Universidad Nacional del Litoral

FACULTAD DE INGENIERÍA
Y CIENCIAS HÍDRICAS

Ingeniería en Informática

Ingeniería de Software I

TEMA VI – Orientación a objetos - UML

Introducción

El lenguaje unificado de modelado (**UML**) es una de las herramientas de mayor uso en el mundo actual para el desarrollo de sistemas. Esto se debe a que permite a los creadores de sistemas poder generar diseños que capturen sus ideas en una forma convencional y uniforme y fácil de comprender para comunicarlas a otras personas.

La comunicación de las ideas es de suma importancia. Antes del advenimiento del UML, los analistas al evaluar los requerimientos de los usuarios, generaban un análisis de requerimientos apoyados en algún tipo de notación que ellos mismos comprendieran (aunque el cliente no lo hiciera) y daban tal análisis a los programadores y esperaban que el producto final cumpliese con lo que el cliente deseaba. Dado que el desarrollo de sistemas es una actividad humana, hay muchas posibilidades de cometer errores en cualquier etapa del proceso.

Conforme aumenta la complejidad del mundo, los sistemas informáticos también lo hacen. Para manejar la complejidad es necesario organizar el proceso de diseño de tal forma que los analistas, desarrolladores, clientes y otras personas vinculadas lo comprendan. El UML proporciona esa organización. Un constructor no podría crear una compleja estructura de edificios sin crear primero un proyecto detallado. Esos planos tienen notaciones estándares y cualquier profesional de esa especialidad lo podría comprender perfectamente. En sistemas de software ese estándar de facto que se ha adoptado es precisamente el UML.

El UML está compuesto por diversos elementos gráficos que se combinan para conformar diagramas. Debido a que el UML es un lenguaje, cuenta con reglas para combinar tales elementos. La finalidad de los diagramas es presentar diversas perspectivas de un sistema a las cuales se les conoce como ***modelo***. El modelo UML describe lo que supuestamente hará un sistema (el **qué**), pero no dice **cómo** implementarlo.

El UML es una creación de *Grady Booch, James Rumbaugh* e *Ivar Jacobson*. Cada uno de ellos durante los años ochenta y noventa diseñó su propia metodología para el análisis y diseño orientado a objetos. Éstas predominaron sobre sus competidores y a mediados de los noventa empezaron a intercambiar ideas entre sí y consecuentemente desarrollaron un trabajo en conjunto. Como se indicara, las distintas ideas eran relativas al análisis y diseño orientado a objetos, razón por la cual para comprender todo lo relacionado con el UML se requiere tener claro todo lo relacionado a la orientación a objetos.

6.1 ORIENTACIÓN A OBJETOS

Antes de comenzar con los fundamentos de la orientación a objetos, conviene mencionar a su oponente (el paradigma anterior a la OO y que se encuentra en proceso de cambio): El **análisis y diseño estructurado** (o funcional). En éste se construyen modelos basados en el procesamiento de datos. Quien utilice este enfoque intentará dividir el problema bajo estudio identificando una serie de procesos, manipulaciones o tratamientos de datos (llamados funciones o procedimientos en las fases de diseño e implementación) que, organizados de modo que puedan llamarse unos a otros, proporcionen la solución. Siempre existe una separación entre datos y procesos: los procesos manipulan y usan datos, pero no se integran con ellos.

El **paradigma estructurado** (análisis estructurado + diseño estructurado + programación estructurada) en la ingeniería de software se basa en la **abstracción por descomposición funcional o por procedimientos**: el problema estudiado se descompone en una serie de capas sucesivas de procesos, hasta que finalmente se descompone en procesos relativamente fáciles de implementar y codificar (desarrollo *top-down*). Un programa estructurado se divide en unidades lógicas (módulos) mediante el uso de funciones y procedimientos; los detalles más internos del programa residen en los módulos de más bajo nivel y los módulos de más alto nivel se encargan del control lógico del programa.

El paradigma estructurado mantendrá su vigencia, posiblemente, por cierto tiempo en la ingeniería del software pues muchas aplicaciones informáticas de gestión utilizan mayoritariamente pseudoobjetos u objetos que no pueden considerarse objetos de pleno derecho (por ejemplo, objetos sin atributos u objetos sin métodos), generalmente asociados a las estructuras tradicionales de datos. Otro de los defectos más comúnmente señalados por los críticos del paradigma estructurado es la separación clara e insalvable entre el análisis y el diseño, es decir, entre lo que se quiere que haga el sistema y cómo lo hace. En el paradigma OO la frontera entre el análisis y el diseño es difusa, y los objetos se introducen desde el principio. Es más natural pasar del análisis a la implementación en el paradigma OO que en el estructurado: el salto conceptual es menor en aquél.

La conversión de los términos del análisis OO hacia construcciones en lenguajes de programación OO es bastante directa, lo que constituye una importante ventaja sobre el paradigma estructurado.

Algunos autores consideran que el enfoque OO ha evolucionado a partir del enfoque estructurado y otros que es un salto revolucionario, cualitativo más que cuantitativo. Desde luego, los partidarios del salto revolucionario suelen ser firmes partidarios de la OO, cuando no creadores de metodologías OO. Así, en la obra *Object- Oriented Modeling and Design* **[Rumbaugh et al, 1991]** se considera que el diseño orientado a objetos es una nueva forma de pensar en los problemas usando modelos sobre conceptos del mundo real y, en *Ingeniería del Software - Un Enfoque práctico* **[Pressman, 1992]**, Pressman refiere que "*a diferencia de otros métodos de diseño, el diseño orientado a objetos da como resultado un diseño que interconecta los objetos de datos y las operaciones, de forma que modulariza la información y el procesamiento en vez de sólo la información. La naturaleza del diseño orientado a objetos está ligada a tres conceptos básicos: abstracción, modularidad y ocultación de la información.*"

Concluyendo, dado que no hay un acuerdo unánime, es que realmente el enfoque OO utiliza abstracciones no presentes en el enfoque estructurado y que no admiten equivalencia.

El paradigma de objetos fomenta una metodología basada en componentes de software de manera que primero se genera un sistema mediante un conjunto de objetos, luego podrá ampliarse agregándole funcionalidad a los componentes ya generados o agregándole nuevos componentes y finalmente, se podrán volver a utilizar los objetos generados para un nuevo sistema.

Para la introducción a la orientación a objetos se tratarán los siguientes conceptos fundamentales (algunos de ellos ya tratados en el tema anterior pero desde una perspectiva acotada a los datos):

- **Abstracción**
- **Encapsulamiento**
- **Herencia**
- **Polimorfismo**
- **Envío de mensajes**
- **Relaciones entre clases**
- **Agregación y composición**
- **Clasificación**

Un objeto es una instancia de una clase (o categoría). Cuenta con una ***estructura*** (es decir atributos o propiedades) y ***acciones***. Éstas son todas las actividades que el objeto es capaz de realizar. Por ejemplo, dada la clase **Persona** ésta puede tener como atributos *altura, peso, fecha de nacimiento* etc. También puede realizar tareas como *comer, dormir, leer, escribir, hablar,* etc. Si se tuviera que crear un sistema que manejara información acerca de las personas, sería muy probable que se incorporen algunos atributos y acciones en el software. Toda clase puede verse desde tres perspectivas distintas pero complementarias:

1. **Como conjunto o colección de objetos.**

   Esta perspectiva apunta a la columna vertebral del concepto de clase: una clase implica clasificación y abstracción. En realidad, una clase no deja de ser la formalización y verbalización de unos procesos que los seres humanos realizan continuamente, a menudo de forma inconsciente y preprogramada. El ser humano piensa con ideas, con abstractos; y, a medio camino entre la percepción y la cognición, se halla la función de clasificar, es decir, la función de poder afirmar que aquello percibido pertenece a un grupo de cosas (clase) o a otro. Pensar consiste en buscar semejanzas y olvidar diferencias; consiste en abstraer, en generalizar.

   Por la propia naturaleza de la percepción humana, resulta difícil razonar con conceptos que rompen clasificaciones establecidas. Por ejemplo, seguramente muchos biólogos no hayan podido entender qué pasaba cuando se descubrió el ornitorrinco. El ornitorrinco rompía una clasificación fundamental de los zoólogos: la de mamífero y no mamífero. Este curioso animal que pone huevos, rompía la tradicional clasificación según la cual los mamíferos tienen pelo o algo similar en alguna parte del cuerpo, son lactantes siendo crías, son de sangre caliente y **no ponen huevos**.

   Todos los seres humanos sabemos que un pájaro volando en vertical no deja de ser un pájaro. En contraste, las máquinas y los sistemas de software (incluso las inteligencias artificiales) carecen de esa capacidad de abstracción y de esa extraña capacidad informe llamada "sentido común". En el campo del reconocimiento digital de imágenes, por ejemplo, resulta difícil que los sistemas de reconocimiento comprendan que un ave – o un avión, o cualquier forma geométrica – volando en vertical, en horizontal o en cualquier otra orientación sigue siendo la misma ave o el mismo objeto que en reposo. Es decir, resulta difícil que comprendan que se encuentran ante distintos estados de un mismo objeto. Pese a los importantes avances logrados en el campo de las inteligencias artificiales, poco se ha avanzado en el desarrollo de sistemas de IA generales. En consecuencia, el análisis OO, basado en la

abstracción y que incluye la identificación de las clases del dominio del problema, continúa siendo una actividad humana.

2. **Como plantilla para la creación de objetos.**
   En esta perspectiva se destaca que la relación entre una clase y los objetos derivados de ella puede considerarse como la existente entre una fábrica y las cosas producidas por ésta. Un ejemplo: una fábrica de automóviles produce automóviles del mismo modo que una clase Automóvil crea objetos automóviles. Una clase Automóvil solamente producirá objetos automóviles, del mismo modo que una fabrica real de automóviles sólo produce coches, no televisores o aviones.

3. **Como definición de la estructura y comportamiento de una clase.**
   En esta se hace hincapié en que la definición de una clase permite definir una sola vez la estructura común, así como usarla cuantas veces sea necesario.

Se llama atributo a la abstracción de una característica común para todas las instancias de una clase, o a una propiedad o característica de un objeto. Las operaciones de una clase son servicios ofrecidos por ésta, que llevan (o pueden llevar) a cambios en el estado de un objeto de dicha clase. Conceptualmente, una operación puede considerarse como una petición a un objeto para que haga algo. Cuando las clases se implementan en lenguajes OO, se habla de variables de instancia (implementaciones software de atributos) y de métodos (implementaciones software de operaciones). Cada instancia de una clase (u objeto) contiene su propio conjunto de variables de instancia, cuyos valores pueden variar con el tiempo. El conjunto de los valores tomados por las variables de instancia de un objeto, en un instante dado, representa el estado del objeto. Dicho en sentido inverso, el estado de un objeto viene representado por los datos almacenados en sus variables de instancia.

Cuando se consideran sistemas de software, enseguida se percibe la tendencia evolutiva de los lenguajes de programación hacia la representación y reproducción de la manera como nosotros contemplamos el mundo que nos rodea. Muy pocas personas comprenden a primera vista la secuencia de código ensamblador que permite sumar dos números enteros; menos aún pueden pensar directamente en binario; pero casi todos entendemos que una clase Entero representa el conjunto de los números enteros, y que éstos tienen su propio comportamiento, en el cual se incluye la operación suma.

Con la evolución y el desarrollo de los lenguajes de programación, se reproduce el modo de pensar al cual nos ha conducido la evolución, o sea que estos lenguajes tienden a ver el mundo tal como nosotros lo vemos de manera totalmente natural.

6.1.1 La abstracción

Es la vía fundamental por la que los humanos combaten la complejidad. Surge de un reconocimiento de las similitudes entre ciertos objetos, situaciones o procesos del mundo real, y la decisión de concentrarse en esas similitudes e ignorar por el momento las diferencias. Es una descripción simplificada que enfatiza ciertos detalles significativos y suprime otros por irrelevantes. Denota las características esenciales de un objeto que lo

distinguen de todos los demás tipos de objeto y proporciona así fronteras conceptuales nítidamente definidas respecto a la perspectiva del observador. En definitiva, se refiere a quitar propiedades y acciones de un objeto para dejar solamente aquellas necesarias. Diferentes problemas requieren distintas cantidades de información, aun si estos problemas pertenecen a un área en común.

6.1.2 El encapsulamiento

La esencia del encapsulamiento es que cuando un objeto trae consigo su funcionalidad, ésta última se oculta. Por lo general, la mayoría de las personas que ven televisión no sabe ni se preocupa de la complejidad electrónica que hay detrás de la pantalla ni de todas las operaciones que tienen que ocurrir para mostrar una imagen. El televisor hace lo tiene que hacer sin mostrar el proceso necesario para ello (como la mayoría de los electrodomésticos). En un sistema de consta de objetos de software, éstos depende unos de otros de alguna manera. Si uno de ellos falla y los especialistas de software tienen que modificarlo de alguna forma, el ocultar sus operaciones de otros objetos significará que tal vez no será necesario modificar los demás objetos. De esta manera, el monitor de la computadora, en cierto sentido, oculta sus operaciones de la CPU, o sea que si algo falla en el monitor, se reparará o reemplazará sin tener que reparar o reemplazar la CPU.

El encapsulamiento entonces es el ocultamiento de la información. Se esconden todos los detalles internos del sistema que se estudia como una caja negra y resulta más importante comprender **qué** hace que el **cómo** lo hace. La abstracción y el encapsulamiento son conceptos complementarios: la abstracción se centra en el comportamiento observable de un objeto, mientras que el encapsulamiento se centra en la implementación que da lugar a este comportamiento. El encapsulamiento se consigue mediante la ocultación de la información, que es el proceso de ocultar todos los secretos de un objeto que no contribuyen a sus características esenciales; típicamente, la estructura de un objeto está oculta, así como la implementación de sus métodos.

Entonces, cada clase debe tener dos partes: *una interfaz* y *una implementación*. La *interfaz* de una clase, captura solamente su vista externa que alcanza a la abstracción que se ha hecho del comportamiento común de todas las instancias de la clase y ahí se listan o declaran las acciones que pueden tener lugar. La *implementación* de una clase comprende la representación de la abstracción así como los mecanismos que consiguen el comportamiento deseado. El encapsulamiento es la característica fundamental que debe poseer la implementación ya que se deben esconder todos los detalles de cómo se hacen las cosas listadas en la interfaz. Para el caso planteado del televisor, la interfaz del mismo viene dada a través de las perillas y botones o el control remoto.

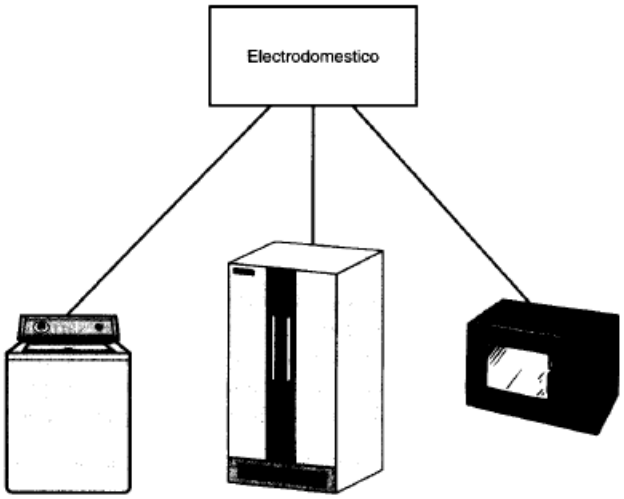
6.1.3 La herencia

La herencia es la jerarquía de clases más importante y es un elemento esencial de los sistemas orientados a objetos. Define una relación entre clases en la que una clase comparte la estructura de comportamiento definida en una o más clases (herencia simple o herencia múltiple respectivamente). La herencia denota una relación ***es un***. A medida que se desarrolla la jerarquía de herencias, la estructura y comportamiento comunes a diferentes clases tenderá a migrar hacia superclases comunes. Por esta razón se dice que la herencia es

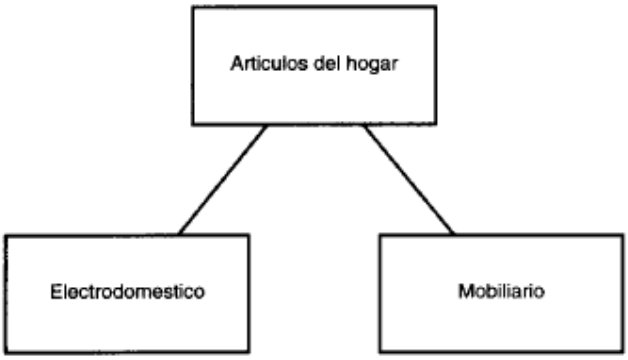
una jerarquía de *generalización/especialización*: las superclases representan abstracciones generalizadas y las subclases representan especializaciones en las que los atributos y métodos de la superclase sufren añadidos, modificaciones o incluso ocultaciones. Si no existiese la herencia, cada clase sería una unidad independiente desarrollada a partir de cero. El hecho que una subclase herede de otra clase implica que se viola el encapsulamiento de la superclase. Los distintos lenguajes de programación, hacen concesiones al respecto, definiendo por ejemplo partes *privadas, protegidas y públicas* las que restringen el acceso de los clientes.

La herencia múltiple es conceptualmente correcta, pero en la práctica introducen ciertas complejidades en los lenguajes de programación. Estos problemas son por ejemplo, colisiones entre nombres de superclases diferentes y la herencia repetida.

Por ejemplo, para el caso del televisor o la lavadora o heladera, microondas, lavaplatos, etc, si bien son clases, forman parte de una clase más genérica: **Electrodomestico**. Un electrodoméstico cuenta con los atributos de interruptor y cable eléctrico y las operaciones de encendido y apagado. Cada una de las clases que dependen de **Electrodomestico** heredarán los mismos atributos y operaciones. Se dice además que **Electrodomestico** es la superclase y que las otras son subclases.



La herencia no tiene porqué terminar ahí, pueden ocurrir que **Electrodomestico** sea a su vez subclase de **Articulos del hogar** como se muestra en la figura:



6.1.4 El polimorfismo

Según Rumbaugh: "*Polimorfismo: Toma de varias formas; propiedad que permite a una operación tener distintos comportamientos en diferentes clases*". En ocasiones puede ocurrir que una misma operación tenga el mismo nombre en diferentes clases, como ser ***abrir()*** se puede abrir una puerta, una persiana, un periódico, una cuenta bancaria, un regalo, etc. y en caso se hará un acción diferente. En la orientación a objetos, cada clase ***sabe*** como realizar tal operación. Esto es el polimorfismo.

En primera instancia, parecería que este concepto es más importante para los desarrolladores que para los modeladores, ya que los primeros tienen que crear el software que implemente tales métodos en los programas y deben estar conscientes de diferencias importantes entre las operaciones que pudieran tener el mismo nombre. No obstante también es importante para los modeladores ya que permite hablar con el cliente en sus propias palabras y terminología.

6.1.5 Envío de mensajes

Se mencionó que en un sistema los objetos trabajan en conjunto. Esto se logra mediante el envío de mensajes entre ellos. Un objeto envía a otro un mensaje para realizar una operación y el objeto receptor la ejecutará.

Un televisor y su control remoto son un ejemplo muy intuitivo. Cuando se presiona el botón de encendido el control remoto le envía un mensaje al televisor para que se encienda. El televisor recibe el mensaje, lo identifica como una petición para encender y ejecuta ese método (encender). Cuando se desea ver otro canal, se presiona el botón correspondiente y nuevamente se envía otro mensaje que el televisor recibe y procesa. Muchas de las cosas que se hace mediante el control remoto también se podrían hacer sobre el televisor presionando los botones correspondientes. La interfaz que el televisor presenta no es la misma que le muestra el control remoto sin embargo el efecto es el mismo.

6.1.6 Relaciones entre clases

Considérense las analogías y diferencias entre las siguientes clases de objetos: *flores, margaritas, rosas rojas, rosas amarillas, pétalos y mariquitas*. Supónganse las siguientes observaciones:

- Una margarita es un tipo de flor,
- Una rosa es un tipo (distinto) de flor,
- Las rosas rojas y las amarillas son tipos de rosas.
- Un pétalo es una parte de ambos tipos de flores.
- Las mariquitas se comen ciertas plagas como los pulgones que pueden infectar ciertos tipos de flores.

Se establecen relaciones entre dos clases por las siguientes razones:

- Una relación entre clases podría indicar un tipo de **compartición**. Por ejemplo, las margaritas y las rosas son tipos de flores, lo que indica que ambas tienen pétalos con colores llamativos, fragancia, etc.
- Una relación entre clases podría indicar algún tipo de **conexión semántica**, así se dice que las rosas rojas y las amarillas se parecen más entre ellas que las margaritas y las rosas, y las margaritas y las rosas se relacionan entre ellas más estrechamente que los pétalos y las flores. Análogamente existe una conexión simbiótica entre las mariquitas y las flores: las mariquitas protegen a las flores de ciertas plagas que a su vez sirven de fuente de alimento a la mariquita.

  En total existen tres tipos básicos de relaciones entre clases:

  - La generalización / especialización que denota **es un**. Por ejemplo, la rosa **es un** tipo de flor lo que quiere decir que una rosa es una subclase especializada de una clase más general (la de las flores). **Herencia**
  - La relación todo/parte que denota una relación **parte de**. Por ejemplo, un pétalo no es un tipo de flor, sino que es una **parte de** la flor. **Agregación**
  - La asociación que denota alguna dependencia semántica entre clases de otro modo independientes como entre las mariquitas y las flores. **Asociación**.

Las asociaciones son el tipo más general pero también el de mayor debilidad semántica. La identificación de asociaciones entre clases, es frecuentemente una actividad de análisis y diseño inicial, momento en el cual se comienza a descubrir las dependencias generales entre las abstracciones. A medida que se continúa el diseño y la implantación, se refinarán a menudo estas asociaciones débiles orientándolas hacia una de las otras relaciones de clase más concretas. Aparecen entonces las agregaciones y la herencia. De todas maneras se necesitan las relaciones de uso que establecen los enlaces entre las clases.

1. **Relación de herencia**

   La herencia permite que unos objetos puedan basarse en otros ya existentes. En términos de clases, la herencia es el mecanismo por el cual una clase **X** puede heredar propiedades de una clase **Y** de modo que los objetos de la clase **X** tengan acceso a los atributos y operaciones de la clase **Y** sin necesidad de redefinirlos. Suelen usarse los términos *hijo/padre* o *subclase/superclase* para designar el par. A veces se dice que la subclase es una especialización de la superclase y que la superclase es una generalización de las subclases.

   La herencia presenta dos cualidades contradictorias entre sí. A saber: una clase hija extiende o amplia el comportamiento de la clase padre, pero también restringe o limita a la clase padre (una subclase se encuentra más especializada que su clase padre). Existe cierta tirantez esencial, constitutiva, entre los dos conceptos (herencia como extensión y herencia como especialización); a veces se olvida esta tensión, pero siempre sigue ahí.

   Suele identificarse la herencia mediante la regla “es un” o “es un tipo de”. Por ejemplo, toda ***Motocicleta*** es un ***Vehículo***, luego la clase ***Motocicleta*** es una subclase de la clase ***Vehículo***. En general, no resulta recomendable emplear la herencia cuando no funcione la regla “X es un Y” o, más precisamente, cuando no pueda justificarse que todo instancia de la clase X es tam-

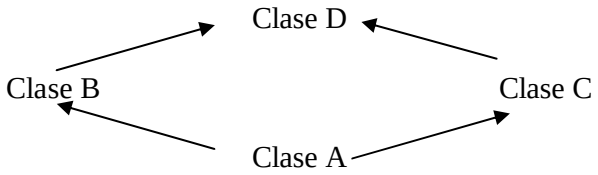
bién una instancia de Y. Por ejemplo: una clase ***ConductorMotocicleta*** que heredase de dos superclases ***Persona*** y ***Motocicleta*** constituiría un mal uso de la herencia ya que no toda instancia de ***ConductorMotocicleta*** es una instancia de ***Motocicleta***. Del mismo modo, no conviene establecer una relación de herencia entre una clase ***Motor*** y una clase ***Coche***, pues un coche no es un género de motores. El uso de la herencia para reutilizar código entre clases que incumplen la regla "es un" suele considerarse incorrecto, aunque a veces se permite y se denomina herencia de implementación o de funcionalidad.

La relación "es un" ilustra una característica crucial de la herencia: un objeto de una subclase puede usarse en cualquier lugar donde se admita un objeto de la superclase. Lo contrario no es cierto (esto también ocurre en el mundo real: los padres, por ejemplo, no heredan los rasgos de los hijos ni pueden intercambiarse por ellos).

La noción de que un objeto de una clase hija puede sustituirse por un objeto de la clase padre conduce inexorablemente a que se incorporen, cuando hablemos de los objetos pertenecientes a una clase, los objetos pertenecientes a todas las subclases. Por claridad, se usa "la clase" de un objeto para referirse a la clase más especializada de la cual es instancia el objeto. Lo dicho en el párrafo inmediatamente superior puede escribirse un poco más precisamente: ***un objeto de una clase especializada puede substituirse por un objeto de una clase más general en cualquier situación donde se espere un miembro de la clase general, pero no al revés***.

La herencia se clasifica asimismo en herencia simple y herencia múltiple. En la herencia simple, una clase sólo hereda (es subclase) de una superclase. En la herencia múltiple, una subclase admite más de una superclase. Entendamos que heredar de más de una superclase no quiere decir que la herencia múltiple consista en que una subclase pueda heredar de una clase que sea, a su vez, subclase de otra clase. Una subclase que herede, mediante herencia múltiple, de dos o más superclases puede mezclar las propiedades de las superclases, en ocasiones de forma ambigua o poco recomendable. Dos problemas fundamentales se presentan con la herencia múltiple:

- Herencia repetida: La clase A hereda de B y C, que a su vez derivan de D. Consecuencia: la clase A hereda dos veces de D.

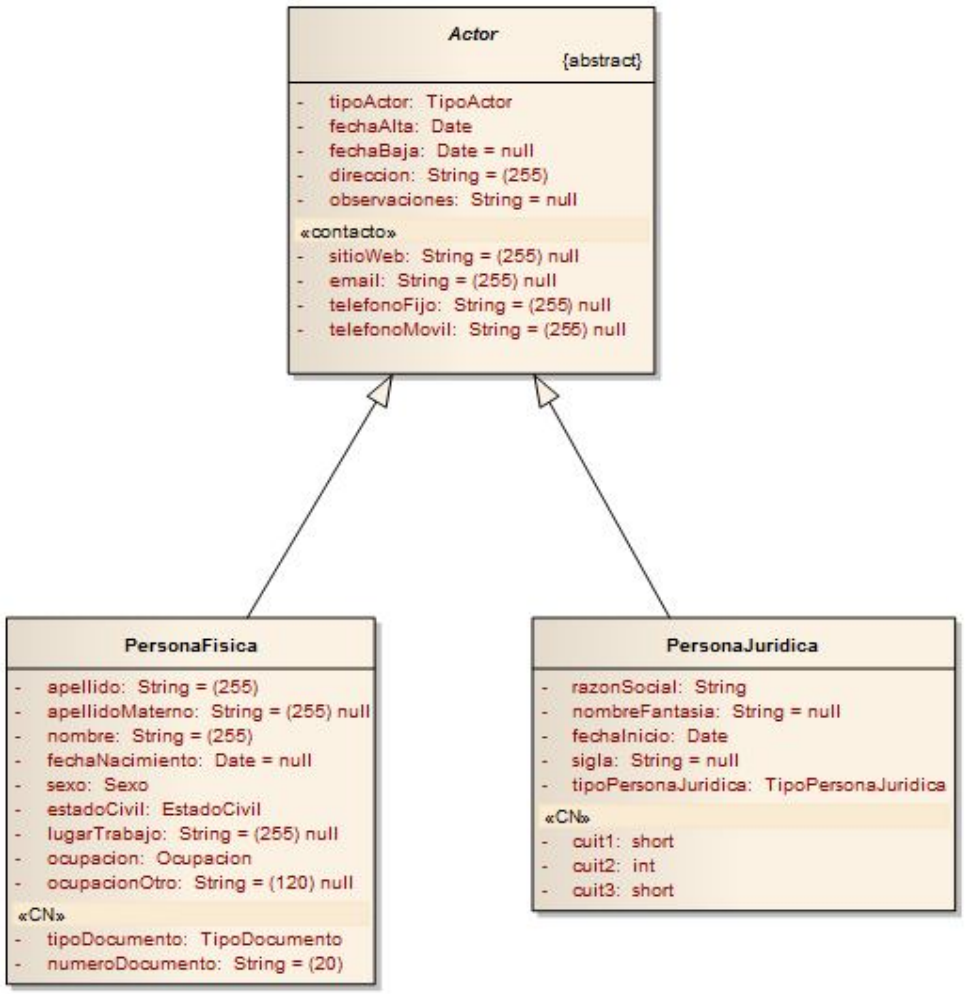


- Conflictos de nombres. Si una clase A hereda simultáneamente de dos superclases B y C, aparecerá un conflicto de nombres si usan el mismo nombre para algún atributo o método. ¿Qué definición usará A del atributo o del método con el mismo nombre? ¿La de la superclase B o la de C? ¿O ninguna de ellas? Generalmente este problema se solu-

ciona redefiniendo en la subclase la propiedad o método con el mismo nombre en las superclases.

Con la herencia simple, cada subclase tiene exactamente una superclase. Sin embargo, fuerza frecuentemente al programador a derivar de una sola entre dos clases factibles. Esto limita la aplicabilidad de las clases predefinidas haciendo muchas veces necesario el duplicar código. Por ejemplo, no existe forma de derivar un gráfico que es a la vez un círculo y una imagen; hay que derivar de uno o del otro y ***reimplantar la funcionalidad de la clase que se excluyó***.

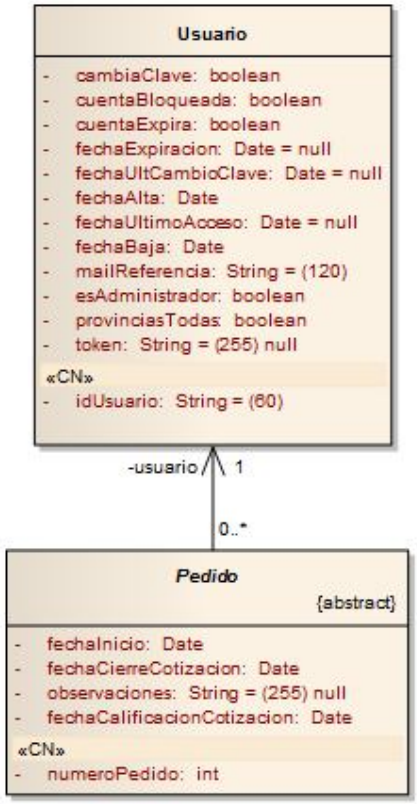
Ejemplo de herencia:



2. Relación de asociación.

Suponga el caso típico de factura y artículos de las factura. Esto se puede representar mediante una asociación simple entre las dos clases. Esta asociación sugiere una relación bidireccional ya que dada una instancia de artículo se debería poder encontrar la factura que denota su venta y dada una instancia de factura se deberían poder localizar todos los artículos que la componen. Esta es una asociación de uno a muchos en la que cada instancia

de artículo puede tener un puntero a su última factura y cada instancia de venta puede tener una colección de punteros que denota los artículos vendidos. La asociación sólo denota una dependencia semántica y no establece la dirección de esa dependencia (a menos que se indique expresamente, una asociación implica relación bidireccional) ni establece la forma exacta en que una clase se relaciona con otra (solo puede denotarse esta semántica nombrando el rol que desempeña cada clase en relación con la otra). Sin embargo, esta semántica es suficiente durante el análisis de un problema ya que es necesario identificar estas dependencias determinando los participantes, rol y su cardinalidad. La cardinalidad indica la multiplicidad de la asociación. Es el mismo concepto de funcionalidad visto anteriormente (uno a uno, uno a muchos y muchos a muchos). En el ejemplo se muestra un Pedido que es realizado por uno solamente un usuario. La visibilidad es desde el pedido hacia el usuario y se indica con una punta de flecha.

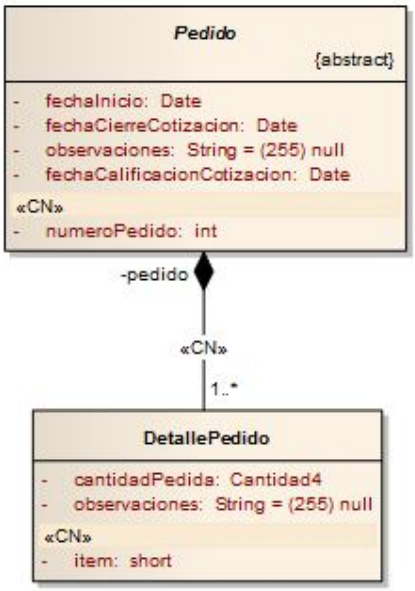


3. Relación de agregación / composición.

Las **agregaciones** son una especialización de las asociaciones, vinculada a una relación de la forma todo-parte. Los objetos parte forman parte del objeto todo, pero la vinculación entre las partes y el todo no es absoluta: se pueden crear y destruir de modo independiente instancias de cada clase involucrada en la relación. La relación todo-parte suele reconocerse porque se manifiesta en la forma de "X está compuesto por Y". Por ejemplo: "los dibujos están formados por elementos gráficos", "la casa está formada por muros", etcétera. Un ejemplo de agregación nos lo da la relación entre un mouse y una computadora. Un mouse puede quitarse de una computadora y colocar-

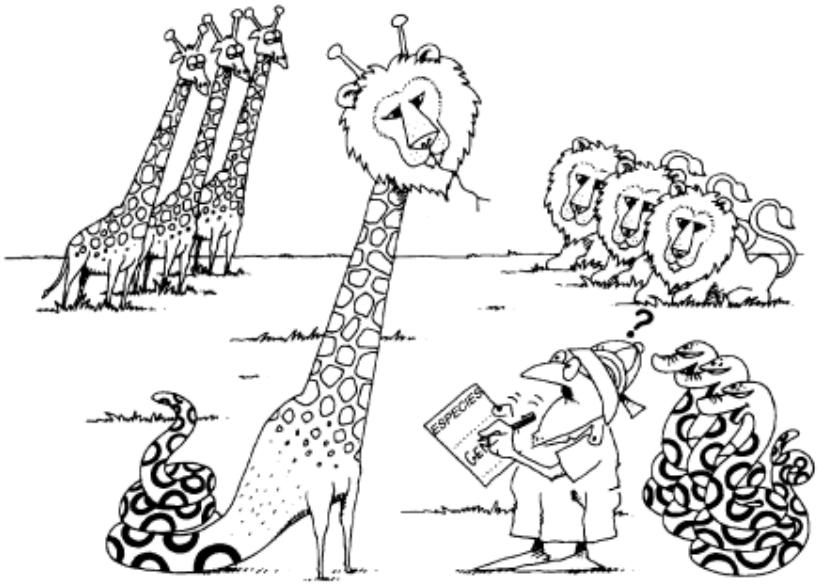
se en otra. Si se considera una clase equipo de club y un jugador, correspondientes a las entidades del mundo real que todos conocemos, la relación entre ambas es una agregación. Todos los objetos jugador de una instancia de equipo de club pueden ser destruidos, y sin embargo la instancia de equipo seguirá existiendo. En el mundo real, puede haber equipos de club sin jugadores. También es posible añadir nuevas instancias de jugadores a un objeto equipo de club ya creado. En el mundo real, pueden aparecer nuevos jugadores e incorporarse a equipos y un equipo puede prestar jugadores a otro.

Las **composiciones** son también una especialización de las asociaciones, vinculada asimismo a una relación del tipo todo-parte, pero con un vínculo absoluto y permanente. La composición indica que los objetos parte están contenidos físicamente en el objeto todo. Como consecuencia, los tiempos de vida de las partes se hallan fuertemente relacionados con el tiempo de vida del todo. Cuando se crea una instancia del objeto todo, se crea una instancia de cada uno de sus objetos parte; y cuando se destruye la instancia todo, se destruyen todas las instancias de los objetos parte. Un objeto parte no puede asignarse a un objeto todo con el cual no se haya creado.



La **agregación** puede o no denotar contención física. Por ejemplo un avión, se compone de alas, motores, tren de aterrizaje, etc. o sea, es un caso de contención física. Una persona puede tener un historial de estados civiles pero estos estados no son de ninguna manera parte física de la persona. Esta relación todo/parte es más conceptual por lo tanto menos directa que la agregación física de las partes que conforman un avión.

6.1.7 La clasificación



La clasificación es el medio por el que ordenamos el conocimiento. En el diseño orientado a objetos, el reconocimiento de la similitud entre las cosas nos permite exponer lo que tienen en común en abstracciones clave y mecanismos, y eventualmente nos lleva a arquitecturas más pequeñas y simples. No existe una receta para la clasificación. No existe a lo que se pueda llamar una estructura de clases *perfecta* ni el conjunto de objetos *correcto*. Al igual que en cualquier disciplina de la ingeniería, nuestras elecciones de diseño son un compromiso conformado por muchos factores que compiten. La identificación de las clases y objetos es la parte más difícil del diseño orientado a objetos. La experiencia muestra que la identificación implica descubrimiento e invención. Mediante el descubrimiento, se llega a reconocer las abstracciones clave y los mecanismos que forma el vocabulario del dominio del problema. Mediante la invención, se idean abstracciones generalizadas. Cuando se clasifica, se persigue agrupar las cosas que tienen una estructura común o exhiben un comportamiento común. La clasificación ayuda a identificar jerarquías de **generalización**, **especialización** y **agregación** entre clases.

Ejemplo de clasificación:

Se ha visto que un objeto es algo que tiene una frontera definida con nitidez. Sin embargo, las fronteras que distinguen un objeto de otro, son a menudo difusas. Por ejemplo, si se fijan en su pierna, ¿dónde empieza la rodilla y donde termina? En el reconocimiento del habla humana, ¿cómo saber que ciertos sonidos se conectan para formar una palabra y no son en realidad parte de otras palabras circundantes? En un procesador de texto, ¿constituyen los caracteres una clase o son las palabras completas una mejor elección? ¿Cómo tratar selecciones arbitrarias no contiguas de texto, y qué hay sobre las oraciones, párrafos o incluso documentos completos: son relevantes para el problema estas clases de objetos?

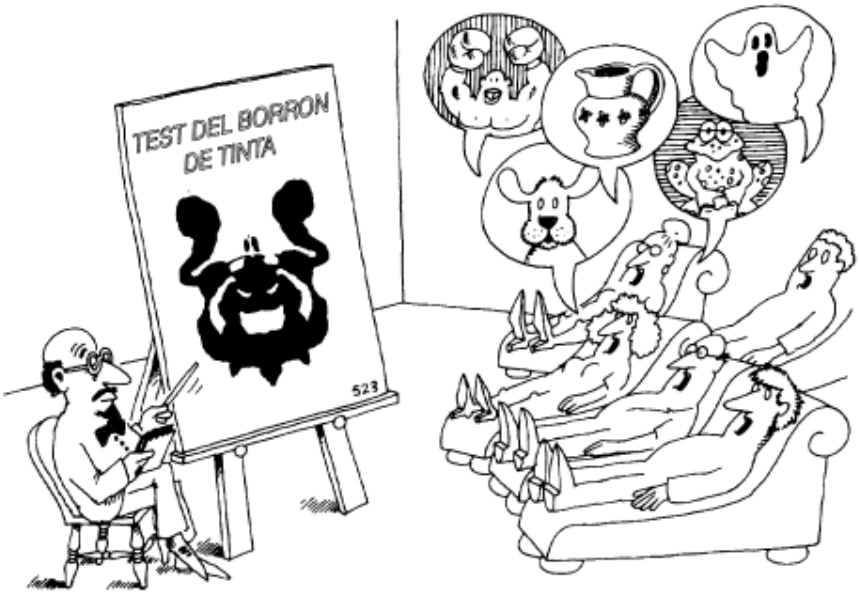
La clasificación es difícil puesto que existen paralelismos con los mismos problemas en el diseño orientado de objetos. Considérese por ejemplo la biología. Hasta el siglo XVIII

la opinión científica más extendida era que todos los organismos vivos podían clasificarse del más simple hasta el más complejo, siendo la medida de la complejidad algo muy subjetivo (¿por ejemplo, por qué será que los humanos fuesen situados en lo más alto de esta lista?). A mediados del mismo siglo, el botánico sueco *Carl Von Linneo* sugirió una taxonomía más detallada para categorizar los organismos de acuerdo con lo que se llama *géneros y especies*. Un siglo más tarde, *Darwin* propuso la teoría de que la selección natural fue el mecanismo de la evolución en virtud de la cual las especies actuales evolucionaron de otras más antiguas. La teoría de *Darwin* dependía de una clasificación inteligente de las especies. Como el propio *Darwin* afirma, los naturalistas *intentan disponer las especies, géneros y familias en cada clase en lo que se denomina el sistema natural*. Algunos autores lo toman a este sistema como un mero esquema para poner juntos aquellos objetos vivos que son más parecidos, y para separar los que son más distintos. En la biología actual, la clasificación denota el establecimiento de un sistema jerárquico de categorías sobre las bases de presuntas relaciones naturales entre organismos. La categoría más general en una taxonomía biológica es el reino, seguido en orden de especialización creciente por el filum, subfilum, clase, orden, familia, género y finalmente, especie. Históricamente un organismo concreto se sitúa en una categoría específica de acuerdo con su estructura corporal, características estructurales internas y relaciones evolutivas. Más recientemente, la clasificación se ha enfocado como la agrupación de organismos que comparten una herencia genética común: los organismos que tienen ADN similar se incluyen en el mismo grupo. La clasificación por ADN es útil para distinguir organismos que son estructuralmente similares, pero genéticamente muy diferentes. Para un informático, la biología parece una disciplina muy madura y con criterios bien definidos para clasificar organismos. Pero esto no es así. Como indican biólogos actuales, a nivel puramente de hechos, no se sabe ni siquiera dentro de un orden de magnitud cuántas especies de plantas y animales existen en el planeta; actualmente se han clasificado menos de 2 millones y las estimaciones del número total varían entre menos de 5 a 50 millones. Además, criterios diferentes para clasificar los mismos organismos arrojan resultados distintos. En definitiva, *todo depende de para qué quiere uno la clasificación*. Si se desea reflejar con precisión las relaciones genéticas entre las especies, ofrecerá una respuesta, pero si en vez de eso, se quiere decir algo sobre niveles de adaptación, la repuesta será distinta. La conclusión es que incluso en disciplinas rigurosamente científicas, la clasificación es altamente **dependiente de la razón por la que se clasifica**.

La clasificación inteligente es un trabajo intelectualmente difícil y la mejor forma de realizarlo es mediante un proceso incremental e iterativo. Primero los problemas se resuelven *ad hoc*. A medida que se acumula la experiencia, se va viendo que algunas soluciones funcionan mejor que otras y se transfiere informalmente una especie de folklore de persona a persona. Eventualmente las soluciones útiles se comprenden de forma más sistemática y se codifican y analizan. Esto permite el desarrollo de modelos que admiten una implantación automática y de teorías que permiten generalizar la solución. Esto a su vez da lugar a un nivel de práctica más sofisticado y nos permite atacar problemas más difíciles a los que con frecuencia se brinda un enfoque *ad hoc* comenzando el ciclo de nuevo. La naturaleza incremental e iterativa de la clasificación tiene un impacto directo en la construcción de jerarquías de clases y objetos en el diseño de un sistema de software complejo. En la práctica, es común establecer una determinada estructura de clases en fases tempranas del diseño y revisar entonces esa estructura a lo largo del tiempo. Sólo en etapas más avanzadas del diseño, una vez que se han construido los clientes que utilizan tal estructura se puede evaluar de forma significativa la calidad de la clasificación. Sobre la base de esta experiencia se puede decidir crear nuevas subclases a partir de otras existentes (derivación) o se puede dividir una clase grande en varias más pequeñas (factorización) o crear una clase mayor

uniendo otras más pequeñas (composición). Ocasionalmente se puede incluso descubrir aspectos comunes que habían pasado desapercibidos e idear una nueva clase (abstracción). Dos conclusiones:

- La clasificación es relativa a la perspectiva del observador que la realiza.
- La clasificación inteligente requiere una tremenda cantidad de perspicacia creativa.



Diferentes observadores pueden clasificar el mismo objeto de distintas formas.

Métodos para la clasificación de objetos y clases

Históricamente han existido tres aproximaciones generales a la clasificación:

- Categorización clásica
- Agrupamiento conceptual
- Teoría de prototipos

1. Categorización clásica

En la aproximación clásica a la categorización, todas las entidades que tienen una determinada propiedad o colección de propiedades en común forman una categoría. Tales propiedades son necesarias y suficientes para definir la categoría. Por ejemplo, las personas casadas constituyen una categoría, *se está casado o no*, y el valor de esta propiedad es suficiente para decidir a qué grupo pertenece un individuo. Por otra parte, las personas altas no forman una categoría, al menos que se determine un criterio absoluto por el que se distinga la propiedad *alto* de la propiedad *bajo*. Esta categorización emplea propiedades relacionadas como criterio de similitud entre objetos. Concretamente se puede dividir los objetos en conjuntos disjuntos dependiendo de la presencia o ausencia de una propiedad particular. En sentido general,

las propiedades pueden denotar algo más que meras características medibles, puede también abarcar ***comportamientos*** observables (hay animales que pueden volar y otros que no, y esta propiedad los puede distinguir). Las propiedades particulares que habría que considerar en una situación dada dependen mucho del dominio, por ejemplo, el color de un auto puede ser importante para el propósito de un control de inventario en una fábrica, pero no es relevante en absoluto para el software que controla los semáforos en una ciudad. Esta es la razón por la que se dice que no hay medidas absolutas para la clasificación aunque una estructura de clases determinada puede ser más adecuada para una aplicación que para otra. Algunas clasificaciones pueden verse mejor que otras, pero sólo respecto a nuestros intereses, no porque representan la realidad de forma más exacta o adecuada. De todas maneras parece prácticamente imposible proponer una lista de propiedades para cualquier categoría natural que excluya a todos los ejemplos que no están en la categoría e incluya a todos los que si están (p.e. la mayoría de los pájaros vuelan pero algunos no lo hacen pero siguen siendo pájaros – *¿o no?*). Estos son realmente problemas fundamentales de la categorización clásica, que el agrupamiento conceptual y la teoría de prototipos intentan resolver.

2. Agrupamiento conceptual

Es una variación más moderna del anterior y deriva en gran medida de los intentos de explicar cómo se representa el conocimiento. En este enfoque, las clases (agrupaciones de entidades) se generan formulando primero descripciones conceptuales de estas clases y clasificando entonces las entidades de acuerdo con las descripciones. Por ejemplo, se puede establecer un concepto como ***una canción de amor***. Éste es más bien un concepto que una propiedad porque la ***cantidad de amor*** de cualquier canción no es algo que se pueda medir empíricamente. Sin embargo, si se decide que cierta canción tiene más de canción de amor que de otra cosa, se la coloca en ésta categoría. Así, el agrupamiento conceptual representa más bien un agrupamiento probabilístico de los objetos.

3. Teoría de prototipos

La categorización clásica y el agrupamiento conceptual son suficientemente expresivos para utilizarse en la mayoría de las clasificaciones que se necesita en el diseño de sistemas de software complejos; sin embargo existen algunas situaciones en las que no resultan adecuadas. Existen algunas abstracciones que no tienen ni propiedades ni conceptos delimitados con claridad. Una categoría como un juego, no encajan en el molde clásico porque no hay propiedades comunes compartidas por todos los juegos. Aunque no hay una sola colección de propiedades que comparten todos los juegos, la categoría de los juegos está unida por *"parecidos familiares"*. Se puede observar que no hay fronteras fijas para la categoría juego. La categoría puede extenderse, pueden introducirse nuevos tipos de juegos siempre que se pareciesen a juegos anteriores de forma apropiada. Ésta es la razón por la que el enfoque se denomina *teoría de prototipos*: una clase de objetos se representa por un objeto prototípico y se considera un objeto como un miembro de esta clase si y solo si se parece a este prototipo de forma significativa. Por ejemplo considérense sillas de comedor con almohadón, sillones de peluquero, sillas plásticas, son sillas no porque compartan algún conjunto de propiedades definito-

rias con el prototipo, sino más bien porque mantienen suficiente parecido familiar con el prototipo. No hace falta que exista un núcleo fijo de propiedades de sillas prototípicas que compartan la silla de comedor y la de peluquero sino que ambas son sillas porque – cada una a su manera – está lo bastante cerca del prototipo. Las propiedades de interacción son sobresalientes entre los tipos de propiedades que cuentan a la hora de determinar si hay suficiente parecido familiar.

Aplicación de las teorías

Se identifican las clases y objetos en primer lugar de acuerdo con las propiedades relevantes para nuestro dominio particular. Aquí se hace hincapié en la identificación de las estructuras y comportamiento que son parte del vocabulario del domino del problema. Si este enfoque fracasa en la producción de una estructura de clases satisfactoria, hay que pasar a considerar la agrupación de objetos por conceptos. Aquí se concentra la atención en el comportamiento de objetos que colaboran. Si ambos intentos fallan al capturar nuestra comprensión del dominio del problema, entonces se considera la clasificación por asociación a través de la cual las agrupaciones de objetos se definen según el grado en el que cada una se parece a algún objeto prototípico.

6.2 DIAGRAMAS DEL UML

El UML documenta los sistemas de software que modela desde dos puntos de vista básicos: dinámico y estático. El modelado dinámico se refiere al comportamiento del sistema en tiempo de ejecución, en tanto que el modelado estático a la descripción de los componentes del sistema así como de sus relaciones. Esto no significa que los modelos deban estar separados en grupos diferenciados; lo que se debe hacer es presentar los sistemas como una colección de distintos puntos de vista (proporcionado por cada tipo de diagrama).

Los diagramas estructurales presentan elementos estáticos del modelo, tales como clases, paquetes o componentes; en tanto que los diagramas de comportamiento muestran la conducta en tiempo de ejecución del sistema, tanto visto como un todo como de las instancias u objetos que lo integran.

Hay que tener en cuenta que cada diagrama sirve para documentar un aspecto distinto del sistema; el criterio para usarlos es el de tener algo que decir, una historia sobre el sistema que se estudia y que debe ser contada; el tipo de diagrama que utilizará será el que dé mayor poder expresivo y la construcción de una serie de diagramas puede ser ordenada por un método de desarrollo utilizado. Algunos sistemas simples serán bien documentados con pocos diagramas, en tanto que algunos sistemas grandes bien podrían beneficiarse de un conjunto mayor. No se hacen diagramas porque un método lo exija sino que se hacen para contar algo importante del modelo, por lo que indicar que en un sistema haya que hacer tantos de tal o cual diagrama es una declaración sin fundamento alguno. Si bien existen varios tipos de diagramas para cada perspectiva, en la materia se verán solamente los siguientes:

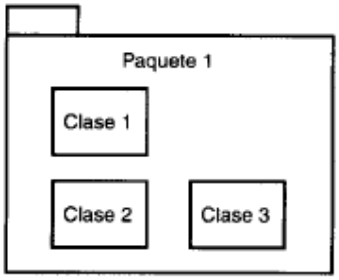
- Estáticos o estructurales:
    - Diagrama de clases

- Dinámicos o de comportamiento:
    - Diagrama de casos de uso
    - Diagrama de transición de estados

Elementos genéricos del UML

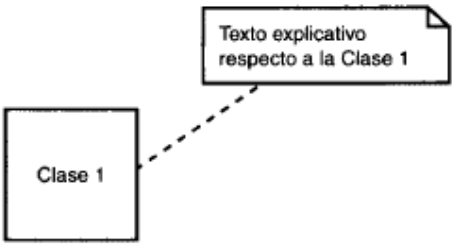
1. Los paquetes

Una de las características de UML es que comúnmente se requiere organizar los elementos de un diagrama dentro de un grupo. Tal vez se desee mostrar ciertas clases o componentes que son parte de un subsistema en particular. Para ello se agruparán en un ***Paquete*** el que se representa por una carpeta tabular como la siguiente:



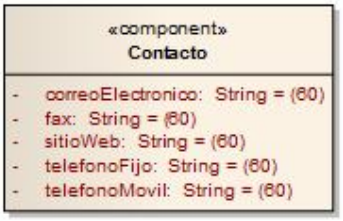
2. Las notas

Es frecuente también que en alguna parte del diagrama no se presenta una clara explicación del porqué algo está en ese lugar o la manera en que trabaja. En este caso se utiliza la ***nota*** UML. Es similar a un sticker pegado al gráfico donde se coloca la explicación. Se adjunta al elemento del diagrama conectándolo con una línea discontinua:



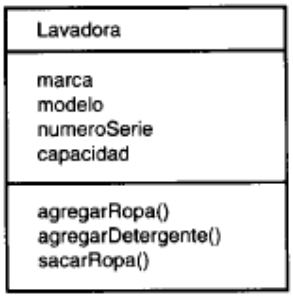
3. Los estereotipos

El UML brinda muchos elementos pero no un conjunto minucioso. A veces se requieren elemento hechos a medida. Los ***estereotipos*** permiten tomar elementos propios del UML y convertirlos en otros. Se representa como un nombre entre dos pares de paréntesis angulares. Su aplicación se verá posteriormente. Por ejemplo:



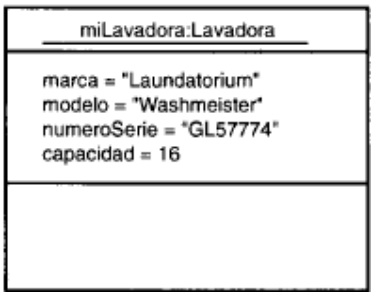
6.2.1 DIAGRAMAS DE CLASES

Supóngase la clase lavadora de la que se conoce la marca, el modelo, el número de serie y la capacidad junto con las acciones de agregar ropa, agregar detergente y sacar ropa. Seguramente también deberá tener un mecanismo para **fabricar** nuevas instancias a partir de ella, es decir, podrá crear nuevos objetos. La figura siguiente muestra la notación de UML que captura los atributos y acciones de una lavadora. Un rectángulo es el símbolo que representa a la clase y se divide en tres áreas. El área superior contiene el nombre, el área central los atributos y la inferior las acciones. Como convención, el nombre de la clase, como lavadora se escribe **Lavadora** y si constara de dos palabras se escribiría como **LavadoraIndustrial** y las características como número de serie se escribirán como **numeroSerie**. De igual manera se procede con las etiquetas de los métodos. De esta manera, la representación de la clase será:



El propósito de la orientación a objeto es desarrollar software que modele un esquema real del mundo. Entre más atributos y acciones se tomen en cuenta, mayor será la similitud del modelo con la realidad. Cabe entonces al diseñador, saber hasta dónde quiere llegar con la profundidad y que ésta sea relevante para la realidad que quiere modelar.

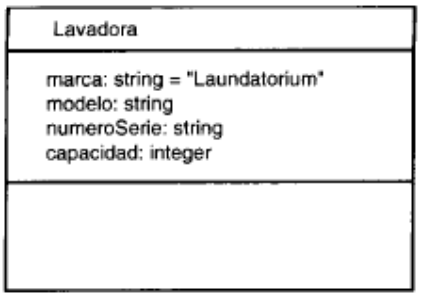
La instanciación de una clase es un objeto, entonces, considerando el ejemplo anterior:



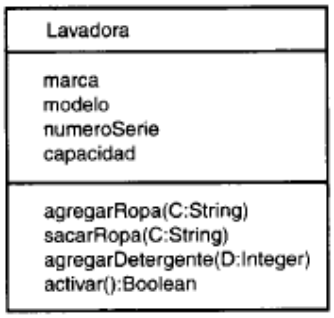
Obsérvese que el nombre de un objeto inicia con una letra minúscula. Se indica el nombre del objeto, dos puntos y luego el nombre de la clase a la que pertenece.

En las clases puede mostrarse además del tipo de dato de cada atributo, el valor por defecto que puede asignársele. Entre los posibles tipos se encuentran las cadenas (string), número de punto flotante (float), entero (integer), entero corto (short), y boolean, además de

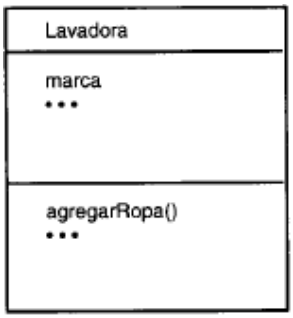
los tipos enumerados. Para indicar un tipo, se utilizan dos puntos para separar el nombre del atributo y su tipo. Por ejemplo:



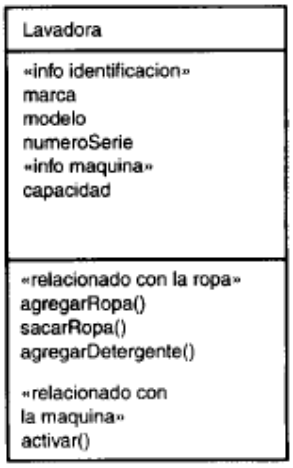
Así como es posible representar información adicional de los atributos, también se puede con las operaciones o métodos. En los paréntesis que preceden al nombre se puede mostrar el parámetro con el que funcionará la operación junto con su tipo de dato. La función devuelve un valor luego que finaliza su trabajo. En una función puede mostrarse el tipo de valor que regresará. Por ejemplo:



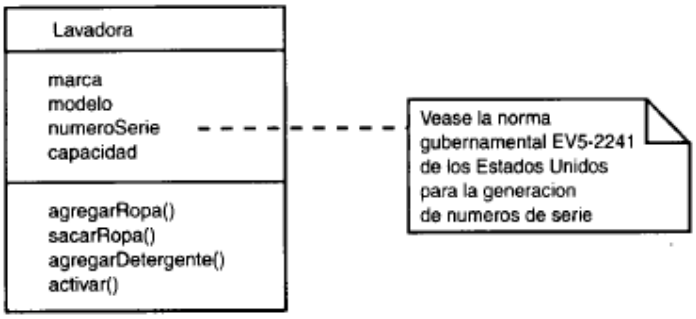
En los diagramas se mostrará más de una clase a la vez por lo que no resulta muy útil que siempre aparezcan todos los atributos y operaciones para no saturarlos demasiado. En lugar de ello se puede mostrar el nombre de la clase y dejar el área de atributos y/o de las operaciones vacías. En ocasiones es bueno mostrar algunos y en este caso seguirá a la lista de los mostrados un renglón con puntos suspensivos:



Si se tiene una larga lista de atributos u operaciones, se puede utilizar un estereotipo para organizarlas de forma que sea más comprensible. Un estereotipo es el modo en que el UML le permite extenderlo es decir, crear nuevos elemento que son específicos de un problema en particular que intente resolver. Los estereotipos se muestran con un nombre entre paréntesis angulares. Para una lista de atributos se puede utilizar un estereotipo como encabezado:



Pueden además agregarse mayor información en una clase mediante las notas adjuntas. Esta nota puede contener tanto una imagen como texto.



Construcción de los diagramas de clases

Las clases representan el vocabulario de un área de conocimiento. Las conversaciones con el cliente o un experto en el área dejarán entrever los sustantivos que se convertirán en clases (o atributos de una clase) en un modelo y los verbos se transformarán en operaciones. Se pueden utilizar diagramas de clase como una forma de estimular al cliente a que diga más respecto a su área y que ponga en evidencia cierta información adicional. Nótese que la manera en que se identifican las clases es similar a la utilizada para encontrar las entidades y sus atributos en el modelo conceptual de datos. Esto es válido para las asociaciones que se presentan entre las clases.

Sin las relaciones, un modelo de clases sería poco menos que una lista de cosas que representarían un vocabulario. Las relaciones muestran cómo se conectan los términos del vocabulario entre sí para dar una idea de lo que se está modelando. La asociación es la conexión conceptual fundamental entre clases. Cada clase en una asociación juega un papel y la multiplicidad muestra cuántos objetos de una clase se relacionan con un objeto de la clase asociada. Una asociación se representa como una línea entre los rectángulos de las clases con los roles y multiplicidades (cardinalidades). Una clase puede heredar atributos y operaciones de otra clase. La clase heredada es secundaria de la clase principal que es de la que se hereda. Se descubre la herencia cuando se encuentran clases en el modelo inicial que

tengan atributos y operaciones en común. Las clases abstractas sólo se proyectan como bases de herencia y no proporcionan objetos por sí mismas.

6.2.2 DIAGRAMAS DE CASOS DE USO

6.2.2.1 Conceptos

Las ideas estáticas ayudan a que el analista se comunique con el cliente, mientras que la dinámica ayuda a la relación con el grupo de desarrolladores. Tanto cliente como equipo de desarrollo es un conjunto importantísimo en el sistema, no obstante falta **el usuario**. Ni la idea estática ni la dinámica muestran el comportamiento del sistema desde el punto de vista del usuario siendo que es un punto clave. El modelado del sistema desde el punto de vista del usuario es el trabajo de los casos de uso. El caso de uso ayuda a los analistas a determinar la forma en que se utilizará un sistema. El caso de uso es como una colección de situaciones respecto al uso de un sistema. Cada escenario describe una secuencia de eventos. Cada secuencia se inicia por una persona, otro sistema, una parte del hardware o por el paso del tiempo. A tales entidades se las conoce como **Actores**. El resultado de la secuencia debe ser algo utilizable ya sea por el actor que la inició o por otro actor.

La visualización de los casos de uso permiten que el usuario al verlos pueda dar más información. Es un hecho que los usuarios saben más de los que dicen y este tipo de diagrama colabora en la comunicación. Además, la representación visual ayuda a combinar los diagramas de casos de uso con otros de otro tipo.

Para su representación gráfica, hay un actor que inicia el caso de uso y otro (posiblemente el mismo) que recibirá algo de valor de él. La representación gráfica es directa. Una elipse representa el caso de uso y en su interior se registra su nombre; una figura esquemática de una persona representa al actor y debajo de él su nombre. El actor que inicia se encuentra a la izquierda y el que recibe a la derecha. Una línea asociativa conecta a un actor con el caso de uso y representa la comunicación entre ellos. Uno de los beneficios del caso de uso es que muestra los confines entre el sistema y el mundo exterior. Generalmente los actores están fuera del sistema mientras que los casos de uso están dentro de él. Se utiliza un rectángulo con el nombre del sistema en algún lugar de su interior para representar las fronteras y este rectángulo tendrá en su interior los casos de uso definidos.

Suponga que se quiere hacer el diseño de la máquina expendedora de gaseosas. Para obtener el punto de vista del usuario final se entrevistarán usuarios potenciales respecto de la manera en que utilizarán la máquina. Cada entrevistado puede que le presente distintos escenarios para el mismo caso de uso. El caso de uso inicial será ***Comprar gaseosa***. El actor será un cliente que desea comprar una gaseosa. El escenario iniciará cuando el cliente inserte el dinero, posteriormente realizará una selección de cual gaseosa quiere y luego la máquina (suponiendo que tenga al menos una de las solicitadas) pondrá a alcance del cliente la gaseosa elegida. Además de la secuencia, hay otros aspectos del escenario que merecen

cierta consideración como la *precondiciones* y las *poscondiciones*. Para el ejemplo se tendrá: Precondición: *¿qué condiciones llevaron al cliente a iniciar el escenario en el caso de uso* ***Comprar gaseosa****?: La sed es la más obvia*. Poscondición: *¿qué se obtiene como resultado?: Cliente con la gaseosa en su poder*.

Cada caso de uso es una colección de escenarios y cada escenario es una secuencia de pasos. Estos pasos no aparecen en el diagrama. No se encuentran en notas adjuntas tampoco. Aunque nada lo prohíbe es mejor que los diagramas sean claros. Las secuencias de pasos se obtienen de documentación asociada al caso de uso que cliente y equipo de desarrollo tomarán como referencia. Cada caso de uso tiene su propio documento, de igual manera que cada escenario de caso de uso lo tendrá. La información que se requiere será:

- Actor que inicia el caso de uso
- Precondiciones
- Pasos en el escenario
- Poscondiciones
- Actor que se beneficia del caso de uso

También pueden enumerarse conjeturas del escenario (por ejemplo, que un cliente a la vez utilizará la máquina de gaseosas) y una breve descripción de una sola frase del escenario.

Supóngase ahora que la máquina no tenga gaseosas o bien que el cliente no tenga el importe exacto. Tales casos el diseño de la máquina deberá contemplarlos.

Para el caso en que la máquina se haya quedado sin gaseosa, existirá una ruta alternativa dentro del caso de uso ***Comprar gaseosa***. El cliente inicia el caso de uso al insertar dinero en la máquina y posteriormente hace una selección. La máquina no cuenta con ninguna lata de la gaseosa seleccionada por lo que mostrará un mensaje al cliente que indicará que no tiene de esa marca. Lo ideal podría ser que el mensaje le pida al cliente que haga otra selección. La máquina también debería dar la opción de devolver el dinero al cliente. La condición previa es un cliente con sed y el resultado es una lata de gaseosa o la devolución del dinero.

Para el caso que el cliente no tiene el importe exacto también existirá una ruta alternativa. El cliente inicia el caso de uso insertando el dinero y luego hace una selección. Se sume que la máquina tiene provisión de la marca elegida. En la máquina hay una reserva de moneda fraccionaria y devuelve la diferencia al despachar la lata. Si la máquina no cuenta con dinero para el vuelto, devolverá el dinero original y mostrará un mensaje que pida al usuario el importe exacto. La precondición sigue siendo la misma y la poscondición la lata de gaseosa y el cambio o bien el importe originalmente depositado.

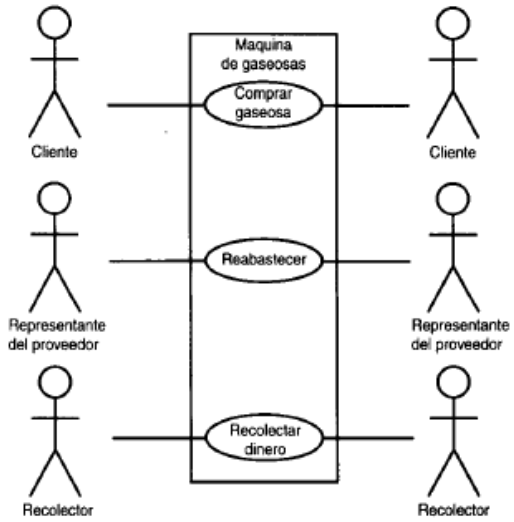
Existen otros usuarios que interactúan con la máquina y estos pueden ser el proveedor que tiene que reabastecer el stock en la máquina, el recolector de dinero, etc. Esto indica que hay que crear otros casos de uso para estos nuevos actores y que podrían ser ***Reabastecer*** y ***Recolectar dinero*** cuyos detalles surgirán durante las entrevistas con los proveedores y los recolectores.

Para el caso de uso ***Reabastecer*** el proveedor (o su representante) lo inicia por su decisión o bien porque habría pasado cierto lapso de tiempo de la última vez que lo hizo. El

representante del proveedor le quita el seguro a la máquina (mediante una llave y un cerrojo, pero eso entra dentro de la implementación), abre la puerta de la máquina y llena el compartimiento de cada marca hasta su capacidad. El representante también llena la reserva de moneda fraccionaria. Luego, cierra el frente de la máquina y vuelve a poner el seguro. La condición previa es la necesidad de reponer unidades y/o el intervalo de tiempo, y el resultado o poscondición es que el proveedor cuenta con nuevo conjunto de ventas potenciales.

Para el caso de ***Recolectar dinero*** el recolector inicia por su decisión o bien porque ha pasado cierto lapso de tiempo desde la última vez que hizo la última recolección. La persona deberá seguir la misma secuencia que en ***Reabastecer*** para abrir la máquina. El recolector sacará el dinero de la máquina y seguirá los pasos de ***Reabastecer*** para cerrar y poner el seguro a la máquina. La condición previa es la necesidad de retirar dinero y/o el intervalo de tiempo y el resultado es el dinero en las manos del recolector.

Considerando los casos de uso definidos y los actores asociados, el siguiente modelo representa tales integrantes:



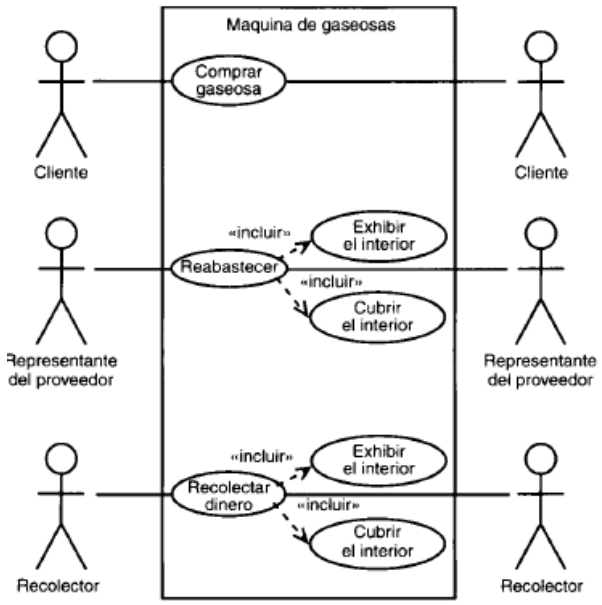
No se tienen en cuenta componentes internos de la máquina, mecanismos, controladores de cambio, etc. sino que lo que se intenta es ver la forma en que la máquina se comporta para con quien tenga que utilizarla.

El objetivo es obtener una colección de casos de uso que finalmente se mostrarán a las personas que diseñen las máquinas de gaseosas y a las personas que las construirán. Además, estos casos de uso reflejan lo que los clientes, recolectores y proveedores desean por lo que el resultado será una máquina que todos esos grupos puedan utilizar con facilidad.

6.2.2.2 Inclusión y extensión de los casos de uso

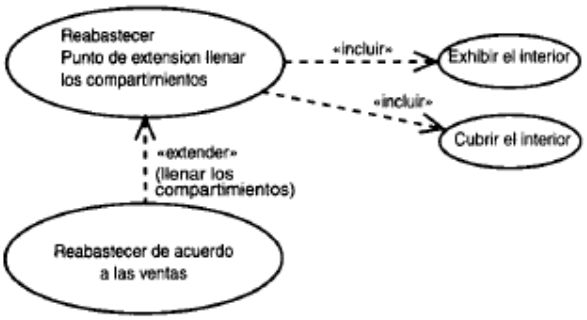
En los caso de uso ***Reabastecer*** y ***Recolectar dinero*** se observa que hay ciertos pasos en común. Ambos comienzan con abrir la máquina y finalizan con el cierre y su aseguramiento. En este caso se puede entonces eliminar esa duplicación de pasos para

ambos casos. La forma de hacerlo es tomar cada secuencia de pasos en común y construir un caso de uso adicional a partir de ellos. Se combinarán los pasos necesarios para *quitar el seguro* y *abrir la máquina* denominándolos ahora ***Exhibir el interior*** y los pasos *cerrar la máquina* y *asegurarla* en otro caso de uso llamado ***Cubrir el interior***. Con estos nuevos casos de uso, el caso de uso ***Reabastecer*** iniciaría con el caso de uso ***Exhibir el interior***. Luego el representante del proveedor seguiría los pasos ya indicados y concluiría con el caso de uso ***Cubrir el interior***. De forma similar, el caso de uso ***Recolectar dinero*** iniciaría con ***Exhibir el interior***, procedería como se indicó y finalizaría con el caso de uso ***Cubrir el interior***. Como se ve, ***Reabastecer*** y ***Recolectar dinero*** incluyen los nuevos casos de uso definidos.



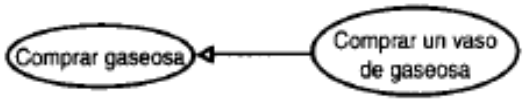
Es posible también utilizar un caso de uso de una forma distinta a la inclusión. En ocasiones se crea un caso de uso agregándole algunos pasos a un caso de uso existente. El caso de uso ***Reabastecer*** antes de colocar nuevas latas de gaseosas en la máquina, el representante del proveedor puede notar que hay marcas que se han vendido bien, así como que otras no lo han hecho. En lugar de solamente reabastecer todas las marcas, el representante podría sacar aquellas que no se han vendido bien y reemplazarlas por las que hay probado ser más populares. De esta forma tendría que indicar al frente de la máquina el nuevo surtido de marcas disponibles. Si se agregan estos pasos a ***Reabastecer***, se está en presencia de un nuevo caso de uso que podría llamarse ***Reabastecer de acuerdo a las ventas***. Este nuevo caso de uso es una extensión del original y se dice entonces que el nuevo caso de uso *extiende* al caso de uso base. La extensión sólo se puede realizar en puntos indicados de manera específica dentro de la secuencia del caso de uso base. A estos puntos se les conoce como puntos de extensión. En el caso de uso ***Reabastecer***, los nuevos pasos (anotar las ventas y abastecer de manera acorde) se darían luego que el representante haya abierto la máquina y esté listo para llenar los compartimientos de las marcas de gaseosas. En este ejemplo, el punto de extensión es *llenar los compartimientos.*

Para denotar la inclusión y la extensión se utilizan líneas de dependencia en donde se deberá mostrar el estereotipo **extender** o **incluir**.

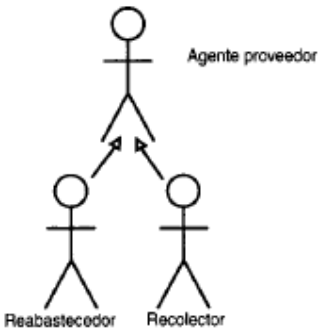


6.2.2.3 Generalización

Las clases puede heredarse entre sí, y esto también se aplica a los casos de uso. En la herencia de los casos de uso, el caso de uso secundario hereda las acciones y significado del primario, y además agrega sus propias acciones. Puede aplicar el caso de uso secundario en cualquier lugar donde se aplique el primario. En el ejemplo puede agregarse el caso de uso ***Comprar vaso de gaseosa*** que se puede heredar de ***Comprar gaseosa***. El caso de uso secundario tiene acciones como *obtener el vaso*, *agregar hielo* y *mezclar* además de las heredadas. La manera de representar la herencia es similar a lo expresado en las clases.



La relación de generalización también puede establecerse entre actores, así como entre casos de uso. Tanto el *Reabastecedor* como el *Recolector* son *Representantes del proveedor*; si se cambia el nombre de *Representante de proveedor* por ***Reabastecedor***, tanto éste como el ***Recolector*** serán secundarios:



6.2.2.4 Diagramas de caso de uso en el proceso de análisis

Las entrevistas al cliente deben iniciar el proceso. Estas entrevistas producirán diagramas de clases que serán como las bases del conocimiento para el dominio del sistema. Una vez que se conozca la terminología general del área del cliente se estará listo para hablar con los usuarios.

Las entrevistas con los usuarios comienzan en la terminología del dominio aunque deberán alternarse hacia la terminología de los usuarios. Los resultados iniciales de las

entrevistas deberán revelar a los actores y casos de uso de alto nivel que describirán los requerimientos funcionales en términos generales. Esta información establece las fronteras del sistema.

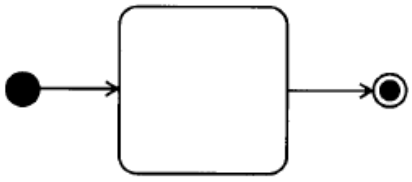
Las entrevistas posteriores con los usuarios profundizarán en estos requerimientos lo que dará por resultado modelos de casos de uso que mostrarán los escenarios y las secuencias detalladamente. Esto podría resultar en otros casos de uso que satisfagan las relaciones de inclusión y extensión. En esta fase, es importante haber logrado una gran comprensión del dominio ya que de lo contrario, podrían crearse demasiados casos de uso y demasiados detalles sin sentido.

6.2.3 DIAGRAMAS DE ESTADOS

Un diagrama de estados es una herramienta que permite caracterizar un cambio en un sistema o sea que los objetos que lo componen modificaron su ***estado*** como respuesta a los sucesos y al tiempo. Estos diagramas presentan los estados en los que puede encontrarse un objeto en un momento determinado junto con las transiciones entre estados, mostrándose además los puntos inicial y final de una secuencia de cambios de estado. También es conocido como **máquina de estados**.

Un diagrama de estados es totalmente distinto de uno de clase o de caso de uso. Los diagramas ya vistos modelan las clases, o comportamientos desde el punto de vista del usuario, mientras que un diagrama de estados **muestra las condiciones de un solo objeto**.

La simbología utilizada es la siguiente:



Se utiliza un rectángulo con los vértices redondeados para representar un estado, junto con una línea continua y una punta de flecha las que representan una transición. La punta de flecha apunta hacia el estado donde se hará la transición. La figura también muestra un círculo relleno que simboliza un punto inicial y la diana que representa un punto final.

Adicionalmente pueden agregarse detalles a la simbología. De la misma manera que se dividen las áreas de una clase (nombre, atributos y operaciones), puede dividirse el ícono de estado. En la parte superior irá el nombre, en el medio las variables de estado y la inferior las actividades.

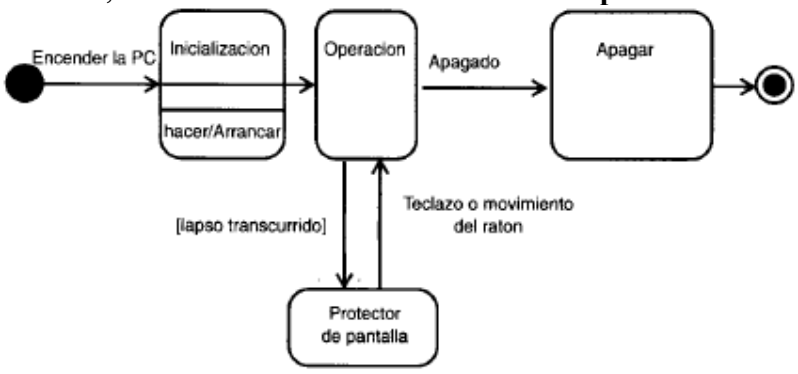
Las variables de estado no aportan demasiado. Las actividades constan de sucesos y acciones; las tres más utilizadas son ***entrada*** (qué sucede cuando el sistema entra al estado), ***salida*** (qué sucede cuando el sistema sale del estado) y ***hacer*** (qué sucede cuando el sistema está en el estado).

En el diagrama también se pueden agregar ciertos detalles a las líneas de transición. Puede indicar un suceso que provoque una transición y la acción que se ejecute. Los sucesos y las acciones se escriben cerca de la línea de transición separados por una diagonal (*suceso / acción*). La interfaz gráfica de usuario (GUI) con que se interactúa en una computadora, da ejemplos de detalles de la transición. Se asume en principio que la GUI puede tener tres estados:

- Inicialización
- Operación
- Apagar

Cuando se enciende el equipo (*Encender la PC*) se ejecuta un proceso de arranque y se desencadena un suceso que provoca que le GUI aparezca luego de una transición desde el estado de **Inicialización** y el arranque es una acción que se realiza durante tal transición. Como resultado de las actividades en el estado de inicialización la GUI entra al modo de **Operación**. Cuando se desea apagar la PC, se desencadena un suceso que provoca la transición hacia el estado de **Apagado** y con ello la PC se apaga.

Para completarlo se agregará que si se deja solo el equipo o si se realizara alguna actividad que no toque el ratón o el teclado aparecerá un protector de pantallas. Para decirlo en términos de cambio de estados sería que si ha pasado cierto tiempo sin que haya interacción con el usuario, la GUI hará una transición del estado **Operación** al uno nuevo.



El intervalo se especifica con algún panel de control del sistema operativo. Cualquier movimiento del ratón o presión sobre alguna tecla provocará la transición del estado **Protector de pantalla** al de **Operación**. Este intervalo constituye una *condición de seguridad* añadida.

Cuando la GUI está en estado de **Operación** muchas otras cosas ocurren aunque no sean evidentes en la pantalla. La GUI espera constantemente que alguien haga alguna

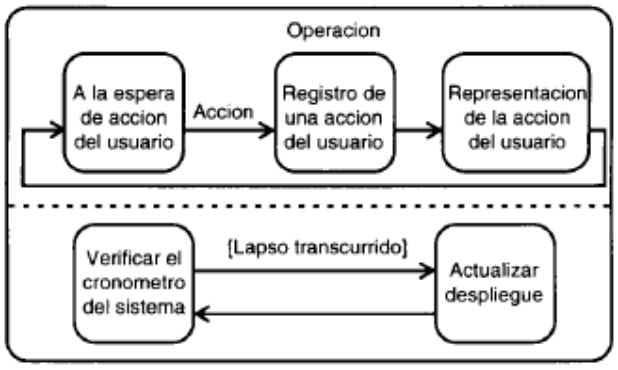
acción (presionar una tecla, accionar el ratón). Luego debe registrar esas acciones y modificar lo que despliega para reflejarlas en la pantalla (mover el puntero al mover el ratón o mostrar una letra **a** cuando se presiona la tecla **a**). De esta manera, la GUI atraviesa varios cambios de estado (subestados) mientras esté en **Operación**. Hay dos tipos de subestados: **secuenciales** y **concurrentes**.

Los **secuenciales** suceden uno detrás de otro, por ejemplo para el caso planteado, los subestados dentro del estado **Operación** podrían ser:

- A la espera de acción del usuario
- Registro de una acción del usuario
- Representación de la acción del usuario

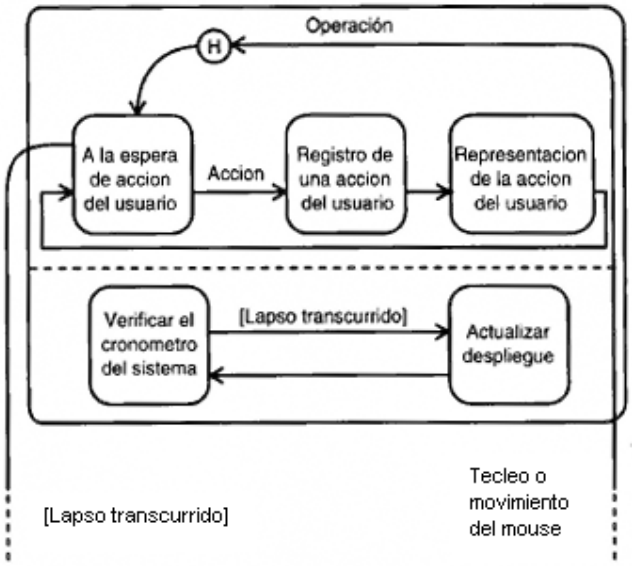
La acción del usuario desencadena la transición a partir de *A la espera de acción del usuario* hacia *Registro de una acción del usuario*. Las actividades dentro del *Registro* trascienden de la GUI hacia la *Representación de la acción del usuario*. Luego del tercer subestado, la GUI vuelve a iniciar *a la espera de acción del usuario*:

Los **concurrentes** suceden al mismo tiempo que los secuenciales. Para el ejemplo, la GUI no solamente aguarda a que se haga algo sino que también verifica el reloj del sistema y posiblemente actualice el despliegue de una aplicación luego de un intervalo específico (por ejemplo una aplicación que incluya un reloj en pantalla que tuviera que actualizar la GUI). Aunque cada secuencia es un conjunto de estados secuenciales, las dos secuencias son concurrentes entre sí.



Cuando se activa el protector de pantalla y luego se mueve el ratón para regresar al estado **Operación** la pantalla no debe retomar el estado inicial como si se hubiese encendido la PC, sino que tiene que mostrarse como se dejó antes que se activara el protector. Para ello, el diagrama de estados **histórico** captura esta idea. Existe un símbolo que muestra que un estado compuesto recuerda su subestado activo cuando el objeto trasciende fuera del

estado compuesto. El símbolo es la letra **H** encerrada en un círculo que se conecta por una línea continua con punta de flecha hacia el subestado a recordar.



Los diagramas de estados permiten a los desarrolladores comprender el comportamiento de los objetos de un sistema. Un diagrama de clases solamente muestra aspectos estáticos del sistema. Muestra jerarquías y asociaciones y le indican cuales son las operaciones pero no muestran ningún detalle dinámico de las operaciones. Los desarrolladores en particular, deben saber la forma en que los objetos se supone que se comportarán, ya que ellos son quienes tendrán que establecer tales comportamientos en el software. No es suficiente con implementar un objeto; los desarrolladores deben hacer que tal objeto haga algo. Los diagramas de estados se aseguran que no tendrán que adivinar lo que se supone qué harán los objetos. Con una clara representación del comportamiento del objeto, aumenta la probabilidad de que el equipo de desarrollo produzca un sistema que cumpla con los requerimientos.