un lenguaje de modelado, porque no es ejecutable, no es un lenguaje de programación; y de un lenguaje formal, porque todos sus elementos han sido definidos formalmente.
  Según los autores, este lenguaje está destinado a la especificación de invariantes en clases y tipos de clases en el modelo de clases; para describir pre y post-condiciones en operaciones y métodos; para describir guardas; para ser empleado como un lenguaje de navegación; para definir reglas de formación; y para definir restricciones de operaciones.

3- HISTORIA

Hasta principios de los noventa se habían desarrollado diferentes métodos, con diferentes notaciones, para el modelado orientado a objetos. Entre los métodos más importantes se encontraban el de Booch (centrado en las etapas de diseño yconstrucción), el método OOSE de Jacobson (que proporcio-naba un buen soporte para los casos de uso) y OMT deRumbaugh (más enfocada al análisis de sistemas de infor-mación con grandes volúmenes de datos). Debido a que cada uno de estos métodos era más adecuado para una determinada fase de desarrollo, estos tres autores decidieron colaborarpara definir un único lenguaje que permitiera modelarsistemas de manera completa, desde el concepto hasta losartefactos ejecutables.
El desarrollo real de UML comenzó en octubre de 1994 cuando Grady Booch y Jim Rumbaugh de Rational Software comenzaron a trabajar en la unificación de los lenguajes de modelado Booch y OMT, es entonces cuando fueron reconocidos mundialmente a la cabeza del desarrollo de metodologías orientadas a objetos. Fue entonces cuando terminaron su trabajo de unificación obteniendo el borrador de la versión 0.8 del denominado *Unified Method* en octubre de 1995. Tras esto también en 1995, Ivar Jacobson padre de la metodología OOSE se unió con Rational Software para obtener finalmente UML 0.9 y 0.91 en junio y octubre de 1996.

4- CUADRO EVOLUTIVO

OCTUBRE 1994	Comienzan a trabajar Grady Booch y Jim Rumbaugh
OCTUBRE 1995	Versión 0.8 del UNIFIED METED
JUNIO 1996	Versión 0.9 Se une a ellos Ivar Jacobson
OCTUBRE 1996	Versión 0.91
1997	Es aprobado por el OMG
1998	aparece el estándar UML 1.2 (revisiones menores)
1999	aparece el estándar UML 1.3
2000	aparece el estándar UML 1.4 (revisiones menores)
2001	aparece el estándar UML 2.0

5- EMPRESAS QUE TRABAJARON EN EL DESARROLLO

A partir de la versión 1.0 de UML, su desarrollo se ve impulsado con el trabajo conjunto de un gran número de empresas. Este hecho hace de UML un estándar de la Industria
Algunas de esas empresas son:

- Rational Software
- Digital Equipment
- Hewlett-Packard
- i-Logix
- IBM
- ICON Computing
- Intellicorp and James Martin & Co.
- MCI Systemhouse
- Microsoft
- ObjetcTime
- Oracle Corp
- Platinium Technology
- Sterling Software
- Taskon
- Texas Instrument
- Unisys

6- UNIFICACIÓN DE STÁNDARES DE DESARROLLO ORIENTADO A OBJETO

Cuando hablamos de UML como el lenguaje que unificó una serie de estándares de desarrollo Orientado a Objetos, inevitablemente debemos mencionar algunos de ellos, entre los cuales destacan:

- Booch
- Rumbaugh
- Jacobson
- Odell
- Meyer (Pre and Post Conditions)
- Harel (State Charts)
- Wirfs-Brock (Responsabilities)
- Fusion (Operation descriptions, message numbering)
- Embly (Singleton classes)
- Gamma et. al. (Frameworks, patterns, notes)
- Shlaer-Mellor (Object life cycles)

7- ELEMENTOS GRAFICOS QUE COMPONEN LA METODOLOGÍA UML

Los elementos gráficos de la metodología, pretenden aportar un lenguaje de descripción común, y fácil de entender, entre todas las personas involucradas en un proyecto. Este es uno de los objetivos del uso de UML: *conseguir intercambiar información sin invertir gran esfuerzo en estas tareas, puesto que se comparte una determinada notación.*

UML dispone de diferentes construcciones para representar gráficamente el sistema. Estas construcciones se denominan *vistas*, las cuales, intentan resaltar determinadas características que componen el software. Durante las etapas del ciclo de vida se utilizarán aquellas que se consideren necesarias según la naturaleza del software que se pretenda modelar, es por ello que en el presente capítulo se utilizarán las siguientes:

7.1- Diagrama de casos de Uso (Use Case)
7.1.1 Introducción

Para realizar la descripción del sistema desde un punto de vista de alto nivel, UML proporciona los denominados diagramas de casos de uso. Éstos describen la funcionalidad de un sistema siguiendo una metodología descendente, es decir, partiendo de una visión de alto nivel del sistema y continuando con una descomposición del mismo en partes más concretas. El propósito de un caso de uso es definir una pieza de comportamiento coherente, sin revelar la estructura interna del sistema.
El diagrama de casos de uso representa la forma en como un Cliente (Actor) opera con el sistema en desarrollo, además de la forma, tipo y orden en como los elementos interactuan (operaciones o casos de uso).

7.1.2 Un diagrama de casos de uso consta de los siguientes elementos:

7.1.2.1 Actor:

Una definición previa, es que un **Actor** es un rol que un usuario juega con respecto al sistema. Es importante destacar el uso de la palabra rol, pues con esto se especifica que un Actor no necesariamente representa a una persona en particular, sino más bien la labor que realiza frente al sistema.
Como ejemplo a la definición anterior, tenemos el caso de un sistema de ventas en que el rol de Vendedor con respecto al sistema puede ser realizado por un Vendedor o bien por el Jefe de Local.

7.1.2.2 Caso de Uso:

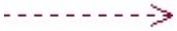
Es una operación/tarea específica que se realiza tras una orden de algún agente externo, sea desde una petición de un actor o bien desde la invocación desde otro caso de uso.

7.1.2.3 Relaciones

7.1.2.3.1 Asociación

Es el tipo de relación más básica que indica la invocación desde un actor o caso de uso a otra operación (caso de uso). Dicha relación se denota con una flecha simple.

7.1.2.3.2 Dependencia o Instanciación



Es una forma muy particular de relación entre clases, en la cual una clase depende de otra, es decir, se instancia (se crea). Dicha relación se denota con una flecha punteada.

7.1.2.3.3 Generalización

Este tipo de relación es uno de los más utilizados, cumple una doble función dependiendo de su estereotipo, que puede ser de **Uso** (<<uses>>) o de **Herencia** (<<extends>>).
Este tipo de relación esta orientado exclusivamente para casos de uso (y no para actores).

extends: Se recomienda utilizar cuando un caso de uso es similar a otro (características).

uses: Se recomienda utilizar cuando se tiene un conjunto de características que son similares en más de un caso de uso y no se desea mantener copiada la descripción de la característica.

De lo anterior cabe mencionar que tiene el mismo paradigma en diseño y modelamiento de clases, en donde esta la duda clásica de **usar** o **heredar**.

7.1.3 Ejemplo**:**

Como ejemplo esta el caso de una Máquina Recicladora:
Sistema que controla una máquina de reciclamiento de botellas, tarros y jabas. El sistema debe controlar y/o aceptar:

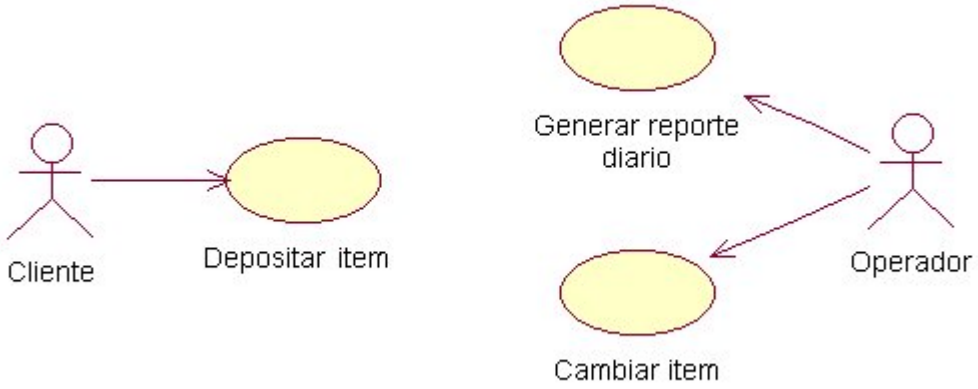
- Registrar el número de ítemes ingresados.
- Imprimir un recibo cuando el usuario lo solicita:
    a. Describe lo depositado
    b. El valor de cada item
    c. Total
- El usuario/cliente presiona el botón de comienzo
- Existe un operador que desea saber lo siguiente:
    a. Cuantos ítemes han sido retornados en el día.

b. Al final de cada día el operador solicita un resumen de todo lo depositado en el día.

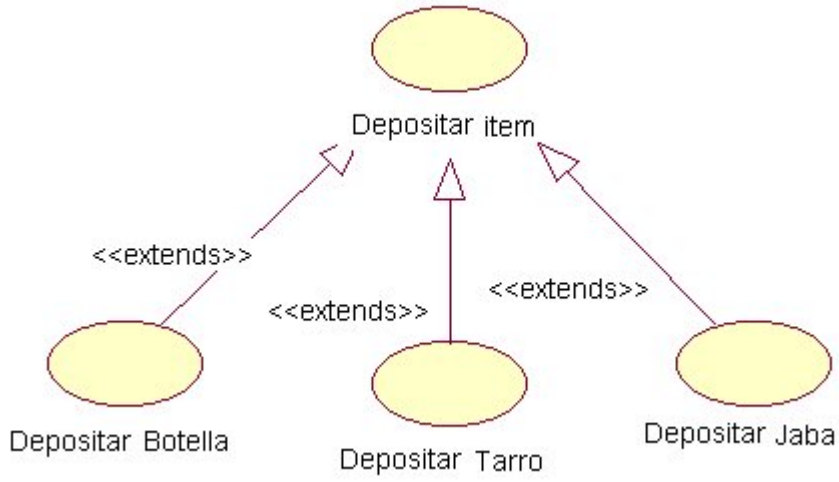
- El operador debe además poder cambiar:
  a. Información asociada a ítemes.
  b. Dar una alarma en el caso de que:
     i. Item se atora.
     ii. No hay más papel.

Como una primera aproximación identificamos a los actores que interactuan con el sistema:

Luego, tenemos que un Cliente puede Depositar Itemes y un Operador puede cambiar la información de un Item o bien puede Imprimir un informe:

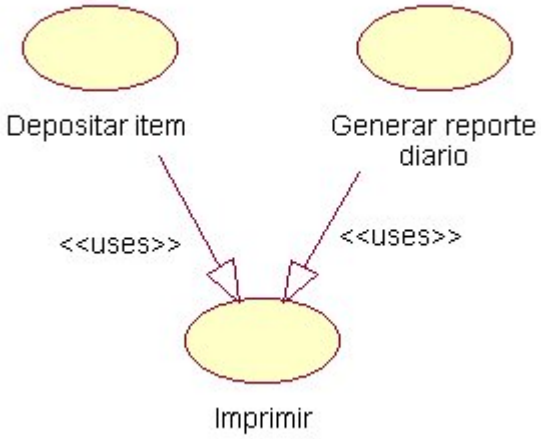


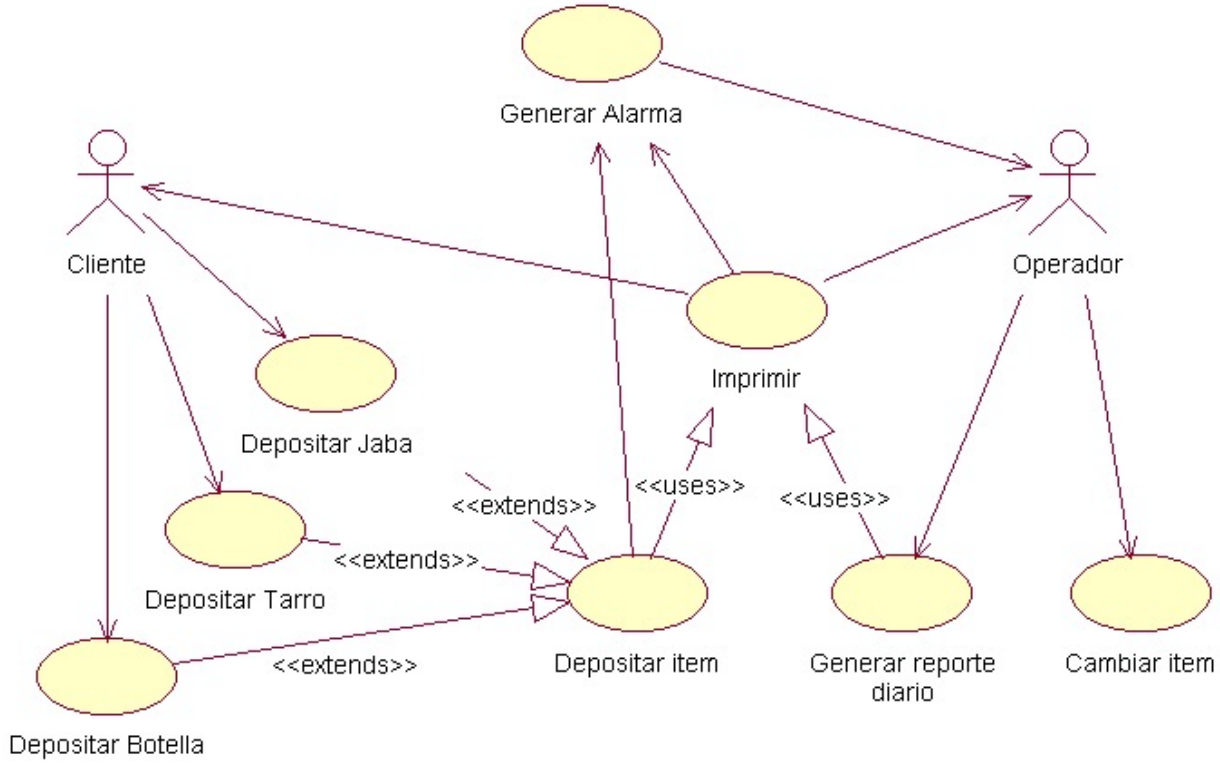
Además podemos notar que un item puede ser una Botella, un Tarro o una Jaba.



Otro aspecto es la impresión de comprobantes, que puede ser realizada después de depositar algún item por un cliente o bien puede ser realizada a petición de un operador.

Entonces, el diseño completo del diagrama Use Case es:





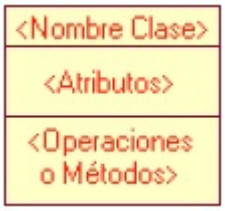
7.2- Modelo de Clases

7.2.1 Introducción
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 contenimiento.
Este diagrama estará formado por objetos, representados con cajas cuadradas, y enlazados entre sí. Estas relaciones junto con los objetos tienen como objetivo conseguir, en el mayor grado posible, una abstracción de la realidad que pretendemos representar, que abarque en su totalidad los elementos y características del sistema que pretendemos representar.

Un diagrama de clases esta compuesto por los siguientes elementos:
7.2.2 Elementos

7.2.2.1Clase
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 es representada por un rectángulo que posee tres divisiones:

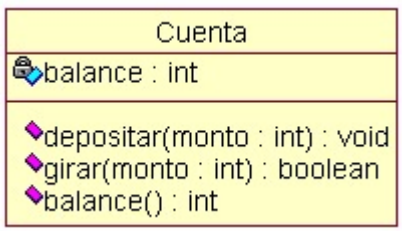


En donde:
- **Superior**: Contiene el nombre de la Clase
- **Intermedio**: Contiene los atributos (o variables de instancia) que caracterizan a la Clase (pueden ser private, protected o public).
- **Inferior**: Contiene los métodos u operaciones, los cuales son la forma como interactúa el objeto con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

7.2.2.1.1 Ejemplo**:**
Una Cuenta Corriente que posee como característica:
- Balance

Puede realizar las operaciones de:
- Depositar
- Girar
- y Balance

El diseño asociado es:



7.2.2.2 Atributos y Métodos:

7.2.2.2.1 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 accsesible desde todos lados.
- **private** :Indica que el atributo sólo será accesible desde dentro de la clase (sólo sus métodos lo pueden accesar).
- **protected**:Indica que el atributo no será accesible desde fuera de la clase, pero si podrá ser accesado por métodos de la clase además de las subclases que se deriven (ver herencia).

7.2.2.2.2 Métodos**:**

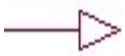
Los métodos u operaciones de una clase son la forma en como ésta interactúa con su entorno, éstos pueden tener las características:

- **public** (+,): Indica que el método será visible tanto dentro como fuera de la clase, es decir, es accsesible desde todos lados.
- **private** (-,): Indica que el método sólo será accesible desde dentro de la clase (sólo otros métodos de la clase lo pueden accesar).
- **protected** (#,): Indica que el método no será accesible desde fuera de la clase, pero si podrá ser accesado por métodos de la clase además de métodos de las subclases que se deriven (ver herencia).

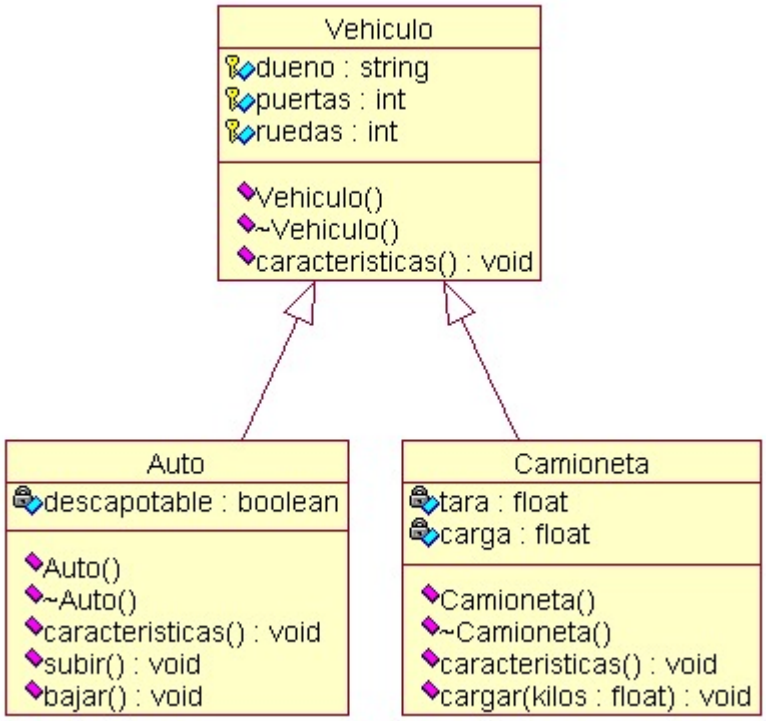
7.2.3 Relaciones entre Clases**:**

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:
uno o muchos: 1..* (1..n)
0 o muchos: 0..* (0..n)
número fijo: m (m denota el número).

7.2.3.1Herencia (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:



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 Caracteristicas aplicable a instancias de Vehículo, Auto y Camión, pues tiene definición publica, en cambio atributos como Descapotable no son visibles por ser privados.

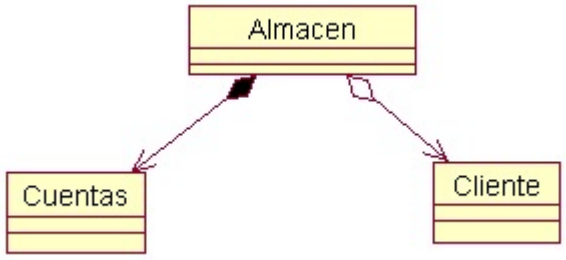
7.2.3.2 Agregación:

Para modelar objetos complejos, n 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 comunmente llamada **Composición** (el Objeto base se contruye 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 comunmente llamada **Agregación** (el objeto base utiliza al incluido para su funcionamiento).

Un Ejemplo es el siguiente:



En donde se destaca que:

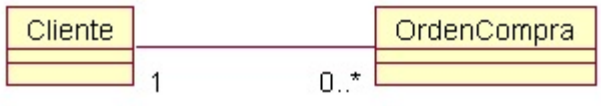
- Un Almacen posee Clientes y Cuentas (los rombos van en el objeto que posee las referencias).
- Cuando se destruye el Objeto Almacen también son destruidos los objetos Cuenta asociados, en cambio no son afectados los objetos Cliente asociados.
- La composición (por Valor) se destaca por un rombo relleno.
- La agregación (por Referencia) se destaca por un rombo transparente.

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

7.2.3.3 Asociación:

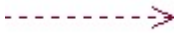
La relación entre clases conocida como Asociación, permite asociar objetos que colaboran entre si. Cabe destacar que no es una relación fuerte, es decir, el tiempo de vida de un objeto no depende del otro.

Ejemplo:



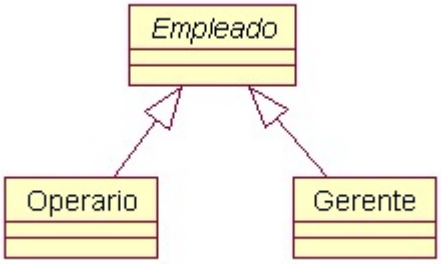
Un cliente puede tener asociadas muchas Ordenes de Compra, en cambio una orden de compra solo puede tener asociado un cliente.

7.2.3.4 Dependencia o Instanciación (uso):

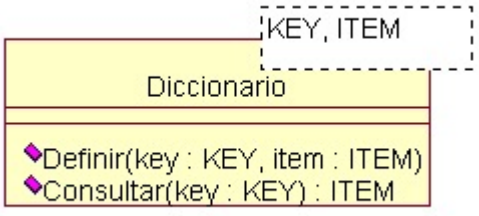


Representa un tipo de relación muy particular, en la que una clase es instanciada (su instanciación es dependiente de otro objeto/clase). Se denota por una flecha punteada.
El uso más particular de este tipo de relación es para denotar la dependencia que tiene una clase de otra, como por ejemplo una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

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).
7.2.4 Casos Particulares:
7.2.4.1 Clase Abstracta:



Una clase abstracta 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.
7.2.4.2 Clase parametrizada:



Una clase parametrizada 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

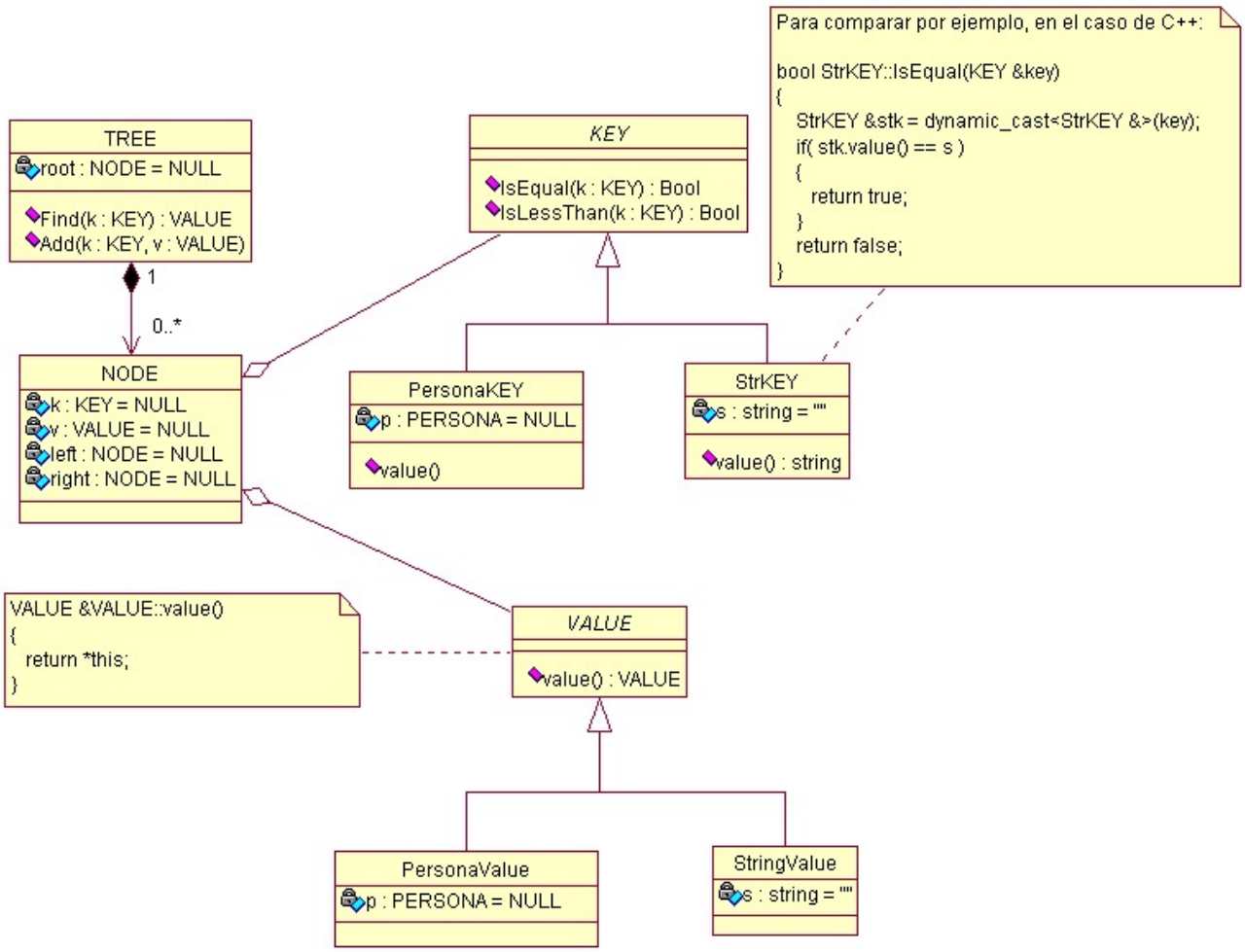
palabra tiene asociado un significado, pero en este caso las llaves y elementos pueden ser genéricos. La genericidad puede venir dada de un Template (como en el caso de C++) o bien de alguna estructura predefinida (especialización a través de clases).
En el ejemplo no se especificaron los atributos del Diccionario, pues ellos dependerán exclusivamente de la implementación que se le quiera dar.

7.2.4.3 Ejemplo:

Supongamos que tenemos tenemos un el caso del Diccionario implementado mediante un árbol binario, en donde cada nodo posee:

- key: Variable por la cual se realiza la búsqueda, puede ser generica.
- item: Contenido a almacenar en el diccionario asociado a "key", cuyo tipo también puede ser genérico.

Para este caso particular hemos definido un Diccionario para almacenar String y Personas, las cuales pueden funcionar como llaves o como item, solo se mostrarán las relaciones para la implementación del Diccionario:

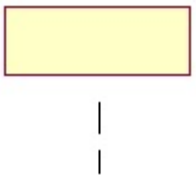


7.3 - Diagrama de Interacción

7.3.1 Introducción

El diagrama de interacción, 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.
La notación UML de los *diagramas de actividad* consiste en un cuadro dividido en una serie de columnas, cada columna representa un objeto/clase, de forma que cuando éste realiza alguna actividad se incluye en un cuadro la descripción de la actividad que realiza. El resultado de una actividad es un conjunto de datos, estos se indican en un cuadro en la *frontera* entre dos columnas, para indicar que esos datos resultantes *fluyen* de una clase a otra. Todo el diagrama está unido por flechas, indicando así el orden en el que se van realizando las diferentes actividades.
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 diágrama de interacción son:
7.3.2 Elementos

7.3.2.1 Objeto/Actor:

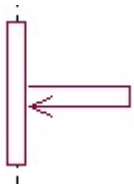


El rectángulo representa una instancia de un Objeto en particular, y la línea punteada representa las llamadas a métodos del objeto.

7.3.2.2 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.

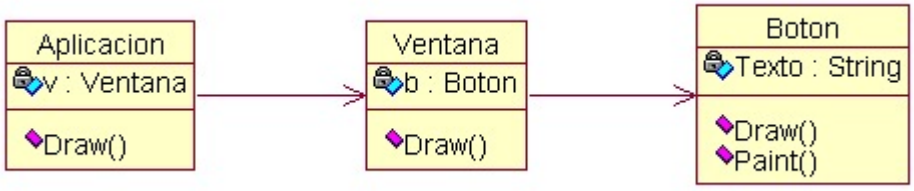
7.3.2.3 Mensaje al Mismo Objeto:



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

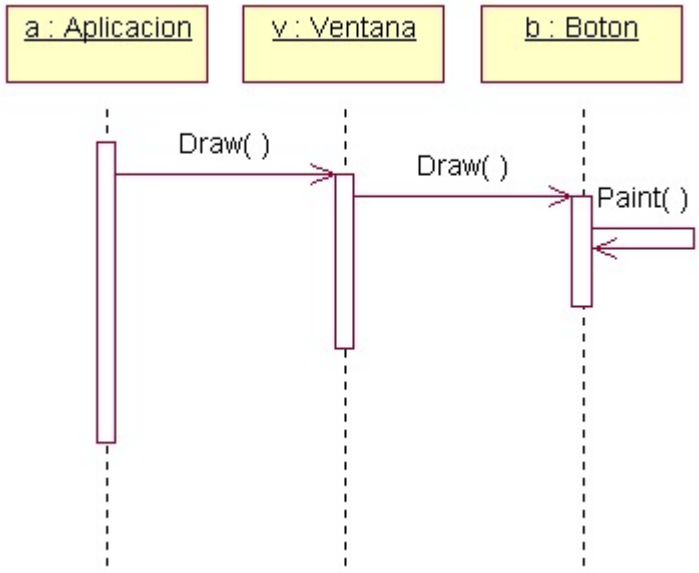
7.3.2.4 Ejemplo

En el presente ejemplo, tenemos el diagrama de interacción proveniente del siguiente modelo estatico:



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

Entonces el diagrama de interacción para dicho modelo es:



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

8- FASES DE CICLO DE VIDA

1. **Análisis global o conceptualización**: Es una primera fase en la que el principal objetivo es marcar las pautas del problema a resolver. Se suelen realizar conjuntos de requisitos, tareas, resultados, etc. que se pretende que el sistema pueda realizar.
2. **Análisis de requisitos**: Es una etapa en la que se pretende organizar los requisitos de la primera fase de forma ordenada y refinada, con la finalidad de poder realizar una *análisis* de los mismos para detectar posibles inconsistencias, omisiones, redundancias, etc.
3. **Diseño del sistema**: Se obtendrá una visión global del sistema que queremos desarrollar desde un punto de vista de alto nivel. En esta fase se ha han usado los *diagramas de casos.*
4. **Diseño de objetos**: Esta fase obtiene como resultado un *modelo de objetos* que podrá ser en la siguiente fase implementado. Este *modelo de objetos* debe representar la realidad que se desea abstraer en el sistema informático fielmente. Esta fase ha usado los elementos de UML denominados *diagrama de clases, diagramas de estado y diagrama de actividades.*
5. **Implementación**: Es necesario por último decidir el lenguaje en el que se va a implementar HALOTIS. Para ello se mostrarán las características o facilidades de las que disponemos en determinados lenguajes para declinarnos finalmente por un determinado lenguaje

9- INCONVENIENTES QUE PRESENTA UML

- UML no es una metodología, por cuanto no hace una definición del proceso de desarrollo, tarea que mucha gente tiende a querer desarrollar con UML.
- UML carece de integración con respecto a otras técnicas tales como patrones de diseño, interfaces de usuario, documentación, etc.
- Se ha generado un monopolio de conceptos, técnicas y métodos en torno a UML

10- VENTAJAS

Una de las grandes ventajas de UML es que es un lenguaje extensible que permite adaptarse a diferentes tipos de aplicaciones. Los mecanismos de extensibilidad de UML incluyen:

- **Estereotipos**: Permiten crear nuevos tipos de bloques deconstrucción a partir de los existentes pero que sean específicos para un determinado problema.
- **Valores etiquetados**: Extienden las propiedades de un bloque de construcción de UML, permitiendo añadir nuevainformación en la especificación de ese elemento.
- **Restricciones**: Extienden la semántica de un bloque deconstrucción UML, permitiendo añadir nuevas reglas omodificar las existentes. El lenguaje OCL (ObjectConstraint Language) permite especificar formalmente las restricciones.

11- ESTADO ACTUAL Y CARACTERÍSTICAS DE UML

El 12 de junio de 2003, en el encuentro técnico de la OMG celebrado en Paris, la Fuerza de Trabajo de Análisis y Diseño (Analysis and Design Task Force) ha votado recomendar la adopción de la especificación del Lenguaje de Modelado Unificado (Unified Modeling Language - UML) 2.0 Superestructura, completando la definición de una actualización mayor de la principal notación de modelado de software de la industria. También fueron recomendadas las especificaciones complementarias MetaObject Facility (MOF) para los núcleos MOF y XMI Metadata Interchange (XMI), actualizando el repositorio que sienta las bases sobre las cuales las herramientas UML y MDA son construidas. La alineación del meta-modelo UML 2.0 con el meta-modelo MOF simplificará el intercambio de modelos vía XMI y la interoperabilidad cruzada entre herramientas.

"Basados en nuestros más de 5 años de experiencia utilizando UML, hemos aprendido mucho sobre la unificación de lenguajes de modelado. Con este conocimiento, UML 2.0 representa literalmente el próximo paso evolutivo en nuestra habilidad para expresar y comunicar especificaciones de sistemas - lo cual provee las bases para MDA", dijo Jim Odell, destacado consultor y escritor así como también uno de los principales miembros de la Fuerza de Trabajo de Análisis y Diseño.

"Con más de 50 compañías contribuyendo con sus mejores tecnologías y prácticas en esta nueva versión, el proceso de la OMG una vez más se ratifica a sí mismo, generando un nuevo estándar que tendrá un impacto importante en el futuro del desarrollo de software", remarcó Dr. Richard Soley, CEO y Chairman de la OMG.

El estándar UML actualizado tiene ahora las siguientes características:
sus propias meta-clases, haciendo más fácil la definición de nuevos UML Profiles y extender el modelado a nuevas áreas de aplicación.
· Soporte incorporado para el desarrollo basado en componentes para facilitar el modelado de aplicaciones desarrolladas con Enterprise JavaBeans, CORBA o COM+. Permite el modelado de objetos y flujo de datos entre diferentes partes del sistema. El soporte para los ejecutables fue mejorado en general.
· Una representación más exacta y precisa de las relaciones mejora el modelado de herencia, composición y agregación, y máquinas de estado.
· Un mejor modelado del comportamiento mejora el soporte para encapsulación y escalabilidad, elimina las restricciones en el mapeo de gráficos de actividad a máquinas de estado y mejora la estructura de los diagramas de secuencias.
· Mejoras generales al lenguaje simplifican la sintaxis y la semántica y permiten organizar mejor la estructura general

Esto no significa que el nuevo estándar UML 2.0 ya está disponible para el público en general. Esto será una realidad en dos fases:
La primera fase implica la primera versión que será liberada en octubre de este año y se espera que aparezcan muchos libros y artículos al respecto.
La segunda fase puede llevar un año. Esta fase incluye un proceso de finalización que resultará en la versión definitiva. El contenido de especificación podría cambiar durante este tiempo debido a que el trabajo para alinear UML con MOF continúa. Luego la OMG liberará

esta nueva especificación al público como la versión oficial del nuevo estándar.

12- FUTURO DE UML

UML será el lenguaje de modelado Orientado a Objetos estándar por excelencia durante los próximos años por cuanto ha venido a llenar el importante vacío existente entre las especificicaciones de diseño y el cliente.
Otras razones por las cuales se justifica esta "predicción" es la participación de importantes empresas que impulsan su uso y desarrollo, la aceptación del OMG como como notación estándard y la gran cantidad de herramientas que actualmente soportan la notación UML (Poseidon, Rational Rose, ModelMaker, otras).

13 - CONCLUSIONES Y RESUMEN

UML se ha convertido, en poco tiempo, no sólo en un estándar de facto, sino también en el lenguaje de modelado más utilizado. Ha sido el resultado de un proceso abierto enel que han contribuido tanto sus “padres” originales, como numerosas organizaciones colaboradoras. El resultado es unlenguaje adaptable a diferentes tipos de aplicaciones.

UML, el lenguaje estándar para el modelado de aplicaciones software, aparece como un leguaje de modelado genérico, es decir, aplicable, al menos en principio, a cualquier tipo de aplicación. Esta facilidad de adaptabilidad de UML se debe a que proporciona mecanismos, los estereotipos, de extensibilidad que permiten adaptar UML a la aplicación concreta que se desea desarrollar.

Bibliografia

www.elrincondelprogramador.com
www.gris.det.uvigo.es
www.creangel.com/uml/
www.mititech.epublish.cl/articles/
www.dte.us.es
www.progrmafacil.com

INDICE PAGINA