UML (Lenguaje de Modelado Unificado)

Introducción

El Lenguaje de Modelado Unificado prescribe un conjunto de notaciones y diagramas estándar para modelar sistemas orientados a objetos, y describe la semántica esencial de lo que estos diagramas y símbolos significan. Mientras que ha habido muchas notaciones y métodos usados para el diseño orientado a objetos, ahora los modeladores sólo tienen que aprender una única notación.

UML es una consolidación de muchas de las notaciones y conceptos más usados orientados a objetos. Empezó como una consolidación del trabajo de Grade Booch, James Rumbaugh, e Ivar Jacobson, creadores de tres de las metodologías orientadas a objetos más populares. UML incrementa la capacidad de lo que se puede hacer con otros métodos de análisis y diseño orientados a objetos. Los autores de UML apuntaron también al modelado de sistemas distribuidos y concurrentes para asegurar que el lenguaje maneje adecuadamente estos dominios.

UML prescribe una notación estándar y semánticas esenciales para el modelado de un sistema orientado a objetos. Previamente, un diseño orientado a objetos podría haber sido modelado con cualquiera de la docena de metodologías populares, causando a los revisores tener que aprender las semánticas y notaciones de la metodología empleada antes que intentar entender el diseño en sí. Ahora con UML, diseñadores diferentes modelando sistemas diferentes pueden sobradamente entender cada uno los diseños de los otros.

Aunque UML está pensado para modelar sistemas complejos con gran cantidad de software, el lenguaje es los suficientemente expresivo como para modelar sistemas que no son informáticos, como flujos de trabajo (*workflow*) en una empresa, diseño de la estructura de una organización y por supuesto, en el diseño de hardware.

Un modelo UML está compuesto por tres clases de bloques de construcción:

• Elementos: Los elementos son abstracciones de cosas reales o ficticias (objetos, acciones, etc.)

• Relaciones: relacionan los elementos entre sí.

• Diagramas: Son colecciones de elementos con sus relaciones.

Un diagrama es la representación gráfica de un conjunto de elementos con sus relaciones. En concreto, un diagrama ofrece una vista del sistema a modelar. Para poder representar correctamente un sistema, UML ofrece una amplia variedad de diagramas para visualizar el sistema desde varias perspectivas.

Se Dispone de 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 clases: muestra las clases, interfaces, colaboraciones y sus relaciones. Son los más comunes y dan una vista estática del proyecto.
- Diagrama 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.
- Diagrama de componentes: Muestran la organización de los componentes del sistema. Un componente se corresponde con una o varias clases, interfaces o colaboraciones.
- Diagrama de despliegue.: 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.

- Diagrama de casos de uso: Muestran los casos de uso, actores y sus relaciones. Muestra quien puede hacer que y relaciones existen entre acciones (casos de uso). Son muy importantes para modelar y organizar el comportamiento del sistema.

Lo diagramas dinámicos son:

- Diagrama de secuencia, Diagrama de colaboración: Muestran a los diferentes objetos y las relaciones que pueden tener entre ellos, los mensajes que se envían entre ellos. Son dos diagramas diferentes, que se puede pasar de uno a otro sin pérdida de información, pero que nos dan puntos de vista diferentes del sistema. En resumen, cualquiera de los dos es un Diagrama de Interacción.
- Diagrama de estados: muestra los estados, eventos, transiciones y actividades de los diferentes objetos. Son útiles en sistemas que reaccionen a eventos.
- Diagrama 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.

Como podemos ver el número de diagramas es muy alto, en la mayoría de los casos excesivos, y UML permite definir solo los necesarios, ya que no todos son necesarios en todos los proyectos.

Diagrama de Clases

Forma parte de la vista estática del sistema. En el diagrama de clases será donde definiremos las características de cada una de las clases, interfaces, colaboraciones y relaciones de dependencia y generalización. Es decir, es donde daremos rienda suelta a nuestros conocimientos de diseño orientado a objetos, definiendo las clases e implementando las ya típicas relaciones de herencia y agregación.

En el diagrama de clases debemos definir a estas y a sus relaciones. Un diagrama de clases está compuesto por los siguientes elementos:

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

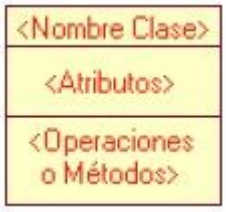
Clase

Una clase es la unidad básica que encapsula toda la información de un Objeto (un objeto es una instancia de una clase). A través de ella podemos modelar el entorno en estudio (una Casa, un Auto, una Cuenta Corriente, etc.), está representada por un rectángulo que dispone de tres apartados, el primero para indicar el nombre, el segundo para los atributos y el tercero para los métodos.

Cada clase debe tener un nombre único, que las diferencie de las otras.

Un atributo representa alguna propiedad de la clase que se encuentra en todas las instancias de la clase. Los atributos pueden representarse solo mostrando su nombre, mostrando su nombre y su tipo, e incluso su valor por defecto.

Un método u operación es la implementación de un servicio de la clase, que muestra un comportamiento común a todos los objetos. En resumen es una función que le indica a las instancias de la clase que hagan algo.



Ejemplo:

Aquí vemos un ejemplo. La clase usuario contiene tres atributos. Nombre que es public, dirección que es protected y situación que es private. Situación empieza con el valor 3. También dispone de tres métodos Entrar, Salir y Trabajar.

Atributos

- Los atributos o características de una Clase pueden ser de tres tipos, los que definen el grado de comunicación y visibilidad de ellos con el entorno, estos son:
  - **public** (+): Indica que el atributo será visible tanto dentro como fuera de la clase, es decir, es accesible desde todos lados.
  - **private** (-): Indica que el atributo sólo será accesible desde dentro de la clase (sólo sus métodos lo pueden acceder).
  - **protected** (#): Indica que el atributo no será accesible desde fuera de la clase, pero si podrá ser accedido por métodos de la clase además de las subclases que se deriven (ver herencia).
  - **Default:** Indica que el atributo es accesible únicamente por la clase que lo contiene y por todas las clases que se encuentren en el mismo paquete que la clase que contiene dicho atributo.(Este caso es para java en particular)

Métodos

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

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

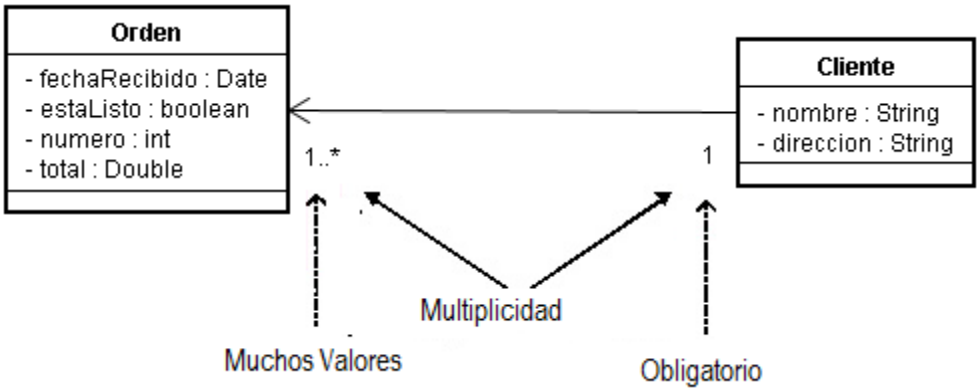
- **protected** (#): Indica que el método no será accesible desde fuera de la clase, pero si podrá ser accedido por métodos de la clase además de métodos de las subclases que se deriven (ver herencia).
- **Default:** Indica que el método es accesible únicamente por la clase que lo contiene y por todas las clases que se encuentren en el mismo paquete que la clase que contiene dicho método. (Este caso es para java en particular)

Relaciones entre clases

Ahora que sea ha definido el concepto de Clase, es necesario explicar cómo se pueden interrelacionar dos o más clases (cada uno con características y objetivos diferentes).

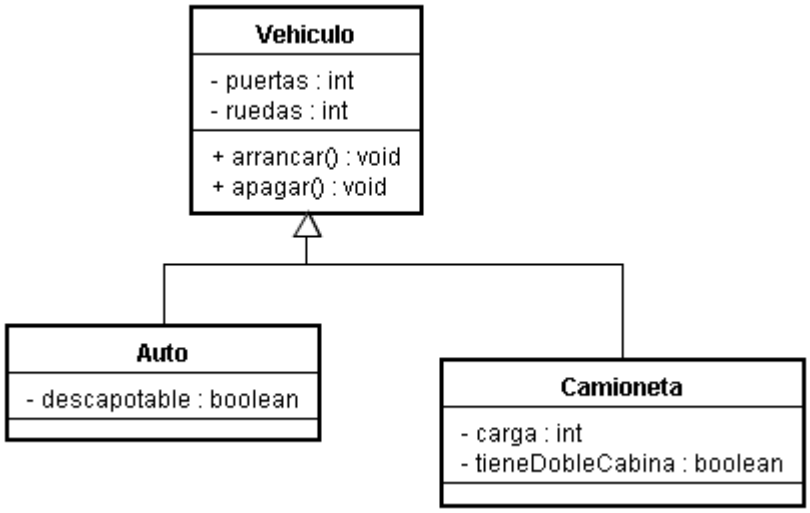
Antes es necesario explicar el concepto de multiplicidad de relaciones: En UML, la multiplicidad de las relaciones indica el grado y nivel de dependencia, se anotan en cada extremo de la relación y éstas pueden ser:

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



Herencia.

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



En la figura se especifica que Auto y Camión heredan de Vehículo, es decir, Auto posee las Características de Vehículo (puertas,ruedas, etc.) además posee algo particular que es Descapotable, en cambio Camión también hereda las características del vehiculo pero posee como particularidad propia carga y tieneDobleCabina

Asociación

Cuando se modela un sistema, ciertos objeto tendrán relación unos con otros y esas relaciones por sí mismas necesitan ser modeladas para ser completamente claros a que se refiere.

Las asociaciones pueden ser de 2 formas:

- Bidireccionales
- Unidireccionales

Una asociación bidireccional Específica que los objetos de una clase están relacionados con los elementos de otra clase. Se representa mediante una línea continua, que une las dos clases. Podemos indicar el nombre, multiplicidad en los extremos, su rol, y agregación.

Cabe destacar que no es una relación fuerte, es decir, el tiempo de vida de un objeto no depende del otro.

 Ejemplo:

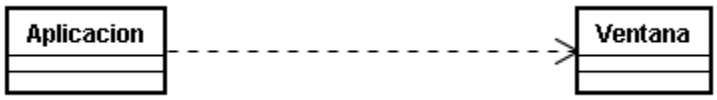
Un cliente puede tener asociadas muchas Órdenes de Compra, en cambio una orden de compra solo puede tener asociado un cliente.

Una asociación unidireccional se dibuja como una línea sólida con una punta de flecha abierta (no la punta de flecha cerrada, o un triángulo, que sirve para indicar la herencia) que apunta a la clase conocida. Al igual que las asociaciones de estándares, la asociación unidireccional incluye un nombre de función y un valor de multiplicidad, pero a diferencia de la asociación estándar bi-direccional, la asociación unidireccional sólo contiene el nombre del rol y el valor de la multiplicidad de la clase conocida

Dependencia o Instanciación

Representa un tipo de relación muy particular, en la que una clase es instanciada (su instanciación es dependiente de otro objeto/clase). Se denota por una flecha punteada.

El uso más particular de este tipo de relación es para denotar la dependencia que tiene una clase de otra, como por ejemplo una aplicación grafica que instancia una ventana (la creación del Objeto Ventana está condicionado a la instanciación proveniente desde el objeto Aplicación):



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

Agregación y Composición.

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

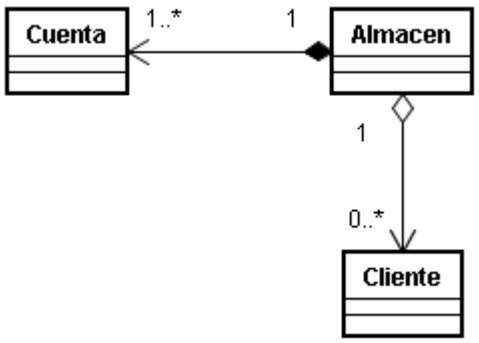
- **Composición**: Es un tipo de relación estática, en donde el tiempo de vida del objeto incluido está condicionado por el tiempo de vida del que lo incluye. (el Objeto base se construye a partir del objeto incluido, es decir, es "parte/todo"), Otra forma de decirlo, es que la existencia del objeto que contiene no tiene sentido alguno sin el objeto contenido.

  En la composición cuando el objeto que contiene nace también lo hace el objeto contenido. Cuando el objeto que contiene muere, también el objeto contenido debe morir.

- **Agregación**: Es un tipo de relación dinámica, en donde el tiempo de vida del objeto incluido es independiente del que lo incluye. (el objeto base utiliza al incluido para su funcionamiento), es decir, el objeto agregado y el objeto que posee al objeto agregado pueden existir de manera independiente.

  En la agregación cuando el objeto que tiene al objeto agregado nace también lo hace el objeto agregado. Cuando el objeto que agrega muere, el objeto agregado no tiene que morir con él y viceversa.

Un Ejemplo es el siguiente:



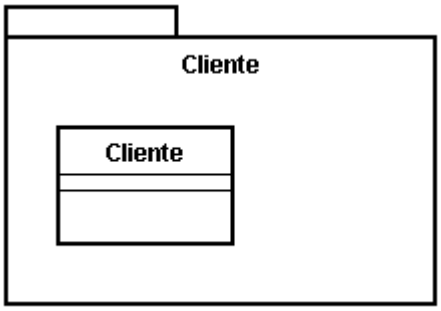
En donde se destaca que:

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

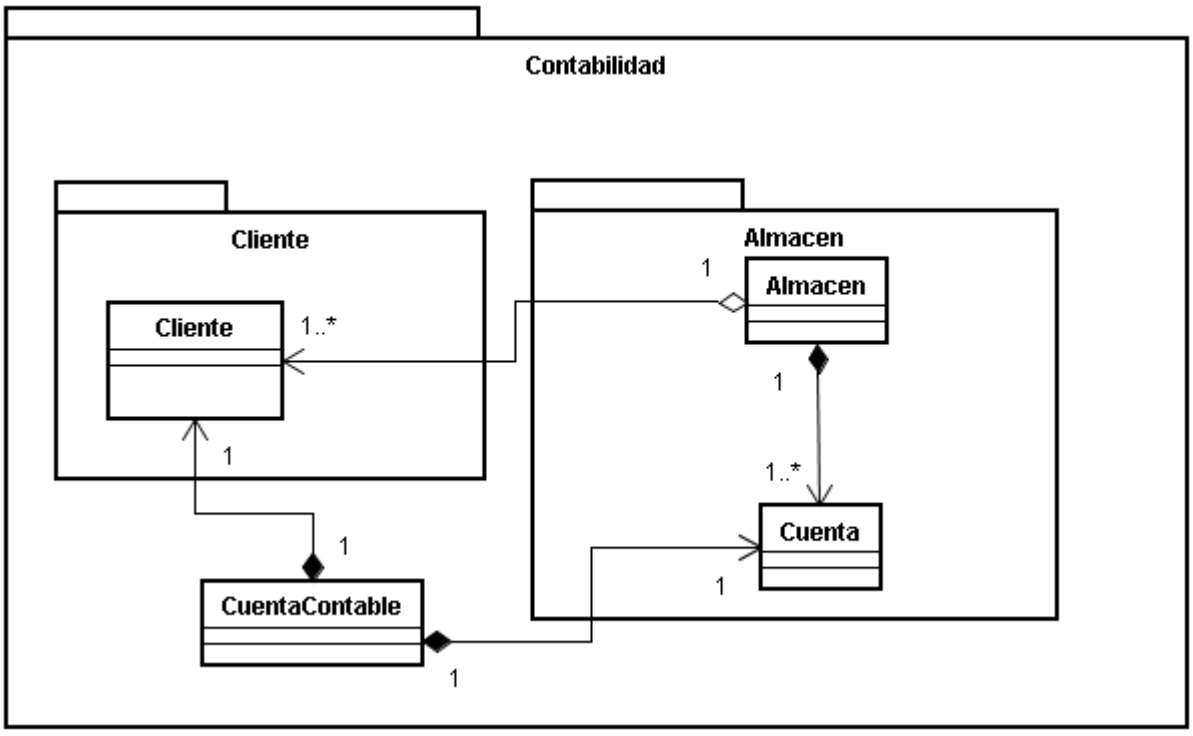
Paquetes

Un paquete es el elemento de organización básica de un modelo de sistema UML. Puede considerar todo el sistema como un paquete que contiene los demás paquetes, diagramas y elementos. Un paquete puede contener paquetes subordinados, diagramas o elementos únicos, y se puede establecer su visibilidad así como la de los elementos que contiene.

Los paquetes sirven para agrupar un conjunto de clases que tienen alguna relación entre sí, a sea una relación por el tipo de comportamiento o por el tipo de propósito que tenga una clase.



Paquete simple



Paquete con subpaquetes

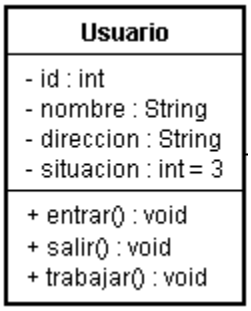
Pasando de UML a Código Java

En esta sección se aprenderá como pasar de un diagrama UML a código en java, en particular lo visto de UML hasta este punto.

Clase

Una clase simple con atributos primitivos se pasa de la siguiente manera:

UML:



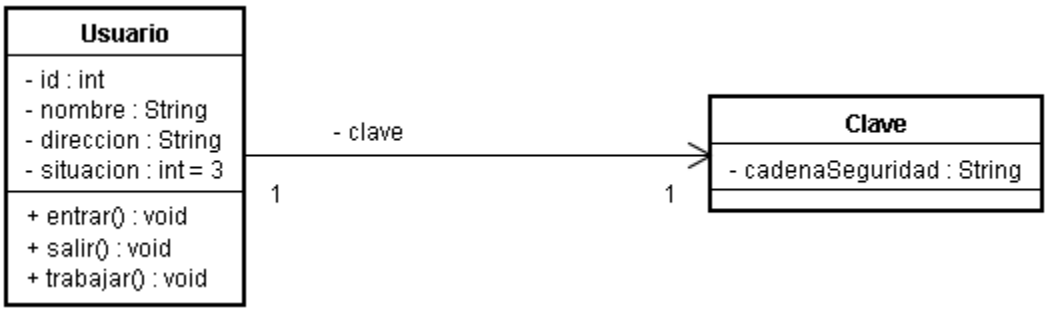
Java:

```
public class Usuario {

	private int id;
	private String nombre;
	private String direccion;
	private int situacion = 3;


	public void entrar(){

	}

	public void salir()	{

	}

	public void trabajar(){

	}

	public int getId() {
		return id;
	}
	public void setId(int id) {
		this.id = id;
	}
	public String getNombre() {
		return nombre;
	}
```

```
	public void setNombre(String nombre) {
		this.nombre = nombre;
	}
	public String getDireccion() {
		return direccion;
	}
	public void setDireccion(String direccion) {
		this.direccion = direccion;
	}
	public int getSituacion() {
		return situacion;
	}
	public void setSituacion(int situacion) {
		this.situacion = situacion;
	}

}
```

Asociación:

UML:



Java:

Clave:

```
public class Clave {

	private String cadenaSeguridad;

	public String getCadenaSeguridad() {
		return cadenaSeguridad;
	}

	public void setCadenaSeguridad(String cadenaSeguridad) {
		this.cadenaSeguridad = cadenaSeguridad;
```

}

Usuario:

```
public class Usuario {

	private int id;
	private String nombre;
	private String direccion;
	private int situacion = 3;
	private Clave clave;


	public void entrar(){

	}

	public void salir()	{

	}

	public void trabajar(){

	}

	public int getId() {
		return id;
	.
	.
	.
}
```

Dependencia:

UML:

Java:

Ventana:

```
public class Ventana
{
	private int coordenadaX;
	private int coordenadaY;


	public void abrir(){

	}

	public void cerrar(){

	}

	public int getCoordenadaX() {
		return coordenadaX;
	}
	public void setCoordenadaX(int coordenadaX) {
		this.coordenadaX = coordenadaX;
	}
	public int getCoordenadaY() {
		return coordenadaY;
	}
	public void setCoordenadaY(int coordenadaY) {
		this.coordenadaY = coordenadaY;
	}



}
```

Aplicación:

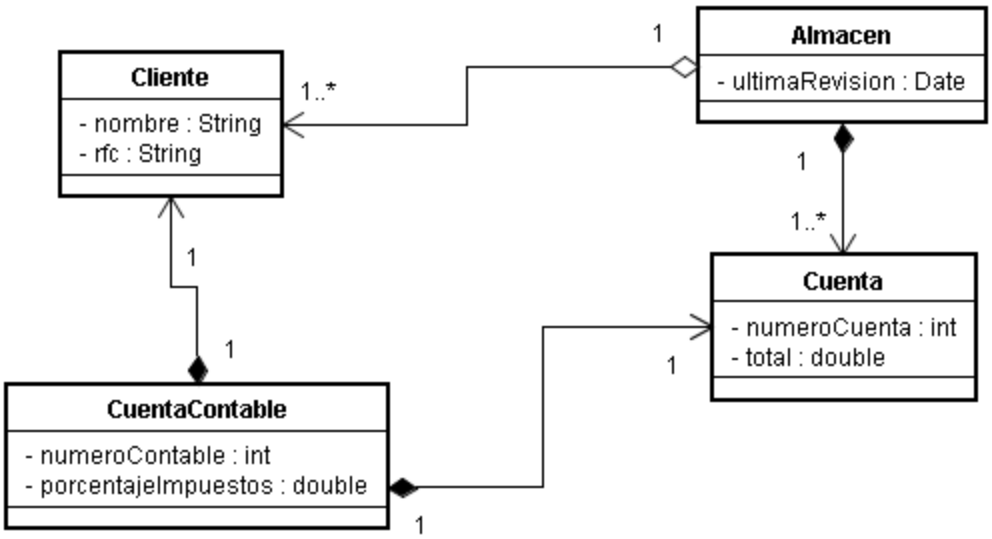
```
public class Aplicacion {

	private int medidaX;
	private int medidaY;

	public void crearVentana(){

		Ventana ventana = new Ventana();
		ventana.abrir();
	}

	public void destruirVentana(){
		Ventana ventana = new Ventana ();
		ventana.cerrar();
	}

	public int getMedidaX() {
```

```
		return medidaX;
	}
	public void setMedidaX(int medidaX) {
		this.medidaX = medidaX;
	}
	public int getMedidaY() {
		return medidaY;
	}
	public void setMedidaY(int medidaY) {
		this.medidaY = medidaY;
	}
```

}

Contención y Agregación:

UML:



Java:

Cliente:

```
public class Cliente {
```

```
	private String nombre;
	private String rfc;

	public String getNombre() {
		return nombre;
	}
	public void setNombre(String nombre) {
		this.nombre = nombre;
	}
	public String getRfc() {
		return rfc;
	}
	public void setRfc(String rfc) {
		this.rfc = rfc;
	}

}
```

Cuenta:

```
public class Cuenta {

	private int numeroCuenta;
	private double total;


	public int getNumeroCuenta() {
		return numeroCuenta;
	}
	public void setNumeroCuenta(int numeroCuenta) {
		this.numeroCuenta = numeroCuenta;
	}
	public double getTotal() {
		return total;
	}
	public void setTotal(double total) {
		this.total = total;
	}


}
```

CuentaContable:

```
public class CuentaContable {

	private int numeroContable;
```

```
	private double porcentajeImpuestos;
	private Cliente cliente;
	private Cuenta cuenta;


	public CuentaContable(int numeroContable, double porcentajeImpuestos,
			Cliente cliente, Cuenta cuenta) {

		this.numeroContable = numeroContable;
		this.porcentajeImpuestos = porcentajeImpuestos;
		this.cliente = cliente;
		this.cuenta = cuenta;
	}

	public int getNumeroContable() {
		return numeroContable;
	}
	public void setNumeroContable(int numeroContable) {
		this.numeroContable = numeroContable;
	}
	public double getPorcentajeImpuestos() {
		return porcentajeImpuestos;
	}
	public void setPorcentajeImpuestos(double porcentajeImpuestos) {
		this.porcentajeImpuestos = porcentajeImpuestos;
	}
	public Cliente getCliente() {
		return cliente;
	}
	public void setCliente(Cliente cliente) {
		this.cliente = cliente;
	}
	public Cuenta getCuenta() {
		return cuenta;
	}
	public void setCuenta(Cuenta cuenta) {
		this.cuenta = cuenta;
	}


}
```

Almacen:

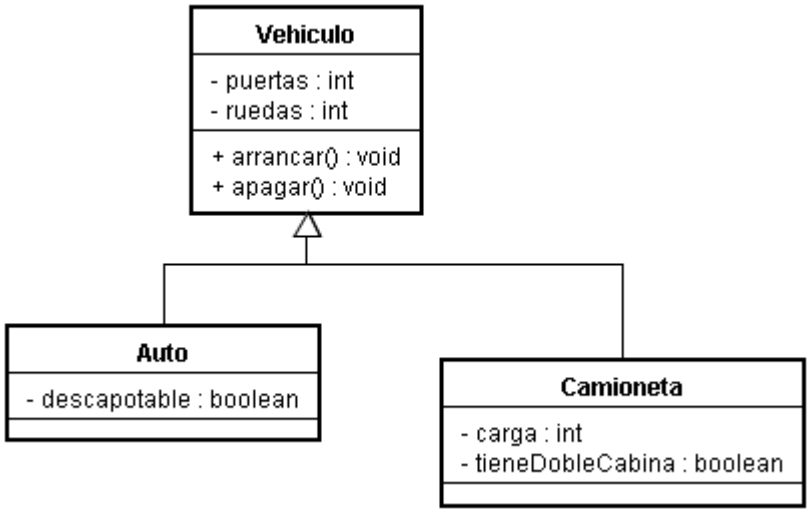
```
public class Almacen {

	private Date ultimaRevision;
	private Cliente cliente;
	private Cuenta[] cuentas;


	public Almacen(Date ultimaRevision, Cliente cliente, Cuenta[] cuentas) {
```

```
            this.ultimaRevision = ultimaRevision;
            this.cliente = cliente;
            this.cuentas = cuentas;
      }

      public Date getUltimaRevision() {
            return ultimaRevision;
      }

      public void setUltimaRevision(Date ultimaRevision) {
            this.ultimaRevision = ultimaRevision;
      }

      public Cliente getCliente() {
            return cliente;
      }

      public void setCliente(Cliente cliente) {
            this.cliente = cliente;
      }

      public Cuenta[] getCuentas() {
            return cuentas;
      }

      public void setCuentas(Cuenta[] cuentas) {
            this.cuentas = cuentas;
      }
```

}

Herencia

UML:



Java:

Vehiculo:

```
public class Vehiculo {

	private int puertas;
	private int ruedas;

	public void arrancar(){

	}

	public void apagar(){

	}

	public int getPuertas() {
		return puertas;
	}

	public void setPuertas(int puertas) {
		this.puertas = puertas;
	}

	public int getRuedas() {
		return ruedas;
	}

	public void setRuedas(int ruedas) {
		this.ruedas = ruedas;
	}

}
```

Auto:

```
public class Auto extends Vehiculo{

	private boolean descapotable;

	public boolean isDescapotable() {
		return descapotable;
	}

	public void setDescapotable(boolean descapotable) {
		this.descapotable = descapotable;
	}
```

```
}
```

Camioneta

```
public class Camioneta extends Vehiculo{

	private int carga;
	private boolean tieneDobleCabina;
	public int getCarga() {
		return carga;
	}
	public void setCarga(int carga) {
		this.carga = carga;
	}
	public boolean isTieneDobleCabina() {
		return tieneDobleCabina;
	}
	public void setTieneDobleCabina(boolean tieneDobleCabina) {
		this.tieneDobleCabina = tieneDobleCabina;
	}


}
```