Final POO

***Tema 2***

**Propiedades de los Objetos**Desde un punto de vista más formal, puede decirse que un objeto queda definido por las siguientes  
propiedades:  
• Estado (atributos)  
• Comportamiento (operaciones)  
• Identidad (propiedad que lo distingue de los demás objetos)  
**Atributos  
color: rojo  
velocidad: 200 km/h**

* El **estado** de un objeto define la situación del mismo en un momento específico en el tiempo. Los valores que toma un objeto se almacenan en variables internas llamadas **atributos**.
* El **comportamiento** describe las acciones y reacciones de un objeto. Es la manera en que responde a mensajes o estímulos recibidos. Las distintas formas de respuesta se denominan **métodos**, y toman la forma de procedimientos o funciones. Se dice que el comportamiento indica lo que el objeto *sabe hacer.*
* La **identidad** es lo que diferencia a un objeto de todos los otros del mismo tipo. Hablando en términos computacionales, la identidad del objeto se puede interpretar como la referencia, es decir el lugar específico en que se almacena el objeto. Dos objetos son distintos incluso aún en el caso de que los valores de todos sus atributos (p. ej. nombre y tamaño) coincidan. Dos manzanas pueden ser totalmente idénticas pero no por eso pierden su identidad: nos podemos comer una u otra.

**Los atributos y los métodos**

son las propiedades que dan estructura al objeto.

• **Atributos**: son las características que describen al objeto. En POO los atributos corresponden a las clásicas "variables" de la programación estructurada. Los atributos de un objeto pueden ser datos simples o puede ser una estructura más compleja de datos (matrices, vectores, listas, etc.). Pueden ser de tipo primitivo (entero, carácter, etc.), o pueden a su vez ser objetos. Sin embargo, existe una diferencia con las "variables", y es que los atributos se pueden heredar de unos objetos a otros. En consecuencia, un objeto puede tener un atributo de maneras diferentes:  
o *Atributo propio:* creado dentro de la cápsula del objeto.  
o *Atributo heredado:* definido en un objeto diferente, antepasado de éste (padre, abuelo, etc.).

También se los denomina atributos o variables miembro, porque el objeto las posee por el mero  
hecho de ser miembro de una clase.

• **Métodos:** operaciones de un objeto que realizan el acceso a sus propios datos, y los procesa para obtener los resultados esperados. Se puede definir un método como el código asociado a un objeto determinado, y cuya ejecución sólo puede desencadenarse a través de un mensaje recibido por éste. Los métodos especifican la forma en que actúa el objeto, y permiten manipular y controlar sus  
propios datos. No deben tener acceso directo a las estructuras de datos de otros objetos. Para utilizar la estructura de datos de otro objeto, deben enviar un mensaje a éste. Son sinónimos de 'método' todos aquellos términos que se han aplicado tradicionalmente a los programas, como procedimiento, función, rutina, etc. Sin embargo, es conveniente utilizar el término 'método' para que se distingan claramente las propiedades especiales que adquiere un programa en el entorno POO, que afectan fundamentalmente a la forma de invocarlo (únicamente a través de un mensaje) y a su campo de acción, limitado a un objeto y a sus descendientes.

**Objeto y encapsulación**

Un ***objeto*** es una unidad que combina datos (atributos) y las operaciones (métodos) que operan sobre esos datos. Los objetos *encapsulan* (agrupan y ocultan) sus operaciones y atributos.

***Encapsulación***es la propiedad del modelo de objetos que describe la vinculación entre un conjunto de  
operaciones y el estado a un objeto particular. Está íntimamente relacionada con la ocultación de la información, definiendo qué partes de un objeto son visibles y qué partes están ocultas.

**Interfaz pública**La encapsulación incluye la ocultación de la información:  
• Algunas partes son visibles (interfaz pública)  
• Otras partes son ocultas (privadas)

La ***parte pública*** o interfaz de una clase describe qué es lo que pueden hacer los objetos de esa clase.

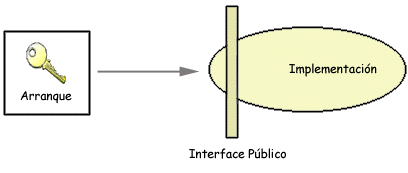
Las ***partes privadas*** de una clase describen la implementación de los métodos, es decir, **cómo** hacen lo que saben hacer.  
En sistemas orientados a objeto puros, todo el estado es privado y sólo se puede acceder a través de operaciones de la interfaz pública. Por ejemplo, un método *frenar* puede cambiar el estado del atributo *velocidad*.

La ventaja de la ocultación de los detalles de implementación es que el objeto puede cambiar  
internamente, y la interfaz pública proporcionada continúa siendo compatible con el original. Como consecuencia, los programas que utilizaban el objeto pueden seguir funcionando sin alteración alguna.  
Esto es extremadamente útil al modificar código, ya que se restringe la propagación de cambios. También se fomenta la reusabilidad, ya que el código puede ser utilizado como una tecnología de caja negra (como los circuitos integrados en la industria electrónica). Desde el punto de vista económico, esto representa una ventaja de indudable valor.

**Interacción entre Objetos: mensajes**Los objetos contribuyen al comportamiento del sistema colaborando con otros objetos. Por ejemplo:  
• El volante interacciona con el objeto intermitente  
• El objeto acelerador interacciona con el objeto motor

El modelado de objetos no sólo modela los objetos en un sistema, sino también sus interrelaciones. Para realizar su tarea, un objeto puede delegar trabajos en otro. Este otro puede ser parte integrante de él o ser cualquier objeto del sistema.

Los objetos interaccionan enviándose mensajes unos a otros. El modo de envío de mensajes depende de la naturaleza de los objetos modelados.

  
Los mensajes son tratados por los métodos de la interfaz pública del objeto que los recibe. Debido a las normas de encapsulación y de ocultación de la información, no hay otro modo de influir o de comunicarse con un objeto. Tras la recepción de un mensaje el objeto actuará. La acción puede ser el envío de otros mensajes, cambiar el estado, o hacer cualquier otra cosa apropiada para el objeto en su estado.

Para que un objeto haga algo, se le envía una petición. Esta hace que se active el método apropiado y se ejecute, llevando a cabo la tarea prevista.  
El mensaje que constituye la petición contiene el nombre del objeto, el nombre de una operación y, opcionalmente, un grupo de argumentos.

Los objetos pueden ser muy complejos, puesto que pueden contener muchos otros objetos, y éstos a su vez pueden contener otros, etc. Sin embargo, gracias a la propiedad de encapsulación, la persona que utilice un objeto no necesita conocer su complejidad interna, sino sólo la forma de comunicarse con él (a través de mensajes) y la forma en que responde (retorno).

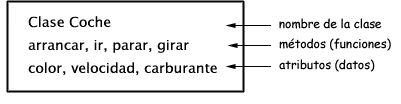
**Concepto de Clase**

Una clase es una descripción de un conjunto de objetos (o tipo de objeto) con características comunes. Es una abstracción que describe un grupo de instancias con propiedades (atributos) comunes, comportamiento (operaciones) común, relaciones comunes con otros objetos y (lo que es más importante) una semántica común.

La definición de una clase no crea ningún objeto. Las clases son sólo modelos o plantillas que describen cómo se construyen los objetos.  
También se puede ver una clase como un molde, esquema, o patrón que define la forma de sus objetos. Es una estructura estática que define qué estado y comportamiento van a tener los objetos y, a partir de ese esquema, dinámicamente durante la ejecución de un programa, se van a ir creando objetos que pertenezcan  
a esa clase. Las Clases son entidades conceptuales que sirven para abstraer y modelar un Sistema. Los problemas se pueden modelar en los términos específicos del dominio. Precisamente, una de las ventajas de la orientación  
a objetos es que propicia que el problema sea representado en términos específicos del dominio. Esto permite a los diseñadores construir modelos que se proyectan directamente en el problema de la aplicación.  
Ejemplos:  
***Sistema Administración de hotel***• Clase Conserje  
• Clase Habitación  
• Clase Cliente  
• Clase Botones  
• Clase Mucama

En cada sistema, se modelan los objetos que interactúan en él. La ventaja de crear modelos cercanos al dominio es que son fáciles de comprender, hay menor probabilidad de cometer errores y descuidos, y los modelos pueden ser verificados por los usuarios que definen los requisitos.

**Propiedades de una clase: atributos y métodos**

Una clase define los atributos y  
comportamiento de un grupo de objetos  
con características similares

**Atributos**: datos internos de los  
objetos pertenecientes a una clase,  
sobre los cuales se realizan las  
operaciones.

Un atributo en una clase representa un dato contenido en cada una de las instancias de la clase. Cada atributo toma un valor particular en cada uno de los objetos que se crean a partir de dicha clase, conformando el “estado” del objeto. Distintas clases pueden tener atributos con el mismo nombre (p. ej. nombre, en las clases Persona y Calle) pero cada atributo debe ser

único dentro de una clase. También pueden existir atributos propios de la clase, que son aquellos que tienen un único valor para toda la clase, es decir, el mismo valor para cada uno de los objetos creados a partir de esa clase. Generalmente se los llama **atributos estáticos**. En Java se implementa anteponiéndoles la palabra *static*.

**Métodos:** también conocidos como operaciones miembro. Son los procedimientos que determinan el comportamiento de los objetos de la clase. Con el mismo criterio que para atributos, existen los métodos de clase, a los que se llama **métodos estáticos**. En Java se implementa anteponiéndoles la palabra *static*.

**Instancias de una clase**

Se pueden crear muchos objetos de la misma clase. Cada vez que se construye un objeto a partir de una clase, se crea lo que se llama una **instancia** de esa clase: Objeto = instancia de una clase.

El proceso de instanciación consiste en crear un nuevo objeto (concreto) a partir de un molde o especificación de referencia (abstracto, conceptual). La clase es justamente quién juega este papel de molde para crear el nuevo objeto o instancia.Podemos ver a la clase, en este sentido, como a una fábrica de objetos.

Todos los objetos creados mediante este proceso de instanciación a partir de una misma clase (molde), tendrán las mismas características estructurales y entenderán el mismo protocolo. La diferencia entre instancia y clase está en el grado de abstracción:  
• un objeto es una abstracción de un objeto del mundo real  
• una clase es una abstracción **de un grupo de objetos** del mundo real.

La abstracción a nivel de clase permite la generalización y evita la redefinición de las características (atributos, comportamiento o relaciones) comunes, de forma que se produce una reutilización de estas

definiciones comunes por parte de cada uno de los objetos. Por ejemplo todas las elipses (instancias) comparten las mismas operaciones para dibujarlas o calcular su área.  
Se dice que un objeto **pertenece** a una clase. Existe una diferencia entre los atributos y los métodos en relación a los objetos pertenecientes a una clase:  
• hay una copia de los atributos en  
cada instancia  
• hay una **única** copia de los métodos, que es compartida por todas las instancias, que se encuentra en la clase que los agrupa.

**Encapsulamiento y Visibilidad**Un objeto sólo muestra lo necesario para poder interactuar con otros objetos. La interfaz de cada componente se define de forma que revele tan poco como sea posible de sus particularidades interiores. El encapsulamiento u ocultación de la información se implementa en los lenguajes de POO a través de la visibilidad o acceso a las variables miembro o atributos y a los métodos. La visibilidad es precisamente la propiedad que define si un atributo u operación de un objeto es accesible desde fuera del propio objeto. Los lenguajes de programación generalmente proporcionan tres niveles de visibilidad: public, protected  
y private.

Las características públicas (public) son accesibles a cualquier usuario de la clase. Constituyen la interfaz de la clase. Pueden ser tanto métodos como atributos. Sin embargo, no es recomendable que los atributos sean públicos. Para respetar los principios del paradigma de objetos, y aprovechar sus beneficios, se deben ocultar (encapsular) todos sus atributos, **y proveer los métodos de acceso apropiados**.

Las características protegidas (protected) están ocultas al mundo exterior pero son accesibles a la clase que los contiene y a cualquier clase derivada de la misma. Es un status mixto entre public private.

Las características privadas (private) sólo son accesibles desde los métodos de la propia clase. Esto es lo que se conoce como encapsulamiento. Es el ocultamiento de información de un objeto hacia los demás. El objeto esconde sus datos de los demás objetos y sólo permite el acceso a los mismos mediante sus propios métodos. La gran ventaja del encapsulamiento consiste en que si un módulo u objeto cambia internamente sin modificar su interfaz, el cambio no desencadenará ninguna otra modificación en el sistema.

En los lenguajes OO la encapsulación garantiza que los usuarios de un objeto sólo pueden interactuar con él y modificarlo a través de su interface (conjunto de métodos públicos).  
El encapsulamiento evita la corrupción de los datos de un objeto. Si los objetos pudieran ser accedidos de cualquier forma, los datos se podrían corromper o ser utilizados de mala manera. El encapsulamiento protege los datos del uso arbitrario y no pretendido.  
El encapsulamiento también oculta los detalles de la implementación de sus métodos. Los usuarios de un objeto conocen las operaciones que pueden solicitar de un objeto, pero desconocen los detalles de cómo se lleva a cabo la operación. Es decir, desconocen la implementación. El encapsulamiento al separar el comportamiento del objeto de su implementación, permite la modificación de ésta sin que se tengan que modificar las aplicaciones que lo utilizan.

**Métodos Constructores**Los objetos son entidades que existen en el tiempo; por ello, deben ser creados o instanciados. Un objeto (entidad real) se crea a partir de la definición de la clase (molde) a la que pertenece. Esta operación se hace a través de métodos especiales llamados constructores o inicializadores. Pueden existir varios constructores (varias formas diferentes de crear objetos a partir de una clase). En java los constructores son métodos especiales que deben tener siempre el mismo nombre de la clase, y no tienen ningún tipo de retorno.

La instanciación la ejecutará implícitamente el compilador o explícitamente el programador, mediante la invocación a los constructores. Cuando se crea una instancia de una clase, el constructor de la misma es llamado automáticamente. El propósito de un constructor es inicializar los atributos del objeto de la clase cuando se crea el objeto. Se le asigna un espacio de memoria y un estado consistente.

***Tema 3***

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

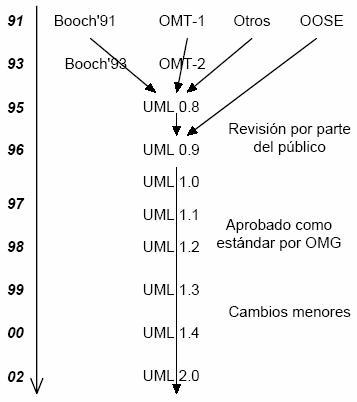
A la hora de desarrollar software industrial, para conseguir un producto de calidad, es completamente necesario seguir ciertas pautas y no abordar los problemas de manera superficial, con el fin de obtener un modelo que represente lo suficientemente bien el problema que se abordará.

**Modelado**El modelado es la espina dorsal del desarrollo de software de calidad. **El modelado es la construcción de un modelo a partir de una especificación.**Se construyen modelos para:  
• explicar el comportamiento del sistema a desarrollar  
• la mejor comprensión del sistema por parte de los mismos desarrolladores  
• facilitar la comunicación entre los participantes del proyecto  
• facilitar la comunicación entre usuarios y desarrolladores  
• controlar el riesgo  
• atacar problemas que, sin el modelado de su resolución seria imposible, tanto desde el punto de vista de los desarrolladores (no se pueden cumplir los plazos estimados, no se consigue ajustar los presupuestos...) como desde el punto de vista del cliente, el cual, cuando finalmente se le entrega el producto del desarrollo, generalmente se encuentra con infinidad de problemas, desde que no se cumplen las especificaciones hasta fallos que dejan inutilizado el sistema.

Al hacer referencia al desarrollo software en el ámbito industrial, se pretende que la capacidad de modelar no se reserve sólo a empresas que disponen de gran número de empleados o empresas que abordan proyectos de gran volumen. Lo que se pretende es la capacidad de obtener un producto comercial, sea cual fuere su costo o tamaño, que cumpla lo que en la industria se denomina **calidad total1** y que reporte beneficios a corto plazo, evitando excesivos períodos de desarrollo debido a la falta de previsión o por haber abordado los problemas muy a la ligera, sin planificación.

**Modelo: definiciones**• Abstracción de la realidad; representación simplificada de algún objeto o fenómeno del mundo real  
• Representación esquemática o conceptual de un fenómeno, que representa una teoría o hipótesis de  
cómo funciona dicho fenómeno  
• Descripción o representación utilizada para ayudar a visualizar algo que no puede ser observado  
directamente. Una representación abstracta de un objeto o sistema, desde un punto de vista particular.  
• Abstracción de algo, que se elabora para comprender ese algo antes de construirlo. El modelo omite detalles que no resultan esenciales para la comprensión del original y por lo tanto facilita dicha comprensión. Es un boceto o una guía de ese algo que vamos a construir para que a la hora de elaborarlo no lo hagamos erróneamente. Es útil también para mostrar a los interesados una aproximación de lo que se construirá. Esto les permite opinar acerca de lo que se modela, y también permite modificarlo si es necesario.

El modelo de un fenómeno es una herramienta que se usa para describirlo, interpretarlo, predecir comportamiento en diferentes situaciones específicas, validar hipótesis y elaborar estrategias para tratarlo. Los modelos son utilizados en muchas actividades de la vida humana: antes de construir una casa el ingeniero realiza un plano, el arquitecto utiliza una maqueta, los músicos representan la música en forma de notas musicales, los artistas pintan sobre el lienzo con carboncillos antes de empezar a utilizar los óleos, etc. Unos y otros abstraen una realidad compleja sobre unos bocetos. Todos ellos son “modelos”.

**Lenguaje de modelado**En la industria del software se ha comprobado que un modelado orientado a objetos proporciona arquitecturas más flexibles y readaptables que otros, por ejemplo orientados a la funcionalidad o a los datos. Desde la aparición de esta tecnología, un gran número de metodólogos hicieron esfuerzos para obtener una notación para representar el modelado. Muchos usuarios de estos métodos tenían problemas al intentar encontrar un lenguaje de modelado que cubriera sus necesidades completamente, alimentando de esta forma  
la llamada *guerra de métodos*. Aprendiendo de estas experiencias, comenzaron a aparecer nuevas generaciones de métodos, entre los que se destacaron unos pocos de manera muy clara. Entre ellos:  
- OOD (Object Oriented Design, Diseño Orientado a Objetos) de Booch  
- OOSE (Object-Oriened Software Engineering, Ingeniería de Software Orientada a Objetos) de Jacobson  
- OMT (Object Modeling Technique – Técnica de Modelado de Objetos) de Rumbaugh  
Cada uno de éstos era un método completo, auque todos tenían sus puntos fuertes y debilidades.  
En la primera mitad de los ‘90, Grady Booch (Rational Software Corporation), Ivan Jacobson (Objectory) y James Rumbaugh (General Electric), empezaron a adoptar ideas de los otros dos métodos, los cuales habían sido reconocidos en conjunto como los tres principales métodos OO a nivel mundial. Estos autores, conocidos como “los tres amigos”, comenzaron a trabajar en forma conjunta para crear un lenguaje unificado de modelado, intentando que la industria software termine su maduración como *Ingeniería.* Es así como en 1996 se publica la primera versión de UML (Unified Modeling Lenguaje, Lenguaje de Modelado Unificado), que luego de varias modificaciones, fue estandarizado por el OMG.

**Lenguaje de Modelado Unificado - UML**La aparición de UML significó un gran avance en la ingeniería del software, ya que proporciona las herramientas necesarias para poder obtener los *planos del software*, equivalentes a los que se utilizan en la construcción, la mecánica o la industria aeroespacial.

El UML es una técnica de modelado de objetos y como tal supone una abstracción de un sistema para llegar a construirlo en términos concretos. El modelado no es más que la construcción de un modelo a partir de una especificación.

Es un lenguaje estándar, orientado a objetos, que permite visualizar, especificar, construir y documentar todos los artefactos que componen un sistema con gran cantidad de software.

Abarca todas las fases del ciclo de vida de un proyecto, soporta diferentes maneras de visualización dependiendo de quién tenga que interpretar los planos y en qué fase del proyecto se encuentre.

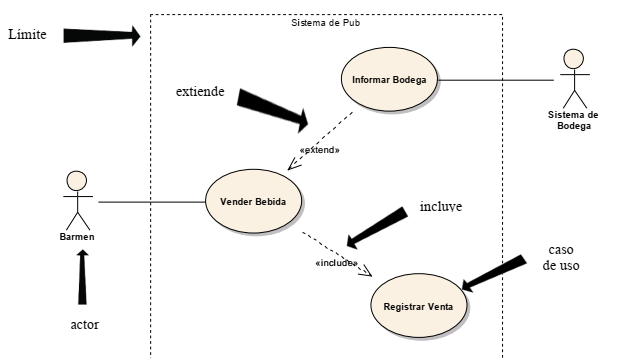
Es importante destacar que UML es solamente un "lenguaje" para especificar y no un método o un proceso. Es el lenguaje en el que está descrito el modelo. UML se puede usar en una gran variedad de formas para soportar una metodología de desarrollo de software (tal como el Proceso Unificado de Rational), pero no especifica en sí mismo qué metodología o proceso usar. Sin embargo, para que el proceso sea óptimo, debe usarse en un proceso dirigido por casos de uso, centrado en la arquitectura, iterativo e incremental.

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

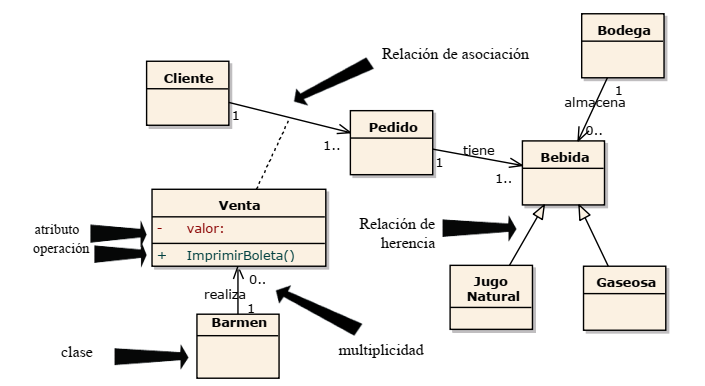
Este lenguaje también intenta solucionar el problema de propiedad de código que se da con los  
desarrolladores: al implementar un lenguaje de modelado común para todos los desarrollos se crea una documentación, también común, que cualquier desarrollador con conocimientos de UML será capaz de entender, independientemente del lenguaje utilizado para el desarrollo  
Ayuda a interpretar grandes sistemas mediante gráficos o mediante texto, obteniendo modelos  
explícitos que ayudan a la comunicación durante el desarrollo ya que al ser estándar, los modelos podrán ser interpretados por personas que no participaron en su diseño (e incluso por herramientas) sin ninguna ambigüedad. Debido a su estandarización y su definición completa no ambigua, y aunque no sea un lenguaje de programación, UML se puede conectar de manera directa a lenguajes de programación como Java, C++ o Visual Basic. Esta correspondencia permite lo que se denomina *ingeniería directa* (obtener el código fuente partiendo de los modelos) pero además es posible reconstruir un modelo en UML partiendo de la implementación, o sea, la *ingeniería inversa*. UML brinda la capacidad de modelar actividades de planificación de proyectos y de sus versiones, expresar requisitos y las pruebas sobre el sistema, representar todos sus detalles así como la propia arquitectura. Mediante estas capacidades se obtiene una documentación que es valida durante todo el ciclo de vida de un proyecto. UML proporciona distintos puntos de vista de la realidad que modela mediante los distintos tipos de diagramas que posee. Un diagrama es una representación gráfica de una colección de elementos del modelo, que habitualmente toma forma de grafo donde los arcos que conectan sus vértices son las relaciones entre los objetos y los vértices se corresponden con los elementos del modelo. Los distintos puntos de vista de un sistema real que se quieren representar para obtener el modelo se dibujan de forma que se resaltan los detalles necesarios para entender el sistema. Los diagramas muestran diferentes aspectos de las entidades representadas. Están agrupados en dos tipos diferentes de diagramas: los que dan una vista estática del sistema y los que dan una visión dinámica.

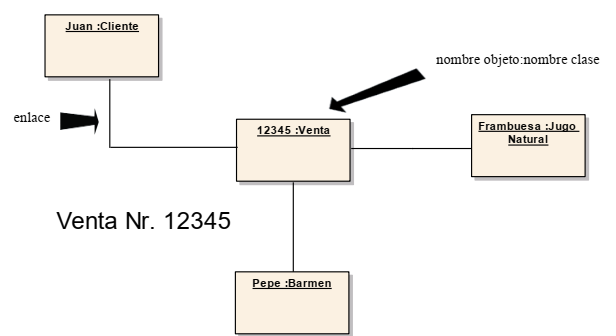
**Los diagramas estáticos son:**• Diagrama de casos de uso  
• Diagrama de clases  
• Diagrama de objetos.  
• Diagrama de componentes  
• Diagrama de despliegue  
**Lo diagramas dinámicos son:**• Diagrama de secuencia  
• Diagrama de colaboración  
• Diagrama de estados  
• Diagrama de actividades

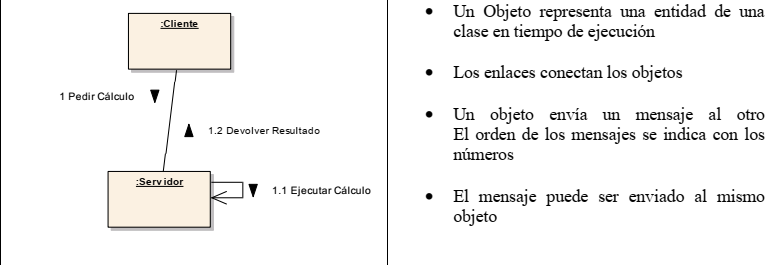
**1. Diagramas de Casos de Uso**Muestra *quién* puede hacer *qué* y las *relaciones* existentes entre acciones (casos de uso).  
Usados Para Comunicarse con el Usuario Final y el Experto de Dominio  
• Proporciona credibilidad en una etapa inicial del desarrollo del sistema  
• Asegura una comprensión mutua de los requisitos  
Usados Para Identificar  
• Quién interactuará con el sistema y qué deberá hacer el sistema  
• Qué interfaz deberá tener el sistema  
Usados Para Verificar  
• Que se hayan capturado todos los requerimientos  
• Que los desarrolladores hayan entendido los requerimientos



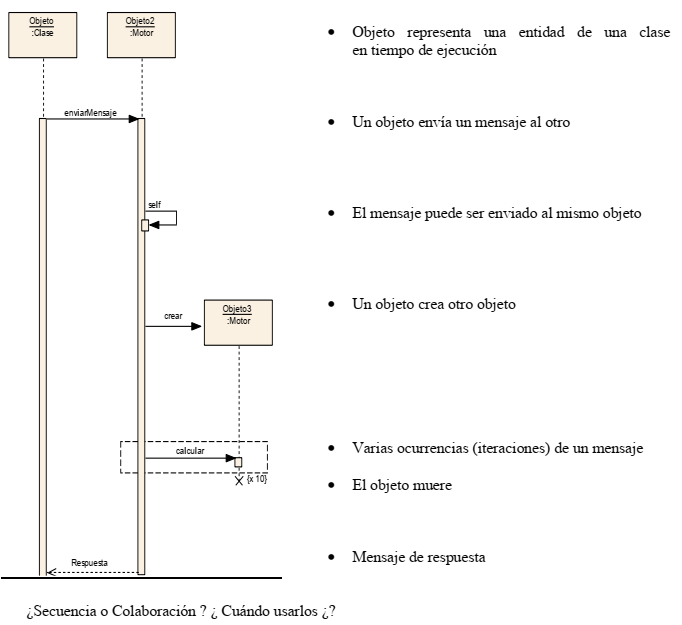
**2. Diagramas de Clases**• Usados para mostrar la Estructura Estática de un sistema computacional o una parte relevante del  
mundo real. Dan una vista estática del proyecto.  
• Son los diagramas más frecuentemente usados. Se los puede considerar con distintas perspectivas:  
– Conceptual: muestra las entidades del mundo real con sus relaciones  
– Implementación: el diseño para el código fuente. Representa la visibilidad de atributos y  
métodos: público (+), privado (-), protegido (#)

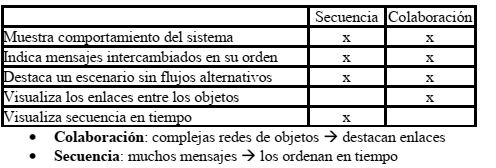


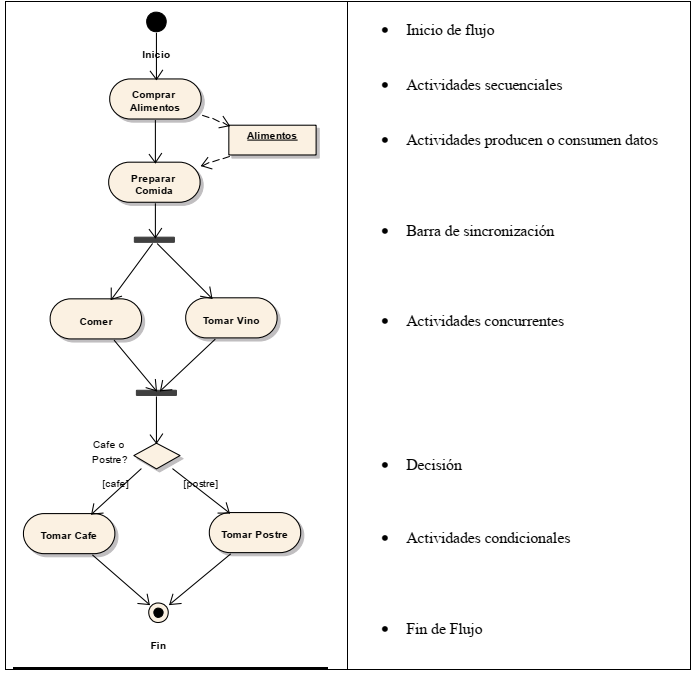
**3. Diagramas de Objetos**Es un diagrama de instancias de las clases mostradas en el diagrama de clases. Muestra las instancias y  
como se relacionan entre ellas. Se da una visión de casos reales.  
• Usados para mostrar la estructura de objetos en tiempo de ejecución del sistema  
• Representan vistas instantaneas (*snapshot*) de una parte del sistema de interés  
• Destacan relaciones entre objetos  
• Útiles para análisis y diseño preliminar e identificación de clases  
• Usados para validar los modelos de clases

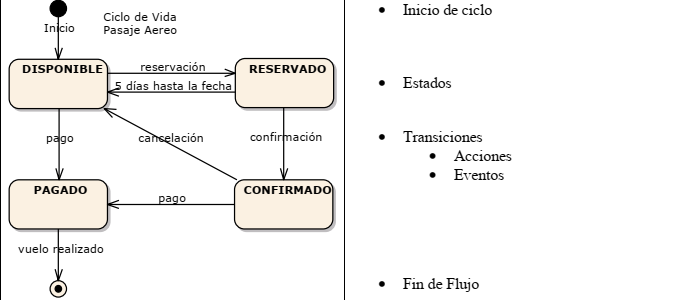
**4. Diagramas de Colaboración**• Usados para representar el comportamiento del sistema  
• Muestran colaboración entre los objetos del sistema  
• Destacan:  
– Mensajes enviados entre los objetos  
– Enlaces entre los objetos  
– Un escenario concreto, sin condiciones  
• Útiles tanto en análisis (identificación de clases), como en diseño (especificación de componentes)

**5. Diagramas de Secuencia**Muestran a los diferentes objetos y las relaciones que pueden tener entre ellos, a través de los mensajes  
que se envían entre ellos. Junto con el de colaboración, constituyen los diagramas de interacción. Son  
dos diagramas diferentes, que se puede pasar de uno a otro sin perdida de información, pero que dan  
puntos de vista diferentes del sistema.  
• Usados para representar el comportamiento del sistema  
• Muestran colaboración a través de mensajes entre los objetos del sistema  
• Destacan:  
o Mensajes enviados entre los objetos  
o Orden secuencial entre los mensajes  
o Un escenario concreto, sin condiciones  
• Útiles tanto en análisis (identificación de clases), como en diseño (especificación de componentes)

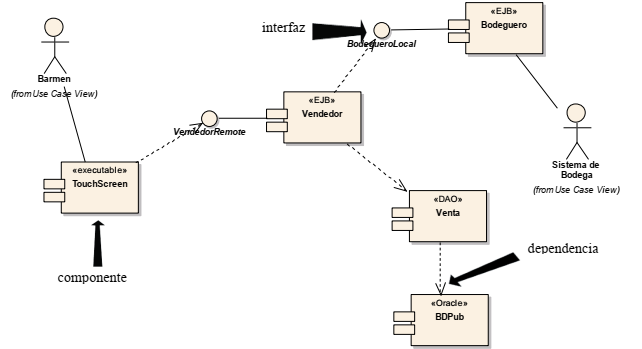


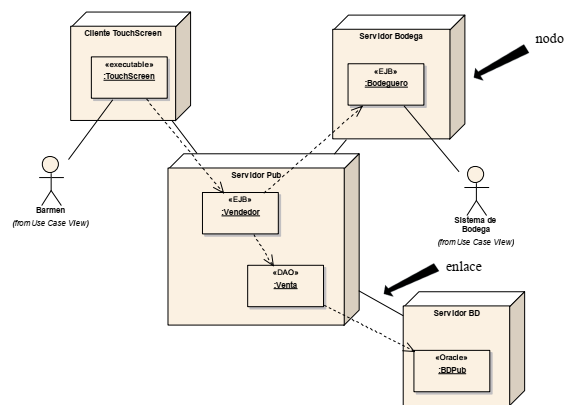


**6. Diagramas de Actividades**Es un caso especial del diagrama de estados. Muestra el flujo entre los objetos. Se utilizan para  
modelar el funcionamiento del sistema y el flujo de control entre objetos.  
• Usados para representar el comportamiento del sistema o negocio  
• Muestran actividades y procesos  
• Destacan:  
o Condiciones y flujos alternativos  
o Tareas y procesos concurentes  
o Responsabilidades sobre ciertas actividades  
• Útiles en análisis de negocio para capturar procesos de alto nivel

**7. Diagramas de Estados**Muestran los estados, eventos, transiciones y actividades de los diferentes objetos. Son útiles en  
sistemas que reaccionen a eventos**.**• Usados para representar el comportamiento INTERNO de un objeto de un módulo del sistema  
• Muestran estados en los cuales un objeto se puede encontrar  
• Destacan:  
o Estados  
o Transiciones y condiciones de las transiciones  
o Actividades realizadas  
• Típicamente usados para describer ciclo de vida de un objeto

**8. Diagramas de Componentes**Muestran la organización de los componentes del sistema. Un componente se corresponde con una o varias clases, interfaces o colaboraciones.



**9. Diagramas de *Deployment* (despliegue) o implementación**Muestra los nodos y sus relaciones. Un nodo es un conjunto de componentes. Se utiliza para reducir la  
complejidad de los diagramas de clases y componentes de un gran sistema. Sirve como resumen e  
índice.  
Usados Para Modelar las Relaciones entre el *Software* y el *Hardware*• Mapeo de los Componentes de Software a los Nodos de Hardware  
• Típicamente contienen elementos tales como: Servidores, Procesadores, Impresoras, Redes  
computacionales, etc. 

**Documentación Interna**Un programa no solo debe ser entendido por la máquina. Frecuentemente el código es leído por muchas personas para repararlo, ampliarlo o, simplemente, para evaluarlo. Por lo tanto, es fundamental que el programa esté bien redactado, con estilo, para que su significado sea claro y comprensible.  
Se logran programas comprensibles  
• Aumentando la legibilidad del código  
• Aplicando técnicas sistemáticas de programación  
• Documentando adecuadamente los programas

Las convenciones de código son la mejor manera de asegurar la legibilidad de los programas. Es  
altamente recomendable atenerse a estándares y convenciones. Existe gran variedad de convenciones, y  
todas son buenas, siempre que sean respetadas.  
Particularmente Sun ofrece para Java una serie de reglas, las “Code Conventions for the Java  
Programming Language”, a las que se puede acceder en **http://java.sun.com/docs/codeconv/.**Un buen estilo de codificación facilita la corrección y la evolución del código, es decir, favorece una de  
las reconocidas cualidades del software, la mantenibilidad (reparabilidad-extensibilidad).  
Para determinar si un programa está bien escrito se evalúa su legibilidad o **capacidad de comunicar lo  
que hace** a otros programadores que tengan que leer el código fuente.  
Un programador dispone de cuatro mecanismos básicos para comunicarse con sus lectores:  
1. El estilo de programación: un orden sistemático facilita la búsqueda al lector  
2. Los comentarios  
3. La nomenclatura: los nombres de las variables, constantes y métodos  
4. La identación: los espacios en blanco

**1. El estilo de programación - Reglas de buen estilo**

|  |
| --- |
| **El código fuente debe organizarse de la siguiente forma:** 1. documentación (javadoc) de la clase o interfaz 2. *class* o *interface* 3. variables de clase (*static*) 4. variables de instancia u objeto 5. constructores 6. accessors 7. demás métodos |
| Un orden sistemático facilita la búsqueda al lector. |

**Sobre las variables**• Las variables de clase u objeto no deben ser públicas → para que nadie pueda alterarlas desde fuera de la clase.  
• Las variables deben ver reducida su existencia al mínimo tiempo posible, es decir, declararse en el ámbito más reducido que sea posible → para que nadie se confunda en su uso.  
• Se preferirán variables de método a variables de objeto.  
• Las variables deben inicializarse inmediatamente → para evitar que la variable contenga un valor fuera de control. Ej: int nMuestras = 0;  
• Las variables con un ámbito muy amplio deberán tener nombres largos y muy significativos. Las de ámbito muy restringido pueden tener nombres muy cortos → intuitivamente el lector pensará que es una variable sin futuro, que no necesita retener. Ej.: int i; char c; String s;  
• Las variables no deben tener nombres ambiguos → dificulta determinar para qué sirve.  
int valor= 0; // incorrecto → valor de qué? Todos son valores

Las variables jamás deben albergar cosas diferentes → dificulta determinar para qué sirve. Tarde o temprano el programador la usará con el contenido inadecuado.

Ej: int numero= 0; numero= cuentaOvejas();  
numero= cuentaSillas();

**Sobre las expresiones**• Evite expresiones (numéricas o condiciones lógicas) complejas. Si una expresión se complica,  
considere la creación de variables auxiliares o métodos auxiliares (posiblemente privados).

**Sobre las estructuras de programa**• No use las construcciones break o continue, salvo que sea estrictamente necesario → para que los bucles sean legibles, sin sorpresas.  
• Un bucle no debe ocupar más de una pantalla (unas 20 líneas de código) desde su inicio hasta su final, incluyendo líneas preliminares para inicializar variables. Si no fuera así, sería conveniente crear métodos auxiliares con nombre propio → para poder leerlos en pantalla en forma completa.  
• Evite el anidamiento excesivo de estructuras for, while, if, switch. Si hay más de tres (3)  
estructuras anidadas, considere la oportunidad de introducir métodos auxiliares con nombre propio.

**2. Los Comentarios**Antiguamente, un principio básico del procesamiento de datos señalaba la práctica de mantener archivos independientes, de código y de documentación, en sincronía. Los resultados de hecho confirman que no es una buena práctica. La documentación de los programas es notablemente pobre y su mantenimiento peor. Los cambios hechos en el programa no aparecen pronto, exacta e invariablemente en el documento.

Actualmente se recomienda combinarlos en un solo archivo, conteniendo toda la información  
incorporados en el archivo fuente, mediante comentarios. Esto es un incentivo poderoso hacia el propio mantenimiento y una seguridad de que la documentación será siempre manejada por el usuario de programa. Dichos programas son llamados auto-documentados.

Todos los lenguajes de programación permiten intercalar comentarios. El contenido de los comentarios puede ser texto arbitrario en lenguaje natural. El compilador ignora los comentarios.

Documentar el código de un programa es añadir suficiente información como para explicar lo que hace, punto por punto, de forma que no sólo los ordenadores sepan qué hacer, sino que además los humanos entiendan qué están haciendo y por qué.  
Hay dos reglas que no se deben olvidar nunca:  
• todos los programas tienen errores y descubrirlos sólo es cuestión de tiempo y de que el programa  
tenga éxito y se utilice frecuentemente  
• todos los programas sufren modificaciones a lo largo de su vida, al menos todos aquellos que tienen éxito

**Qué documentar**Hay que añadir explicaciones a todo lo que no es evidente. No hay que repetir lo que se hace, sino explicar por qué se hace. Esto se traduce en:  
• Una descripción de la funcionalidad **(¿qué hace?)** de cada clase y de cada método.  
• Una descripción del comportamiento **(¿cómo lo hace?)** de cada clase y de cada método.  
• Una descripción del propósito de cada constante, declaración de tipo y variable.  
• ¿qué algoritmo se está usando? ¿qué limitaciones tiene el algoritmo? ¿... la implementación?  
• ¿qué se debería mejorar ... si hubiera tiempo?

**Tipos de Comentarios**Java dispone de tres notaciones para introducir comentarios:  
• **javadoc**: Su propósito es generar documentación externa. Comienzan con los caracteres "/\*\*", se pueden prolongar a lo largo de varias líneas (que probablemente comiencen con el carácter "\*") y terminan con los caracteres "\*/".

• **una línea:** Comienzan con los caracteres "//" y terminan con la línea. Se usa paradocumentar código que no se necesita que aparezca en la documentación externa (que genere javadoc). Este tipo de comentarios se usa incluso cuando el comentario ocupa varias líneas, cada una de las cuales comenzará con "//".

• **tipo C**: Su propósito es anular código, cuando por algún motivo se desea mantener código que es obsoleto. Para que no se ejecute, se comenta. (En inglés se suele denominar "**C**omment out". De allí su nombre). Comienzan con los caracteres "/\*", se pueden prolongar a lo largo de varias líneas (que probablemente comiencen con el carácter "\*") y terminan con los caracteres "\*/".

**Cuándo colocar un comentario**Por obligación (**javadoc**):  
1. al principio de cada clase  
2. al principio de cada método  
3. ante cada variable de clase  
Por conveniencia (**una línea**):  
4. al principio de fragmento de código no evidente  
5. a lo largo de los bucles  
Y por si acaso (**una línea**):  
6. siempre que se haga poco habitual  
7. siempre que el código no sea evidente  
Es primordial que cuando un programa se modifica, los comentarios se modifiquen al mismo tiempo, de modo que los comentarios no acaben haciendo referencia a un algoritmo que ya no se utiliza.

**Javadoc**Javadoc es una herramienta integrada al SDK de Java, que permite documentar de manera rápida y sencilla las clases y los métodos que se proveen. Genera un conjunto de páginas web (formato html) a partir de los archivos de código. Esta herramienta toma en consideración algunos comentarios para generar una documentación bien presentada de clases y componentes de clases (variables y métodos). Para ello es preciso que los comentarios tengan una sintaxis especial. Deben comenzar por "/\*\*" y terminar por "\*/", incluyendo una descripción y algunas etiquetas especiales:  
/\*\*  
\* Parte descriptiva.  
\* Que puede consistir de varias frases o párrafos.  
\*  
\* @**etiqueta** texto específico de la etiqueta  
\*/  
Estos comentarios especiales deben aparecer justo antes de la declaración de una clase, un atributo o un método, en el mismo código fuente. En las siguientes secciones se detallan las etiquetas (tags) que javadoc sabe interpretar en cada uno de los casos. Como regla general, hay que destacar que la primera frase (el texto hasta el primer punto) recibirá un tratamiento destacado, por lo que debe aportar una explicación concisa y contundente del elemento documentado. Las demás frases entrarán en detalles.

**Documentación de constructores y métodos**Deben usarse al menos las etiquetas:  
•**@param**: una por argumento de entrada  
•**@return**: si el método no es *void*•**@exception**: una por tipo de *Exception* que se puede lanzar  
La tabla muestra todas las etiquetas posibles y su interpretación:

|  |  |  |
| --- | --- | --- |
| @param | nombre del parámetro | descripción de su significado y uso |
| @return | descripción de lo que se devuelve |  |
| @exception | nombre de la excepción | excepciones que pueden lanzarse |
| @since | indica desde qué versión o fecha existe este constructor o método en la clase |  |
| @deprecated | Indica que este método no debería usarse pues puede desaparecer en próximas versiones |  |

**Documentación de atributos**Ninguna etiqueta es obligatoria.  
La tabla muestra todas las etiquetas posibles y su interpretación:

|  |  |
| --- | --- |
| @since | Indica desde qué versión o fecha existe este atributo en la clase |
| @deprecated | Indica que este atributo no debería usarse pues puede desaparecer en próximas versiones |

**3. Nomenclatura**Los nombres de clases, variables y métodos son un mecanismo básico para transmitir al lector lo que piensa el programador. En principio, cualquier convenio sería bueno, con el único requisito de que se siguiere sistemáticamente.

A continuación se describe el convenio más habitual en la comunidad de programadores Java.  
**Reglas de nomenclatura  
REGLA GENERAL**: cuando un nombre conste de varias palabras, se escribirán una tras otra, sin solución de continuidad, comenzando cada palabra por mayúscula:  
**V**ariable**D**e**T**res**P**alabras  
**N**umero**D**e**T**elefonos**E**n**L**a**O**ficina  
Las excepciones a esta regla general se señalan en cada caso.  
• Los nombres para las clases deben ser sustantivos. Se sigue la REGLA GENERAL.  
ColaLlamadas, Estancia, ListaDeNombres  
• Los nombres de las variables siguen la REGLA GENERAL; pero la primera letra será minúscula.  
int numeroDeCarneros;  
• Las constantes ("variables" final) se escribirán íntegramente con mayúsculas. Si constan de varias palabras, se separarán estas por "\_". static final int COLOR\_ROJO= 0xFF0000;  
• Los nombres de los métodos deben ser verbos. Se sigue la REGLA GENERAL; pero la primera letra será minúscula.  
String getNombre (); void setNombre(String string);  
• Si una palabra es un acrónimo, se sigue la REGLA GENERAL de escribir solo el primer carácter con mayúscula (con las salvedades antes apuntadas).  
class Dni, dniDelSujetoPasivo, getUrl()

**4. Identación**El identación o i**n**dentación (términos utilizados indistintamente), indican los cambios de línea y espacio en blanco al principio de una línea o párrafo. Es un mecanismo básico para transmitir al lector lo que piensa el programador. Se realiza una alineación de los párrafos para los diferentes elementos del código para mostrar los niveles de anidamiento.  
A continuación se describe el convenio más habitual en la comunidad de programadores Java.

***Tema 4***

**Objeto**En el paradigma de objetos, los objetos son los elementos primarios utilizados para construir programas. Dichos objetos son abstracciones de alguna entidad del mundo real (tangible o no), circunscripta al dominio del problema a resolver (espacio del problema).

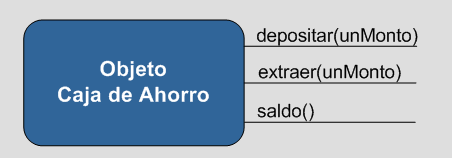
Al hablar de abstracción se hace referencia al hecho de aislar los aspectos relevantes, en un determinado contexto, descartando los detalles que no son significativos en el dominio del problema.

Para pasar del problema al espacio de la solución (resolver el problema en cuestión) en el paradigma OO solo se cuenta con objetos, que se comportan de una determinada manera, definida por sus métodos, e interactúan entre ellos enviándose mensajes. Este es el concepto básico de la POO.

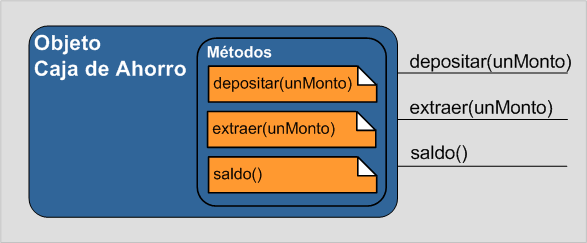
Un objeto es una entidad que queda caracterizada por:

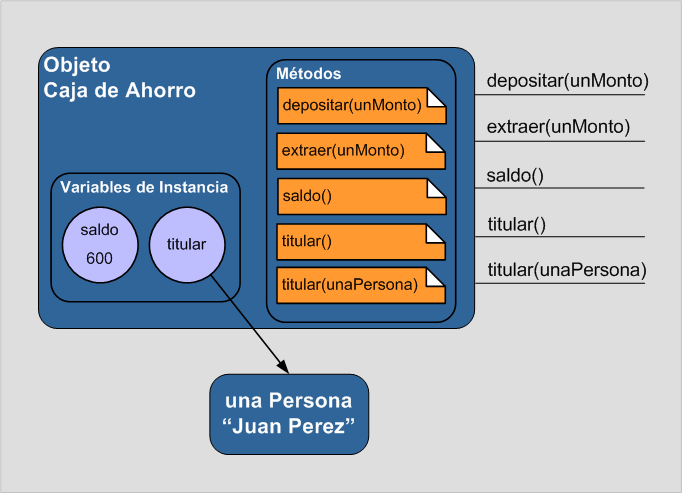
* **su estado**: representado por el valor de sus atributos o variables de instancia (v.i.). Al crear un objeto, sus atributos toman valores particulares, representando las características intrínsecas de una entidad particular. Estos atributos se utilizan para representar aquellas características propias del objeto (como puede ser el saldo de una cuenta bancaria) y a los objetos con los que deberá colaborar a lo largo de su vida (por ejemplo, un titular). En este último caso se da una relación de conocimiento, tema que será tratado más ampliamente.
* **su comportamiento**: el cual está dado por los mensajes a los que puede responder. El comportamiento del objeto puede verse como los servicios que presta, su utilidad. Esto determina para qué existe. Por ejemplo, la administración de una cuenta bancaria, representada por los métodos extraer, depositar, consultar saldo, etc.
* **una identidad**: ni un objeto es idéntico a otro, incluso aunque tengan los mismos valores en sus atributos. Porejemplo, dos titulares tendrán distinta identidad, a pesar de llamarse ambos “Juan”.

Otra de las propiedades importantes de los objetos es el encapsulamiento, que asegura que un objeto no puede invadir el espacio privado (estado) de otro objeto. Es el propio objeto el único que puede manipular su estado, es decir, ver o modificar los valores de sus atributos.

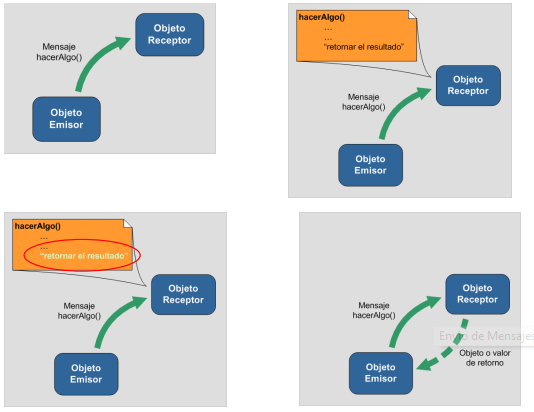
**Comportamiento: ¿Qué hace?**El comportamiento indica qué sabe hacer el objeto, es decir cuáles  
son sus responsabilidades. El comportamiento se especifica a través del  
conjunto de mensajes que el objeto sabe responder. Un objeto se define  
en términos de su comportamiento, por lo que al modelar con objetos  
se debe establecer **qué sabe hacer** cada objeto. El objeto debe verse  
como una entidad activa, con la capacidad de realizar tareas. Además,  
se debe tener en cuenta que, para llevar a cabo su trabajo, un objeto  
puede recibir la colaboración de otros objetos.  
Al conjunto de mensajes que un objeto puede responder se lo denomina **protocolo**. 

**Comportamiento: ¿Cómo lo hace?**La implementación de un objeto indica **cómo hace** el objeto para responder a los mensajes que recibe. Se  
especifica a través de un conjunto de métodos. Esto significa que en el cuerpo de cada método está el código que  
realiza la tarea que debe llevar adelante el objeto.  
La encapsulación también alcanza a la implementación del objeto. En efecto, el código es privado. Solamente el  
propio objeto puede accederlo de manera directa, y hacer uso de él.

**Implementación: Método vs. Mensaje**Cuando un objeto envía un mensaje a otro, el objeto  
receptor responde activando el método asociado a ese  
mensaje. Básicamente un método es la implementación  
(código) asociada a un mensaje. Una vez definido el  
protocolo de un objeto (qué es lo que va a hacer, cuáles  
son sus responsabilidades), es necesario especificar  
cómo va a llevar a cabo sus responsabilidades y esto se  
hace por medio de la codificación de los métodos.

**El estado**El estado está representado por los valores que toma el conjunto de atributos de un objeto, representado por unnconjunto de variables de instancia (**v.i**.). Las variables de instancia acompañan al objeto durante todo su ciclo de vida.  
Una v.i. puede hacer referencia a una propiedad intrínseca del objeto que representa, o bien **a otro objeto quencolabora con el objeto primario** para que éste lleve a cabo sus responsabilidades. A lo largo de la vida del objeto, las propiedades y colaboradores pueden cambiar. Por ejemplo, al depositar $100 en una cuenta bancaria la variable de  
instancia saldo cambiará su valor. Este estado es privado del objeto. Es una de las características fundamentales del paradigma, y resulta crucial para lograr software escalable. Por definición, las v.i. (estructura interna) de un objeto son privadas a éste. El único que puede acceder y manipular sus v.i. es el propio objeto. Si un objeto externo quiere acceder a una v.i. lo tiene que hacerlo por medio del envío de un mensaje. Esto le da la posibilidad al objeto de decidir qué es lo que quiere publicar, o cómo quiere hacerlo.  
En la siguiente figura puede observarse un ejemplo concreto: un objeto ***Caja de Ahorro***, cuyo estado queda representado por sus v.i.: toma un valor particular en el caso del ***saldo***, e indica una relación de conocimiento con el objeto ***Persona***, que colabora con él a través de la v.i. ***titular***. A su vez puede determinarse el comportamiento (qué hace el objeto), a través del conjunto de métodos (***depositar(), extraer(), saldo(),*** etc), que responde a sendos mensajes.  
Atención: No se debe confundir el **atributo** saldo con el **método** saldo()

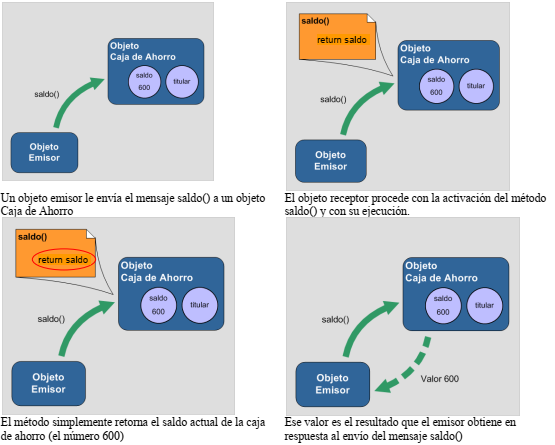
**Mensajes**

Para poder enviarse mensajes, los objetos deben conocerse. Esto significa que debe haber una “relación” entre ellos. El objeto receptor responde en forma activa, ejecutando el método asociado al mensaje. El método, es decir la  
implementación (el código), especifica qué hace el objeto receptor al recibir el mensaje.  
Cuando un objeto (emisor) le envía un mensaje a otro objeto (receptor), el objeto receptor responde activando el método asociado a este mensaje, siempre que el mensaje recibido forme parte del protocolo del objeto receptor. Caso contrario, pueden darse distintas situaciones, dependiendo del lenguaje: algunos lenguajes pueden evitar esta situación mediante un sistema de tipos fuerte que chequee estas situaciones en tiempo de compilación (java), mientras que otros brindan una mayor libertad en cuanto al envío de mensajes y responden, eventualmente, con un error en tiempo de ejecución (smalltalk).

**Formato de mensajes**Típicamente un mensaje tiene un nombre, puede o no tener argumentos, y se invoca sobre un objeto. Los distintos lenguajes de programación orientados a objeto proponen una sintaxis particular para indicar el envío de un mensaje. El formato aceptado habitualmente es el siguiente:  
 objeto\_receptor.nombre\_de\_mensaje(argumentos);  
Ejemplo:  
 unaCuenta.depositar(100);  
Los valores pasados como argumento en el mensaje (***parámetros reales***), son reemplazados en los ***parámetros formales*** del método, al momento de ejecutarse.

**Métodos**Un método es un conjunto de sentencias que llevan a cabo una acción. Puede tener parámetros de entrada que regularán dicha acción y, puede tener un valor de salida (o valor de retorno). La diferencia con un procedimiento o función usados en programación estructurada es que un método, al estar asociado con un objeto o clase en particular, puede acceder y modificar los datos privados del objeto correspondiente de forma tal que sea consistente con el comportamiento deseado para el mismo. Así, es recomendable entender a un método no como una secuencia de instrucciones sino como la forma en que el objeto es útil (el modo de hacer su trabajo). Por lo tanto, podemos considerar al método la respuesta de un objeto ante el pedido para que realice una tarea. Cada método tiene:  
• una parte que es pública (la **firma** del método), que constituye la interfaz, lo que los demás objetos pueden ver. Expone lo que sabe hacer.  
• una parte que es privada (el cuerpo del método – el código), que está oculta. El ocultamiento garantiza que no se propaguen los efectos de los cambios en las estructuras de datos encapsuladas dentro del objeto, sobre otras partes del sistema.

En la declaración de un método se establece un nombre (se recomienda que tenga significado en el dominio del problema, dado que propicia la legibilidad), la visibilidad, el tipo de retorno, y los **parámetros formales.** Los parámetros formales actúan como variables locales en el cuerpo del método, y se inicializan al valor que se pasa como argumento en la invocación del método (**parámetros reales**):  
public void asignarSaldo(double unMonto){  
 saldo = unMonto;  
}  
Un método es la contraparte funcional del mensaje. Por cada mensaje que el objeto debe responder, declarado como parte del protocolo del objeto, debe existir un método asociado que exprese la forma de llevar a cabo la tarea solicitada por dicho mensaje, es decir, **cómo se hace**. El *cómo* en un método puede implicar básicamente tres cosas:  
**1) Modificar el estado del objeto**. Esto implica asignar a una v.i. un valor distinto al que tiene. Por ejemplo, en el caso de la cuenta bancaria, se podría tener un método como el siguiente:  
public void asignarSaldo(double unMonto){  
 saldo = unMonto;  
}

  
**2) Retornar y terminar**. Un método puede retornar un resultado. Por ejemplo, en el caso de la cuenta bancaria es lógico que le podamos pedir el saldo:  
public double saldo(){  
 return saldo;  
}

**3) Colaborar con otros objetos**. En la mayoría de los casos, para llevar a cabo una determinada responsabilidad, el objeto necesita la colaboración de otros objetos. Por ejemplo, para alquilar una película es necesario saber si el cliente adeuda alguna película. Por lo tanto, el método *alquilar* en el objeto *videoClub* sería algo de la forma:

public void alquilar(unaPelicula, unCliente){  
 if (**unCliente.esDeudor**()) {  
 System.out.println(“No alquilar”);  
} else {  
 unaPelicula.alquilar**A**(unCliente)  
 }  
}  
El video club básicamente funciona como un coordinador de comportamiento del cliente y de la película.

**Observadores y Mutadores (Accessors: getters y setters)**Dado que el estado de los objetos es privado, se debe proveer el mecanismo para que los demás objetos puedan acceder a sus atributos, de modo consistente. Para ello se utiliza lo que se conoce como observadores (getters), que solo muestran sus atributos, sin modificarlos, y los mutadores (setters), que asignan valores o modifican los atributos. Estos no son nombres obligatorios. Solo son utilizados por su significado: get=obtener, set=asignar.  
Por ejemplo, si se desea obtener el saldo de una caja de ahorro, en java se escribirá lo siguiente:  
/\*\* Retorna el saldo actual de la cuenta.  
\* @return Un valor de tipo double.  
\* @throws No dispara ninguna excepcion.  
\*/  
public double getSaldo(){  
return saldo;  
}  
En el caso de querer asignar un monto particular al saldo, en java se procederá de la siguiente manera:  
/\*\* Asigna unMonto al saldo actual de la cuenta.  
\* @param double unMonto.  
\* @return No devuelve ningún valor.  
\* @throws No dispara ninguna excepcion.  
\*/  
private void setSaldo(double unMonto){  
saldo = unMonto;  
}  
El **encapsulamiento** es la característica del paradigma de objetos que garantiza que los usuarios (otros objetos) de un objeto sólo pueden modificar su estado e interactuar con él a través de sus métodos. La gran ventaja del encapsulamiento consiste en que, si un objeto cambia internamente sin modificar su interfaz, el cambio no desencadenará ninguna otra modificación en el sistema. Esta característica puede ser llevada a un nivel superior, o **doble encapsulamiento**, lo cual significa que el propio objeto no accede a sus atributos directamente sino a través de sus métodos. Para la implementación de este concepto, por ejemplo en los métodos constructores, se utilizan los métodos set para las asignaciones:  
CajaAhorro(int p\_codigo, double p\_saldo) {  
 setCodigo(p\_codigo);  
setSaldo(p\_saldo);  
}  
En el caso de un método de actualización, como por ejemplo una extracción de una caja de ahorro, se implementa mediante los *accessors*, como puede observarse en el siguiente ejemplo:  
public void extraer(double p\_importe){  
 setSaldo(getSaldo()- p\_importe);  
}  
En conclusión, la implementación del mecanismo de **doble encapsulamiento** mediante los *accessors* permite acceder a los atributos de modo consistente y evita la modificación en cadena del sistema

**Objeto y Clase**Una clase representa un nivel superior de abstracción. Un objeto es una abstracción de una entidad del mundo real, en el modelo del dominio. En el dominio del problema aparece un conjunto de objetos que se comportan en forma similar. Surge la necesidad de encontrar una forma de definir el comportamiento de todos los objetos en forma general.

Por ejemplo, si el objeto que se está analizando es una caja de ahorro, es evidente que en un banco habrá una gran cantidad de objetos de este tipo, uno por cada caja de ahorro existente en el banco. Se puede afirmar que todos estos objetos tendrán el mismo comportamiento, y si hay algún cambio (por ejemplo, el interés devuelto), el mismo se reflejará en todos los objetos de este tipo.

Esto lleva a pensar que sería útil algún mecanismo que permita agrupar el comportamiento común a un conjunto de objetos, de manera que pueda ser reutilizado sin tener que ser especificado individualmente. Por otra parte, el estado de cada caja de ahorro será diferente (por ejemplo, tendrán distinto saldo), pero todas compartirán la misma estructura (todas conocen su saldo, titular, etc.). Desde el punto de vista de la reutilización, es así como surge la noción de clase: como una técnica que permite factorizar y reutilizar estructura y comportamiento común. La clasificación es un mecanismo de abstracción, concepto que se desarrollará más ampliamente en temas posteriores.

Con el objetivo de reutilizar estructura y comportamiento común, en el ejemplo de cajas de ahorro, aparece la noción de ***clase*** “Caja de Ahorro”, la cual se utiliza para describir el comportamiento de todas y cada una de sus ocurrencias (***instancias***). Cada objeto caja de ahorro es un elemento único de la clase en la que se basa. La ventaja que representa una clase está en que permite describir en un sólo lugar el comportamiento de un conjunto de objetos. Una vez definido este comportamiento, es posible crear objetos que lo reutilicen y se comporten de la manera allí descripta.

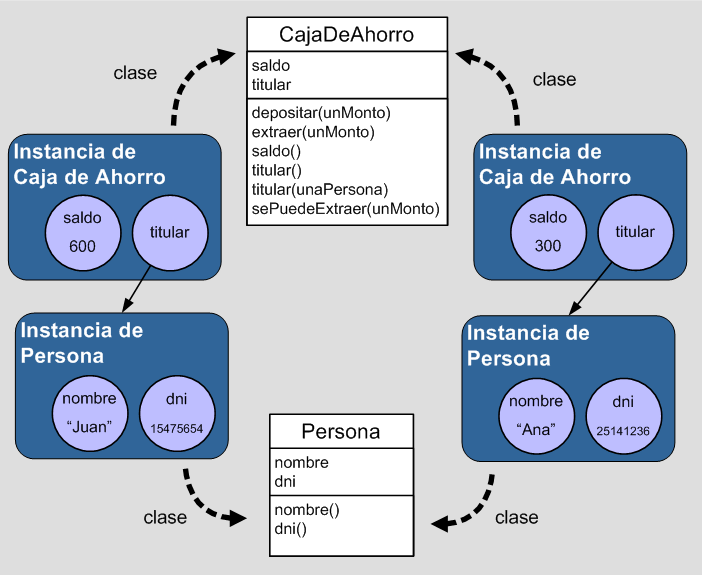
Por lo tanto*: “una clase es una descripción abstracta de un conjunto de objetos”.*

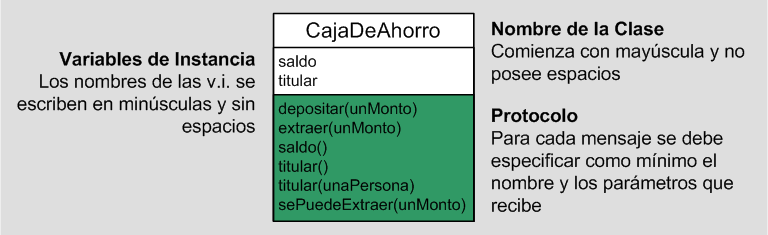
Una clase especifica qué forma tendrán sus instancias (sus variables de instancia) y cómo responderán a los mensajes que se le envíen. De esta forma, una clase puede ser vista como una descripción de sus instancias y como un repositorio de comportamiento. Cuando se le envía un mensaje a un objeto, lo que hace el intérprete es buscar **en la clase** del objeto receptor un método con el cual responder al mensaje recibido. Por lo tanto, todos los objetos que comparten una misma clase responden de la misma forma al envío de un mensaje.

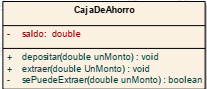
En el caso de la caja de ahorro, dado que en un banco habrá muchas, lo que se hace es modelar la noción de caja de ahorro en una clase. Esta clase definirá qué atributos tendrán sus instancias, así como los mensajes a los que puede responder. Por ejemplo, la clase se puede llamar CajaDeAhorro, y especificar que sus instancias (cada caja de ahorro) tiene dos atributos (saldo y titular) y que puede responder a los mensajes saldo(), titular(), depositar(unMonto) y extraer(unMonto).

Para crear una nueva caja de ahorro se debe pedir a la clase CajaDeAhorro una nueva instancia. Para ello, las clases cuentan con métodos especiales, llamados “constructores”. Cuando se desea crear un objeto (instanciar una clase), se invocan estos métodos, que definen un proceso de inicialización que prepara al objeto para ser usado: le asigna un espacio de memoria y los valores apropiados a sus atributos.

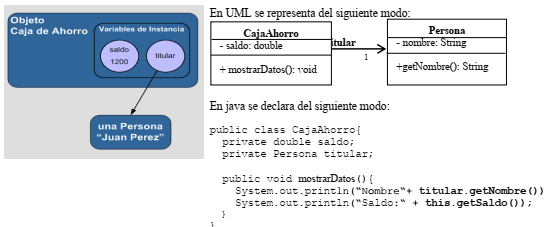
Una vez creada una instancia, esta tiene sus **propios** atributos (v.i.). Por ejemplo, al crear dos cuentas bancarias, cada una tendrá **su** saldo y **su** titular. Sin embargo, al ser instancias de la misma clase, al recibir un mensaje, el método que se activará se buscará en la clase CajaDeAhorro, que es común a las dos.



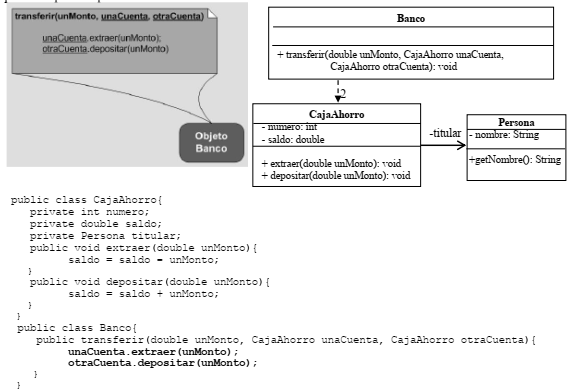
**Clase y UML**Las clases se especifican por medio de un nombre, la estructura que tendrán sus instancias y los métodos que definen su comportamiento. El nombre debe comenzar con mayúscula y no contener espacios. Las v.i. representan el estado o estructura interna de las instancias de la clase. El conjunto de mensajes que entenderá una instancia de la clase es llamado ***protocolo***. Es la declaración de lo que el objeto sabe hacer. En función del protocolo los otros objetos saben qué mensajes le pueden enviar. Por eso se dice que “es el conjunto de mensajes que el objeto sabe responder”. La representación gráfica de una clase, utilizando una versión simplificada del lenguaje UML es: 

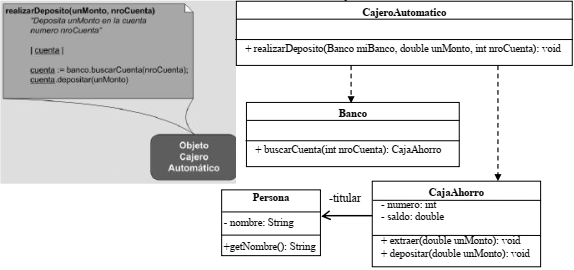
La representación de la clase puede variar según la herramienta  
utilizada. Sin embargo, generalmente tiene la forma de la figura  
adjunta, donde puede observarse el tipo de dato, de retorno de los  
métodos, y un signo que indica el tipo de acceso: público (+),  
privado (-), protegido (#)

**Formas de conocimiento entre objetos**Es fundamental tener presente que, para que un objeto le envíe un mensaje a otro, debe conocerlo, o sea, nombrarlo. Se dice que existe una ligadura (binding) entre un objeto y su nombre. Ej: Persona titular = unaPersona;  
En este ejemplo, **titular** es el nombre del objeto al que hace referencia.  
También se dice que hay una “relación” entre los objetos, que puede ser de distintos tipos. A la hora de programar existen diversas formas de conocimiento:  
1. Variables de instancia  
2. Parámetros  
3. Variables temporales  
4. Seudo-variables (son una forma especial de conocimiento)

**1. Variables de instancia**Definen una relación entre un objeto y sus atributos. Constituyen la estructura de la clase/objeto. Las variables de instancia acompañan al objeto desde su creación hasta que muere. 

**2. Parámetros**Se utilizan para nombrar objetos que el objeto receptor necesita para cumplir un requerimiento. Se refiere a los parámetros de un mensaje. La relación de conocimiento se establece dentro del cuerpo del método, y dura el tiempo que el método se encuentra activo.

Por ejemplo, para hacer una transferencia de una cuenta a otra, el banco debe saber cuáles son las cuentas intervinientes. En este caso las cuentas son parámetros que se envían junto con el monto, en el mensaje *transferir* para poder cumplir el requerimiento

**3. Variables temporales**Definen relaciones temporales dentro de un método. Estas variables se utilizan para nombrar objetos temporalmente. La relación con el objeto se crea durante la ejecución del método, y dura el tiempo que éste se encuentra activo. Al finalizar la activación del método, estas variables dejan de existir. 

**4. Seudo-Varibles**Así como un objeto necesita conocer a otro objeto para enviarle un mensaje, es necesario que tenga una forma de nombrarse a sí mismo para poder enviarse mensajes. Este autoreferenciamiento se hace por medio de una seudovariable, que toma distintos nombres en los lenguajes de programación. En algunos casos es *self*, y en el caso de Java, C# y C++ es *this*. Se dice que es una seudo-variable ya que funciona similar a una variable de instancia, excepto por dos motivos: no está definida en ninguna clase y no puede ser asignada (no se le puede asignar un valor). Todos los objetos tienen la referencia implícita **this,** que apunta a sí mismo.

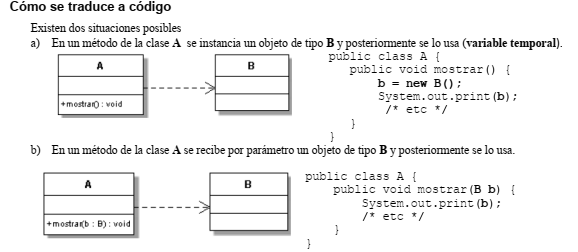
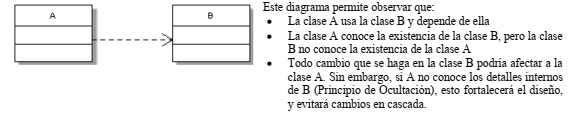
**Conocimiento entre objetos y Relaciones en UML**Para enviarse mensajes, los objetos deben estar relacionados, es decir, deben “conocerse”. Los mensajes “navegan” por las relaciones existentes entre las clases a las que pertenecen los objetos. Las relaciones entre las clases indican cómo se comunican los objetos de esas clases entre sí.  
Existen cuatro tipos de relaciones entre los elementos de un modelo UML: asociación, dependencia, realización y generalización.

  
**1 - Asociación**Es una relación estructural entre entidades (una entidad se construye a partir de otras entidades /una clase tiene en  
su estructura a otra clase. Al codificar esto se representa como un atributo que es una instancia de otra clase.

|  |
| --- |
| La asociación se representa con una línea continua que a veces incluye una etiqueta y otros elementos para indicar la multiplicidad y roles de los objetos involucrados. |

**Navegación de las asociaciones**Aunque las asociaciones suelen ser bidireccionales (se pueden recorrer en ambos sentidos), en ocasiones es necesario hacerlas unidireccionales, es decir, restringir la navegación en un solo sentido.

**Relaciones involutivas**Son las relaciones en las que interviene el mismo tipo de entidad desempeñando distintos roles. Es decir, la misma clase aparece en los dos extremos de la asociación.

**2 – Dependencia**Es una relación de uso entre dos entidades. Una clase depende de la funcionalidad que ofrece otra clase. Puede verse desde el punto de vista “Cliente / Servidor”, es decir, existe una entidad que necesita de un “servicio” y otra entidad que lo provee. Una clase es “cliente” del servicio de otra clase. Se dice que es una relación “débil”, ya que no forma parte de la estructura de la clase dependiente.  
En UML se representa con una flecha discontinua que va desde la clase “cliente” hasta la clase que ofrece el “servicio/funcionalidad”. Un diagrama genérico sería: 

**3 – Realización**Es una relación de contrato con otra clase: una clase (interfaz) especifica un contrato que otra clase garantiza que cumplirá. Se la utiliza para implementar una interfaz. En lenguajes como java o php se utiliza la palabra reservada “implements”.

|  |
| --- |
| La parte realizante cumple con una serie de especificaciones propuestas por la clase realizada (interfaces). Se representa con una línea discontinua con una punta de flecha cerrada y vacía |

**4 - Generalización**

Es una relación en la que el elemento generalizado (padre) puede ser substituido por  
cualquiera de los elementos hijos, ya que comparten su estructura y comportamiento.  
Gráficamente, la generalización se representa con una línea con punta de flecha  
cerrada y vacía.