UNIDAD

3

DIPLOMATURA EN PROGRAMACION JAVA

MÓDULO 1: PROGRAMACIÓN ORIENTADA A OBJETOS EN JAVA

Relaciones y Relaciones en Java

Relaciones y Relaciones en Java

Esta unidad explica Relaciones y Relaciones en Java. Se explica cómo se relacionan los objetos entre sí en Java utilizando técnicas sencillas como declaraciones de variables o vectores.

Asociaciones y enlaces

Tanto las clases como los objetos se relacionan de diferentes maneras. Cada relación depende de cómo interaccionan unos con otros. Por ejemplo, se puede declarar un objeto como atributo de una clase, pasar un objeto como parámetro a un método, devolver un objeto desde un método o declararlo como un elemento de un vector de objetos. Todas estas posibilidades dependen siempre de la capacidad del lenguaje utilizado, pero más allá de la cuestión técnica, definen una relación en la cual se establece como interactúan dos objetos del tipo de las clases a las que pertenecen los miembros de dicha relación.
Por eso, se puede definir a toda relación entre clases y objetos como una asociación que puede ser diagramada en UML. Para indicar que dos clases u objetos están relacionados por medio de una asociación, se diagrama una línea que une a ambos a la cual se denomina enlace. Resumiendo
Asociación:

- Se refiere a una relación representada por una línea en un diagrama de colaboración de clases u objetos
- La línea puede ser horizontal o vertical
- La línea de la relación posee un nombre que describe la relación

Enlace:

- Se refiere a la relación mostrada en un diagrama de clases u objetos entre dos de ellos

Asociaciones

La representación de las asociaciones y sus enlaces en un diagrama de objetos es como se presenta a continuación

En el caso de los objetos, el diagrama puede ser calificado o no. Esto quiere decir, si se conoce el nombre del objeto se lo coloca antecediendo al nombre de la clase que define su tipo, pero si no se posee dicha información al momento de realizar el diagrama, este se puede hacer poniendo sólo el nombre de la clase antecediéndolo con dos puntos y subrayándolo, como muestra el diagrama anterior. El siguiente ejemplo muestra cuando se determino inclusive el nombre de los objetos

El nombre que se ve en el enlace es una frase verbal que define el tipo de asociación que se refleja en el enlace. Así, por ejemplo, este último diagrama puede leerse:

Juan, que es del tipo Persona, posee un Fiat que es del tipo Automóvil

Relaciones entre clases

Los diagramas de clases muestran los elementos definidos dentro de ellas total o parcialmente, pero no indican como interaccionan con otras, ya sea desde el punto de vista del diseño como en tiempo de ejecución.

Se pueden diagramar las interacciones entre clases desde dos puntos de vista, el estático y el dinámico. El punto de vista estático esta asociado con la descripción del tipo de interacción que se define entre dos clases mientras que, el dinámico, muestra el comportamiento de los objetos del tipo de dichas clases cuando se utiliza esa relación en tiempo de ejecución.

A los diagramas estáticos se los denomina diagramas de clases, mientras que a los dinámicos se los denomina diagramas de objetos.

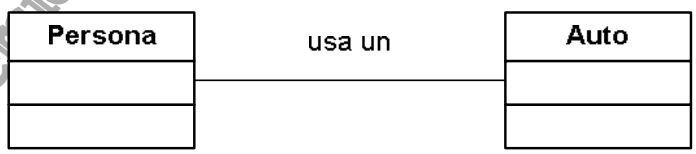
Las relaciones entre clases definen una asociación y se las puede clasificar según el tipo de interacción que definen. Los tipos que definen la mayoría de los lenguajes orientados a objetos son:

- Asociaciones simples
- Agregaciones
- Composiciones
- Herencia

Asociaciones simples

El primer paso para representar la relación que existe entre dos clases, es dibujar una línea entre ellas con el nombre que se le asigna a esta. La sola existencia de una línea entre dos clases define una asociación. Dentro de las posibles asociaciones existen las más débiles llamadas asociaciones simples

Para detectar una asociación simple entre dos clases u objetos se utilizan las palabras "**usa un**" o "**usa una**". Por ejemplo, "una persona usa un auto". Partiendo de este ejemplo, se puede realizar un diagrama como el que muestra la figura:



Este tipo de asociaciones muestran la interacción entre dos clases y definen su comportamiento. Es claro que la relación entre la persona y el auto es temporal e independiente de la vida de los objetos involucrados. Por ejemplo, el hecho que la "persona use el auto" no implica que la persona cree el auto para usarlo.

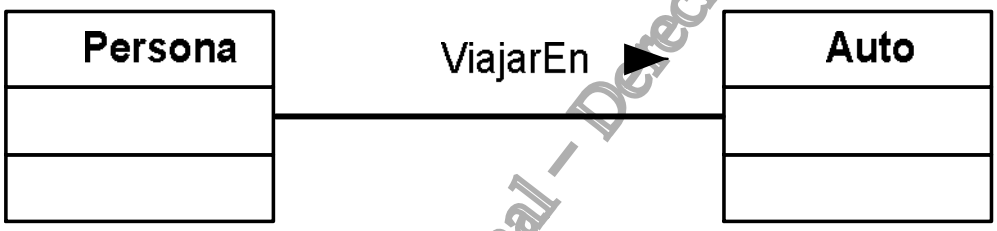
La asociación es inherentemente bidireccional, lo cual significa que la puede crear de cualquiera de las clases que intervienen para definir un recorrido "lógico" en ambas direcciones.

Nota: El recorrido es puramente abstracto, **no** se trata de una sentencia sobre conexiones entre entidades de software.

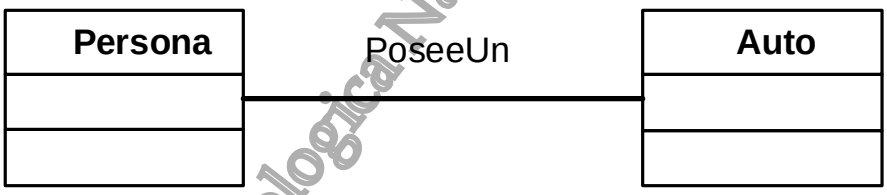
Una "flecha de dirección de lectura" opcional indica la dirección de cómo leer el nombre de la asociación. NO indica ningún tipo de visibilidad o navegación. Si no esta presente, por convención se lee de izquierda a derecha o de arriba hacia abajo.

Nota: la flecha de dirección de lectura no tiene significado en términos del modelo, sólo es una ayuda para quien lee el diagrama.

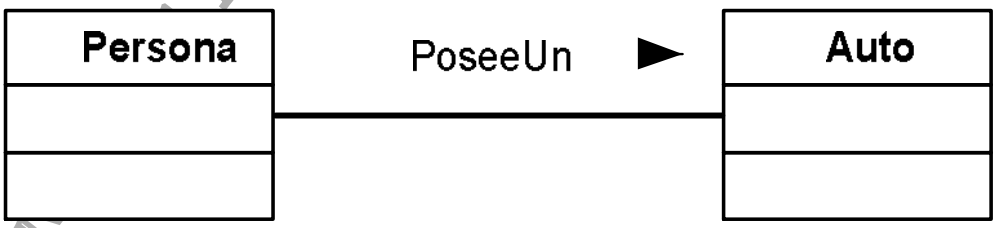
Por otro lado, se le puede asignar a una relación un nombre y una dirección. Por ejemplo, si para definir la interacción entre dos clases se enuncia que: una persona usa un auto para viajar, el diagrama descriptivo podría ser:



En este caso, la asociación simple se define como "viajarEn". La dirección de la asociación indica la forma en que se relacionan estas clases, NO la navegabilidad.

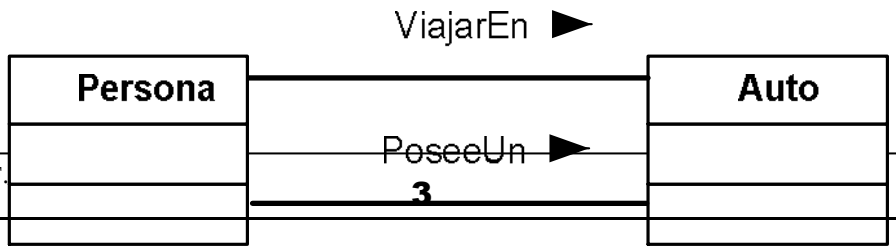


Dos clases pueden tener más de una asociación, por ejemplo para las mismas clases si se quiere representar que la persona posee un auto (siguiendo la convención), el diagrama

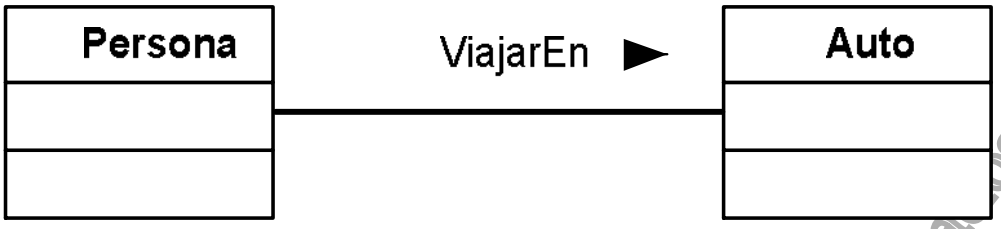


seria:

El cual deberá leerse según la convención como si el diagrama fuera:
Como en el caso anterior se puede observar dos asociaciones entre las mismas clases,



ambas se pueden colocar en un mismo gráfico, como muestra la figura:



Asignación de nombres en las asociaciones

Para asignar un nombre a una asociación, se debe seguir la siguiente guía:

NombreTipo ***FraseVerbal*** NombreTipo

Donde frase verbal es un nombre que califica la relación, en este caso los nombres utilizados son:

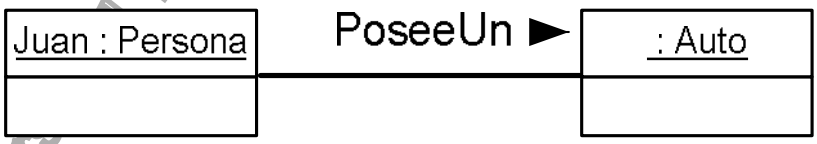
- ViajarEn
- PoseeUn

Otro formato válido es

- Viajar-en
- Posee-un

Cuando se diagrama una asociación en tiempo de ejecución (diagrama dinámico) se diferencia del diagrama estático porque el nombre del objeto antecede al de la clase y se los separa con dos puntos. Ambos nombres se colocan subrayados. Si no se conociera el nombre del objeto cuando se realiza el diagrama, sólo se colocan los dos puntos y el nombre de la clase. Ambas situaciones se muestran en el diagrama:

O también:



Roles

Cada extremo de una asociación es un rol. Los nombres de rol proporcionan la herramienta para identificar cada extremo de una relación y son los que sirven para navegarla en caso de requerirlo. El nombre deberá ser único en ambos lados para diferenciar cada extremo de la asociación y por lo general es el mismo nombre de la variable de referencia que se utiliza en un lenguaje de programación para declarar la relación.

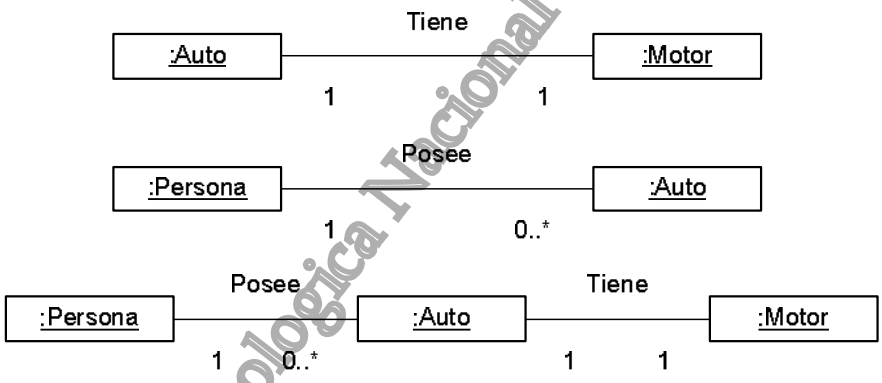
Si en el diagrama existe más de una asociación, todos los nombres de roles en cada una de las que se muestren deberán ser diferentes para no crear ambigüedades.
A pesar de todas estas restricciones, los nombres de roles son opcionales y pueden ser omitidos.
Los roles pueden tener, opcionalmente:

- Nombre
- Expresión de multiplicidad
- Navegabilidad
- Visibilidad

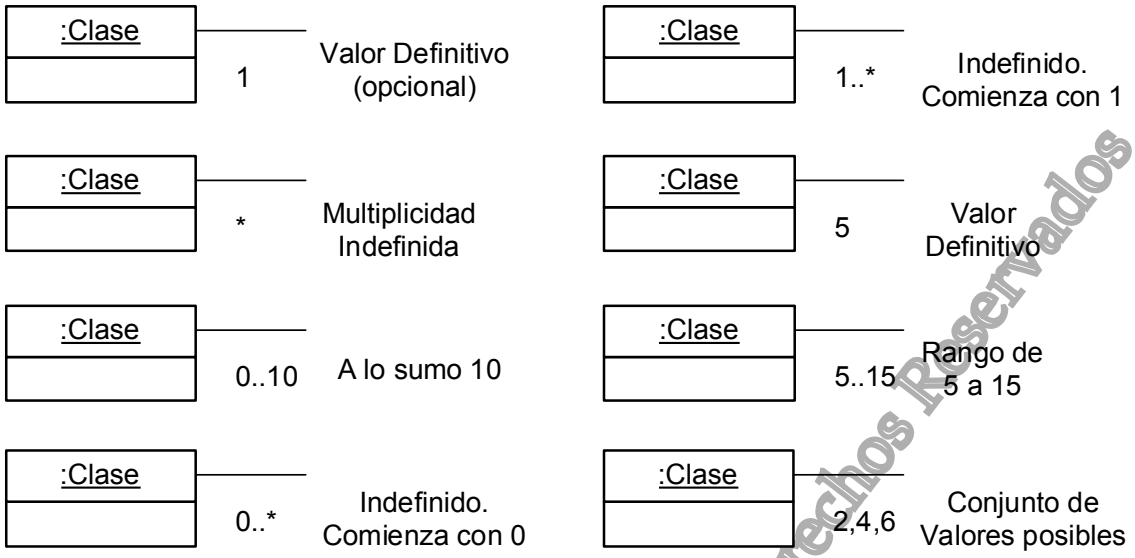
Pueden especificarse las cuatro opciones, ninguna, o cualquier combinación de ellas.

Asociaciones y Multiplicidad

La multiplicidad define cuantas instancias de una clase, por ejemplo, A pueden asociarse con una instancia de otra clase B.
Las multiplicidades se agregan en un extremo de la relación puesto que definen en si misma un rol, a pesar que este no tenga nombre.
En la siguiente figura se pueden apreciar varias asociaciones que definen distinta multiplicidades



De esta manera, el diagrama especifica que existe una asociación llamada “tiene” entre Auto y Motor del tipo “Auto-tiene-Motor” donde por cada instancia de Auto existe una instancia de Motor.
En cada rol se pueden definir distintas multiplicidades y sus significados son como los que muestra la siguiente figura:



Ejemplo

Para el siguiente ejemplo se presentará un análisis básico del enunciado de un problema y el posterior análisis de objetos candidatos para realizar un diagrama de objetos. El fin de este ejemplo es exponer parte del diseño de solución pero no así todo el proceso involucrado en un análisis y diseño orientado a objetos

Enunciado del problema:

Se debe manejar las asignaciones de cursos en un instituto de enseñanza informando a instructores y alumnos las asignaciones de curso que tienen designadas. Se cuenta, para esto, con las direcciones de ambos con los datos completos. Cada alumno puede ser asignado como mínimo a un curso y los cursos deben tener un mínimo de tres alumnos para que se coloquen en el calendario y comiencen

Por otra parte, los cursos son asignados según un calendario en el cual pueden tener más de un horario (el mismo curso en diferentes horarios). Cada curso es asignado a un aula las cuales tiene diferentes cantidades de escritorios, sillas, pizarrones y computadoras.

Determinación del contexto del problema

El problema a resolver sólo tiene en cuenta el manejo para las asignaciones de cursos, aulas, alumnos e instructores. Todo manejo más allá de esto se analiza en la resolución de otros problemas enunciados.

Lista de objetos candidatos:

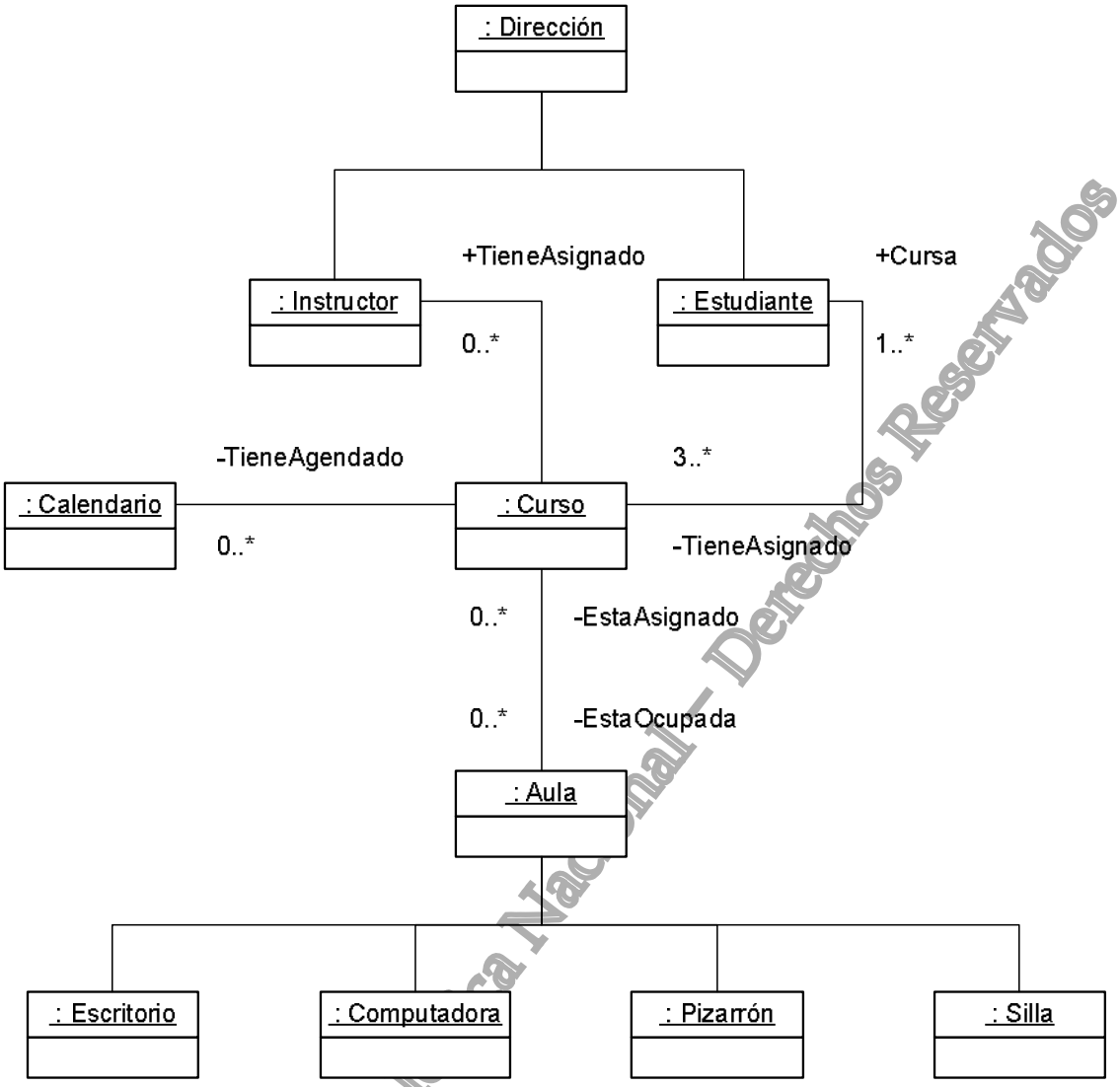
- Instituto de enseñanza
- Instructor
- Alumno
- Curso
- Calendario
- Horario
- Aula

- Escritorio
- Silla
- Pizarrón
- Computadora

Análisis de los objetos encontrados

Objetos Candidatos	Descripción	Estado	Motivo
Instituto de enseñanza	Es la entidad en donde se dictan los cursos a asignar	Descartado	Esta fuera del contexto las particularidades del Instituto
Instructor	Es quien dicta el curso	Activo	Es parte del contexto
Alumno	Es quien asiste al curso	Activo	Es parte del contexto
Dirección	Lugar de residencia	Descartado	Es un atributo
Curso	Es la materia a dictar	Activo	Es parte del contexto
Calendario	Es la organización de horarios para el dictado de los cursos	Activo	Es parte del contexto
Horario	Es el tiempo en el cual se pueden asignar un o más cursos	Descartado	Es el atributo del calendario que define el rango de validez del mismo
Aula	Es el lugar físico donde se dictan los cursos	Activo	Es parte del contexto
Escritorio	Es donde trabajan los alumnos y el instructor	Activo	Es parte del contexto
Silla	Es donde se sientan los alumnos y el instructor	Activo	Es parte del contexto
Pizarrón	Es la herramienta sobre la que se escribe para la clase cuando es necesario	Activo	Es parte del contexto
Computadora	Es la herramienta utilizada para las prácticas y o seguimientos de las clases por los alumnos e instructores	Activo	Es parte del contexto

Diagrama de colaboración entre objetos



Ejercicio 1

Los temas de los que se tratan en este ejercicio son los siguientes:

- Asociaciones
- Multiplicidad

Dado un enunciado de problema, se debe realizar un análisis y diseño (preliminar) en base a este que refleje asociaciones y multiplicidad

Asociaciones complejas

Existen situaciones en las cuales las clases se relacionan con otras por medio de multiplicidades en ambos roles de la asociación que no se pueden determinar en el momento del diseño. Una causa puede ser que en ambos lados de la asociación pueden existir múltiples instancias de objetos del tipo de la clase que interviene en la relación.
En lo diagramas UML, se identifican cuando los demarcadores de la multiplicidad no definida (el "*") aparece en ambos lados de la relación (en términos de bases de datos, son relaciones de muchos a muchos).
Este tipo de situaciones requieren una forma de manejar la asociación de manera que pueda ser fácilmente identificada cada relación. Para ello, se descompone una relación de muchos a muchos en dos relaciones, una de muchos a uno y otra de uno a muchos.
Sin embargo, en el medio de esta descomposición debe existir un mecanismo que las gestione. Este último presenta un nuevo problema que se puede resolver de dos maneras diferentes. Una de ellas es utilizando una clase de asociación, la cual gestionará ambos extremos de cada relación uno a muchos. La otra, como se verá posteriormente, se llama asociación calificada.
Cuando existe una relación del tipo:

> *Una persona puede reservar muchos libros en una biblioteca. A la vez, un mismo libro puede ser reservado por muchas personas.*

Se identifica claramente una asociación de "muchos a muchos", la cual se puede diagramar como muestra la siguiente figura:

Se debe notar que en cada rol de la asociación se identifica la multiplicidad con un "*".

Clase de asociación

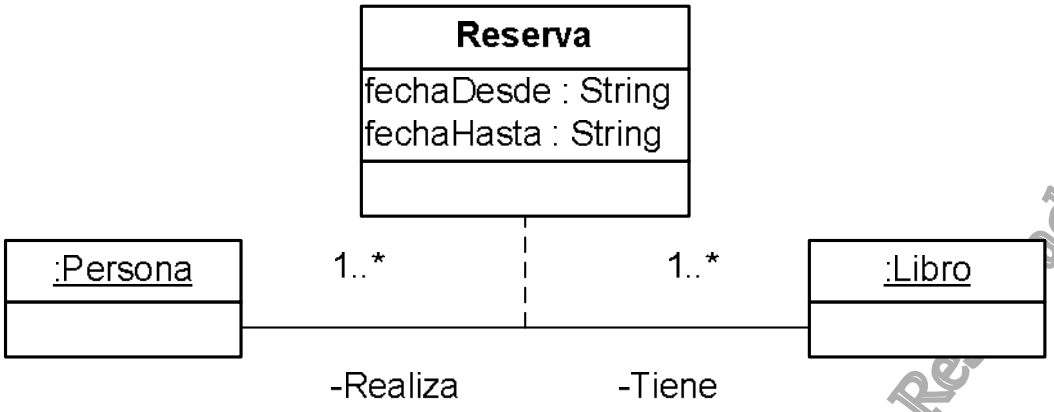
Como se mencionó anteriormente, cuando aparecen asociaciones complejas, se necesita un mecanismo que las maneje. Una técnica para manejar asociaciones complejas es intercalar en la relación, una clase dedicada a manejarla. De esta manera, la clase se utiliza para resolver las relaciones de muchos a muchos.

Sin embargo, la existencia de una clase de este tipo es dedicada a resolver la relación, por lo tanto sus métodos y atributos están dedicados a resolver el conflicto y es una buena práctica de diseño que este sea su único fin.
Volviendo al enunciado previo del problema de diseño a resolver:

> *Una persona puede reservar muchos libros en una biblioteca. A la vez, un mismo libro puede ser reservado por muchas personas.*

La primera solución dada por el diagrama

Se puede reformular como:



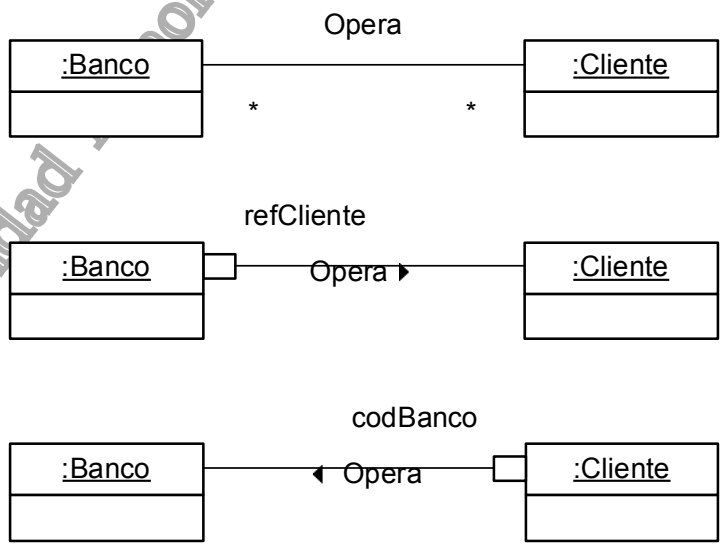
Lo cual permite el manejo de la relación.

Asociación calificada

Las clases de asociación no son la única técnica que existe para resolver este tipo de asociaciones. Por ejemplo, se puede tener un elemento que pertenezca a una clase que realice la misma función que la clase de asociación un extremo de la relación para cada clase. La salvedad, es que el rol en cada extremo es más complejo que un rol común y se define y diagrama diferente.
Un ejemplo en la codificación, es si la clase tiene un vector o una colección de objetos entre sus atributos para manejar la relación.
El atributo deberá almacenar o tener la capacidad de referenciar los elementos individuales de la relación compleja.
Suponiendo el siguiente ejemplo:

> *Un banco opera con muchos clientes, pero estos, a su vez, pueden operar con muchos bancos*

Una solución posible es la que se muestra en el diagrama



Los atributos definidos, refCliente y codBanco, manejan múltiples instancias de la clase con la que se relaciona. Estos atributos almacenan la referencia pero la clase que los posee debe proveer un mecanismo para acceder a dichos elementos.

Ejercicio 2

El temas del que trata este ejercicio es el siguiente:

- Asociaciones Complejas

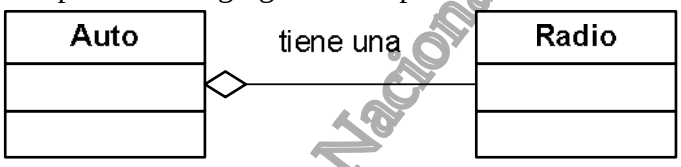
En base al diagrama realizado en el ejercicio anterior, analizar las relaciones complejas y proponer un diseño adecuado

Agregaciones

Es una relación más fuerte entre clases. Por lo general, cuando se enuncia el problema de diseño se puede reconocer fácilmente una agregación porque se caracteriza por las palabras "**tiene un**" o "**tiene una**". A esta frase se la llama "frase de validación" y es la que la determina.
Otra forma de reconocer las asociaciones es la relación "totalidad – parte de". Por ejemplo, se puede definir como problema de diseño que:

Un auto tiene una radio

El auto es la totalidad y la radio es una parte que se reconoce como un objeto independiente. Sin embargo, es claro que aunque la relación puede interpretarse como "totalidad – parte de", esto no implica que ambos objetos mantengan su relación durante todo el tiempo que cada uno este activo (por ejemplo, un auto puede cambiar el algún momento la radio que tiene), aunque también, la relación puede durar la vida del objeto (por ejemplo, un auto nunca cambia la radio que tiene aunque pueda).
Los diagramas que representan las agregaciones se dibujan con un rombo en blanco del lado de la clase que representa la "totalidad" y como es un tipo de relación, todo lo expuesto para las relaciones anteriormente (nombres, enlaces, roles, multiplicidad) es válido también para ellas.
Un diagrama que representa la agregación del problema de diseño antes enunciado, es el siguiente:

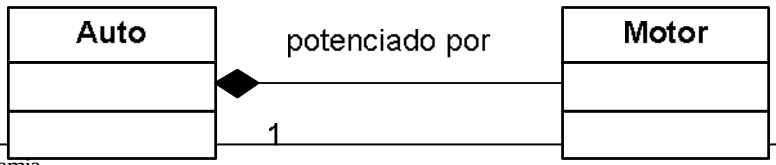


Composiciones

Es la relación más fuerte entre clases, ya que cuando una clase compone un objeto del tipo de otra, el problema de diseño indica que para resolverlo, una clase no tiene sentido sin la instancia del objeto de la clase que compone.
La frase de validación se caracteriza por las palabras "**siempre tiene**". Por ejemplo, un problema de diseño donde se puede apreciar lo afirmado es el siguiente:

Un auto siempre tiene un motor
Una factura de siempre tiene detalles de facturación

Las composiciones tiene como característica que duran el mismo tiempo que la vida del objeto siempre, ya que si faltase el objeto compuesto, el que lo compone no tiene sentido. Por ejemplo, no tiene sentido tratar de diseñar un auto sin su motor o una factura sin detalles de facturación (en este caso la composición también incluye una asociación compleja).
Los diagramas que representan las composiciones se dibujan con un rombo en negro del lado de la clase que representa la "totalidad" y como es un tipo de relación, todo lo expuesto para las relaciones anteriormente (nombres, enlaces, roles, multiplicidad) es válido también para ellas.
Un diagrama que representa la composición del problema de diseño antes enunciado, es el siguiente:



Vectores

Declaración de vectores

Los vectores en Java, al igual que en cualquier lenguaje, son un grupo de variables del mismo tipo agrupadas por un único nombre.
En Java, los vectores pueden contener entre sus elementos tipos primitivos o referencias a objetos, por ejemplo:

```
char s[];
Point p[];
char[] s;
Point[] p;
```

Sin embargo, en Java, cuando se crea un vector, este no se aloja en memoria hasta crearlo con el operador **new** y es, hasta ese momento, la declaración de una referencia a un objeto de tipo vector. Por lo tanto, la declaración en si misma no crea otra cosa que un espacio en memoria para alojar una referencia.
En Java los vectores funcionan como una clase (esto es porque internamente lo es), de la cual se debe crear una instancia con el operador **new**, como cualquier objeto.

Creación de vectores

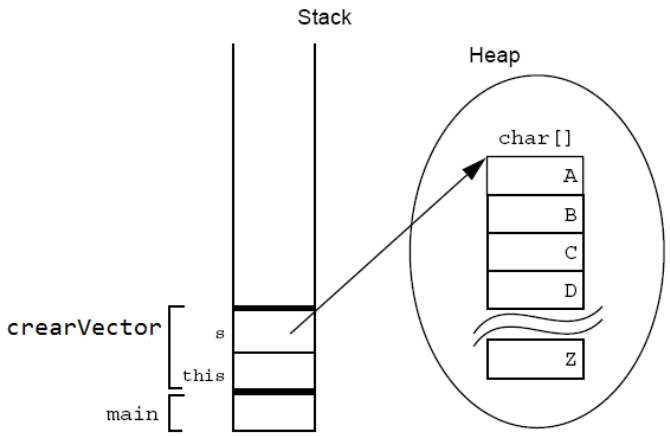
Como se mencionó anteriormente, se puede crear un vector con tipos primitivos. Para esto se debe indicar a continuación del operador **new** el tipo primitivo a utilizar y seguido de éste, entre corchetes, la cantidad de elementos que poseerá el vector.
Para acceder a cada elemento del vector se utiliza el nombre del mismo con un subíndice, el cual puede ser una variable o una constante, que indica el elemento del vector con el que se quiere operar.
El siguiente ejemplo ilustra lo expuesto

```
public char[] crearVector() {
        char[] s;
        s = new char[26];
        s[0] = (char) ('A');
        s[1] = (char) ('B');

        :
        :

        s[22] = (char) ('W');
        s[23] = (char) ('X');
        s[24] = (char) ('Y');
        s[25] = (char) ('Z');
        return s;
}
```

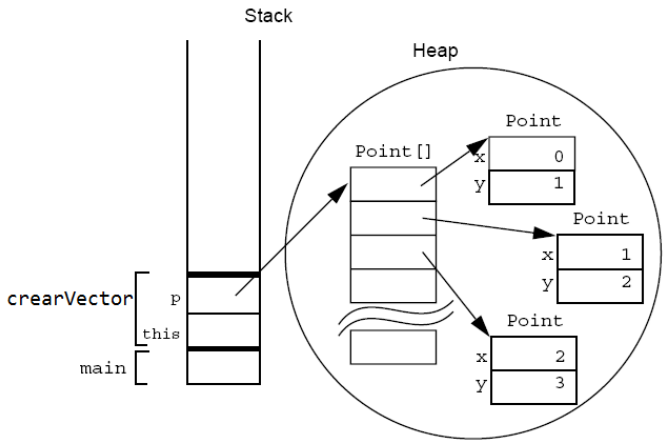
Tener en cuenta que la referencia a un vector puede utilizarse como valor retornado por un método
La forma en que se aloja en memoria el vector lo muestra la siguiente figura



Notar que el contenido del vector se aloja en el heap, como los atributos de cualquier objeto.
También existen los vectores de objetos. Si bien el manejo es similar, la comprensión del concepto es un poco más confusa.
Cuando se crea un vector de objetos, al igual que un vector de tipos primitivos, se crea una referencia. Sin embargo, cuando se genera el espacio para almacenar los elementos del vector, cada uno de estos es a la vez "una referencia a objetos del tipo del cual fue declarado el vector".
Por lo tanto, por cada elemento del vector, se deberá crear un objeto cuya referencia se guardará en dicho elemento. El siguiente código muestra la operatoria de creación del vector. Para crear cada elemento se utilizará una clase de Java llamada Point que recibe dos parámetros en el constructor

```
public Point[] crearVector() {
        Point[] p;
        p = new Point[10];
        p[0] = new Point(0, 1);
        p[1] = new Point(1, 2);
        p[2] = new Point(8, 9);
        p[3] = new Point(2, 1);
        p[4] = new Point(3, 6);
        p[5] = new Point(4, 5);
        p[6] = new Point(5, 4);
        p[7] = new Point(6, 3);
        p[8] = new Point(7, 2);
        p[9] = new Point(10, 1);
        return p;
}
```

El siguiente gráfico ilustra los lugares de almacenamiento para este vector de objetos



Inicialización de vectores

A los vectores, tanto de objetos como de tipos primitivos, se les puede asignar valores elemento a elemento o en bloque. Cuando es en bloque, se lo denomina "bloque de inicialización", se encierra entre llaves los elementos que determinan el valor inicial de cada elemento del vector y el tamaño de este se ajusta automáticamente a la cantidad de elementos declarados en el bloque.
Se debe tener en cuenta que cuando se inicializan vectores de objetos se lo hace mediante su constructor **siempre**.
Como ejemplo de asignación de valores iniciales elemento a elemento, se muestra el siguiente código.

```
String nombres[];
nombres = new String[3];
nombres[0] = "Silvina";
nombres[1] = "Silvana";
nombres[2] = "Silvia";

Fechas fechas[];
fechas = new Fechas[3];
fechas[0] = new Fechas(22, 7, 1964);
fechas[1] = new Fechas(1, 1, 2000);
fechas[2] = new Fechas(22, 12, 1964);
```

~~el~~El siguiente código se muestra como ejemplo de declaración con un bloque de asignación,

```
String nombres2[] = {
                    "Alejo",
                    "Alexis",
                    "Alejandro"
                    };

Fechas fechas2[] = {
                    new Fechas(2, 5, 1963),
                    new Fechas(12, 2, 2012),
                    new Fechas(13, 8, 1963)
                    };
```

Asociaciones y enlaces

Para definir una relación en Java se puede declarar un objeto dentro de una clase, pasar un objeto como parámetro a un método, devolver un objeto desde un método o declararlo como un elemento de un vector de objetos.
Salvo un tipo de relación un poco más compleja, la herencia, el resto se explicará en este módulo.

Asociaciones simples

Están definidas cada vez que dos clases interaccionan entre si. Este tipo de asociaciones es la más débil de todas y por lo general se representa en el código como una variable auxiliar, un parámetro de una función o cualquier tipo de referencia que una tenga sobre la otra de manera que no cree mayor dependencia entre ellas más que por un lapso pequeño de tiempo.
La frase de verificación para una asociación simple es “usa un” o “usa una” y la relación siempre es temporal e independiente de la vida de la utiliza el servicio de la otra.
Por ejemplo, se puede suponer el siguiente problema de diseño:

Una persona usa un auto de la compañía cuando lo requiere para visitar clientes

La frase de verificación indica claramente una asociación simple entre la clase Persona y la clase Auto. El diagrama que se puede generar a partir del enunciado es el siguiente

En UML cuando se omite la multiplicidad se debe interpretar la relación de izquierda a derecha y uno a uno.
El código que se expone a continuación resuelve el problema de diseño

Ejemplo

Clase Auto.java

```
public class Auto {
	private String marca;
	private String modelo;
	private String anio;

	public Auto(String ma, String mo, String an) {
		marca = ma;
		modelo = mo;
		anio = an;
	}

	public String getAnio() {
		return anio;
	}

	public void setAnio(String an) {
		anio = an;
	}
```

```
	public String getMarca() {
		return marca;
	}

	public void setMarca(String ma) {
		marca = ma;
	}

	public String getModelo() {
		return modelo;
	}

	public void setModelo(String mo) {
		modelo = mo;
	}
}
```

Clase Persona.java

```
public class Persona {
	private String primerNombre;
	private String segundoNombre;
	private String apellido;
	private String documento;

	public Persona(String p,
	String s,
	String a,
	String d) {
		primerNombre = p;
		segundoNombre = s;
		apellido = a;
		documento = d;
	}

	public String getApellido() {
		return apellido;
	}

	public String getDocumento() {
		return documento;
	}

	public String getPrimerNombre() {
		return primerNombre;
	}

	public String getSegundoNombre() {
		return segundoNombre;
	}

	public void setApellido(String string) {
		apellido = string;
	}

	public void setDocumento(String string) {
```

```
            documento = string;
      }


      public void setPrimerNombre(String string) {
            primerNombre = string;
      }

      public void setSegundoNombre(String string) {
            segundoNombre = string;
      }


      public void usaAuto(Auto a) {
            System.out.println("La persona esta usando un " + a.getMarca()
                  + " modelo " + a.getModelo() + " del año " + a.getAnio());
      }
}
```

Clase UsaAsociacion.java

```
public class UsaAsociacion {

      public static void main(String[] args) {
            Auto a = new Auto("Fiat", "Uno","1998");
            Persona p = new Persona("Juan", "Ignacio",
                  "Perez","21111222");

            p.usaAuto(a);

      }
}
```

Asociaciones y multiplicidad

En el presente curso, por razones de simplicidad en la codificación, las multiplicidades superiores a 1 se representarán como vectores y se interaccionará con ellos elemento a elemento, sin recorrerlos.
La razón de esto es focalizar en los conceptos de la programación orientada a objetos sin entrar en los detalles de codificación propios de Java, lo cuales se verán en otro curso.

Ejercicio 1

Los temas de los que se tratan en este ejercicio son los siguientes:

- Asociaciones
- Multiplicidad

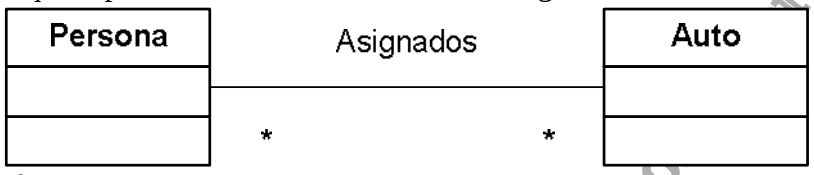
Basándose en el diseño realizado en el módulo anterior, codificar en Java las asociaciones con su respectiva multiplicidad

Asociaciones complejas

Existen muchas situaciones en las cuales se crean relaciones de muchos a muchos en el diseño de clases. Por ejemplo, se puede suponer el siguiente problema a resolver:

> *Una persona puede tener asignados para su uso varios autos de la compañía. Los mismos son compartidos por varias personas, por lo tanto, un auto puede estar asignado a varias personas a la vez.*

El diagrama UML que representaría este enunciado es el siguiente:



Existen distintas técnicas y herramientas para resolver este tipo de relaciones, tanto de diseño como de codificación.

Por simplificación y para minimizar la cantidad de lenguaje necesario para aprender estos conceptos, el lado "muchos" de la relación se manejará con un vector cuando fuese necesario a lo largo de todas las explicaciones. Otra posibilidad son las colecciones, pero es un tema fuera del alcance de este curso.

En cuanto al diseño, las posibilidades son dos:

- Clases de asociación
- Asociaciones calificadas

Clase de asociación

Es evidente que el problema plantea la necesidad de utilizar alguna técnica de diseño o codificación para resolver la relación entre clases.

Una posible solución es:

> *Crear una clase que maneje por un lado la relación de una a muchos entre cada persona y sus posibles autos asignados y por el otro las personas asignadas a cada auto*

El siguiente código plantea una de las posibles soluciones

Ejemplo

Clase Asignados.java

```
public class Asignados {
	private Persona refPersona;
	private Auto refAuto;
	private Persona personas[];
	private Auto autos[];
	private int i;
	private int j;

	public Asignados(Persona p){
		refPersona = p;
		autos = new Auto[10];
		i = 0;
	}
```

```
	public Asignados(Auto a){
		refAuto = a;
		personas = new Persona[10];
		j = 0;
	}

	public void setAutos(Auto auto) {
		autos[i] = auto;
		i = i + 1;
	}

	public void setPersonas(Persona persona) {
		personas[j] = persona;
		j = j + 1;
	}

	public Auto getAuto(int i) {
		return autos[i];
	}

	public Persona getPersona(int i) {
		return personas[i];
	}

}
```

Clase Auto.java

```
public class Auto {
	private String marca;
	private String modelo;
	private String anio;
	private Persona refPersona[] = new Persona[10];

	public Auto(String ma, String mo, String an) {
		marca = ma;
		modelo = mo;
		anio = an;
	}


	public String getAnio() {
		return anio;
	}

	public void setAnio(String an) {
		anio = an;
	}

	public String getMarca() {
		return marca;
	}

	public void setMarca(String ma) {
		marca = ma;
```

```
	}

	public String getModelo() {
		return modelo;
	}

	public void setModelo(String mo) {
		modelo = mo;
	}

	public Persona getRefPersona(int i) {
		return refPersona[i];
	}

	public void setRefPersona(Persona persona, int i) {
		refPersona[i] = persona;
	}

}
```

Clase Persona.java

```
public class Persona {
	private String primerNombre;
	private String segundoNombre;
	private String apellido;
	private String documento;
	private Auto auto;

	public Persona(String p,
	String s,
	String a,
	String d) {
		primerNombre = p;
		segundoNombre = s;
		apellido = a;
		documento = d;
	}

	public String getApellido() {
		return apellido;
	}

	public String getDocumento() {
		return documento;
	}

	public String getPrimerNombre() {
		return primerNombre;
	}
	public String getSegundoNombre() {
		return segundoNombre;
	}

	public Auto getAuto() {
		return auto;
```

```
	}

	public void setApellido(String string) {
		apellido = string;
	}

	public void setDocumento(String string) {
		documento = string;
	}

	public void setPrimerNombre(String string) {
		primerNombre = string;
	}

	public void setSegundoNombre(String string) {
		segundoNombre = string;
	}

	public void setAuto(Auto auto) {
		this.auto = auto;
	}
}
```

Clase UsaClaseAsociacion.java

```
public class UsaClaseAsociacion {

	public static void main(String[] args) {
		Persona p1 = new Persona("Juan", "Ignacio", "Perez","21111222");
		Persona p2 = new Persona("Pedro", "Alberto", "Almeida","23222111");
		Persona p3 = new Persona("Carlos", "Alejo", "Garcia","21444555");
		Persona p4 = new Persona("Sergio", "Ricardo", "Burr","21666777");

		Auto a1 = new Auto("Fiat", "Uno","1996");
		Auto a2 = new Auto("Fiat", "Dos","1997");
		Auto a3 = new Auto("Fiat", "Tres","1998");
		Auto a4 = new Auto("Fiat", "Cuatro","1999");
		Auto a5 = new Auto("Fiat", "Cuatro","2000");

		Asignados ap1 = new Asignados(p1);
		Asignados ap2 = new Asignados(p2);
		Asignados ap3 = new Asignados(p3);
		Asignados ap4 = new Asignados(p4);

		Asignados aa1 = new Asignados(a1);
		Asignados aa2 = new Asignados(a2);
		Asignados aa3 = new Asignados(a3);
		Asignados aa4 = new Asignados(a4);
		Asignados aa5 = new Asignados(a5);

		//Asignar los autos de la persona 1
		ap1.setAutos(a1);
		ap1.setAutos(a3);
		// Registrar que el auto fue asignado a
		// la persona 1
		aa1.setPersonas(p1);
```

```
aa3.setPersonas(p1);

//Asignar los autos de la persona 2
ap2.setAutos(a2);
ap2.setAutos(a4);
// Registrar que el auto fue asignado a
// la persona 2
aa2.setPersonas(p2);
aa4.setPersonas(p2);

//Asignar los autos de la persona 3
ap3.setAutos(a1);
ap3.setAutos(a5);
// Registrar que el auto fue asignado a
// la persona 3
aa1.setPersonas(p3);
aa5.setPersonas(p3);

//Asignar los autos de la persona 4
ap4.setAutos(a2);
ap4.setAutos(a4);
// Registrar que el auto fue asignado a
// la persona 4
aa2.setPersonas(p4);
aa4.setPersonas(p4);

System.out.println("Autos Asignados a la Persona 1");
System.out.println("Marca: " + ap1.getAuto(0).getMarca()
      + " Modelo: " + ap1.getAuto(0).getModelo()
      + " Año: " + ap1.getAuto(0).getAnio());
System.out.println("Marca: " + ap1.getAuto(1).getMarca()
      + " Modelo: " + ap1.getAuto(1).getModelo()
      + " Año: " + ap1.getAuto(1).getAnio());
System.out.println("------------------------------");
System.out.println("Autos Asignados a la Persona 2");
System.out.println("Marca: " + ap2.getAuto(0).getMarca()
      + " Modelo: " + ap2.getAuto(0).getModelo()
      + " Año: " + ap2.getAuto(0).getAnio());
System.out.println("Marca: " + ap2.getAuto(1).getMarca()
      + " Modelo: " + ap2.getAuto(1).getModelo()
      + " Año: " + ap2.getAuto(1).getAnio());
System.out.println("------------------------------");
System.out.println("Autos Asignados a la Persona 3");
System.out.println("Marca: " + ap3.getAuto(0).getMarca()
      + " Modelo: " + ap3.getAuto(0).getModelo()
      + " Año: " + ap3.getAuto(0).getAnio());
System.out.println("Marca: " + ap3.getAuto(1).getMarca()
      + " Modelo: " + ap3.getAuto(1).getModelo()
      + " Año: " + ap3.getAuto(1).getAnio());
System.out.println("------------------------------");
System.out.println("Autos Asignados a la Persona 4");
System.out.println("Marca: " + ap4.getAuto(0).getMarca()
      + " Modelo: " + ap4.getAuto(0).getModelo()
      + " Año: " + ap4.getAuto(0).getAnio());
System.out.println("Marca: " + ap4.getAuto(1).getMarca()
```

```
                    + " Modelo: " + ap4.getAuto(1).getModelo()
                    + " Año: " + ap4.getAuto(1).getAnio());
            System.out.println("------------------------------");
            System.out.println("******************************");
            System.out.println("------------------------------");
            System.out.println("Personas Asignadas al auto 1");
            System.out.println("Nombre: " + aa1.getPersona(0).getPrimerNombre()
            + " "+ aa1.getPersona(0).getSegundoNombre()
            + " " + aa1.getPersona(0).getApellido());
            System.out.println("Nombre: " + aa1.getPersona(1).getPrimerNombre()
                        + " "+ aa1.getPersona(1).getSegundoNombre()
                        + " " + aa1.getPersona(1).getApellido());
            System.out.println("------------------------------");
            System.out.println("Personas Asignadas al auto 2");
            System.out.println("Nombre: " + aa2.getPersona(0).getPrimerNombre()
                        + " "+ aa2.getPersona(0).getSegundoNombre()
                        + " " + aa2.getPersona(0).getApellido());
            System.out.println("Nombre: " + aa2.getPersona(1).getPrimerNombre()
                        + " "+ aa2.getPersona(1).getSegundoNombre()
                        + " " + aa2.getPersona(1).getApellido());
            System.out.println("------------------------------");
            System.out.println("Personas Asignadas al auto 3");
            System.out.println("Nombre: " + aa3.getPersona(0).getPrimerNombre()
                        + " "+ aa3.getPersona(0).getSegundoNombre()
                        + " " + aa3.getPersona(0).getApellido());
            System.out.println("------------------------------");
            System.out.println("Personas Asignadas al auto 4");
            System.out.println("Nombre: " + aa4.getPersona(0).getPrimerNombre()
                        + " "+ aa4.getPersona(0).getSegundoNombre()
                        + " " + aa4.getPersona(0).getApellido());
            System.out.println("Nombre: " + aa4.getPersona(1).getPrimerNombre()
                        + " "+ aa4.getPersona(1).getSegundoNombre()
                        + " " + aa4.getPersona(1).getApellido());
            System.out.println("------------------------------");
            System.out.println("Personas Asignadas al auto 5");
            System.out.println("Nombre: " + aa5.getPersona(0).getPrimerNombre()
                        + " "+ aa5.getPersona(0).getSegundoNombre()
                        + " " + aa5.getPersona(0).getApellido());
        }
}
```

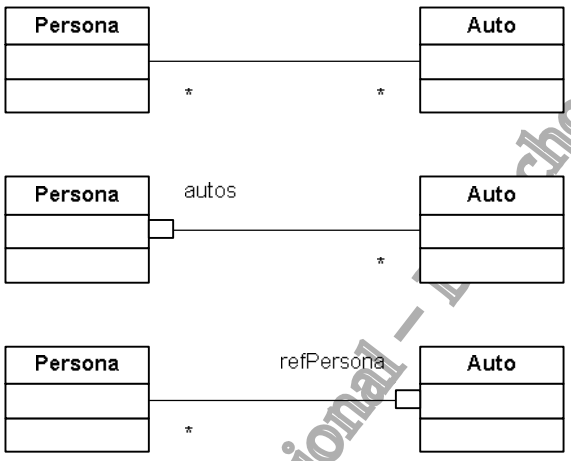
Asociación calificada

Es otra técnica para resolver asociaciones complejas
Se realiza mediante la declaración de un atributo que maneje la complejidad de la relación.
El atributo deberá almacenar o tener la capacidad de referenciar los elementos individuales de la relación compleja.
Existen diferentes técnicas para lograr este objetivo. Como en el curso se utiliza un vector para simplificar el código, la clase que lo implementa debe generar un servicio que permita interactuar con los elementos que éste almacena.
Utilizando un ejemplo similar al de la clase de asociación para que se puedan comparar ambas técnicas, se realiza el siguiente enunciado de problema

Crear una clase que maneje por un lado la relación de una a muchos entre cada persona y sus posibles autos asignados y en otra clase las personas asignadas a cada auto

Por la limitación impuesta que el manejo de la relación se encuentre para el caso de "muchos" en clases diferentes, la solución evidente es la de la asociación calificada. Las candidatas para manejar este extremo de la relación en cada caso son las mismas clases que intervienen en ella y no una diferente y aparte como en el caso de la clase de asociación. El siguiente gráfico muestra el diseño resultante

El diseño deriva en el siguiente código. Por simplicidad, se limito el tamaño del vector para



no manejar la complejidad de un posible cambio de tamaño ya que Java exige que para cambiar el tamaño de un vector, se cree uno nuevo con más elementos y se copie los que estaban almacenados en el vector original. Para peor, esto se considera una mala práctica de programación y es uno de los argumentos de la creación de colecciones

Ejemplo

Clase Auto.java

```
public class Auto {
	private String marca;
	private String modelo;
	private String anio;
	private Persona refPersona[] = new Persona[10];
	private int i;

	public Auto(String ma, String mo, String an) {
		marca = ma;
		modelo = mo;
		anio = an;
		i = 0;
	}

	public String getAnio() {
		return anio;
	}

	public void setAnio(String an) {
```

```
            anio = an;
      }

      public String getMarca() {
            return marca;
      }

      public void setMarca(String ma) {
            marca = ma;
      }

      public String getModelo() {
            return modelo;
      }
      public void setModelo(String mo) {
            modelo = mo;
      }

      public Persona getRefPersona(int p) {
            return refPersona[p];
      }

      public void setRefPersona(Persona persona) {
            refPersona[i] = persona;
            i = i + 1;
      }
}
```

Clase Persona.java

```
public class Persona {
      private String primerNombre;
      private String segundoNombre;
      private String apellido;
      private String documento;
      private Auto autos[];
      private int i;

      public Persona(String p,
      String s,
      String a,
      String d) {
            primerNombre = p;
            segundoNombre = s;
            apellido = a;
            documento = d;
            i = 0;
            autos = new Auto[10];
      }

      public String getApellido() {
            return apellido;
      }

      public String getDocumento() {
            return documento;
```

```
	}

	public String getPrimerNombre() {
		return primerNombre;
	}

	public String getSegundoNombre() {
		return segundoNombre;
	}

	public Auto getAuto(int p) {
		return autos[p];
	}

	public void setApellido(String string) {
		apellido = string;
	}

	public void setDocumento(String string) {
		documento = string;
	}

	public void setPrimerNombre(String string) {
		primerNombre = string;
	}

	public void setSegundoNombre(String string) {
		segundoNombre = string;
	}

	public void setAuto(Auto a) {
		autos[i] = a;
		i = i + 1;
	}
}
```

Clase UsaAsociacionCalificada.java

```
public class UsaAsociacionCalificada {

	public static void main(String[] args) {
		Persona p1 = new Persona("Juan", "Ignacio", "Perez","21111222");
		Persona p2 = new Persona("Pedro", "Alberto", "Almeida","23222111");
		Persona p3 = new Persona("Carlos", "Alejo", "Garcia","21444555");
		Persona p4 = new Persona("Sergio", "Ricardo", "Burr","21666777");

		Auto a1 = new Auto("Fiat", "Uno","1996");
		Auto a2 = new Auto("Fiat", "Dos","1997");
		Auto a3 = new Auto("Fiat", "Tres","1998");
		Auto a4 = new Auto("Fiat", "Cuatro","1999");
		Auto a5 = new Auto("Fiat", "Cuatro","2000");


		//Asignar los autos de la persona 1
		p1.setAuto(a1);
		p1.setAuto(a3);
		// Registrar que el auto fue asignado a
		// la persona 1
```

```
a1.setRefPersona(p1);
a3.setRefPersona(p1);

//Asignar los autos de la persona 2
p2.setAuto(a2);
p2.setAuto(a4);
// Registrar que el auto fue asignado a
// la persona 2
a2.setRefPersona(p2);
a4.setRefPersona(p2);

//Asignar los autos de la persona 3
p3.setAuto(a1);
p3.setAuto(a5);
// Registrar que el auto fue asignado a
// la persona 3
a1.setRefPersona(p3);
a5.setRefPersona(p3);

//Asignar los autos de la persona 4
p4.setAuto(a2);
p4.setAuto(a4);
// Registrar que el auto fue asignado a
// la persona 4
a2.setRefPersona(p4);
a4.setRefPersona(p4);

System.out.println("Autos Asignados a la Persona 1");
System.out.println("Marca: " + p1.getAuto(0).getMarca()
       + " Modelo: " + p1.getAuto(0).getModelo()
       + " Año: " + p1.getAuto(0).getAnio());
System.out.println("Marca: " + p1.getAuto(1).getMarca()
       + " Modelo: " + p1.getAuto(1).getModelo()
       + " Año: " + p1.getAuto(1).getAnio());
System.out.println("-----------------------------");
System.out.println("Autos Asignados a la Persona 2");
System.out.println("Marca: " + p2.getAuto(0).getMarca()
       + " Modelo: " + p2.getAuto(0).getModelo()
       + " Año: " + p2.getAuto(0).getAnio());
System.out.println("Marca: " + p2.getAuto(1).getMarca()
       + " Modelo: " + p2.getAuto(1).getModelo()
       + " Año: " + p2.getAuto(1).getAnio());
System.out.println("-----------------------------");
System.out.println("Autos Asignados a la Persona 3");
System.out.println("Marca: " + p3.getAuto(0).getMarca()
       + " Modelo: " + p3.getAuto(0).getModelo()
       + " Año: " + p3.getAuto(0).getAnio());
System.out.println("Marca: " + p3.getAuto(1).getMarca()
       + " Modelo: " + p3.getAuto(1).getModelo()
       + " Año: " + p3.getAuto(1).getAnio());
System.out.println("-----------------------------");
System.out.println("Autos Asignados a la Persona 4");
System.out.println("Marca: " + p4.getAuto(0).getMarca()
       + " Modelo: " + p4.getAuto(0).getModelo()
       + " Año: " + p4.getAuto(0).getAnio());
System.out.println("Marca: " + p4.getAuto(1).getMarca()
       + " Modelo: " + p4.getAuto(1).getModelo()
       + " Año: " + p4.getAuto(1).getAnio());
System.out.println("-----------------------------");
System.out.println("*****************************");
System.out.println("-----------------------------");
```

```
                System.out.println("Personas Asignadas al auto 1");
                System.out.println("Nombre: " + a1.getRefPersona(0).getPrimerNombre()
                        + " "+ a1.getRefPersona(0).getSegundoNombre()
                        + " " + a1.getRefPersona(0).getApellido());
                System.out.println("Nombre: " + a1.getRefPersona(1).getPrimerNombre()
                        + " "+ a1.getRefPersona(1).getSegundoNombre()
                        + " " + a1.getRefPersona(1).getApellido());
                System.out.println("-----------------------------");
                System.out.println("Personas Asignadas al auto 2");
                System.out.println("Nombre: " + a2.getRefPersona(0).getPrimerNombre()
                        + " "+ a2.getRefPersona(0).getSegundoNombre()
                        + " " + a2.getRefPersona(0).getApellido());
                System.out.println("Nombre: " + a2.getRefPersona(1).getPrimerNombre()
                        + " "+ a2.getRefPersona(1).getSegundoNombre()
                        + " " + a2.getRefPersona(1).getApellido());
                System.out.println("-----------------------------");
                System.out.println("Personas Asignadas al auto 3");
                System.out.println("Nombre: " + a3.getRefPersona(0).getPrimerNombre()
                        + " "+ a3.getRefPersona(0).getSegundoNombre()
                        + " " + a3.getRefPersona(0).getApellido());
                System.out.println("-----------------------------");
                System.out.println("Personas Asignadas al auto 4");
                System.out.println("Nombre: " + a4.getRefPersona(0).getPrimerNombre()
                        + " "+ a4.getRefPersona(0).getSegundoNombre()
                        + " " + a4.getRefPersona(0).getApellido());
                System.out.println("Nombre: " + a4.getRefPersona(1).getPrimerNombre()
                        + " "+ a4.getRefPersona(1).getSegundoNombre()
                        + " " + a4.getRefPersona(1).getApellido());
                System.out.println("-----------------------------");
                System.out.println("Personas Asignadas al auto 5");
                System.out.println("Nombre: " + a5.getRefPersona(0).getPrimerNombre()
                        + " "+ a5.getRefPersona(0).getSegundoNombre()
                        + " " + a5.getRefPersona(0).getApellido());
        }
}
```

Ejercicio 2

El tema del que trata en este ejercicio es el siguiente:

- Asociaciones complejas

Basándose en el diseño realizado en el módulo anterior, codificar en Java las asociaciones complejas que se hayan detectado

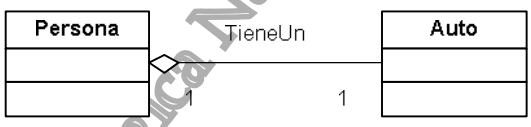
Agregaciones

Es una relación más fuerte entre clases ya que se mantiene por un período de tiempo mucho más largo que con una asociación simple. Las agregaciones son parte fundamental de la programación porque permiten el acceso a otros objetos mediante los servicios públicos que estos prestan incorporándolos como parte del objeto que lo agrega.
Al no tener la limitación temporal de la composición que necesite el objeto a lo largo de todo el tiempo de vida del que lo compone, resulta una técnica de programación mucho más flexible y proclive a los cambios dinámicos que suceden en los sistemas. Por eso la agregación es la relación más utilizada en la programación orientada a objetos, compitiendo inclusive con la herencia que, como se verá más adelante, es la base de la reutilización del código.
Se caracteriza por las palabras "tiene un" o "tiene una" y traducido a la programación esto significa que la clase declara al menos una variable de instancia que almacena la referencia al objeto agregado
Aunque no es mandatorio, la relación puede durar la vida del objeto que tiene declarada la agregación. Sin embargo, para que sea una agregación, debe existir al menos una forma en la cual el objeto agregado pueda ser remplazado por otro del mismo tipo.
Si se tiene el siguiente enunciado

Una persona tiene un auto

La resolución en UML estaría dada por el siguiente gráfico

Y la correspondiente solución en código



Ejemplo

Clase Auto.java

```
public class Auto {
	private String marca;
	private String modelo;
	private String anio;

	public Auto(String ma, String mo, String an) {
		marca = ma;
		modelo = mo;
		anio = an;
	}

	public String getAnio() {
		return anio;
	}

	public void setAnio(String an) {
		anio = an;
```

```
	}

	public String getMarca() {
		return marca;
	}

	public void setMarca(String ma) {
		marca = ma;
	}

	public String getModelo() {
		return modelo;
	}

	public void setModelo(String mo) {
		modelo = mo;
	}
}
```

Clase Persona.java

```
public class Persona {
	private String primerNombre;
	private String segundoNombre;
	private String apellido;
	private String documento;
	private Auto auto;

	public Persona(String p,
	String s,
	String a,
	String d) {
		primerNombre = p;
		segundoNombre = s;
		apellido = a;
		documento = d;
	}

	public String getApellido() {
		return apellido;
	}

	public String getDocumento() {
		return documento;
	}

	public String getPrimerNombre() {
		return primerNombre;
	}

	public String getSegundoNombre() {
		return segundoNombre;
	}

	public Auto getAuto() {
		return auto;
```

```
	}

	public void setApellido(String string) {
		apellido = string;
	}

	public void setDocumento(String string) {
		documento = string;
	}

	public void setPrimerNombre(String string) {
		primerNombre = string;
	}

	public void setSegundoNombre(String string) {
		segundoNombre = string;
	}

	public void setAuto(Auto auto) {
		this.auto = auto;
	}
}
```

Clase UsaAgregacion.java

```
public class UsaAgregacion {
	public static void main(String[] args) {
		Auto a = new Auto("Fiat", "Uno","1998");
		Persona p = new Persona("Juan", "Ignacio", "Perez","21111222");

		p.setAuto(a);

		System.out.println("La Persona tiene un auto " +
				p.getAuto().getMarca() + " " +
				p.getAuto().getModelo()+ " del año " +
				p.getAuto().getAnio());
	}
}
```

Composiciones

Es la relación más fuerte entre clases ya que los objetos, el que compone y el compuesto, están ligados durante tiempo de vida, es decir, cuando se crea uno se crea el otro y cuando se destruye uno se destruye al otro. Cualquier caso fuera de esta consideración lo convierte en una agregación.

Se caracteriza por las palabras "siempre tiene" porque su relación es durante todo el tiempo de vida de ambos objetos. Debe quedar el claro que el objeto compuesto puede tener una instancia diferente e independiente al de la composición, pero cuando se crea dentro de un objeto que lo compone no puede escapar a la limitación temporal que impone la composición, dura el mismo tiempo que la vida del objeto siempre.

Si se tiene el siguiente enunciado

Una persona siempre tiene una dirección

La resolución en UML estaría dada por el siguiente gráfico

Y la correspondiente solución en código. Notar que en el caso de la composición se puede tener una referencia al objeto, pero no se la puede cambiar

Ejemplo

Clase Direccion.java

```
public class Direccion {
	private String nombreCalle;	// nombre completo de la calle
	private int nro;		// nro de la casa o departamento

	public Direccion(
		String nomCalle,	// nombre completo de la calle
		int n) {		// nro de la casa
		nombreCalle = nomCalle;
		nro = n;

	}

	public String getNombreCalle() {
		return nombreCalle;
	}

	public int getNro() {
		return nro;
	}

	public void setNombreCalle(String string) {
		nombreCalle = string;
	}

	public void setNro(int i) {
		nro = i;
	}
}
```

Clase Persona.java

```
public class Persona {
	private String primerNombre;
	private String segundoNombre;
	private String apellido;
	private String documento;
	private Direccion dir;

	public Persona(String p,
	String s,
	String a,
	String d,
	String direccion,
	int num) {
		primerNombre = p;
		segundoNombre = s;
		apellido = a;
		documento = d;
		dir = new Direccion(direccion, num);
```

```
	}

	public String getApellido() {
		return apellido;
	}

	public String getDocumento() {
		return documento;
	}

	public String getPrimerNombre() {
		return primerNombre;
	}

	public String getSegundoNombre() {
		return segundoNombre;
	}

	public void setApellido(String string) {
		apellido = string;
	}

	public void setDocumento(String string) {
		documento = string;
	}

	public void setPrimerNombre(String string) {
		primerNombre = string;
	}

	public void setSegundoNombre(String string) {
		segundoNombre = string;
	}

	public String getDireccion() {
		return dir.getNombreCalle()+ " "+dir.getNro();
	}
}
```

Clase UsaComposicion.java

```
public class UsaComposicion {

	public static void main(String[] args) {
		Persona p = new Persona("Juan", "Pedro", "Diaz",
				"22333444","Venezuela",1301);
		System.out.println("La persona vive en " + p.getDireccion());
	}
}
```

Ejercicios 3 y 4

Los temas de los que se tratan en este ejercicio son los siguientes:

- Agregación
- Composición

En el ejercicio 3, basándose en el diseño realizado en el módulo anterior, codificar en Java las agregaciones y composiciones que se hayan detectado.
En el ejercicio 4, codificar un diagrama UML respetando solamente el diseño del mismo sin conocer el enunciado del problema o como se analizó y diseñó.