**UML – Diagramas de Clases – Conceptos**

**Introducción**

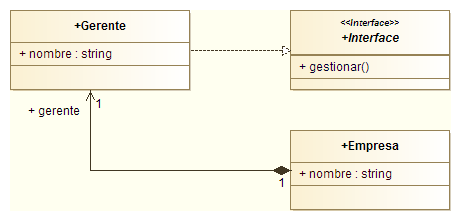
Hay que recordar que **UML** contempla la **descripción de una aplicación informática** bajo el prisma del analisis/diseño orientado a objetos, lo que supone que los conceptos de este paradigma se ven reflejados en la esquematización de sus diagramas.

Los diagramas de clases como herramientas de documentación de la estructura estática de una aplicación informática según los principios de UML.

**Componentes de un diagrama de clases**

Los diagramas de clases son **representaciones estáticas bidimensionales** de la **estructura del sistema a describir**. Así pues, un diagrama de clases se compone de los siguientes elementos:

* **Clases**. Las clases representan las **entidades del sistema**, elementos diferenciados y protagonistas que tienen unas características y comportamientos bien definidos.
* **Interfaces**. Representan **capacitaciones o habilidades**. Son **implementados** o **realizados** por las clases.
* **Relaciones**. Son **vinculaciones** que se establecen generalmente de forma binaria entre los elementos anteriores.



**Visibilidad**

Hay cuatro **tipos de visibilidad** aplicables a los componentes de un diagrama de clases.

* **Publica**. Los elementos marcados con este alcance son accesibles desde cualquier parte, esto es **desde cualquier clase de cualquier paquete**. Se señalizan con un **signo más** [ + ] o un icono característico.
* **Paquete**. Los elementos marcados con este alcance son visibles **desde cualquier clase del mismo paquete**. Se señalizan con un **signo tilde** [ ~ ] o un icono característico.
* **Protegida**. Los elementos marcados con este alcance son visibles **únicamente dentro de la clase a la que pertenecen y desde las clases descendientes de ella**. Se señalizan con un **signo almohadilla**[ # ] o un icono característico.
* **Privada**. Los elementos marcados con este alcance son visibles **únicamente dentro de la clase a la que pertenecen**. Se señalizan con un **signo menos**[ – ] o un icono característico.

**Estereotipos**

Cuando un componente de un diagrama de clases tiene un **significado especial**, se le asigna una etiqueta que lo caracteriza con ese significado. Esta etiqueta recibe el nombre de **estereotipo**.

Un estereotipo está formado por una **palabra**, que identifica ese significado especial, delimitada por **comillas francesas**, por ejemplo **«interface»**.

Hay **estereotipos predefinidos**, pero pueden definirse **estereotipos a conveniencia** en caso de necesitar expresar una semántica que se escape a los elementos predefinidos.

En caso de necesitar de algún estereotipo a medida, el analista/diseñador puede definir el suyo propio declarándolo con su significado. A partir de ese momento ya puede utilizarse aplicándolo a los elementos que corresponda.

Los estereotipos son específicos de cada tipo de elemento.

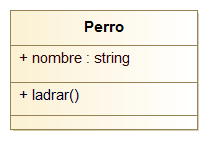
**UML – Diagramas de Clases – Clases**

**Representación de una clase**

Las clases se representan en UML como una caja con varias zonas:

1. **Zona superior**. En ella se escribe el **identificador** de la clase, generalmente un nombre en singular que empieza por mayúscula. Adicionalmente se pueden representar también la **visibilidad** de la clase y el **estereotipo** aplicado.
2. **Zona intermedia**. En ella aparecen los **atributos**. Cada atributo identifica una **propiedad** de la clase. Un atributo se representa con tres elementos:
   * **Identificador**, generalmente un **nombre en singular** que empieza por minúscula.
   * **Tipo de datos**, refiere la naturaleza del atributo.
   * **Visibilidad**, expresa el alcance del atributo en el sistema.
3. **Zona inferior**. En ella aparecen las **operaciones**o **métodos**. Expresan las **capacidades funcionales** de una clase y se nombran con un **verbo en infinitivo** que empieza por minúscula y que expresa una acción.

La imagen siguiente corresponde a la representación de la clase **Perro**. Esta clase tiene un atributo de tipo texto llamado **nombre**y un método llamado **ladrar** que no tiene parámetros de llamada ni valor de retorno.



Obsérvese el signo más a la izquierda del atributo y también a la izquierda del método, indica visibilidad pública para ambos. Obsérvese también que no se ha expresado la visibilidad de la clase.

El proceso de **Codificar** esta clase en un **lenguaje de programación orientado a objetos** como **Java** resulta sencillo, por ahora, y se correspondería con el siguiente código:

|  |  |
| --- | --- |
| 2  3  4  5  6 | class Perro {     public String nombre;     public void ladrar() {        System.out.println("¡Guau, guau!");     }  } |

El siguiente gráfico muestra la clase de ejemplo **Nif** que permite representar los datos de un NIF que son el **número de DNI** y el **dígito de control**.

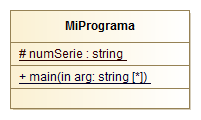


En la clase del diagrama anterior el número de DNI está representado por el atributo **dni** que es de tipo entero, el dígito de control está representado por el atributo **letra** que es de tipo carácter. Ambos tienen **visibilidad pública** y, como antes, no se ha expresado visibilidad para la clase.

La **codificación del diagrama de clases** para la clase anterior podría corresponderse con el siguiente código:

|  |  |
| --- | --- |
|  | class Nif {     public int dni;     public char letra;  } |

El siguiente gráfico muestra la clase de ejemplo **MiPrograma** que ejecuta un programa Java que muestra un número de serie.



En la clase del diagrama anterior se puede apreciar:

* La **clase** se llama **MiPrograma** y no tiene expresada su visibilidad.
* Un **atributo** llamado **numSerie** de tipo **String** que está marcado con **visibilidad protegida**. El **subrayado** significa que el atributo es **estático**, lo cual tiene por consecuencia que **se puede utilizar sin necesidad de instanciar la clase (sin crear un objeto de la misma)**.
* Un **método** llamado **main** que tiene como **parámetro formal** un **array de String** llamado **arg**y no tiene valor de retorno. Este método también está marcado como **estático** con lo que, al igual que el atributo, **puede ser llamado sin necesidad de instanciar la clase**. En este caso, el método **main** tiene visibilidad pública.

La **codificación del Diagrama de Clases** para la clase anterior podría corresponderse con el siguiente código:

|  |  |
| --- | --- |
| 1  2  3  4  5 | class MiPrograma {     protected static String numSerie;     public static void main(String[] arg) {        System.out.printf("Serie: %s\n", numSerie);     }  } |

# UML – Diagramas de Clases – Interfaces

**Interfaz – Definición**

Los interfaces son mecanismos que permiten a los lenguajes orientados a objetos con herencia lineal, (léase Java), acceder a la semántica de los lenguajes orientados a objetos con herencia múltiple ( léase C++ ). Sin embargo en el grano fino las posibilidades de los interfaces difieren de aquellas que aporta la herencia múltiple.

En el fondo, de lo que se trata, es de aportar nueva funcionalidad a las clases, aunque la forma en la que esto se procura no es precisamente de forma directa.

**Interfaz – Funcionamiento**

La idea es que los **interfaces** son como **contratos** para las **clases** que las obliga a definir los **métodos** establecidos en **cláusulas de funcionalidad**.

Las **clases** que se **comprometen** y **cumplen** con estos **contratos** tienen implementada esa **funcionalidad** y por tanto son capaces de llevarla a cabo. El “premio” es que **pueden instanciar objetos**.

Las **clases** que se **comprometen** con estos contratos pero **no los cumplen** no son funcionales. El “castigo” es que son **abstractas** y **no pueden instanciar objetos**.

Cuando alguien entra en un vehículo, lo primero con lo que se encuentra al entrar en la **plaza del conductor** es esa pieza grande y redonda al frente, que parece que esté hecha para cogerse a ella, que se llama **volante**. A pesar de que **es la primera vez** que entramos en ese coche en concreto, damos por sentado que si gira el volante el coche girará también. Probablemente, no sabemos cómo se las ingenia el coche para girar con el volante, y ni falta que le hace para poder conducir.

**Eso es un interfaz** …

El interfaz es la razón por la que cuando se gira el volante, el vehículo también gira. Así, una vez se sabe girar un vehículo se sabe girar cualquier otro, porque todos funcionan igual, porque todos realizan la misma funcionalidad impuesta por el hecho de que haya un volante, todos implementan el mismo interfaz.

“Si una clase implementa un interfaz, significa que los objetos instanciados de ella disponen de una determinada funcionalidad adicional en forma de métodos que podrán ser utilizados a conveniencia. Por tanto esa clase podrá ser utilizada para un determinado fin.”

**Interfaz – Representación**

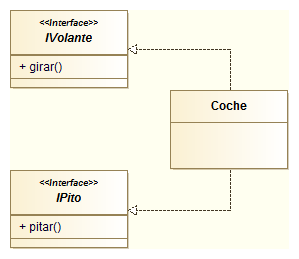
En los Diagramas de Clases UML los interfaces se pueden representar de varias formas.



Una clase puede implementar a la vez tantos interfaces como se desee, pero con un solo método de un solo interfaz que falte por implementar la clase será abstracta.

**Interfaz – Utilización**

Un coche puede girar y puede pitar. Así, una clase ***Coche*** puede implementar simultáneamente el interfaz  **IVolante**  y el interfaz  **IPito**, cuya representación podría quedar como sigue:



Obsérvese que la línea que vincula una clase y sus interfaces es dirigida, **parte de la clase y acaba en el interfaz**. Además se representa con **línea discontinua** y con **punta de flecha cerrada**.

**Interfaz – Codificación**

El ejemplo anterior podría **codificarse** en **Java** de la siguiente **manera**:

|  |  |
| --- | --- |
|  | public interface IVolante {     public void girar();  }  public interface IPito {     public void pitar();  }  class Coche implements IVolante, IPito {     public void girar() {        System.out.println("¡Girando, girando!");     }     public void pitar() {        System.out.println("¡Pitando, pitando!");     }  } |

**UML – Diagramas de Clases – Herencia**

**Introducción**

Si el paso de la Programación Estructurada a la Programación Orientada a Objetos supuso dotar al programador de mayor potencia y control sobre su código, la evolución de la Programación Modular en la Herencia Orientada a Objetos ha supuesto un mecanismo muy potente para aumentar la productividad evitando las continuas reinvenciones de la rueda.

**Contexto**

Técnicamente hablando la **herencia** en el contexto de la programación orientada a objetos tiene muchas cualidades, pero siendo pragmáticos la herencia supone un **mecanismo muy eficaz para reducir y simplificar el código**.

Para que demuestre todo su potencial hay que partir de una **concepción del sistema como composición/colaboración/interacción de clases**, (o interfaces), huyendo de la concepción puramente funcional.

**Identificación**

Partiendo de las **especificaciones** hay que identificar las **clases del sistema** y, acto seguido, averiguar las **relaciones** que hay entre ellas analizando cuales están vinculadas entre sí, de una forma u otra.

Una vez **identificadas las relaciones** existentes entre las clases de un sistema hay que averiguar cuáles de ellas pueden ser **modelizadas en forma de herencia**.

Para **detectar una relación de herencia** hay que examinar las clases involucradas en cada una de las diferentes relaciones binarias que existan en el sistema y, para cada relación, averiguar si existe una **derivación semántica entre ambas clases**.

Dicho de otro modo, para detectar una relación de herencia entre dos clases hay que averiguar si una de ellas es una **consecuencia o evolución** de la otra.

En ese caso es muy probable que esa relación se pueda modelizar en forma de herencia.

**Ventajas**

Y … ¿En qué beneficia la detección de una relación de herencia entre dos clases de un sistema?

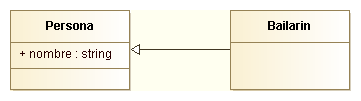
Pues, en primer lugar permite expresar la **relación natural** existente entre ellas de forma correcta. Si en general, expresar las cosas de forma incorrecta genera muchos problemas, en los diagramas de clases más.

Y en segundo lugar supone que no hay que duplicar los recursos existentes en la entidad base dentro de la entidad especializada, lo que ayuda a **simplificar el diagrama** y a **clarificar su significado**.

**Clases**

Es posible encontrar una **relación de herencia entre dos clases**. En este caso la **clase derivada** posee todos los recursos de la **clase matriz** a **excepción** de aquellos recursos marcados con **visibilidad privada**.

En el Diagrama de Clases siguiente se muestra un ejemplo de **relación de herencia** entre dos clases: **Persona** y **Bailarín**.



Como se puede observar la **línea** que vincula ambas clases es **dirigida**, de **trazo continuo** y acaba en una **punta de flecha cerrada**. **Empieza en la entidad derivada y termina en la entidad base**.

En la sintaxis UML el diagrama anterior expresa que la clase **Persona** es la **generalización** de la clase **Bailarín,** lo cual, puesto por pasiva,  también significa que la clase **Bailarín** es la **especialización** de la clase **Persona**.

En el ejemplo, la clase **Persona** dispone de un atributo llamado **nombre** de tipo **String** que es **público**. Como consecuencia de la herencia la clase **Bailarín** también tiene un atributo llamado **nombre** de tipo **String** que hereda de la clase **Persona**. Como se puede observar **los recursos heredados no se representan**.

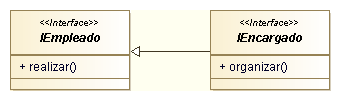
El diagrama anterior se puede **codificar en Java** dando lugar a un código fuente parecido al siguiente.

|  |  |
| --- | --- |
|  | class Persona {     public String nombre;  }  class Bailarin extends Persona {  } |

**Interfaces**

La misma formulación que se ha hecho con las clases puede realizarse entre dos interfaces.

En el Diagrama de Clases siguiente se muestra un ejemplo de **relación de herencia** entre dos interfaces: **IEmpleado** y **IEncargado**. El interfaz **IEmpleado** declara la funcionalidad que tienen que realizar los **empleados de una empresa**. A veces algún empleado tiene que hacer de **encargado** organizando la actividad. En esa circunstancia debe implementar el interfaz **IEncargado** que declara la funcionalidad que tiene que hacer un encargado.



El interfaz **IEmpleado** declara un método llamado **realizar** cuya implementación  debe definir la capacidad funcional que debe tener un **empleado**. El interfaz **IEncargado** declara un método llamado **organizar** que debe definir la capacidad funcional que debe tener un **encargado**. Pero además, como el interfaz **IEncargado** extiende al interfaz **IEmpleado**, también dispone de la declaración heredada del método **realizar**.

Asimismo, la codificación de diagrama de clases anterior en Java podría corresponder al siguiente código.

|  |  |
| --- | --- |
|  | public interface IEmpleado {     public void realizar();  }  public interface IEncargado extends IEmpleado {     public void organizar();  } |

**UML–Diagramas de Clases – Realización**

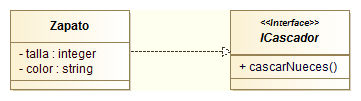
**Concepto**

El concepto de **realización** concierne a la **implementación de un interfaz por parte de una clase**. Este proceso tiene **dos partes**, en primer lugar **la clase debe declarar la implementación del interfaz**. En segundo lugar **la clase debe de definir el cuerpo de los métodos impuestos por el interfaz**.

**Representación**

En los Diagramas de Clases **no se expresa el contenido del cuerpo de los métodos**, únicamente se expresa la declaración de que la clase implementa un interfaz.

En el Diagrama de Clases del ejemplo que se expone a continuación se puede observar como la clase **Zapato** implementa el interfaz **ICascador** que permite al zapato utilizarlo como cascanueces.



Obsérvese como la **relación entre la clase y el interfaz** implementado se representa como una **línea de trazos dirigida**, que **parte de la clase y termina en el interfaz** con una **punta de flecha cerrada**. Por supuesto, **los métodos impuestos por el interfaz no se representan en la clase**.

**Codificación**

El código fuente Java equivalente al diagrama anterior podría corresponder al que se expone a continuación.

|  |  |
| --- | --- |
|  | public interface ICascador {     public void cascarNueces();  }  class Zapato implements ICascador {     private int talla;     private String color;     public void cascarNueces() {        System.out.println("Crick Crack");     }  } |

**UML– Diagramas de Clases – Relaciones**

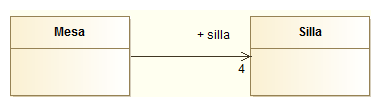
**Introducción**

Las **relaciones** son el **tercer pilar fundamental** en el que se basan los Diagramas de Clases, después de las **clases** mismas y los **interfaces**. Las relaciones **se aplican exclusivamente entre clases** y pueden ser **binarias o de orden superior**.

Decir que dos clases están relacionadas entre si viene a significar que esas clases tienen algo que ver entre sí. De cómo sea la **naturaleza de la relación** definirá un tipo u otro de vinculación. De lo que se trata aquí es de identificar, caracterizar y ejemplarizar cada una de ellas.

**Asociación**

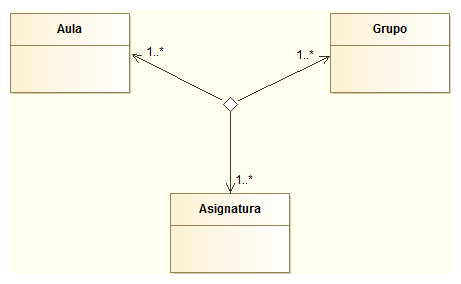
La forma más sencilla de relación es aquella denominada **asociación**. La asociación se utiliza para expresar simplemente que dos clases están vinculadas entre sí. En ella se expresa la **navegabilidad** entre la **clase origen** y la **clase destino**, y la **cardinalidad** de la **clase destino** en la asociación.



El Diagrama de Clases del ejemplo anterior permite representar, en el contexto de UML,  el hecho de que una mesa tiene 4 sillas a juego.  Obsérvese que en el diagrama no se expresa más vinculación que la navegabilidad que expresan que las sillas van con la mesa, ni más restricción que la cardinalidad que expresa que con la mesa van cuatro sillas.

En la figura se observa como aparece el rol **silla** para vincular la clase **Silla** a la clase **Mesa**. Esta vinculación se sustanciará convirtiendo el **rol de la relación** en **un atributo de la clase origen que referencia la clase destino**.

Estrictamente hablando una asociación no tiene que ser únicamente de navegabilidad en un solo sentido, Puede ser en ambos con los que ambas clases son origen y destino a la vez. Un tipo especial de esta situación acontece cuando la asociación involucra más de dos clases. En ese caso todas las clases asociadas son origen y destino a la vez.



El ejemplo anterior se modeliza la siguiente situación.

Cada aula alberga uno o más grupos a los que se imparten una o más asignaturas, a su vez cada grupo tiene asignada una o más aulas en donde recibe docencia de una o más asignaturas, y además cada asignatura se imparte en una o más aulas a uno o más grupos.

En las asociaciones, el peso de la definición de la relación recae enteramente sobre la parte [TODO]:

* La vinculación se define en la parte [TODO] que incluye la referencia a la parte [PARTE].
* La multiplicidad de  la parte [PARTE] debe expresarse en la parte [TODO] a través de algún tipo de colección, a la cual debe de acompañar, en la mayoría de los casos, al menos de sendos métodos para incorporar/desvincular elementos.

El concepto de asociación entre clases permite representar la semántica de muchas situaciones. Sin embargo la realidad ofrece situaciones mucho más complejas cuyo modelizado exige evolucionar este concepto de asociación introduciendo el **concepto de pertenencia** y el **concepto de autonomía**.

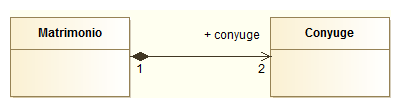
**Pertenencia**

Para explicar la semántica de pertenencia de una relación, ayuda plantearla desde un punto de vista del binomio **[PARTE] – [TODO]**. Desde esta perspectiva, una relación binaria está constituida por un componente **[PARTE]** y un componente **[TODO]**.

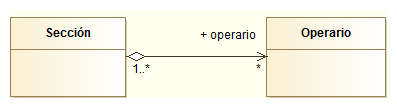
El componente **[PARTE]** se caracteriza porque es una pieza, en el sentido constructivo, del componente **[TODO]**. El componente **[TODO]** tiene la capacidad de albergar al componente **[PARTE]** integrándolo dentro de sí mismo.

Antes de seguir un buen ejemplo ayudaría a fijar los conceptos de **[PARTE]** y **[TODO]** en una relación entre clases.

Bien, considérese el ejemplo de la relación de un **matrimonio** respecto de sus **cónyuges**. En esa relación el **matrimonio** seria la parte **[TODO]**, mientras que los **cónyuges** serian la parte **[PARTE]** de la relación. Trasladando el ejemplo al contexto UML, si se considera la clase **Matrimonio** y la clase **Conyuge**, y dejando para más adelante la explicación de los detalles involucrados en la relación, el **Diagrama de Clases** que representaría esta relación podría ser el siguiente:



Otro ejemplo. Considérese la relación que existe entre los **operarios de una fábrica** y las **secciones de trabajo** de la misma. En este caso cada operario trabaja en un momento dado en una sección de la fábrica, aunque después puede trabajar en otra. Así pues, cada sección de la fábrica alberga un número determinado de operarios.



**Autonomía**

De lo que se trata de dilucidar en este apartado es, qué **ciclo de vida** tienen los objetos de la **clase [TODO]** y qué **ciclo de vida** tienen los objetos de la **clase [PARTE].** Dependiendo de cómo se concreta esta cuestión la semántica de la situación define un tipo de asociación u otro entre las respectivas clases.

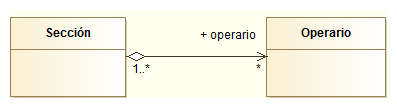
En concreto, la **regla para determina el tipo de asociación** es fijarse en el **ciclo de vida** de los objetos de la **clase [TODO]**, en concreto en el **momento en que se destruye**. La pregunta que hay que hacerse es **¿Qué ocurre con los objetos de la clase [PARTE] cuando se destruye la parte [TODO]?**. La respuesta a esta pregunta determina **dos tipos de asociaciones**:

* **Agregación**. Cuando el objeto [TODO] se destruye, los objetos [PARTE] pueden seguir existiendo autónomamente.
* **Composición**. Cuando el  objeto [TODO] se destruye también desaparecen los objetos [PARTE], cuya existencia ya no tiene sentido.

**Agregación**

Es un tipo de asociación en donde el ciclo de vida de la parte [TODO] está desvinculado del ciclo de vida de la parte [PARTE], de tal manera que cuando desaparece la parte [TODO] la parte [PARTE] puede seguir existiendo. A este tipo de vinculación se la denomina también **asociación débil** o **asociación funcional**.

Para explicar este tipo de relación considérese el caso expuesto anteriormente respecto a los operarios y las secciones de una fábrica.



En el ejemplo, la clase **Seccion** referencia las instancias de la clase **Operario**, que se corresponden con los operarios que están trabajando en ella.

Tomando como referencia el ejemplo anterior, se inferirán las correspondientes reglas respecto a las agregaciones en los diagramas de clases:

* La **clase [TODO]** se identifica con un **rombo en blanco**.
* La **clase [PARTE]** se identifica con una **flecha de navegación**.
* La **relación** se identifica por su **rol situado en la clase [PARTE]**.
* La **clase [TODO] no tiene un atributo para expresar el rol**.
* La **multiplicidad** de la clase [TODO] es **diferente de la unidad**.
* El **constructor** de la clase [TODO] **no instancia** la clase [PARTE].
* El **destructor** de la clase [TODO], cuando existe, **no altera** la clase [PARTE].

La codificación del Diagrama de Clases del ejemplo anterior en Java podría corresponderse con el siguiente código:

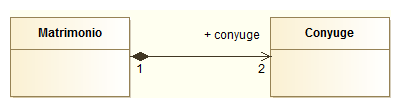
|  |  |
| --- | --- |
|  | class Operario {     ...  }  class Seccion {     ...     ArrayList<Operario> listaOperarios = new ArrayList<Operario>();     ...  } |

Como se puede observar en el código anterior la multiplicidad de la clase [TODO] no se expresa porque depende de las restricciones funcionales de la clase que instancia la parte [TODO].

**Composición**

Es un tipo de asociación en donde el ciclo de vida de la parte [PARTE] está vinculado al ciclo de vida de la parte [TODO], de tal manera que cuando desaparece la parte [TODO] la parte [PARTE] también desaparece. A este tipo de vinculación se la denomina también **asociación fuerte**o **asociación existencial**.

Para explicar este tipo de relación considérese el caso expuesto anteriormente respecto de los cónyuges y el matrimonio.



En el ejemplo, la clase **Matrimonio** referencia cada una de las dos instancias de la clase **Conyuge**, generalmente a través de algún tipo de documento en el registro civil.

Tomando como referencia el ejemplo anterior, se inferirán las correspondientes reglas respecto a las agregaciones en los diagramas de clases:

* La **clase [TODO]** se identifica con un **rombo en negro**.
* La **clase [PARTE]** se identifica con una **flecha de navegación**.
* La **relación** se identifica por su **rol situado en la clase [PARTE]**.
* La **clase [TODO] no tiene un atributo para expresar el rol**.
* La **multiplicidad de la clase [TODO] es siempre la unidad**.
* El **constructor** de la clase [TODO] **suele instanciar** la clase [PARTE].
* El **destructor** de la clase [TODO], cuando existe, **destruye también** la clase [PARTE].

La codificación del Diagrama de Clases del ejemplo anterior en Java podría corresponderse con el siguiente código:

|  |  |
| --- | --- |
|  | class Conyuge {     ...  }  class Matrimonio {     ...     Conyuge[] conyuge = new Conyuge[2];     ...     Matrimonio() {        conyuge[0] = new Conyuge();        conyuge[1] = new Conyuge();        ...     }  } |

Al igual que en la agregación la multiplicidad de la clase [TODO] no se expresa porque depende de la restricciones funcionales de la clase que instancia la parte [TODO].