



**Verificación  
de la  
Implementación de  
Elementos  
UML en Java**

(Proyecto fin de carrera - 2001)

Belén Criado Sánchez

Tutor: Gonzalo Génova

Ingeniería Superior en Informática

Universidad Carlos III de Madrid

# Índice

<b>1. INTRODUCCIÓN .....</b>	<b>3</b>
<b>2. DEFINICIÓN EN RATIONAL ROSE DE ELEMENTOS DEL DIAGRAMA DE CLASES.....</b>	<b>6</b>
2.1. CLASE .....	6
2.1.1. <i>Especificación</i> .....	7
2.1.2. <i>Ejemplo de clase en Rational Rose</i> .....	9
2.1.3. <i>Tipos de clases</i> .....	10
2.2. GENERALIZACIÓN .....	11
2.2.1. <i>Especificación</i> .....	12
2.2.2. <i>Ejemplo de Generalización en Rational Rose</i> .....	14
2.3. ASOCIACIÓN .....	15
2.3.1. <i>Especificación</i> .....	16
2.3.2. <i>Ejemplo de Asociación en Rational Rose.</i> .....	17
2.3.3. <i>Navegación</i> .....	17
2.3.4. <i>Restricciones de una Asociación</i> .....	18
2.3.5. <i>Asociación Derivada</i> .....	19
2.4. AGREGACIÓN .....	20
2.4.1. <i>Ejemplo de Agregación en Rational Rose</i> .....	20
2.5. COMPOSICIÓN .....	21
2.5.1. <i>Ejemplo de Composición en Rational Rose</i> .....	21
2.6. CLASE ASOCIACIÓN.....	22
<b>3. ESTRUCTURA GENERAL DE FICHERO RATIONAL ROSE .....</b>	<b>24</b>
3.1. VISTA LÓGICA (REPRESENTACIÓN DE DIAGRAMA DE CLASES) ..	26
3.1.1. <i>Diagramas a nivel principal</i> .....	27
3.1.2. <i>Paquete</i> .....	27
<b>4. IMPLEMENTACIÓN EN JAVA DE ELEMENTOS DEL DIAGRAMA DE CLASES.....</b>	<b>28</b>
4.1. CLASE .....	28
4.1.1. <i>Ejemplo de clase en Java</i> .....	30
4.2. GENERALIZACIÓN .....	30
4.2.1. <i>Ejemplo de Generalización en Rational Rose</i> .....	31
4.3. ASOCIACIÓN .....	32
4.3.1. <i>Ejemplos de Asociación en Java</i> .....	34
4.3.2. <i>Complejidad de las cardinalidades</i> .....	36
4.3.3. <i>Enlaces</i> .....	38
4.3.4. <i>Integridad</i> .....	39
4.3.5. <i>Múltiples asociaciones</i> .....	40
4.3.6. <i>Clase Asociación</i> .....	41
4.3.7. <i>Ejemplo de Clase Asociación en Java</i> .....	45
4.3.8. <i>Asociación Derivada</i> .....	46
4.3.9. <i>Ejemplo de Asociación Derivada</i> .....	46
4.4. COMPOSICIÓN .....	47

---

<b>5.</b>	<b>ANÁLISIS Y DISEÑO DE LA HERRAMIENTA DE VERIFICACIÓN.....</b>	<b>52</b>
5.1.	ANÁLISIS DE LA APLICACIÓN .....	52
5.1.1.	<i>Clase</i> .....	52
5.1.2.	<i>Generalización</i> .....	55
5.1.3.	<i>Asociación</i> .....	55
5.1.4.	<i>Clase-Asociación</i> .....	59
5.1.5.	<i>Composición</i> .....	60
5.2.	DISEÑO DE LA APLICACIÓN .....	61
5.2.1.	<i>Clases</i> .....	61
5.2.2.	<i>Generalización</i> .....	62
5.2.3.	<i>Asociaciones</i> .....	63
5.2.4.	<i>Clases-Asociaciones</i> .....	64
5.2.5.	<i>Composición</i> .....	65
<b>6.</b>	<b>MANUAL DE USUARIO.....</b>	<b>66</b>
6.1.	VERIFICACIÓN DE CLASES .....	69
6.2.	VERIFICACIÓN DE GENERALIZACIONES .....	74
6.3.	VERIFICACIÓN DE ASOCIACIONES.....	77
6.4.	VERIFICACIÓN DE CLASES ASOCIACIONES.....	82
6.5.	VERIFICACIÓN DE COMPOSICIÓN .....	85
<b>7.</b>	<b>RESULTADOS, CONCLUSIONES Y PROPUESTAS .....</b>	<b>89</b>
<b>8.</b>	<b>BIBLIOGRAFÍA.....</b>	<b>91</b>

# 1. INTRODUCCIÓN

El proyecto trata de comprobar que las relaciones existentes en un diagrama de clases UML<sup>1</sup> creado por una herramienta CASE<sup>2</sup> específica como es Rational Rose, se corresponden con la posterior codificación en un lenguaje de programación orientada a objetos como Java. Del mismo modo, se comprueba que las relaciones que contiene un programa en Java están debidamente recogidas por el diagrama de clases UML que lo representa.

UML es un lenguaje para crear modelos de análisis y de diseño. Los diagramas de análisis son abstracciones (simplificaciones) del mundo real, que usamos para comprender el problema. Los diagramas de diseño son abstracciones (simplificaciones) del mundo informático, que usamos para plantear la solución. La herramienta desarrollada no tiene sentido para verificar la correspondencia entre diagramas de análisis e implementación, ya que esta correspondencia no tiene por qué existir.

UML es el resultado de un largo proceso iniciado por tres de los metodologistas más reputados: Grady Booch, Ivar Jacobson y Jim Rumbaugh. Sus trabajos respectivos representaban las diversas facetas de un mismo problema, por lo que unieron sus esfuerzos trabajando en "Rational Software Corporation" y apoyados por otros especialistas del sector, lograron definir una nueva aproximación al modelado de programas a base de objetos. UML es, a la vez, la síntesis y el sucesor natural de sus diferentes trabajos.

UML es una notación pensada para servir de lenguaje de modelado de objetos, independientemente del método de implementación. Los diseñadores de la notación han buscado ante todo la simplicidad; UML es intuitivo, homogéneo y coherente.

UML se concentra sobre la descripción de los artefactos del desarrollo de programa, en lugar de en la formalización del propio proceso de desarrollo. UML no es una notación cerrada: es genérica, extensible y configurable por el usuario. UML define nueve tipos de diagramas para representar los diferentes puntos de vista del modelado:

- *diagramas de actividades* que representan el comportamiento de una operación en términos de acciones;
- *diagramas de caso de uso* que representan las funciones del sistema desde el punto de vista del usuario;
- *diagramas de clases* que representan la estructura estática en términos de clases y relaciones;
- *diagramas de colaboración* que son una representación espacial de los objetos, enlaces e interacciones;
- *diagramas de componentes* que representan los componentes físicos de una aplicación;
- *diagramas de despliegue* que representan el despliegue de los componentes sobre los dispositivos materiales;

---

<sup>1</sup> Unified Modeling Language

<sup>2</sup> Computer Aided Software Engineering

- *diagramas de estados-transiciones* que representan el comportamiento de una clase en términos de estados;
- *diagramas de objetos* que representan los objetos y sus relaciones y corresponden a diagramas de colaboración simplificados, sin representación de los envíos de mensaje;
- *diagramas de secuencia* que son una representación temporal de los objetos y sus interacciones.

El diagrama de clases expresa de manera general la estructura estática de un sistema, en términos de clases y de relaciones entre estas clases. Al igual que una clase describe un conjunto de objetos, una asociación describe un conjunto de enlaces; los objetos son instancias de clases y los enlaces son instancias de asociaciones. Un diagrama de clases no expresa nada de particular sobre los enlaces de un objeto dado, pero describe de manera abstracta los enlaces potenciales de un objeto hacia otros objetos.

Teniendo en cuenta que en este proyecto se trata de hacer una correspondencia entre la definición del diagrama de clases en UML y una implementación Java, se puede comprobar que el diagrama de clases es el que mejor representa los elementos encontrados en un lenguaje de programación orientado objetos, donde localizamos elementos tales como clases, atributos, etc.

Para este proyecto se ha utilizado la herramienta “Rational Rose” creada por la compañía “Rational Software Corporation”, que permite crear de forma sencilla los distintos diagramas utilizados en la notación UML. La versión seleccionada para el trabajo ha sido la 4.0 del año 1996. Esta versión tiene algunas limitaciones y existen versiones posteriores que son mucho más potentes, pero se ha escogido por permitir la representación de los elementos básicos del diagrama de clases y lo más importante, por ser una versión de libre distribución y por lo tanto accesible a todas aquellas personas interesadas en esta herramienta.

La programación orientada a objetos (POO) es una forma de escribir un software que se puede volver a utilizar y cuyo mantenimiento es realmente sencillo. Java es el lenguaje de programación orientado a objetos utilizado en este proyecto. La Core API de Java es un conjunto de componentes POO que proporcionan funciones comunes a todas las plataformas que pueden trabajar con Java para facilitar el desarrollo de proyectos. Java además se caracteriza por:

- permitir la escritura de programas fiables;
- permitir la construcción de aplicaciones en casi cualquier plataforma sin tener que volver a compilar el código;
- permitir la distribución de aplicaciones a través de una red “poco fiable” siguiendo el sistema tradicional.

Para el estudio de este proyecto se ha utilizado el código del fichero con extensión .MDL que genera Rational Rose en la versión 4.0. Cabe destacar que en este caso no es de vital importancia la versión utilizada, puesto que se han omitido muchas líneas del fichero que son útiles para la herramienta, pero no para el caso que nos interesa. De las líneas resultantes de este fichero, se han escogido aquellas que nos dan la información deseada, sin hacer referencia en ninguna parte de este trabajo al resto de ellas.

Este fichero .MDL contiene varios tipos de información, como son: información acerca de la representación gráfica de los elementos incluidos en los distintos diagramas, información analítica sobre otros diagramas incluidos en la notación UML e información analítica acerca de los diagramas de clases. De toda esta información sólo se ha estudiado la referente a la información analítica de los diagramas de clases. Por ello es importante destacar que en este trabajo no se ofrece la información suficiente acerca de los ficheros con extensión .MDL, como para poder realizar modificaciones sobre él y por lo tanto cambiar la representación gráfica en Rational Rose, por ejemplo.

## 2. DEFINICIÓN EN RATIONAL ROSE DE ELEMENTOS DEL DIAGRAMA DE CLASES

Para obtener la información deseada, primero debemos saber qué elementos contiene un Diagrama de Clases UML y qué información nos interesa capturar del fichero resultante de la representación de ese diagrama en Rational Rose.

De este tipo de diagramas nos interesan los siguientes conceptos:

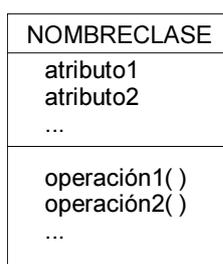
- Clase
- Asociación
- Agregación
- Agregación de tipo composición
- Generalización
- Clase abstracta

Todos estos conceptos pueden ser representados en un diagrama de clases UML, pero Rational Rose en su versión 4.0 no permite recoger todos los elementos y características definidas por el estándar UML. Uno de los elementos más importantes que no permite representar son las asociaciones ternarias, y en general asociaciones que no sean binarias y tampoco permite la representación gráfica de clases abstractas aunque sí internamente.

Rational Rose codifica los diferentes diagramas que permite representar, a través de un fichero con extensión .MDL. Entre esta información encontramos los diferentes elementos pertenecientes a un Diagrama de Clases UML, como veremos a continuación:

### 2.1. CLASE

Una clase UML se compone de un nombre, atributos (o propiedades) y operaciones (o métodos) que implementa. Esto es representado de la siguiente forma:



La estructura general de un objeto de este tipo, dentro de un fichero .MDL es la siguiente:

```

(object Class "NOMBRECLASE"
  ...
  operations (list Operations
    (object Operation "operación1"
      parameters (list Parameters
        (object Parameter "Nombre_Parámetro"
          type "Tipo_Parámetro"1
          initv "Valor_Inicial"))
      result "Tipo_Resultante"1
      opExportControl "Tipo_Visibilidad_Operación"2
    )

    (object Operation "operación2"
      opExportControl "Tipo_Visibilidad_Operación"2
    ..))

  class_attributes (list class_attribute_list
    (object ClassAttribute "atributo1"
      type "Tipo_Atributo"*1
      exportControl "Tipo_Visibilidad_Atributo"2
    ...)

    (object ClassAttribute "atributo2"
      initv "Valor_Inicial"
      exportControl "Tipo_Visibilidad_Atributo"2)))

```

<sup>1</sup> Tipo\_Parámetro, Tipo\_Resultante y Tipo\_Atributo: Definen de qué tipo es el parámetro que se pasa, el que devuelve el método ó el tipo de los propios atributos de la clase, que puede ser de cualquiera de las clases que contiene el propio diagrama de clases. Al ser un campo libre, puede aparecer incluso en blanco o una cadena de caracteres escrita por el diseñador.

<sup>2</sup> Tipo\_Visibilidad\_Operación y Tipo\_Visibilidad\_Atributo: Podrán tomar valores de "Public", "Private", "Protected" ó "Implementation".

Es necesario recordar que existen líneas intermedias en el código creadas por el programa, pero que no se han incluido por no ser importantes para este estudio.

### 2.1.1. Especificación

Como ya se ha comentado, una clase UML se compone de un nombre, propiedades y operaciones ó métodos que implementa. En este apartado veremos con mas detalle donde aparece esta información.

- Nombre

Cuando se describe una clase se comienza con el siguiente código donde se especifica el nombre de la clase:

```
(object Class "Nombre-Clase"
```

- Operaciones

Posteriormente se identifican las operaciones, como vemos a continuación:

```
operations (list Operations
```

Para cada operación se identifica el nombre de la operación y su visibilidad, definida por el parámetro "*opExportControl*", que puede ser de tipo: Public (Público), Private (Privado), Protected (Protegido) ó Implementation (Implementación):

```
(object Operation "operación"
  opExportControl "Public"
  ..))
```

En una operación también podemos identificar los parámetros que tiene, su tipo y el tipo del valor que devuelve. Esto es codificado dentro del fichero, a partir de la línea:

```
parameters (list Parameters
```

Este código aparece después de la línea que identifica el nombre de la operación (object Operation "*operación*"). A partir de aquí se define para cada parámetro la siguiente información:

Nombre de parámetro:

```
(object Parameter "Nombre_Parámetro"
```

Tipo del parámetro:

```
type "Tipo_Parámetro"
```

Valor por defecto que va a tomar ese parámetro:

```
initv "Valor_Inicial"))
```

Tipo que devuelve la operación y tipo de visibilidad, como se ha comentado anteriormente:

```
result "Tipo_Resultante"
opExportControl "Tipo_Visibilidad_Operación"
)
```

- Atributos

Por último encontramos los atributos de la clase, que comienzan a definirse después de la siguiente línea:

```
class_attributes (list class_attribute_list
```

Para cada atributo de la clase se obtiene el siguiente código:

Nombre:

```
(object ClassAttribute "Nombre_Atributo"
```

Tipo del atributo:

```
type "Tipo_Atributo"
```

Valor por defecto que va a tomar ese atributo:

```
initv "Valor_Inicial"
```

Visibilidad, que como se ha indicado anteriormente podrá tomar los valores de: public, private, protected ó implementation:

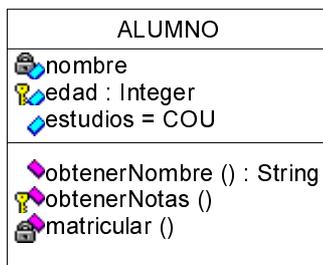
```
exportControl "Tipo_Visibilidad_Atributo"))
```

Hay que destacar que, al contrario de las operaciones, en la definición de la visibilidad de los atributos, Rational Rose omite la especificación del parámetro *exportControl* cuando se trata de un atributo con visibilidad Privada, quedando codificado de la siguiente forma:

```
(object ClassAttribute "Atributo-Privado"
initv "Valor_Inicial")
```

### 2.1.2. Ejemplo de clase en Rational Rose

Este ejemplo contiene todos los posibles tipos de atributos y operaciones, que quedan registrados por Rational Rose de la siguiente forma:



```
(object Class "ALUMNO"
operations (list Operations
(object Operation "obtenerNombre"
result "String"
opExportControl "Public"
)
(object Operation "obtenerNotas"
opExportControl "Protected"
)
(object Operation "matricular"
opExportControl "Private"
))
))
```

```

class_attributes      (list class_attribute_list
                      (object ClassAttribute "nombre"
                      )
                      (object ClassAttribute "edad"
                      type          "Integer"
                      exportControl "Protected")
                      (object ClassAttribute "estudios"
                      initv         "COU"
                      exportControl "Public"))))

```

### 2.1.3. Tipos de clases

Rational Rose permite definir otros tipos de clases. Algunas de estas clases están definidas en UML, como son las parametrizables y utilidad, y otras no, como son parametrizable-utilidad o instanciada-utilidad.

La codificación de este tipo de clases es equivalente a la clase estándar que hemos estudiado anteriormente, con la diferencia de que la primera línea en la que aparecía el siguiente código

```
(object Class "Nombre-Clase"
```

es sustituida por el tipo de clase representada.

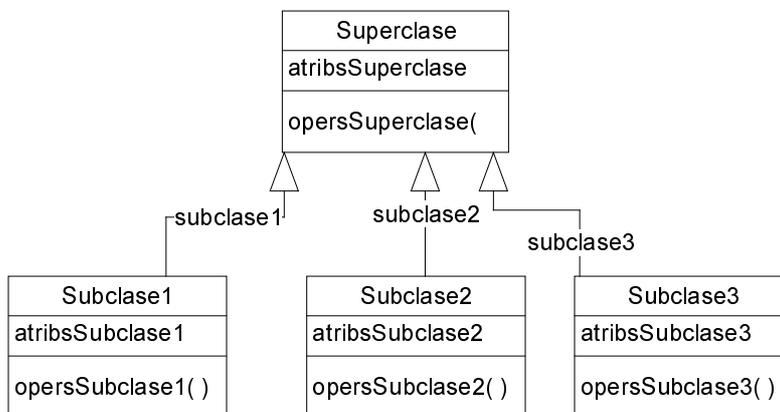
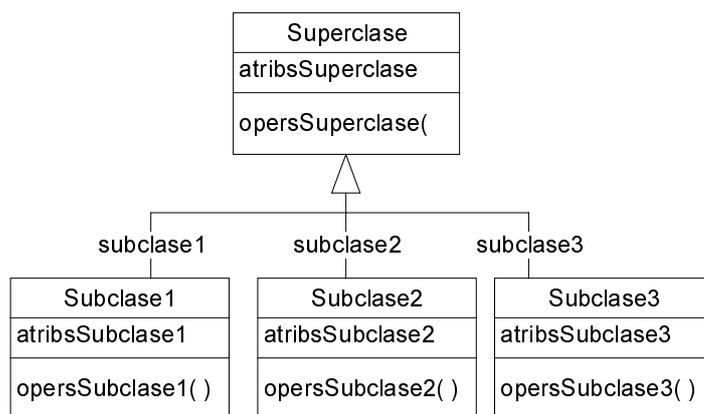
Los tipos de clases y la codificación correspondiente que podemos encontrar son las siguientes:

- **Clase Parametrizable**  
(object Parameterized\_Class "Nombre-Parametrizable"
- **Clase Instanciada**  
(object Instantiated\_Class "Nombre-Instanciada"
- **Clase Utilidad**  
(object Class\_Utility "Nombre-Utilidad"
- **Clase Parametrizable-Utilidad**  
(object Parameterized\_Class\_Utility "Nombre-ParamUtilid"
- **Clase Instanciada-Utilidad**  
(object Instantiated\_Class\_Utility "Nombre-InstancUtilid"
- **Metaclase**  
(object Metaclass "Nombre-Metaclase"

Estas son las clases que permite definir Rational Rose, pero en este documento sólo se van a tratar las clases ordinarias especificadas en el apartado anterior "Clases".

## 2.2. GENERALIZACIÓN

Una generalización es básicamente una relación entre un elemento más general y uno más específico. Se compone de una clase superclase y una subclase, donde los atributos, operaciones y asociaciones de la superclase se heredan en la subclase. Además una superclase puede tener varias superclases y heredar de todas ellas (herencia múltiple). Su representación gráfica puede ser de dos formas como vemos a continuación, siendo la primera la más conveniente porque refleja con mayor claridad que las subclases son 3 especializaciones del mismo tipo con respecto a la superclase:



La estructura general de una generalización es la siguiente:

```

(object Class "Superclass"
  operations (list Operations
    (object Operation "opersSuperclass"
      opExportControl "Public"
    )
  ))
class_attributes (list class_attribute_list
  (object ClassAttribute "atribsSuperclass"
  )))
  
```

```
(object Class "Subclase1"
  superclasses (list inheritance_relationship_list
    (object Inheritance_Relationship
      label "subclase1"
      supplier "Superclase"
    ))
  operations (list Operations
    (object Operation "opersSubclase1"
      opExportControl "Public"
    ))
  class_attributes (list class_attribute_list
    (object ClassAttribute "atribSubclase1"
    )))

(object Class "Subclase2"
  superclasses (list inheritance_relationship_list
    (object Inheritance_Relationship
      label "subclase2"
      supplier "Superclase"
    ))
  operations (list Operations
    (object Operation "opersSubclase2"
      opExportControl "Public"
    ))
  class_attributes (list class_attribute_list
    (object ClassAttribute "atribSubclase2"
    )))

(object Class "Subclase3"
  superclasses (list inheritance_relationship_list
    (object Inheritance_Relationship
      label "subclase3"
      supplier "Superclase"
    ))
  operations (list Operations
    (object Operation "opersSubclase3"
      opExportControl "Public"
    ))
  class_attributes (list class_attribute_list
    (object ClassAttribute "atribSubclase3"
    )))
```

### 2.2.1. Especificación

Una generalización se define básicamente en dos partes: la superclase y las subclases.

- Superclase

La generalización comienza definiendo la superclase, como una clase normal de las estudiadas en el apartado 3.1:

```
(object Class "Superclass"
operations      (list Operations
                 (object Operation "opersSuperclass"
                  opExportControl  "Public"
                  ))
class_attributes (list class_attribute_list
                 (object ClassAttribute "atribSuperclass"
                  )))
```

- Subclases

Posteriormente se definen las subclases, de forma parecida a una clase normal, pero añaden algo de código diferente:

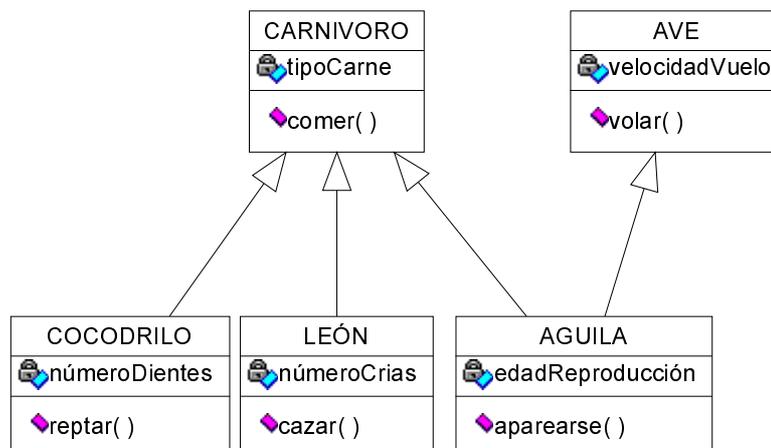
```
superclasses    (list inheritance_relationship_list
                 (object Inheritance_Relationship
                  supplier          "Superclass"
                  ))
```

La primera línea indica que se trata de una subclase y las otras dos hacen referencia al nombre de su superclase. Con esto, el código de una subclase queda de la siguiente forma, como ya hemos visto anteriormente:

```
(object Class "Subclass1"
superclasses    (list inheritance_relationship_list
                 (object Inheritance_Relationship
                  supplier          "Superclass"
                  ))
operations      (list Operations
                 (object Operation "opersSubclass1"
                  opExportControl  "Public"
                  ))
class_attributes (list class_attribute_list
                 (object ClassAttribute "atribSubclass1"
                  )))
```

El resto de las subclases son definidas del mismo modo.

## 2.2.2. Ejemplo de Generalización en Rational Rose



```

(object Class "CARNIVORO"
  operations (list Operations
    (object Operation "comer"
      concurrency "Sequential"
      opExportControl "Public"
    ))
  class_attributes (list class_attribute_list
    (object ClassAttribute "tipoCarne"
    )))

(object Class "AGUILA"
  superclasses (list inheritance_relationship_list
    (object Inheritance_Relationship
      supplier "CARNIVORO"
    )
    (object Inheritance_Relationship
      supplier "AVE"
    )
  )
  operations (list Operations
    (object Operation "aparearse"
      concurrency "Sequential"
      opExportControl "Public"
    ))
  class_attributes (list class_attribute_list
    (object ClassAttribute "edadReproducción"
    )))

(object Class "LEÓN"
  superclasses (list inheritance_relationship_list
    (object Inheritance_Relationship
      supplier "CARNIVORO"
    )
  )
)

```

```

        operations      (list Operations
            (object Operation "cazar"
                concurrency    "Sequential"
                opExportControl "Public"
            ))
        class_attributes (list class_attribute_list
            (object ClassAttribute "númeroCrias"
            )))

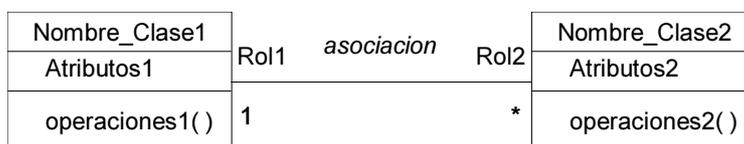
(object Class "COCODRILO"
    superclasses (list inheritance_relationship_list
        (object Inheritance_Relationship
            supplier    "CARNIVORO"
        ))
    operations      (list Operations
        (object Operation "reptar"
            concurrency    "Sequential"
            opExportControl "Public"
        ))
    class_attributes (list class_attribute_list
        (object ClassAttribute "númeroDientes"
        )))
(object Class "AVE"
    operations      (list Operations
        (object Operation "volar"
            concurrency    "Sequential"
            opExportControl "Public"
        ))
    class_attributes (list class_attribute_list
        (object ClassAttribute "velocidadVuelo"
        )))

```

### 2.3. ASOCIACIÓN

En una asociación entre dos clases, encontramos el nombre de la asociación, las clases que asocia, el rol o papel que desempeña cada una de ellas, las cardinalidades en ambos sentidos (hacia las dos clases que une) y la navegabilidad.

Se representa del siguiente modo:



Una asociación es codificada por Rational Rose de la siguiente forma:

```
(object Association "Nombre_Asociacion"
  roles (list role_list
    (object Role "Rol1"
      label "Rol1"
      supplier "Nombre_Clasel"
      client_cardinality (value cardinality "1") *I
      is_navigable TRUE)

    (object Role "Rol2"
      label "Rol2"
      supplier "Nombre_Clase2"
      client_cardinality (value cardinality "n") *I
      is_navigable TRUE)))
```

<sup>I</sup> Las multiplicidades más habituales son 0..1, 1..1, 0..\*, y 1..\*, dónde "\*" también puede ser representado como "n" o "N". Los intervalos 1..1 y 0..\* se pueden expresar abreviadamente como 1 y \*.

### 2.3.1. Especificación

Vemos mas detalladamente la especificación de la asociación.

- Nombre

Es identificado con el siguiente código:

```
(object Association "Nombre-Asociacion"
```

- Clases asociadas

Posteriormente se definen las dos clases que asocia, con el rol que representa, la cardinalidad y la especificación de si es navegable o no, a partir del siguiente código:

```
roles (list role_list
```

Rol que representa la clasel en la asociación y su etiquetado en el diagrama cuando se ha definido un nombre de rol (cuando no se indica el nombre del rol para esa clase, Nombre-Rol1 aparece con un valor \$UNNAMED\$). Si se define un nombre de rol entonces tendremos una etiqueta:

```
(object Role "Nombre-Rol1"
  label "Rol1"
```

Nombre de la clase asociada a ese rol (clasel):

```
supplier "Nombre_Clasel"
```

Cardinalidad:

```
client_cardinality(value cardinality "1")
```

Navegabilidad o no en este sentido:

```
is_navigable TRUE)
```

Por último, nos encontramos los parámetros correspondientes a la otra clase asociada:

```
(object Role "Rol2"
  label "Rol2"
  supplier "Nombre_Clase2"
  client_cardinality(value cardinality "n"))))
```

En este caso se ha puesto que el rol2 no es navegable desde el rol1, suprimiendo la etiqueta “is\_navegable”.

### 2.3.2. Ejemplo de Asociación en Rational Rose.



```
(object Association "estudia"
  roles      (list role_list
    (object Role "$UNNAMED$3"
      supplier      "Materia"
      client_cardinality (value cardinality "1..n")
      is_navigable  TRUE)

    (object Role "$UNNAMED$4"
      supplier      "Estudiante"
      client_cardinality (value cardinality "1")
      is_navigable  TRUE)))
```

En este ejemplo no se han definido los roles que representan cada una de las clases de la asociación, por ello aparecen identificados con “\$UNNAMED” seguido de un código interno propio de Rational Rose.

### 2.3.3. Navegación

En principio, las asociaciones que aparecen en un diagrama de clases son navegables en ambas direcciones, lo que quiere decir que la comunicación es posible en los dos sentidos y los enlaces son vistos como canales de navegación entre objetos. Existe la posibilidad de tener una asociación navegable en un único sentido y esto se vería como una *Semiasociación*. Esta semiasociación se representa con una flecha que indica el sentido de la navegación:



En Rational Rose esta restricción no añade información al código, lo que ocurre es que cuando no es navegable se elimina la línea `is_navigable TRUE` de la clase que no permite navegación, como vemos en este ejemplo donde la línea es eliminada de “Clase\_Asociada2”:

```
(object Association "Nombre_Asociacion"
  roles          (list role_list
    (object Role "$UNNAMED$2"
      supplier    "Clase_Asociada2"
      client_cardinality (value cardinality "n"))

    (object Role "$UNNAMED$3"
      supplier    "Clase_Asociada1"
      client_cardinality (value cardinality "n")
      is_navigable TRUE))
```

La navegación también es utilizada para las agregaciones y clases-asociaciones y su representación es equivalente.

#### 2.3.4. Restricciones de una Asociación

En general, en UML se puede definir una restricción sobre cualquier elemento del modelo, por ejemplo una clase, pero Rational Rose sólo permite hacerlo sobre una asociación, clase-asociación o agregación. Las restricciones dentro de una asociación son recogidas en el código por el parámetro que aparece en **negrita** en el siguiente fragmento, dentro de la definición de asociación:

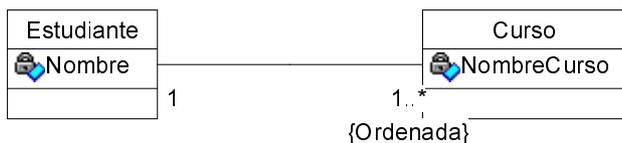
```
(object Role "Nombre-Rol"
  supplier    "Nombre-Clase"
  client_cardinality (value cardinality "1")
  Constraints  "Restricción"
  is_navigable TRUE)
```

En Rational Rose, las restricciones son recogidas en formato libre de forma que cuando escribimos la restricción en una línea, la codificación es la que descrita anteriormente, pero si tenemos que la restricción ocupa varias líneas la codificación es la siguiente:

```
(object Role "Nombre-Rol"
  supplier    "Nombre-Clase"
  client_cardinality (value cardinality "1")
  Constraints
|Restricción1
|Restricción2

  is_navigable TRUE)
```

Veamos un ejemplo en Rational Rose:



```

(object Association "$UNNAMED$0"
  roles          (list role_list
    (object Role "$UNNAMED$1"
      supplier      "Curso"
      client_cardinality (value cardinality "1..n")
      Constraints  "Ordenada"
      is_navigable  TRUE)

    (object Role "$UNNAMED$2"
      supplier      "Estudiante"
      client_cardinality (value cardinality "1")
      is_navigable  TRUE))))
  
```

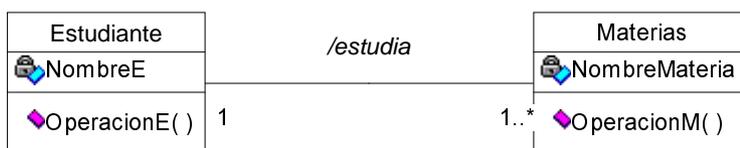
### 2.3.5. Asociación Derivada

Cuando representamos una asociación derivada, la codificación es la misma que para las asociaciones normales, pero añaden la siguiente línea de fichero al final de la definición:

```

derived          TRUE) )
  
```

Veamos un ejemplo de Asociación Derivada en Rational Rose, tomando como referencia la asociación del ejemplo que aparece en el apartado 3.2.2 y suponiendo que esta asociación fuera derivada, tendríamos la siguiente codificación:



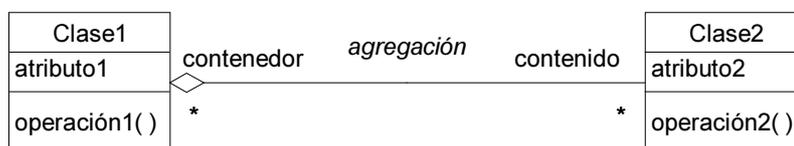
```

(object Association "estudia"
  roles          (list role_list
    (object Role "$UNNAMED$0"
      supplier      "Materias"
      client_cardinality (value cardinality "1..n")
      is_navigable  TRUE)

    (object Role "$UNNAMED$1"
      supplier      "Estudiante"
      client_cardinality (value cardinality "1")
      is_navigable  TRUE))
  derived          TRUE) )
  
```

## 2.4. AGREGACIÓN

En una agregación UML encontramos una clase principal y una clase agregada o subordinada. Su representación general es la siguiente:



En este caso la codificación asociada es igual a la utilizada para una Asociación (apartado “ASOCIACIÓN” de este documento), con la única variación de que se añade la característica “**is\_aggregate TRUE**”, para la clase contenedora, es decir, la que contiene el rombo en el diagrama UML, como vemos en la estructura general del fichero Rational Rose:

```

(object Association "agregación"
  roles          (list role_list
    (object Role "contenedor"
      label          "contenedor"
      supplier       "Clase1"
      client_cardinality (value cardinality "n")
      is_navigable   TRUE
      is_aggregate   TRUE)

    (object Role "contenido"
      label          "contenido"
      supplier       "Clase2"
      client_cardinality (value cardinality "n")
      is_navigable   TRUE))))
  
```

### 2.4.1. Ejemplo de Agregación en Rational Rose



```

(object Association "$UNNAMED$0"
  roles          (list role_list
    (object Role "$UNNAMED$1"
      supplier       "TITULACION"
      client_cardinality (value cardinality "1..n")
      is_navigable   TRUE
      is_aggregate   TRUE)

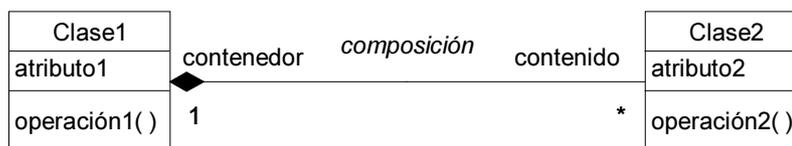
    (object Role "$UNNAMED$2"
      supplier       "ASIGNATURA"
      client_cardinality (value cardinality "n")
      is_navigable   TRUE))))
  
```

```
(object Role "$UNNAMED$2"
  supplier      "ASIGNATURA"
  client_cardinality (value cardinality "n")
  Containment    "By Reference"
  is_navigable   TRUE)))))
```

Como observamos en el ejemplo, el código que difiere de una asociación normal y que define una agregación es el indicado con la letra negrita.

## 2.5. COMPOSICIÓN

Se trata de un caso particular de agregación, donde los componentes son físicamente contenidos por la clase contenedora. Esto implica una restricción sobre la cardinalidad en el lado del contenedor, que solo podrá tomar el valor 1 (Rational Rose no controla esta restricción). Se representa gráficamente del mismo modo que la agregación, pero el rombo correspondiente a la parte del contenedor aparece en negro.



Este tipo de relación es representado por Rational Rose de la misma forma que en la agregación normal, pero con la diferencia de que en este caso se incluye un parámetro “**Containment “By Value”**” que identifica que esta agregación es de tipo composición.

### 2.5.1. Ejemplo de Composición en Rational Rose



```
(object Association "$UNNAMED$0"
  roles (list role_list
    (object Role "contenido"
      label      "contenido"
      supplier   "Coche"
      client_cardinality (value cardinality "1")
      is_navigable TRUE
      is_aggregate TRUE)
```

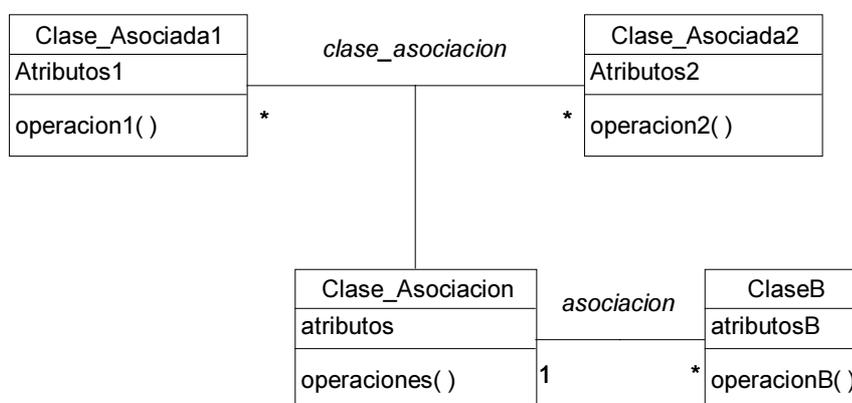
```

(object Role "contenedor"
 label          "contenedor"
 supplier       "Pieza"
 client_cardinality (value cardinality "n")
 Containment   "By Value"
 is_navigable  TRUE)))

```

## 2.6. CLASE ASOCIACIÓN

Este tipo de relaciones se caracteriza porque la asociación se representa por una clase, para tener la posibilidad de añadir atributos y operaciones de la asociación. Esta clase asociación es una clase instanciable como el resto, por ello es posible que participe en otras relaciones del modelo. Veamos su representación de forma genérica:



La clase asociación se representa como una asociación conectada mediante una línea punteada a una clase, y podemos observar que se trata de una clase normal, que a su vez se asocia de forma común con la clase B.

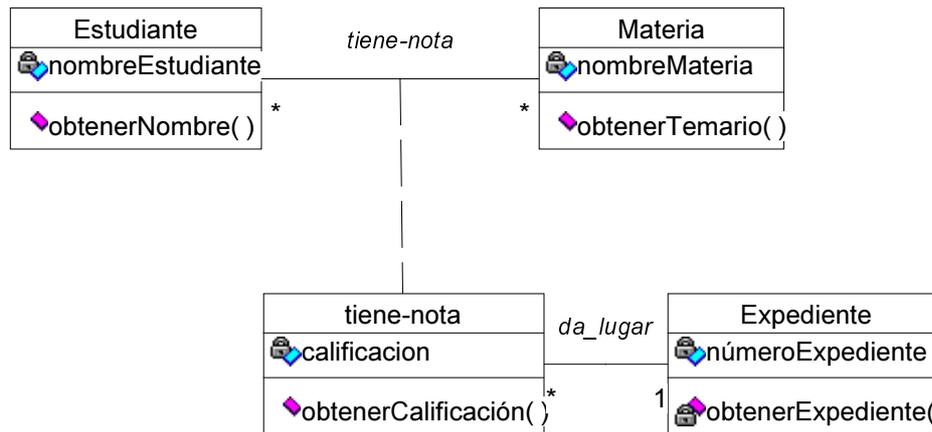
En Rational Rose estas asociaciones aparecen codificadas del mismo modo que las demás asociaciones, pero al final de la definición añade la siguiente línea:

```
AssociationClass "Clase_Asociacion")
```

para identificar la clase correspondiente. En el código de la clase en sí no hay ninguna señal distintiva que remita al código de la asociación (supongo que esto es así, comprobar y decir cómo es en todo caso).

La codificación de las *asociaciones atribuidas* (iguales a las anteriores pero no reciben un nombre específico y contienen atributos sin participar en relaciones con otras clases) es la misma que la anterior.

A continuación se presenta un ejemplo de clase asociación según Rational Rose:



```

(object Association "tiene-nota"
  roles (list role_list
    (object Role "$UNNAMED$2"
      supplier "Materia"
      client_cardinality (value cardinality "n")
      is_navigable TRUE)
    (object Role "$UNNAMED$3"
      supplier "Estudiante"
      client_cardinality (value cardinality "n")
      is_navigable TRUE))
  AssociationClass "tiene-nota")
  
```

```

(object Association "da_lugar"
  roles (list role_list
    (object Role "$UNNAMED$4"
      supplier "Expediente"
      client_cardinality (value cardinality "1")
      is_navigable TRUE)
    (object Role "$UNNAMED$5"
      supplier "tiene-nota"
      client_cardinality (value cardinality "n")
      is_navigable TRUE))))
  
```

### 3. ESTRUCTURA GENERAL DE FICHERO RATIONAL ROSE

Una vez vista la representación o codificación individual de los distintos elementos que pueden aparecer en un diagrama de clases, observamos la estructura general del fichero.

Dentro del fichero se definen 3 grandes vistas:

1. Vista de Casos de Uso (Use Case View)
2. Vista Lógica (Logical View)
3. Vista de Componentes (Component View)

De todas ellas, nos centraremos en la *vista lógica*, que es donde se almacena la información perteneciente a los diagramas de clases.

Vemos como queda la estructura general del fichero .MDL de Rational Rose:

```
(object Design "Logical View"

  file_name "C:\.....\fichero.mdl"

  root_usecase_package (object Class_Catogoy "Use Case View"
    .....*1

  root_category (object Class_Category "Logical View"

    logical_models (list unit_reference_list
      .....*2

      (object Class_Category "Paquetel"
        exportControl "Public"
        subsystem "Component View"*3
        logical_models (list unit_reference_list
          .....*2

          (object Class_Category "Paquetel_1"
            exportControl "Public"
            logical_models (list unit_reference_list
              .....*2

              (object Class_Category "Paquete2"
                exportControl "Public"
                logical_models (list unit_reference_list
                  .....*2

                logical_presentations (list unit_reference_list
                  .....*4

                root_subsystem (object SubSystem "Component View"*5
                  ...))
                quid "as2124gj351"))
```

\*1 Información referente a la vista de casos de uso, donde encontramos los propios Diagramas de Casos de Uso, Diagramas de Colaboración y Diagramas de Secuencia.

\*2 Información interna de Rational Rose, utilizada para la representación de los distintos elementos en pantalla.

\*3 Identifica que el diagrama contiene paquetes.

\*5 Codificación referente a los *diagramas de componentes* que representan los componentes físicos de una aplicación.

\*4 Información de clase, asociación, clase-asociación, agregación, composición, navegación y/o generalización del tipo estudiado en el apartado “Definición Rational Rose De Elementos del Diagrama de Clases” de este proyecto. Se definen primero las clases y subclases y posteriormente el resto de relaciones. Las posibles estructuras que podemos encontrar son:

- La codificación referente a clases, con sus atributos y operaciones:

```
(object Class "Clase"
```

- La codificación de generalización que comienza con el siguiente código:

```
(object Class "Subclase-Generalizacion"
superclasses (list inheritance_relationship_list
.....
```

- La codificación de asociación entre dos clases:

```
(object Association "Asociacion"
roles (list role_list
.....
```

- La codificación de asociación entre dos clases, a través de una clase-asociación:

```
(object Association "Asociacion"
roles (list role_list
.....
AssociationClass "ClaseAsociacion")
```

- La agregación:

```
(object Association "Agregacion"
roles (list role_list
(object Role "RolA"
.....
is_aggregate TRUE)
.....
```

- La codificación de composición, que varía en la anterior añadiendo una característica de **Containment "By Value"** a una de las clases:

```

(object Association "Composicion"
  roles (list role_list
    (object Role "RolA"
      .....
      is_aggregate      TRUE)
    (object Role "RolB"
      .....
      Containment "By Value"
      .....

```

La información de los paquetes se encuentra definida en la vista "Logical View" (vista lógica) cuya estructura se define en el siguiente apartado.

### 3.1. VISTA LÓGICA (REPRESENTACIÓN DE DIAGRAMA DE CLASES)

La parte que nos interesa estudiar dentro del fichero .MDL creado por Rational Rose, es la correspondiente a la Vista Lógica, que contiene todos los conceptos referentes a los diagramas de clases UML representados por Rational Rose.

Siguiendo la estructura general del fichero que aparece en el apartado 4 encontramos lo siguiente:

Identificamos el comienzo de la sección Vista Lógica a través de la siguiente línea:

```
(object Design "Logical View"
```

Posteriormente aparece el nombre del fichero Rational Rose donde se ha guardado la información, identificado por la etiqueta:

```
file_name "C:\.....\fichero.mdl"
```

Después pasa a definir la Vista de Casos de Uso, que en este trabajo no se va a estudiar. Aparece identificada con la siguiente codificación, donde se incluyen los objetos o elementos pertenecientes a los Diagramas de Casos de Uso, a los de Secuencia y a los Diagramas de Colaboración. El comienzo viene identificado de la siguiente forma:

```
root_usecase_package(object Class_Categoy "Use Case View"
```

Posteriormente llegamos a la parte interesante para este proyecto, que es la descripción de la vista lógica, identificada de la siguiente forma:

```
root_category (object Class_Category "Logical View"
```

La Vista Lógica contiene: Clases, Relaciones entre objetos (asociación, agregación, composición,...) y Paquetes donde se pueden agrupar Diagramas de Clases. También pueden incluir Diagramas de Colaboración y de Secuencia, pero no son objeto de este estudio.

Centrándonos en los diagramas de clases de la vista lógica, podemos encontrar varios diagramas de clases a nivel principal (con sus clases, asociaciones, etc.) y varios paquetes que incluyen a su vez otros diagramas de Clases.

### 3.1.1. Diagramas a nivel principal

Si encontramos diagramas de clases a nivel principal, lo encontraremos después del siguiente código:

```
logical_models (list unit_reference_list
```

Esto aparece posteriormente a la línea *root\_category (object Class\_Category “Logical View”*.

Inmediatamente después aparecen las clases y relaciones del diagrama principal o “Main”, como se ha visto en el apartado “Estructura General del Fichero Rational Rose”:

```
(object ....
```

### 3.1.2. Paquete

Una vez definidos los componentes que se encuentran en un primer nivel, pasan a definirse los Paquetes que contiene la Vista Lógica y que son definidos de la siguiente forma:

```
(object Class_Category “Nombre-Paquete”  
  exportControl “Public”
```

Si contiene subpaquetes aparecerá la siguiente línea:

```
  subsystem “Component View”
```

A partir de aquí la codificación de un paquete es igual a la utilizada en el nivel principal, comenzando con la siguiente codificación que identifica que ese paquete contiene elementos de un diagrama de clases:

```
logical_models (list unit_reference_list
```

Dentro de un paquete podemos encontrar subpaquetes que se definen como un objeto mas dentro del propio paquete (ver esquema general del apartado 3).

## 4. IMPLEMENTACIÓN EN JAVA DE ELEMENTOS DEL DIAGRAMA DE CLASES

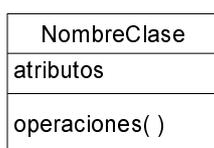
En este apartado se trata de identificar la correspondencia entre los elementos que podemos encontrar en un diagrama de clases Rational Rose y su respectiva implementación Java. Por lo tanto, en este apartado no se tomarán en cuenta aquellos elementos que no pueden ser representados en Rational Rose, como son las asociaciones ternarias.

Hay que destacar que aunque aquí se presentan conceptos básicos del lenguaje Java, para la comprensión de este proyecto es recomendable poseer unos conocimientos mínimos de Java, ya que aquí no se trata de explicar el lenguaje y no es el ámbito de este trabajo.

A continuación aparecen los distintos elementos que podemos encontrar en un diagrama de clases representado en Rational Rose:

### 4.1. CLASE

Una clase UML se compone de un nombre, atributos y operaciones que implementa. Esto es representado de la siguiente forma:



Cada clase representada en Rational Rose se corresponde con un fichero Java con extensión .java que contiene el código fuente de la definición de propiedades y métodos de la clase, o un fichero con extensión .class si se trata del ejecutable. La estructura de una clase sencilla sería la siguiente:

```
public class NombreClase {
    //Definición de propiedades de la clase Ejemplo
    private*1 tipo*2 propiedad;
    .....

    //Definición del constructor de la clase
    public NombreClase() {
        .....
    }

    //Definición de los métodos de la clase Ejemplo
    public*1 void*3 nombreMetodo (tipo*2 nombreAtributo, ..... )
    {
        .....
    }
    .....
```

```
//Definición del método main
public static void main(String[] args) {
    .....
}
}
```

\*<sup>1</sup> Podrá tomar el valor de public, protected o private

\*<sup>2</sup> Tipo de dato Java. Puede ser simple como char, int, etc., perteneciente a una clase Java como Vector, Hashtable o definido por el programador.

\*<sup>3</sup> Tipo de dato que devuelve el método de la clase. Void cuando el método no devuelve ningún dato o \*<sup>2</sup> en caso contrario.

En cada clase Java podemos encontrar la siguiente información:

- Nombre

```
public class NombreClase {
```

El nombre de la clase es identificado al inicio de la codificación, después de “public class”. Este nombre también queda reflejado en el nombre del fichero con extensión .java que contiene el código fuente de la clase. El ejecutable identificado con este mismo nombre con extensión .class aparece una vez compilado el código.

- Propiedades

```
private tipo propiedad;
```

Las propiedades de una clase UML, son los atributos de clase definidos en la misma. El tipo de visibilidad del atributo (público, privado ó protegido), se indica al principio de la definición de la propiedad, teniendo en cuenta que si éste no aparece se toma como público. El tipo al que pertenece ese atributo viene definido inmediatamente después del tipo de Visibilidad del mismo.

- Métodos

```
public void nombreMetodo (tipo nombreAtributo, ..... ) { .. }
```

Los nombres de los métodos que implementa esa clase, vienen identificados dentro del código como el nombre de las funciones o métodos de la clase.

### 4.1.1. Ejemplo de clase en Java

Se puede encontrar una clase “Contador.java” definida con el siguiente código:

```
public class Contador {  
  
    //Definición de propiedades de la clase Contador  
    private int contadorInterno;  
  
    //Definición del constructor de la clase  
    public Contador() {  
        contadorInterno=0;  
    }  
  
    //Definición de los métodos de la clase Ejemplo  
    public void incrementarIndice (int incremento)  
    {  
        contadorInterno=contadorInterno+incremento;  
    }  
}
```

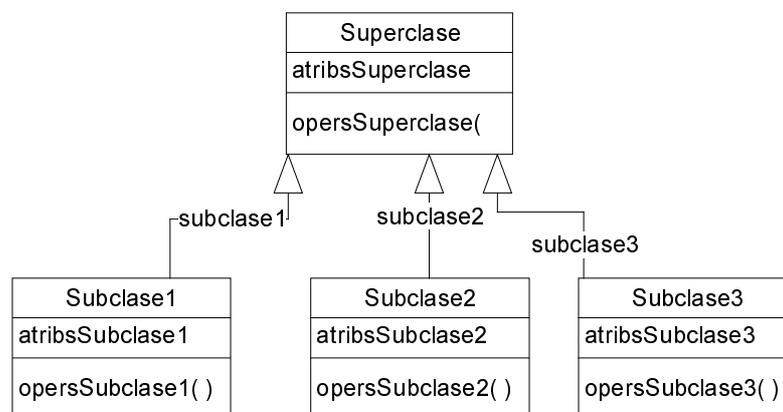
En el ejemplo identificamos:

- Nombre de la clase: Contador
- Atributos: contadorInterno con visibilidad privada
- Métodos: incrementarIndice

Una vez compilado el código, aparecerá el ejecutable “Contador.class”.

## 4.2. GENERALIZACIÓN

Como ya se ha explicado anteriormente en este documento, una generalización es básicamente una relación entre una clase superclase y una subclase, siendo su representación la siguiente:



Una generalización se define básicamente en dos conceptos: la superclase y las subclases.

Detectaremos que existe una generalización cuando en la definición de una clase Java encontremos la palabra “**extends**” en la cabecera:

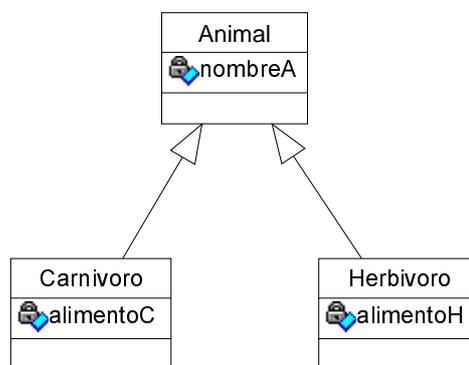
```
public class NombreSubclase extends NombreSuperclase
```

dónde encontramos la superclase “NombreSuperclase” y la subclase “NombreSubclase”.

Una de las características importantes de Java es que no permite la herencia múltiple, aunque UML sí lo permita, con lo que la herencia o generalización siempre será de una única clase.

#### 4.2.1. Ejemplo de Generalización en Rational Rose

Veamos un ejemplo de generalización:



La correspondiente codificación Java sería:

```
public class Animal {
    private string nombreA;

    Animal() {
    }
}
```

```
public class Carnivoro extends Animal
{
    private string alimentoC;

    Carnivoro() {
    }
}
```

```
public class Herbivoro extends Animal
{
    private string alimentoH;

    Herbivoro() {
    }
}
```

### 4.3. ASOCIACIÓN

En una asociación UML entre dos clases, encontramos el nombre de la asociación, las clases que asocia, el rol o papel que desempeña cada una de ellas, las cardinalidades y la navegación en ambos sentidos (hacia las dos clases que une).

En UML se representa del siguiente modo:



Las asociaciones no son localizadas en una implementación Java tan fácilmente como los elementos del diagrama de clases vistos anteriormente (clases y generalizaciones). Para identificar cuándo existe una asociación en Java y qué valor toman los elementos que representan la misma, lo que se hace es observar los atributos de clase. Si se encuentra un atributo de tipo igual a una clase creada por el programador (no propia de Java como son las clases Button, Label,.. o las básicas como int o String) nos encontramos ante una asociación.

Una asociación Java podría ser de la siguiente forma:

```

public class Nombre_Clase1 {
    //Atributos de clase
    private Nombre_Clase2 rol1[]1;
    .....

    public Nombre_Clase1() {
        .....
    }
    .....
}

public class Nombre_Clase2 {
    //Atributos de clase
    private Nombre_Clase1 rol2;
    .....

    public Nombre_Clase2() {
        .....
    }
    .....
}
  
```

<sup>1</sup>En este y sucesivos ejemplos, para implementar la cardinalidad n de un atributo se ha utilizado el tipo de datos **Array**. Este tipo de dato, representa una lista de elementos, pero no representa realmente una cardinalidad indeterminada “n”, puesto que debe ser creado con un límite acerca de los elementos que va a contener:

```
private Nombre_Clase2 rol1[]=new Nombre_Clase2[10];
```

Como se puede observar, en este caso el array tendrá como máximo 10 objetos de tipo Nombre\_Clase2.

Los tipos de datos que de verdad representan una cardinalidad “n”, son los que implementan alguno de los siguientes interfaces: List, Map, Collection ó Set. Entre estos tipos tenemos tipos de datos como son “Vector”, “Stack”, etc., que realmente son creados sin imponer un límite de tamaño, con lo que pueden ser rellenos con tantos objetos como se desee.

Se ha utilizado este tipo de dato para poder ofrecer una visión de los datos que va a contener el Array. Si utilizásemos el vector la definición sería:

```
private Vector rol2;
```

De este modo no tendríamos forma de saber que clase de objetos va a contener rol2 y por lo tanto no se podría deducir que existe una asociación entre Nombre\_Clase1 y Nombre\_Clase2. Este problema es tratado en el apartado “Complejidad de las Cardinalidades” de este documento.

Veamos detalladamente la especificación de la asociación.

- Nombre

En Java no existe un nombre de asociación como tal, como ocurre en la representación UML.

- Clases asociadas

```
Nombre_Clase1 --> private Nombre_Clase2 rol1[];  
Nombre_Clase2 --> private Nombre_Clase1 rol2;
```

Para localizar las dos clases que forman parte de la asociación UML (Nombre\_Clase1 y Nombre\_Clase2) lo que se hace es identificar en una clase Java, un atributo de tipo “clase creada por el programador” (rol1 de tipo Nombre\_Clase2) y buscar en la otra clase (Nombre\_Clase2) el atributo que sea de tipo igual a la primera (rol2 de tipo Nombre\_Clase1). En una clase identificaremos tantas asociaciones como atributos contenga de tipo “clase creada por el programador”.

- Rol

El rol de las clases que forman cada asociación se obtiene del nombre que toman los atributos de clase que forman la asociación. De este modo tenemos que el rol que toma la clase Nombre\_Clase2 en la clase Nombre\_Clase1 es rol1 y el rol que toma la clase Nombre\_Clase1 en la clase Nombre\_Clase2 es rol2.

- Navegabilidad

En el caso de que en alguna de las dos clases que forman la asociación no aparezca el atributo que referencia a la otra, tendremos una asociación no navegable en un sentido, es decir, no navegable desde la que no define el atributo a la que sí lo hace. En el ejemplo que estamos viendo es navegable en ambos sentidos, pero a continuación veremos otros ejemplos de Asociación donde esto no es así.

- Cardinalidad

La cardinalidad de la asociación (cardinalidades máximas en ambos extremos) se puede determinar según se defina la propiedad que hace referencia a la clase que asocia. Si tenemos una propiedad simple como el atributo rol2 en la clase Nombre\_Clase2, la cardinalidad en ese extremo será “1”. Si lo que tenemos es una propiedad definida como un conjunto de valores o lista, tendremos una cardinalidad “n” como ocurre con el atributo rol1 en la clase Nombre\_Clase1.

La cardinalidad mínima no se puede especificar en la definición de la clase Java, al igual que una máxima con valor limitado (sólo puede ser 1 ó n), como se estudiará mas adelante.

#### 4.3.1. Ejemplos de Asociación en Java

Si tenemos la siguiente asociación en Rational Rose:



Obtendríamos la siguiente implementación Java:

```

public class Profesor {
    private String nombreProfesor;
    private Materia materia[];

    Profesor() {
        .....
    }

    public void impartirClase() {
        .....
    }
}
  
```

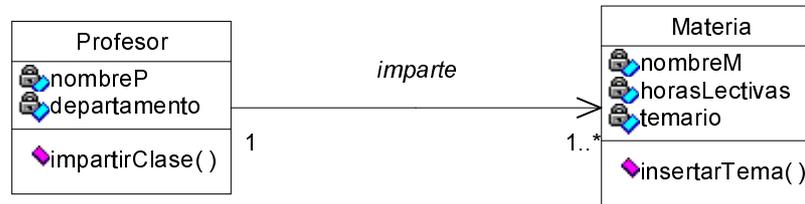
```

public class Materia {
    private String nombreMateria;
    private Profesor profesor;

    Materia() {
        .....
    }

    public void asignarTema() {
        .....
    }
}
  
```

En esta implementación se puede observar que, a diferencia del diagrama UML, aquí no aparece el nombre de la asociación “*imparte*”. Las cardinalidades máximas están recogidas con “1” para el atributo profesor en la clase Materia y “n” para el atributo materia (materia[]) en la clase Profesor. Además se puede observar que en este ejemplo la navegabilidad es en ambos sentidos, veamos algunos ejemplos donde esto no ocurre:



```

public class Profesor {
    private string nombreP;
    private string departamento;
    private Materia impone[];

    Profesor() {
    }
    public void impartirClase() {
    }
}

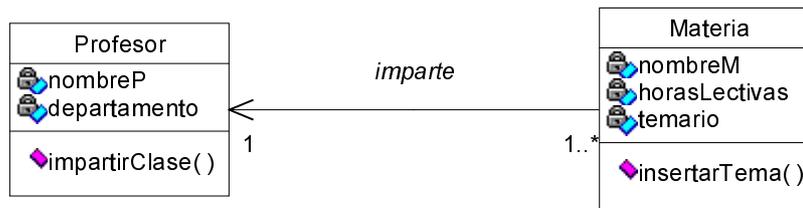
public class Materia {
    private string nombreM;
    private int horasLectivas;
    private string temario;

    Materia() {
    }
    public void insertarTema() {
    }
}
  
```

En este caso observamos que no aparece el nombre del rol, con lo que el nombre que toma la propiedad Java que identifica la asociación es el nombre de la misma asociación, “*imparte*”.

Esta relación representa que la asociación entre Profesor y Materia es únicamente navegable en un sentido (de Profesor a Materia). Como vemos no existe propiedad en la clase Materia que relacione ésta con Profesor, de forma que desde la clase Materia no se puede llegar hasta Profesor y por lo tanto no hay forma directa de conocer qué profesor la impone.

Si ahora indicamos que la navegabilidad es en sentido contrario, veremos como desaparece de la clase Profesor la propiedad que le relaciona con las materias que impone y se incluye una propiedad en la clase Materia que referencia al Profesor que la impone:



```

public class Profesor {
    private string nombreP;
    private string departamento;

    Profesor() {
    }

    public void impartirClase() {
    }
}

public class Materia {
    private string nombreM;
    private int horasLectivas;
    private string temario;
    private Profesor imparte;

    Materia() {
    }
    public void insertarTema() {
    }
}
  
```

En este caso se pretende representar que desde el Profesor no se puede deducir directamente las materias que éste imparte.

#### 4.3.2. Complejidad de las cardinalidades

- Cardinalidad “n”

Como se ha adelantado en el apartado anterior, para recoger una cardinalidad “n” en un lado de la asociación se necesita un atributo de tipo "lista" que contenga los objetos de la clase correspondiente.

```
private Nombre_Clase2 rol1[];
```

En Java, un conjunto de elementos no tiene por qué definirse siempre de este modo, en el que queda claramente definido que se trata de una lista de objetos del tipo Nombre\_Clase2. También existen otros tipos de datos (*colecciones de Java*) que permiten trabajar con grupos de objetos y que realmente representan la cardinalidad indefinida “n”, pero que en su definición no ofrecen información acerca de los tipos de datos que va a almacenar

posteriormente. De esta forma podremos encontrarnos con una definición de atributo de la siguiente forma:

```
private Vector roll;
```

Donde sabemos que el vector roll va a contener una serie de objetos, por lo tanto cardinalidad n, pero no podemos identificar el tipo de los mismos, lo que impide saber si existe una asociación o no con otra clase y cuál es el nombre de esta clase con la que se asocia ésta que contiene el atributo.

Hay un número limitado de estos tipos de datos que permiten guardar objetos: ArrayList, Dictionary, HashMap, HashSet, Hashtable, LinkedList, Stack, TreeMap, TreeSet, Vector. También existen otros tipos de datos que, aunque son abstractos, también permiten guardar objetos: Enumeration, Iterator, Set, ListIterator, Map, SortedMap, Collection.

Todos estos tipos tienen en común que son clases que implementan alguno de los siguientes interfaces: List, Map, Collection ó Set; pero cada uno tiene sus propias funciones de inserción y extracción de objetos que hacen que resulte más complejo el estudio.

- Cardinalidad máxima distinta de “n”

Hasta el momento sólo se ha hablado de cardinalidades máximas en ambos lados de la asociación, pero no se han tratado las cardinalidades mínimas ni las cardinalidades máximas que contengan un valor concreto y no “n”.

Además de las cardinalidades máximas en extremos 1:n, 1:1 ó n:n podemos encontrarnos otro tipos de cardinalidades máximas en una asociación como es X:Y donde X e Y representan un número entero definido de antemano.

Se pueden comprobar las cardinalidades máximas estudiando la definición de los atributos de las clases que forman parte de la asociación, de forma que si tenemos 1 como cardinalidad máxima en ambos extremos encontraremos una definición simple del atributo que forma la asociación en las dos clases:

```
public class ClaseAsociada1
{
    private ClaseAsociada2 nombreAtributoC1;
    ...
}

public class ClaseAsociada2
{
    private ClaseAsociada1 nombreAtributoC2;
    ...
}
```

Si se trata de unas cardinalidades máximas X:Y tendremos la siguiente definición de atributos:

```
public class ClaseAsociada1
{
    private ClaseAsociada2 atributoC1[]=new ClaseAsociada2[X*1];
    ...
}

public class ClaseAsociada2
{
    private ClaseAsociada1 atributoC2[]=new ClaseAsociada1[Y*1];
    ...
}
```

\*1X e Y serán enteros que indican el tamaño de las listas

En estos ejemplos Java, se han usado atributos de tipo array para contener una lista de objetos, lo que permite que por la propia definición obtengamos el número de objetos que va a contener (definido por el entero X) y por lo tanto la cardinalidad exacta de la asociación.

No obstante, como ya se ha estudiado, puede que se encuentre una relación del tipo X:Y implementada como una colección de Java de la cuál no se podría deducir la cardinalidad, debido a que no se especifica en la propia definición del atributo como en el caso de los arrays. Sólo se puede obtener la cardinalidad exacta cuando la aplicación se encuentre en ejecución, ya que también es posible que exista en el código fuente un bucle donde se introducen objetos al atributo de tipo colección, pero no sabemos exactamente cuántos.

- Cardinalidades mínimas

No es posible recoger las cardinalidades mínimas en la definición de la propiedad Java (en la implementación sí es posible, ya que se puede implementar cualquier regla deseada; lo que ocurre es que no será de forma estándar o autocontrolada). Se trata de un dato semántico del diagrama de clases que indica el mínimo número de objetos que pueden estar asociados, pero en Java este dato no siempre podemos obtenerlo sobre el código y necesitaríamos ejecutarlo para obtener esta información, de forma que no podemos decir si es distinto de 0 simplemente cuando dentro del código asignamos un objeto al atributo.

### 4.3.3. Enlaces

No hay que confundir las asociaciones vistas hasta el momento con los enlaces. Es posible que se encuentre a lo largo de la definición de una clase, asociaciones no permanentes, por ejemplo con objetos pasados por parámetro; estos objetos no han sido definidos como atributos de mi clase y por lo tanto no tienen una asociación permanente con esta clase, sino que son usados por ella ocasionalmente, por ejemplo dentro de una operación. En este caso lo que tenemos son enlaces temporales que no llegan a ser una instancia de asociación como las vistas anteriormente.

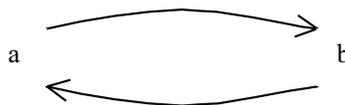
Recordemos que un enlace, normalmente una instancia de asociación, especifica una ruta por la cual un objeto puede enviar un mensaje a otro (o el mismo) objeto. La mayoría de las veces esto es insuficiente para especificar que tal camino existe. Si es necesario ser más preciso sobre cómo existe ese camino, cómo el emisor conoce al receptor, se puede usar cualquiera de los siguientes estereotipos estándar:

- asociación: el objeto receptor es visible por una asociación.
- self: el objeto receptor es el mismo que el objeto emisor de la operación.
- global: el objeto receptor está dentro del alcance que engloba el objeto emisor.
- local: el objeto receptor está en un alcance local.
- parameter: el objeto receptor es un parámetro.

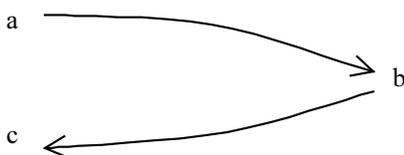
(The unified modeling language user guide, p. 210)

#### 4.3.4. Integridad

Llegados a este punto hay que aclarar que en este apartado nos hemos centrado en la comprobación de las cardinalidades representadas en el diagrama y posteriormente implementadas en Java, pero no podemos asegurar la integridad debido a la implementación en Java con punteros cruzados. Veamos esto detalladamente, si implementásemos una asociación 1:1 entre dos clases a y b, tendríamos un objeto “a” de la primera clase que apunta a un objeto “b” de la segunda clase y para cerrar la asociación se debe asignar al objeto b de esta última clase, el objeto “a” creado primeramente. Si lo representamos gráficamente tendríamos la siguiente figura:



Así es como se deben enlazar ambos objetos. Sin embargo, en la implementación Java no existe una forma implícita de saber que exactamente se ha realizado este enlace y no otro, es decir, que por ejemplo el objeto “b” apunta realmente a ese objeto “a” de la primera clase y no ha otro objeto “c” de esta misma clase. Véase la siguiente figura:



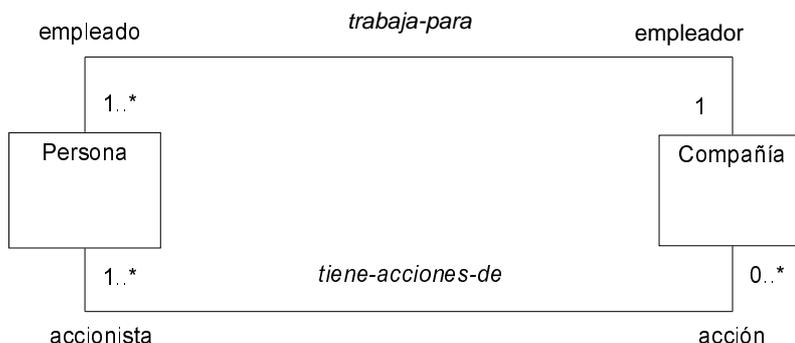
Como se puede observa en este último gráfico no se ha respetado la integridad de la asociación 1:1.

La única forma de comprobar que se respeta la integridad es mediante reglas programadas que no están en la propia definición de las clases, con los inconvenientes que esto tiene: no hay forma estándar de programarlas, quedan ocultas en el código, etc.

### 4.3.5. Múltiples asociaciones

Hasta el momento hemos visto asociaciones entre clases donde tenemos una única asociación entre las dos mismas clases, pero debemos pensar que existe la posibilidad de encontrar varias asociaciones entre las dos clases. En este caso volvemos a encontrar dificultades en la búsqueda de asociaciones UML sobre la implementación Java.

Supongamos el siguiente ejemplo:



La codificación resultante sería:

```

public class Persona {
    private Compañía empleador;
    private Compañía acción;

    .....
}

public class Compañía {
    private Persona empleado;
    private Persona accionista;

    .....
}
  
```

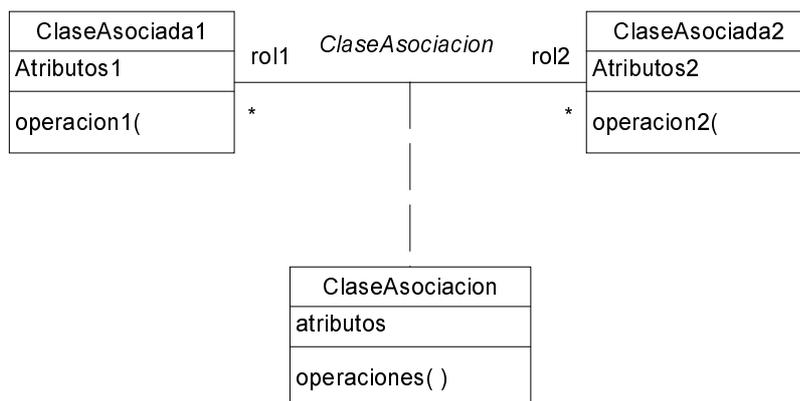
En este caso vemos claramente que existen dos asociaciones bidireccionales entre Persona y Compañía, pero ¿cuáles son las propiedades que implementan la asociación trabaja-para y cuáles las de tiene-acciones-de? Tenemos un problema de ambigüedad en la traducción UML-Java, se podría decir que existen dos asociaciones pero en lugar de empleador-empleado y acción-accionista podríamos hacer corresponder, erróneamente, empleador-accionista y acción-empleado. Esto es derivado del problema de los punteros cruzados, tratado en el apartado anterior.

Este mismo problema nos aparece cuando tenemos dos atributos definidos en una clase y en la otra sólo existe uno. En este caso ¿qué podemos pensar?, ¿existen dos asociaciones y sólo una de ellas es bidireccional ó quizás tenemos tres asociaciones unidireccionales y ninguna bidireccional?

En el caso en que los nombres de las propiedades Java coincidan con los nombre de rol UML o el nombre de la asociación, la ambigüedad deja de existir, por lo que el problema se resuelve si se dispone de un modelo UML de referencia.

#### 4.3.6. Clase Asociación

Una clase asociación se caracteriza por ser a la vez una asociación y una clase, lo que permite añadir atributos y operaciones a la asociación. Recordemos su representación de forma genérica:

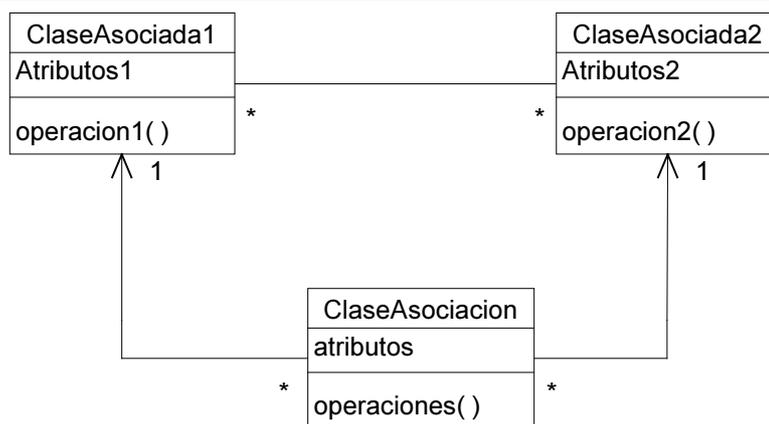


La clase asociación es representada con una conexión a través de una línea punteada y podemos observar que se trata de una clase normal, que a su vez puede asociarse de forma normal a otras clases.

En Java, al igual que las asociaciones, no es posible implementar directamente una clase asociación como tal. Podríamos intentar implementarlas de alguna de las siguientes formas:

- Clase externa

Ya que la idea de clase asociación es poder tener una clase intermedia donde poder recoger atributos pertenecientes a la asociación entre las dos clases, la primera idea que se nos ocurre es implementarla a través de una clase externa, formando una asociación circular. Este tipo de asociación sería navegable desde la clase asociación a las otras dos, de forma que para conocer los datos de la clase asociación tendríamos que recurrir a esta clase intermedia, que es la que contiene toda la información. Gráficamente tendríamos algo así:



La implementación básica resultante sería la siguiente:

```

public class ClaseAsociada1
{
    private tipo atributo1;
    private ClaseAsociada2 atributoC2 [];
    .....
}

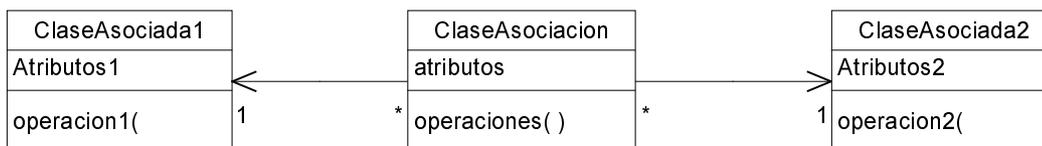
public class ClaseAsociada2
{
    private tipo atributo2;
    private ClaseAsociada1 atributoC1 [];
    .....
}

public class ClaseAsociacion {
    private tipo atributo;
    private ClaseAsociada1 claseAsociada1;
    private ClaseAsociada2 claseAsociada2;
    .....
}
  
```

Pero tenemos una dificultad que consiste en asegurar la coherencia. Al tratarse de un ciclo de asociaciones, existen dos caminos distintos para ir de una clase a otra y hay que controlar que se obtiene el mismo resultado por ambos, lo que resulta muy complejo de tener en cuenta en una programación Java donde las asociaciones se implementan a través de punteros.

- Clase intermedia no navegable

Otra forma de implementar este tipo de asociación, evitando ciclos, sería a través de una clase intermedia de forma que tendríamos asociaciones binarias dos a dos entre ClaseAsociada1 y ClaseAsociación y entre ClaseAsociada2 y ClaseAsociación. Tendríamos lo siguiente:



La implementación básica resultante sería la siguiente:

```

public class ClaseAsociada1
{
    private tipo atributo1;

    .....
}

public class ClaseAsociada2
{
    private tipo atributo2;

    .....
}

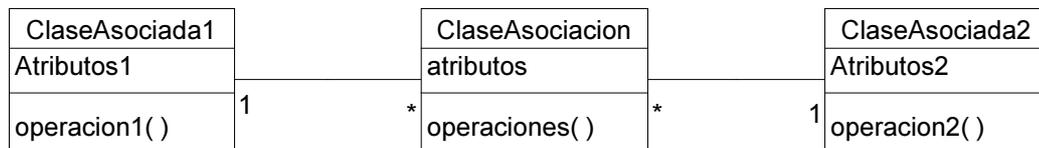
public class ClaseAsociacion {
    private tipo atributo;
    private ClaseAsociada1 claseAsociada1;
    private ClaseAsociada2 claseAsociada2;
    .....
}
  
```

Como se ha comentado, una clase asociación se caracteriza porque la asociación se representa por una clase ClaseAsociacion, que permite la posibilidad de añadir atributos y operaciones de la asociación y como vemos esto es posible en la implementación sin necesidad de ningún control adicional. Pero nos encontramos con un problema y es que esta implementación permite enlaces repetidos, es decir, se pueden crear varias instancias de ClaseAsociacion que hagan referencia a las mismas clases ClaseAsociada1 y ClaseAsociada2 y esto es una de las características principales de una clase asociación que no se permite en la semántica ofrecida por la clase asociación UML.

Además surge otro problema, ya que no se puede recoger de ninguna forma qué tipo de navegabilidad tiene, ni tampoco las cardinalidades máximas ó mínimas, puesto que si la clase está asociada de forma no navegable con las otras dos, las cardinalidades originales no quedan recogidas en el código.

## Clase intermedia navegable

Otra forma que se nos ocurre para arreglar parte del problema, es la misma clase intermedia pero navegable en ambos sentidos. Tendríamos lo siguiente:



La implementación básica resultante sería:

```

public class ClaseAsociada1
{
    private tipo atributo1;
    private ClaseAsociacion claseAsociacion[];
    .....
}

public class ClaseAsociada2
{
    private tipo atributo2;
    private ClaseAsociacion claseAsociacion[];
    .....
}

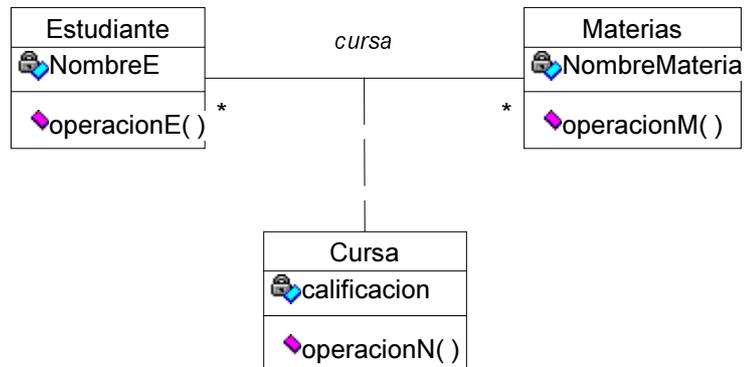
public class ClaseAsociacion {
    private tipo atributo;
    private ClaseAsociada1 claseAsociada1;
    private ClaseAsociada2 claseAsociada2;
    .....
}
  
```

En este caso podemos comprobar que es posible recoger las cardinalidades y el tipo de navegabilidad que tiene, pero seguimos sin arreglar el problema de instancias repetidas.

Aunque esta última representación es la más completa, se puede observar que no es posible diferenciar la implementación Java de una claseAsociación, de la obtenida al implementar asociaciones normales. ¿Quién nos asegura que esto es una clase asociación y no realmente dos asociaciones binarias entre dos clases?

### 4.3.7. Ejemplo de Clase Asociación en Java

Tenemos la siguiente clase asociación en Rational Rose:



La implementación Java que se obtiene, siguiendo la última forma de implementación descrita, es la siguiente:

```

public class Estudiante {
    private String nombreE;
    private Nota nota[];

    Estudiante() {
    }
}

public class Materia {
    private string nombreM;
    private Nota nota[];

    Materia() {
    }
}

public class Nota {
    private int calificacion;
    private Materia materia;
    private Estudiante alumno;

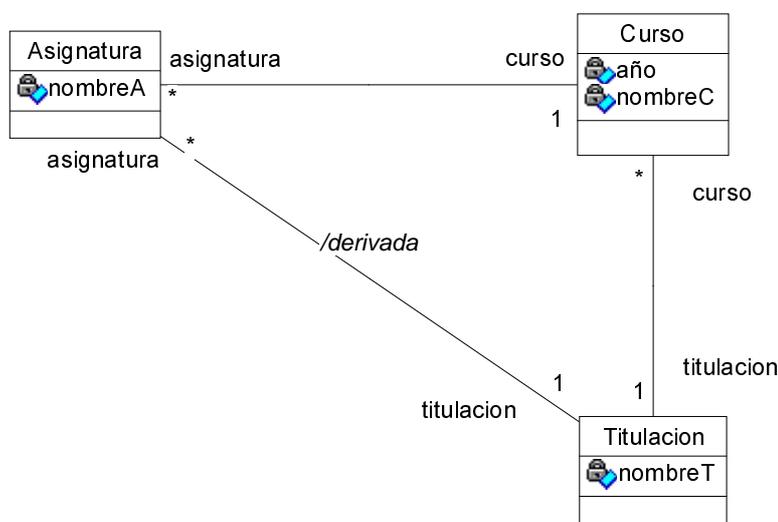
    Nota() {
    }
}
  
```

Observamos que la implementación es realizada como asociaciones binarias, navegables dos a dos, entre Estudiante y Notas y entre Materias y Notas.

### 4.3.8. Asociación Derivada

Una asociación derivada se caracteriza por permitir que una clase pueda enviar mensajes directamente a otra que se encuentre asociada a ella a través de clases intermedias. Esto no se puede implementar directamente en Java, si no se utilizan restricciones que garanticen la redundancia de la asociación. Recordemos que una asociación derivada es una asociación redundante (se puede deducir de otras asociaciones declaradas) que se refleja en el modelo por claridad, especialmente en el modelo de análisis. En el modelo de diseño se decide si se va a implementar o no la asociación derivada. Si se decide que no, no se reflejará en el modelo. Si se decide que sí, hay que tener en cuenta que junto a los punteros hay que implementar las restricciones que garanticen la redundancia de la asociación.

### 4.3.9. Ejemplo de Asociación Derivada



En el ejemplo gráfico observamos cómo podemos enviar mensajes directamente desde Asignatura a la clase Titulación, sin necesidad de llegar a ella a través de la clase Curso. Su implementación en Java sería la siguiente:

```

public class Asignatura {
    private int nombreA;
    public Curso curso;
    public Titulacion titulacion;
    .....
}

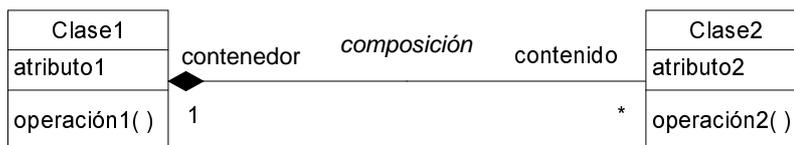
public class Curso {
    private int año;
    private int nombreC;
    public Asignatura asignatura[];
    public Titulacion titulacion;
    .....
}
  
```

```
public class Titulacion {
    private int nombreT;
    public Curso curso[];
    public Asignatura asignatura[];
    .....
}
```

Lo que se hace es tratar la asociación derivada como una asociación normal, puesto que no se puede implementar directamente en Java si no se utilizan restricciones que garanticen la redundancia de la asociación (que desde Asignatura a Titulación se permita llegar por los dos caminos existentes obteniendo el mismo resultado, es decir, un mecanismo que asigne a la asignatura su titulación).

#### 4.4. COMPOSICIÓN

Una composición UML es representada con un nombre, la clase principal ó contenedora y la clase subordinada ó contenida, donde los componentes son físicamente contenidos por la clase contenedora, lo que implica una restricción sobre la cardinalidad en el lado del contenedor, que solo podrá tomar el valor 1. Su representación general es la siguiente:



La implementación en Java de este tipo de relaciones es la siguiente:

```
public class Clase1 {
    private String atributo1;
    private Clase2 contenedor[];
    .....
}

public class Clase2 {
    private String atributo2;
    private Clase1 contenido;
    .....
}
```

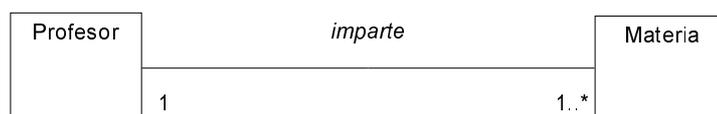
En este caso la implementación asociada es igual a la utilizada para una asociación 1:n y no tenemos ninguna forma de saber si realmente es una composición o se trata de una asociación sin más. Del mismo modo que para la asociación, en la composición obtenemos el nombre, las clases que pertenecen a la composición, el rol, la navegabilidad y la

cardinalidad (apartado “Asociación”). La composición también puede tomar cardinalidades máximas 1:1 o 1:x donde x será “n” o cualquier otro valor.

Estudiemos este hecho más en detalle:

- Asociación 1:x y Composición

Para explicar la problemática, se va a comparar una asociación 1:n con una composición con la misma cardinalidad máxima en ambos extremos. Por lo tanto, supongamos la siguiente asociación 1:n representada en Rational Rose del siguiente modo:



Esto representa una asociación entre dos clases de forma que un Profesor puede impartir una o varias Materias y una Materia es impartida por un único Profesor.

La implementación en Java de este tipo de asociaciones sería la siguiente:

```

public class Profesor {
    private Materia imparte[];
    .....
}

public class Materia {
    private Profesor imparte;
    .....
}
  
```

Si lo que tenemos es una composición tendríamos un diagrama en Rational Rose del siguiente tipo:



El diagrama representa que las Piezas son físicamente contenidas por un Coche. Esto implica una restricción sobre el valor de la multiplicidad en el lado del agregado (Coche) que sólo puede tomar el valor 1.

La implementación en Java de este tipo de relación sería la siguiente:

```

public class Coche {
    private Pieza contenido[];
    .....
}
  
```

```
public class Pieza {
    private Coche contenedor;
    .....
}
```

Observando la implementación de la asociación 1:n y la composición, podemos comprobar que obtenemos el mismo formato de código. Para ambas relaciones, lo que refleja el tipo de relación que implementamos es la definición de los atributos de clase, como hemos visto en los ejemplos. En principio no se ve ninguna diferencia entre ambos ejemplos, pero podríamos pensar que ésta se encuentra en la forma de crear los objetos.

Si estudiamos el caso de la composición, podemos pensar que la diferencia con la implementación de una asociación esta en que la creación de los objetos Pieza debería realizarse dentro del código de la clase Coche (ya sea en el propio constructor o en un método de la clase). Con esto podríamos decir que la clase Coche es la que contiene objetos de tipo Pieza, a diferencia de la asociación donde los objetos se crean desde fuera de ambas clases y posteriormente se asocian. Veamos un ejemplo:

- Ejemplo Agregación-Composición:

```
public class Coche
{
    private Pieza parte[];

    Coche()
    {
        parte=new Pieza[X*1];
        Pieza p=new Pieza();
        parte[0]=p;
        Pieza p2=new Pieza();
        Parte[1]=p2;
        .....
    }
}
```

\*1 X representa un número entero que será el número máximo de piezas que podrá contener un coche.

```
public class Pieza {
    private Coche contenedor;

    Pieza() {
    }
}
```

En este caso tenemos una composición, pero si no conociésemos el significado de los conceptos también podríamos decir que tenemos una asociación. De hecho una composición es un tipo de asociación que incluye otra semántica. Si la creación de objetos se hiciera desde fuera de estas clases el resultado sería el mismo:

```
public class Coche
{
    private string matricula;
    private Pieza contenedor[];
    private int ultimaPieza=0;

    Coche()
    {
        contenedor=new Pieza[X*1];
    }

    private void insertarPieza(Pieza p)
    {
        pieza[ultimaPieza]=p;
        ultimaPieza++;
    }
}
```

\*1 X representa un número entero que será el número máximo de piezas que podrá contener un coche.

```
public class Pieza {
    private string nombre;
    private Coche contenedor;

    Pieza() {
    }
    private void insertarCoche(Coche c)
    {
        contenedor=c;
    }
}
.....
```

```
Coche c=new Coche();
Pieza p=new Pieza();
Pieza p2=new Pieza();

c.insertarPieza(p);
c.insertarPieza(p2);
p.insertarCoche(c);
```

En este ejemplo obtenemos el mismo resultado que en el anterior, de forma que se asegura que la clase Pieza pertenecerá a un único Coche, por lo tanto multiplicidad 1 y que la clase Coche contendrá un conjunto de Piezas. Pero esto podría ser perfectamente una asociación del tipo 1:n entre Coche y Pieza.

Como vemos en el ejemplo, independientemente de dónde se creen los objetos obtendremos el mismo tipo de relación, puesto que la diferencia entre una asociación y una composición está en la semántica que cada una recoge y no en la propia implementación.

Para verlo claramente podemos partir de cualquiera de los dos ejemplos y sustituir la palabra “Coche” por “Profesor” y “Pieza” por “Materia”, obteniendo la implementación de la asociación 1:n entre Profesor y Materia del ejemplo anterior. La diferencia es solamente semántica, de forma que podemos recogerla en el diagrama de Clases UML pero no en la implementación Java.

La escasa semántica que tiene la agregación es oficialmente reconocida: “La diferencia entre agregación y asociación es a menudo un asunto de gusto mas que una diferencia semántica... La única semántica real que añade a la asociación es la restricción que cadenas de enlaces de agregación no pueden formar ciclos... A pesar de la semántica mínima que corresponde a la agregación, muchos piensan que es necesaria (por diferentes razones). Considérese una forma inocua de modelar.” (The unified modeling language user guide, p. 148).

La composición expresa que las partes está físicamente contenidas en el todo. Esto es imposible de implementar en Java, porque las propiedades de una clase siempre son punteros, o referencias, a objetos de otra clase, nunca es un valor físicamente contenido. En la asociación normal, y en la agregación, los objetos asociados no son físicamente contenidos, sino que se almacena una referencia o puntero hacia ellos, por tanto la implementación Java es adecuada. Dicho de otro modo: Java ofrece una implementación adecuada para asociación y agregación, pero no para composición, porque en Java no existe el contenido físico.

## 5. ANÁLISIS Y DISEÑO DE LA HERRAMIENTA DE VERIFICACIÓN

Después de realizar el estudio anterior, la aplicación queda bastante más sencilla de lo esperado inicialmente, al no poder realizar comparaciones con muchos de los elementos que aparecen en un diagrama de clases UML, ya sea por una limitación de Rational Rose ó por la implementación en Java.

El análisis y la aplicación por tanto quedan restringidos a la verificación de los siguientes elementos:

- Clases, atributos y métodos (operaciones)
- Generalizaciones
- Asociaciones, cardinalidades máximas y navegabilidad
- Clases asociaciones, cardinalidades y navegabilidad
- Composición, cardinalidades y navegabilidad

### 5.1. *Análisis de la aplicación*

El análisis de la aplicación se ha realizado para cada elemento del diagrama de clases UML a verificar y en función de las características Java que tiene. Para cada uno de estos elementos encontramos lo siguiente:

#### 5.1.1. Clase

- Localización de las clases en un proyecto JAVA

Una aplicación Java normalmente está compuesta de varias clases pertenecientes a un mismo paquete o incluidas en otros paquetes creados anteriormente. Esto hace que los ficheros .java que componen una aplicación, puedan residir en varios directorios. Para identificar la localización de las clases que pertenecen a otros paquetes se utiliza la palabra clave *import*, especificando el nombre del paquete como ruta y nombre de clase o asterisco si deseamos utilizar varias clases. Esto permite localizar físicamente los directorios donde se encuentran las clases a las que pertenecen los objetos creados o referenciados, dentro de la clase que contiene el import.

- Ejemplo:

Supongamos el directorio EJEMPLO con las siguientes clases que pertenecen al paquete con el mismo nombre del directorio:

A.java

B.java

Por otro lado tenemos el directorio EJEMPLO2 que contiene la clase C.java perteneciente al paquete con el mismo nombre del directorio. Supongamos la siguiente estructura de clases:

```
public class A
{
    package EJEMPLO;
    .....
    public static void main(String[] args)
    {
        .....
    }
    .....
}

public class B
{
    package EJEMPLO;
    import EJEMPLO2.C;

    .....
}

public class C
{
    package EJEMPLO2.C; // ó package EJEMPLO2.*;

    .....
}
```

Desde la clase A se pueden crear objetos de la clase B y al revés porque se encuentran en el mismo paquete. Para crear un objeto de la clase C, se necesita importar esa clase haciendo referencia al paquete al que pertenece, como vemos en la clase B.

Para buscar objetos creados en la clase A, usaremos su directorio (EJEMPLO) puesto que al no importar otras clases todas las clases creadas pertenecerán a su mismo paquete.

Para buscar objetos creados en la clase B, usaremos su directorio (EJEMPLO) si se tratan de clases pertenecientes a su mismo paquete o el directorio EJEMPLO2 si se trata de un objeto de la clase C.

Por todo esto, la búsqueda de clases comienza solicitando al usuario que introduzca la ruta de la clase principal del proyecto Java que ha creado o el nombre del paquete (directorio principal). Si seleccionamos una clase específica, el proceso de localización del resto de clases será a través de los objetos creados o referenciados en ésta, cuya localización se encuentra en la cláusula import. Finalmente en estas nuevas clases realizaremos la misma operación, hasta que no encontremos referencias a clases nuevas. Si por el contrario seleccionamos el paquete principal de la aplicación, hará el mismo proceso anterior pero para todas las clases incluidas en el paquete especificado.

- Codificación del nombre de clase:

Cuando en Java tenemos una clase, lo que nos encontramos es un fichero con nombre igual al de la clase que contiene y extensión .java (fuente) o .class (ejecutable). Por lo tanto, es de este nombre de fichero de donde obtenemos el nombre de la clase.

- Codificación de los métodos

Para ello se utiliza la API “Reflection” de Java que permite, partiendo del fichero con extensión .class, conocer los métodos de la clase, su visibilidad (público, protegido o privado) y los parámetros que recibe. Esta forma de obtener los métodos y atributos (como veremos en el siguiente punto), hace que para la ejecución de la aplicación necesitemos el código fuente y el compilado, lo que permite asegurarnos de que el fuente ha sido verificado y no contiene errores.

- Codificación de atributos

En este caso también se utiliza la API “Reflection” de Java, para obtener de una clase los atributos, el tipo del atributo y la visibilidad (público, protegido o privado ) de estos.

- Ejemplo de codificación de clase dentro de la aplicación:

Si tenemos el siguiente código Java:

```
public class Contador {
    public int contadorInterno;
    private Tiempo hora;
    .....
    public void incrementar(int incremento)
    {
        .....
    }
}
```

```
public class Tiempo {
    private int minutos;
    private int segundos;

    .....
}
```

En la aplicación, aparecerá la siguiente codificación para el código Java anterior :

Clases Java

- Contador

Propiedades

contadorInterno Visibilidad: public Tipo: int

hora Visibilidad: private Tipo: class Tiempo

Metodos

incrementar Visibilidad: public

- Tiempo
  - Propiedades
    - minutos Visibilidad: private Tipo:int
    - segundos Visibilidad: private Tipo:int
  - Metodos

### 5.1.2. Generalización

Java permite hacer herencia de clases predefinidas, por ello necesitamos diferenciar éstas de las que realmente nos interesan, que son las clases creadas por el programador. Para ello y para poder decir que existe generalización, se debe comprobar que la superclase encontrada (gracias a la cláusula “extends”) no pertenece a alguno de los siguientes paquetes: "com", "java", "javax", "launcher", "META-INF", "org", "sun", "sunw", que son los que identifican las clases predefinidas por Java.

- Ejemplo de codificación de generalización dentro de la aplicación.

Si tenemos el siguiente código java:

```
public class Carnivoro extends Animal
{
.....
}

public class Herbivoro extends Animal
{
.....
}
```

Encontraremos en la aplicación una ventana con la siguiente información:

SuperclaseJava-SubclaseJava

- Animal-Carnivoro
- Animal-Herbivoro

### 5.1.3. Asociación

- Codificación del nombre

Como ya sabemos, no existe nombre de asociación en Java, así que se ha tomado como nombre de asociación los nombres de las clases asociadas unidos por el carácter “-”.

- Codificación de las cardinalidades

Las cardinalidades son codificadas de dos formas según el tipo de atributo encontrado. Si tenemos un atributo de tipo lista, éste será codificado como “0..n”, si por el contrario tenemos atributo simple, este será codificado como “0..1”.

- Ejemplo de codificación de cardinalidades:

```
public class Profesor {
    private String nombreProfesor;
    private Materia materia[];
    .....
}

public class Materia {
    private String nombreMateria;
    private Profesor profesor;
    .....
}
```

Tendremos una asociación Profesor-Materia, dónde el rol profesor tiene una cardinalidad “0..1” y el rol materia “0..n”.

- Codificación de las cardinalidades incorrectas

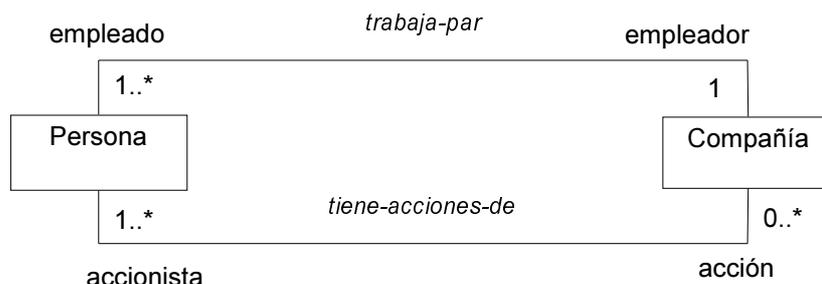
Rational Rose permite introducir cardinalidades nulas en las asociaciones. Esto hace que si nos encontramos una cardinalidad “null” en el fichero de Rational Rose, se tomará como cardinalidad no especificada, con lo que no se tomará en cuenta en la comparación con la cardinalidad de la implementación Java.

Cabe destacar que aunque Rational Rose también permite representar la cardinalidad 0, incluso como opción por defecto, tampoco se tomará en cuenta puesto que se trata de una multiplicidad inválida como especifica el estándar UML (versión 1.3, p. 2-79):

*“Multiplicidad: En el metamodelo una multiplicidad define un conjunto no vacío de enteros no negativos. Un conjunto que solo contiene cero ( $\{0\}$ ) no es considerado una multiplicidad válida. A cada multiplicidad le ha de corresponder al menos una cadena que la representa. Rango de Multiplicidad: En el metamodelo un rango de multiplicidad se define como un rango de enteros. El valor mayor del rango no puede estar por debajo del valor menor. El valor menor debe ser un entero no negativo. El valor mayor debe ser un entero no negativo o el valor especial "ilimitado", el cuál indica que no hay un valor mayor en el rango.”*

- Codificación de varias asociaciones entre las dos mismas clases

Ya se ha comentado el problema de las multiasociaciones entre dos clases. Como no podemos distinguir los roles Java representados en una asociación UML, lo que se hace es ir formando asociaciones según el orden en que se encuentren los atributos dentro de la clase Java. Es decir, si encontramos las siguientes asociaciones UML:



Y el siguiente código Java:

```

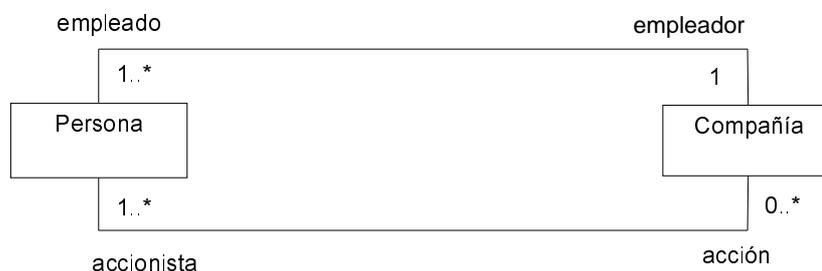
public class Persona {
    private Compañía empleador;
    private Compañía accionista;
    .....
}

public class Compañía {
    private Persona empleado;
    private Persona accionista;
    .....
}
  
```

La aplicación comenzaría estudiando la clase Persona, que tiene un primer atributo de tipo Compañía con rol empleador. Posteriormente se estudiaría la clase Compañía donde encontraría un primer atributo de tipo Persona con rol empleado. En este momento se formaría una asociación Persona-Compañía con roles empleador-empleado.

Se continuaría con el estudio de la clase Persona donde se encuentra otro atributo de tipo Compañía con rol acción. Volveríamos a la clase Compañía donde existe un segundo atributo de tipo Persona con rol accionista. Ahora se formaría otra asociación Persona-Compañía con roles acción-accionista.

En este caso, obtendríamos del código java, dos asociaciones que se corresponden con el diagrama UML inicial.



Pero si el código Java se hubiese definido de este modo:

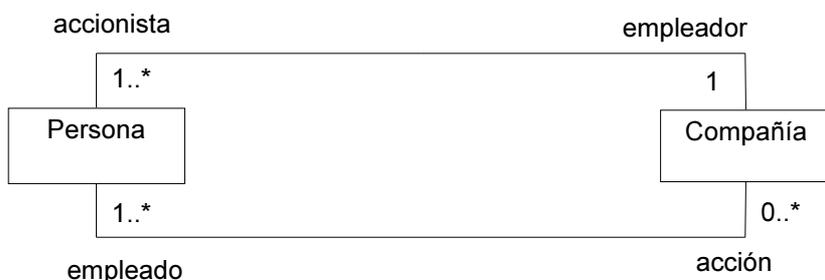
```

public class Persona {
    private Compañía empleador;
    private Compañía acción;
    .....
}

public class Compañía {
    private Persona accionista;
    private Persona empleado;
    .....
}

```

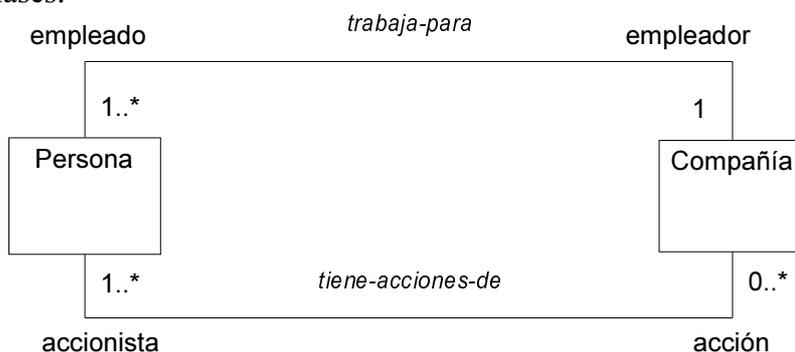
La aplicación, al igual que en el ejemplo anterior, comenzaría con la clase Persona donde encontraría un atributo de tipo Compañía con rol empleador. Al buscar en la clase Compañía un atributo de tipo Persona, el primero que localizaría sería Persona con rol accionista, con lo que la asociación formada sería Compañía-Persona con roles empleador-accionista. Del mismo modo localizaría la segunda asociación, formando erróneamente dos asociaciones que no se correspondería con el diagrama UML inicial y que sería el siguiente:



Por tanto, hay que tener en cuenta que cuando se ejecute la aplicación será bastante importante el orden de definición de los atributos que representan el rol de una asociación dentro de cada clase.

- Ejemplo completo de codificación de asociación dentro de la aplicación

Diagrama de clases:



Codificación Java:

```

public class Persona {
    private Compañía empleador;
    private Compañía acción[];
    .....
}

public class Compañía {
    private Persona empleado[];
    private Persona accionista[];
    .....
}

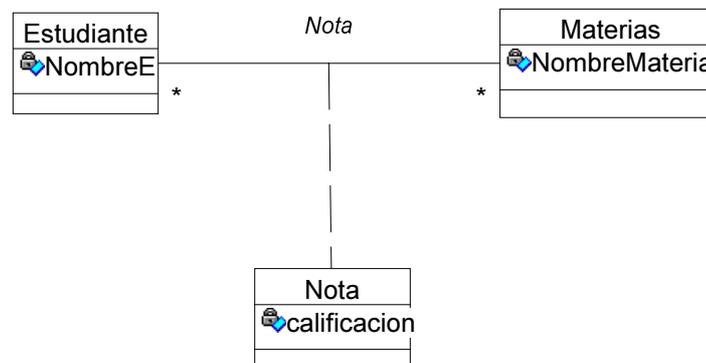
```

Las codificación de asociaciones dentro de la herramienta sería:

- Persona-Compañía
  - Clase1: Persona Rol: empleado Visibilidad: private Card: 0..n Naveg: true
  - Clase2: Compañía Rol: empleador Visibilidad: private Card: 0..1 Naveg: true
- Persona-Compañía
  - Clase1: Persona Rol: accionista Visibilidad: private Card: 0..n Naveg: true
  - Clase2: Compañía Rol: acción Visibilidad: private Card: 0..n Naveg: true

#### 5.1.4. Clase-Asociación

Los elementos del tipo clase asociación, como ya se ha indicado, se codifican como asociaciones dos a dos, como vemos en el siguiente ejemplo (obsérvese que el nombre de la asociación Nota está con la inicial mayúscula, contra lo que es habitual, porque debe ser el mismo que el nombre de la clase, ya que la asociación y la clase son una y la misma cosa y por tanto deben tener el mismo nombre):



Código Java:

```

public class Estudiante {
    private String nombreE;
    private Nota Nota[];
    .....
}

```

```

}

public class Materia {
    private string nombreM;
    private Nota Nota[];
    .....
}

public class Nota {
    private int calificacion;
    private Materia materia;
    private Estudiante alumno;
    .....
}

```

Codificación dentro de la aplicación:

- Estudiante-Nota
  - Clase1: Estudiante Rol: alumno Visibilidad: private Card: 0..1 Naveg: true
  - Clase2: Nota Rol: Nota Visibilidad: private Card: 0..n Naveg: true
- Materia-Nota
  - Clase1: Materia Rol: materia Visibilidad: private Card: 0..1 Naveg: true
  - Clase2: Nota Rol: Nota Visibilidad: private Card: 0..n Naveg: true

### 5.1.5. Composición

En este caso la codificación es equivalente a la utilizada en la asociación, puesto que como ya se ha explicado en apartados anteriores la diferencia es semántica. Veamos el ejemplo:



La implementación en Java de este tipo de relaciones es la siguiente:

```

public class Coche {
    private Pieza parte[];
    .....
}

public class Pieza {
    private Coche contenedor;
    .....
}

```

El resultado es el siguiente:

- Coche-Pieza
  - Clase1: Coche Rol: contenedor Visibilidad: private Card: 0..1 Naveg: true
  - Clase2: Pieza Rol: parte Visibilidad: private Card: 0..n Naveg: true

## 5.2. Diseño de la aplicación

La aplicación comienza con una pantalla inicial donde se introduce la información de directorios y ficheros, necesaria para localizar los fuentes y los ejecutables Java, la clase o nombre del paquete Java a verificar y el fichero de Rational Rose donde encontramos el diagrama de clases. Si la información es correcta, aparecen unos botones para verificar los distintos elementos del diagrama de clases:

- Clases (Atributos, Operaciones)
- Asociaciones (cardinalidad, navegabilidad)
- Clase Asociación (cardinalidad, navegabilidad)
- Composición (cardinalidad, navegabilidad)
- Generalizaciones

Dentro de cada elemento a verificar, aparecen unos botones para poder verificar los mismos elementos y otros para verificar las características de cada uno de ellos como son atributos, cardinalidades, etc... En todas las verificaciones aparecen unos iconos que simbolizan el resultado de la misma, de forma que el icono  identifica elemento correcto, el icono  elemento incorrecto y el icono  indeterminación.

Hay que tener en cuenta que el lenguaje de programación Java es sensible a mayúsculas y minúsculas, con lo que un elemento a estudiar será diferente si recibe el mismo nombre en un lado y en otro pero contiene alguna letra diferente en mayúsculas o minúsculas.

### 5.2.1. Clases

Una vez seleccionada la opción correspondiente a las clases, aparece en el lado izquierdo la codificación de las clases encontradas en el diagrama de clases UML y en la parte derecha la codificación correspondiente a las clases encontradas en la implementación Java.

Dentro de esta pantalla nos encontramos con varias opciones:

- Verificar las clases. Una vez seleccionemos esta opción aparecerán los distintos iconos según la verificación, donde:
  - las clases con tick verde representan las clases correctas en la verificación, es decir, aquellas cuyo nombre coincide en el diagrama UML y en la implementación Java
  - las clases con aspa roja representan aquellas de las que no se ha encontrado correspondencia en el lado contrario, es decir, clases UML no implementadas en Java y clases Java no representadas en UML (nombres de clases no coincidentes)
- Verificar atributos/propiedades. Esta opción se habilitará cuando la clase seleccionada tenga su correspondencia en el lado contrario, es decir, cuando podamos verificar la correspondencia entre atributos en UML y propiedades en Java. De este modo nos aparecerá:

- tick verde para los atributos que coinciden en nombre, visibilidad y tipo de atributo. También se considerarán atributos válidos a aquellos que coincidan en nombre y Visibilidad, cuando el tipo no se haya definido en el diagrama de clases UML
  - interrogación naranja para los atributos que coinciden únicamente en el nombre, sin tener el mismo tipo de Visibilidad ó tipo de atributo
  - aspa roja para aquellos atributos que no coinciden en el nombre.
- Verificar métodos/operaciones. Esta opción, al igual que en el caso anterior, se habilitará cuando la clase seleccionada tenga su correspondencia en el lado contrario, es decir, cuando podamos verificar la correspondencia entre operaciones UML y métodos Java. De este modo nos aparecerán:
    - con un tick verde, los atributos que coinciden en nombre y visibilidad de la operación ó método. Para decidir esto, no se han tenido en cuenta otros aspectos como son el tipo de parámetros que recibe el método o el tipo de resultado que devuelve. Esto es debido a que en Rational Rose este valor es libre y no se adapta a la posterior codificación Java, con lo que su comparativa en la aplicación realizada con el código Java generado podría dar lugar a error
    - con interrogación naranja los métodos que coinciden únicamente en el nombre
    - y con aspa roja aquellos métodos que no coinciden en el nombre.

Cuando seleccionamos una clase, aparecerá sombreada su correspondiente al otro lado. Si no es así, es porque esa clase no tiene correspondencia, con lo cuál aparecerá con aspa roja si realizamos la verificación de clases.

### 5.2.2. Generalización

Al seleccionar la opción de generalizaciones volvemos a tener, por un lado las generalizaciones encontradas en el diagrama de clases y por el otro las encontradas en Java. En este caso nos encontramos la siguiente opción:

- Verificar generalizaciones. Una vez seleccionada esta opción nos aparecerán distintos iconos, donde:
  - el tick verde identifica aquellas generalizaciones cuya superclase y subclase coinciden en el diagrama UML y en la implementación Java
  - el aspa roja representa aquellas de las que no se ha encontrado correspondencia en el lado contrario, es decir, generalizaciones UML no implementadas en Java y generalizaciones Java no representadas en UML (nombres de superclase y subclase no coincidentes)

Al igual que para las clases, cuando seleccionamos un elemento en un lado de la pantalla, nos aparecerá su correspondiente generalización al otro lado.

### 5.2.3. Asociaciones

En este caso vuelven a aparecer por un lado las asociaciones encontradas en el diagrama de clases y por el otro las encontradas en Java. Las posibles opciones son:

- Verificar asociaciones. Tendremos:
  - asociaciones con tick verde que representan las asociaciones correctas, es decir, aquellas coincidentes en el nombre de clases que forman parte de la asociación y en el rol representado para esas dos clases. También aparecerán con este icono aquellas que coincidan en el nombre de clases que asocia y no se haya definido un rol en el diagrama UML (\$UNNAMED), tomando como nombre de rol en java el nombre de la asociación. Además se incluirán las asociaciones en las que no se haya definido el nombre del rol ni el de la asociación, pero sólo exista una asociación de este tipo en el diagrama UML y en el código Java
  - el aspa roja será para aquellas donde no coinciden los nombres de clases
  - con interrogación naranja aparecerán el resto de asociaciones no descritas en los casos anteriores, como son las asociaciones que coincidan en los nombres de clases que asocian pero no en los roles.

Como en los casos anteriores, si seleccionamos un elemento a un lado de la pantalla se seleccionará su homólogo al otro lado, pero aquí encontramos alguna diferencia:

- Si las asociaciones no han sido verificadas, la asociación sólo aparecerá seleccionada al otro lado cuando sea totalmente correcta, es decir, si realizásemos la verificación de asociaciones ésta aparecería con tick verde
  - Si se ha realizado la verificación, entonces al otro lado aparecerán, además de las totalmente correctas, aquellas asociaciones que tengan relación con las indeterminadas, de forma que es posible que aparezca una única selección o varias.
- Verificar cardinalidad/navegabilidad. Esta opción se habilitará cuando la asociación seleccionada tenga su correspondencia en el lado contrario, es decir, cuando aparecen una asociación sombreada en cada uno de los lados. En este caso aparece una descripción de la verificación realizada sobre las cardinalidades UML y Java y sobre la navegabilidad.
    - Cardinalidades: la verificación puede dar los siguientes resultados:

Cardinalidad UML	Cardinalidad Java	Resultado
0..1	0..1	OK
0..1	0..n	Error
1	0..1	OK
1	0..n	Error
0..n	0..1	Error
0..n	0..n	OK
1..n	0..1	Error
1..n	0..n	OK
0..3	0..1	Error
0..3	0..n	OK
2..n	0..1	Error

2..n	0..n	Error
2..4	0..1	Error
2..4	0..n	Mensaje informativo indicando que la cardinalidad UML tiene definida un máximo y en java es ilimitado

Normalmente, además del resultado aparece un mensaje de aclaración, como por ejemplo en el último caso que aunque no se considera erróneo presenta una breve aclaración indicando que existe una cardinalidad limitada a 3 en UML y en Java es ilimitada.

- Navegabilidad. Aparecerá un mensaje con el resultado de la verificación de la navegabilidad, de forma que si en UML una de las clases de la asociación tiene navegabilidad (Naveg. en la aplicación) true, pero en la parte Java esta misma clase aparece como false, aparecerá un mensaje de error y lo mismo en caso contrario.

#### 5.2.4. Clases-Asociaciones

Este caso es bastante parecido al de las asociaciones. Por un lado vemos los elementos de tipo clase-asociación encontrados en el diagrama de clases y en el otro todas las asociaciones que se han encontrado en el código Java. Esto es debido a que, como ya se ha explicado anteriormente, no podemos detectar los elementos de tipo clase-asociación en Java, por lo tanto lo que se hace es intentar localizar las encontradas en UML entre todas las asociaciones que existen en Java. Volvemos a encontrarnos las mismas opciones que para la asociación:

- Verificar clases asociaciones. Tenemos los distintos iconos:
  - tick verde para las clases asociaciones UML y las asociaciones Java que se han detectado como asociaciones correspondientes a una clase-asociación UML. Estas son aquellas que se corresponden con el nombre de rol y con la cardinalidad. También aparecerán como válidas aquellos elementos de la clase-asociación que no corresponden en el nombre de rol, porque éste aparece indefinido (\$UNNAMED) en UML y sólo existen dos asociaciones Java para esa clase asociación
  - el aspa roja será para aquellas donde no coincidan los nombres de las asociaciones con los nombre de clases que pertenecen a las clases asociaciones UML
  - con interrogación naranja aparecerán el resto de casos, como son el encontrar mas de dos asociaciones Java para una clase-asociación UML, ó la no correspondencia de las cardinalidades ó la no correspondencia de los roles cuando se han definido en UML (valor distinto a \$UNNAMED).

Esta verificación se realiza según lo explicado en el apartado “Clase-Asociación”, teniendo en cuenta que se ha seleccionado la opción de utilizar una clase intermedia navegable como forma de representación más correcta.

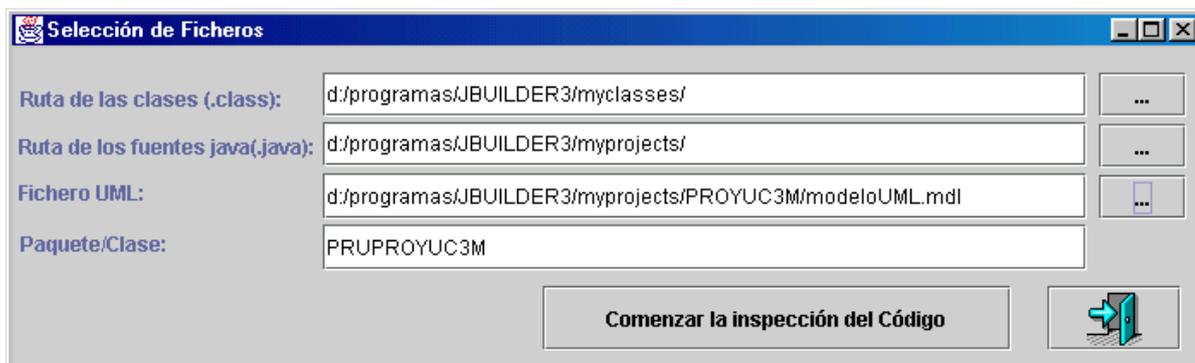
De nuevo, si seleccionamos un elemento a un lado de la pantalla se seleccionará su homólogo al otro lado. Hay que tener en cuenta que si se selecciona una clase asociación indeterminada es posible que aparezcan varias selecciones.

### 5.2.5. Composición

Esta opción es equivalente a la asociación. Tenemos en la parte de UML las composiciones encontradas y en la parte Java todas las asociaciones Java, ya que al igual que los elementos de tipo clase-asociación no tenemos forma de identificar éstas en Java y lo que se hace es intentar localizar cada composición entre las distintas asociaciones Java. Tenemos las mismas opciones que para las asociaciones, con el mismo significado.

## 6. MANUAL DE USUARIO.

La aplicación comienza solicitando al usuario los ficheros y directorios necesarios para ejecutar la aplicación. Para ello aparecerá la siguiente pantalla:

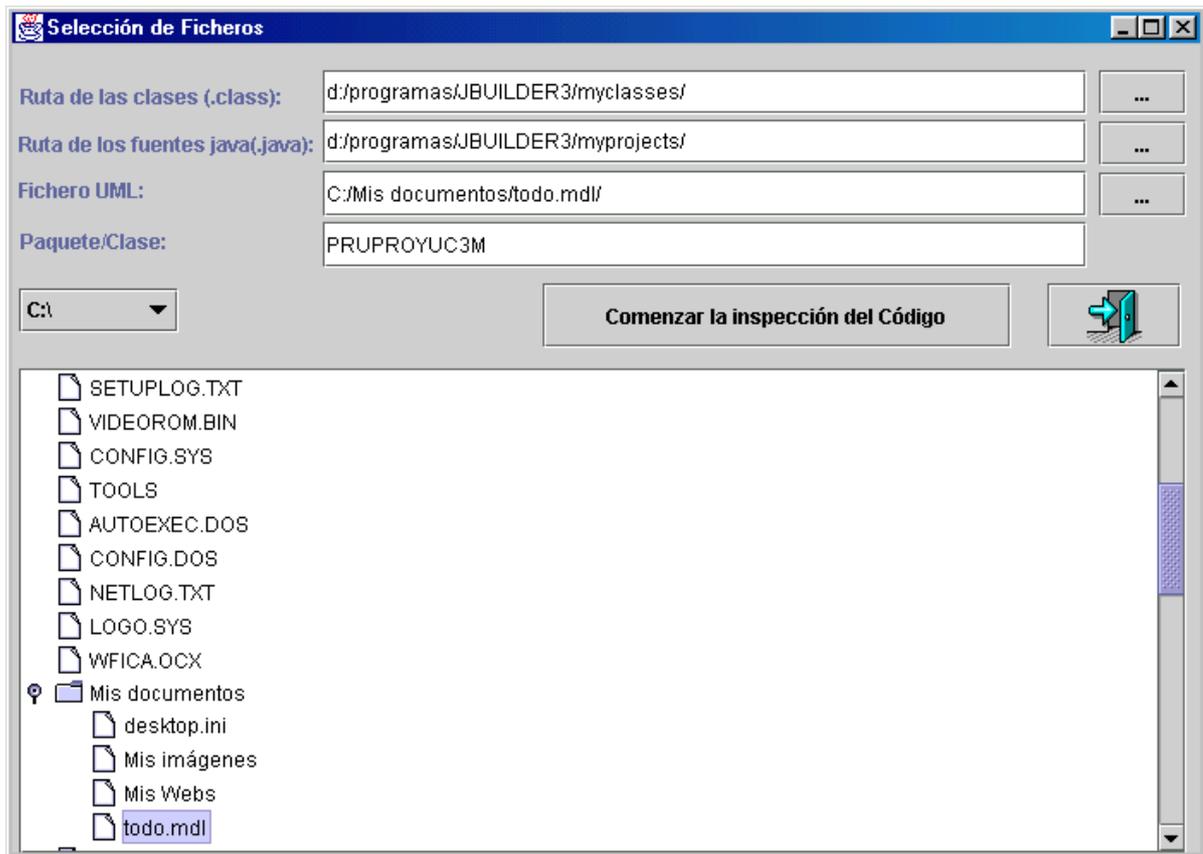


(Figura 6.1)

Los directorios y ficheros solicitados son:

- Ruta de las clases Java (.class): Directorio principal donde se encuentran los ejecutables Java
- Ruta de los fuentes Java (.java): Directorio principal donde se encuentran los fuentes del programa que vamos a analizar
- Fichero UML Rational Rose (.mdl): Fichero creado por Rational Rose que contiene el diagrama de clases UML a verificar
- Paquete/Clase principal: Podemos poner el paquete del programa Java a analizar o podemos especificar la clase principal del mismo (Ejemplo: PRUPROYUC3M.ClasePrincipal)

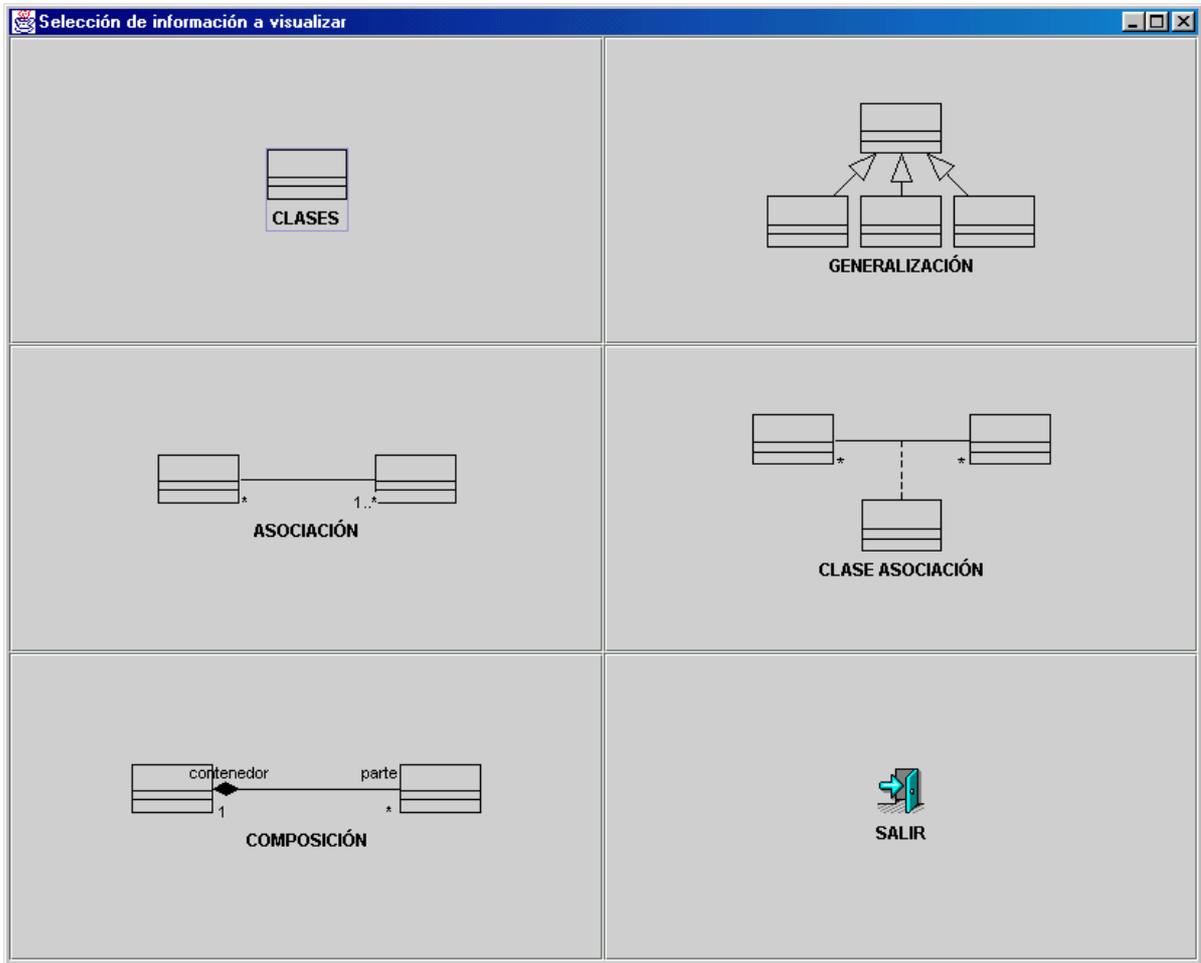
Podemos modificar las rutas directamente en la caja de texto, o pulsando el botón “...” que se encuentra a la derecha del campo de texto que queremos modificar. Una vez pulsemos este botón, aparecerán nuevas opciones en pantalla que permitirán seleccionar directorios o ficheros:



(Figura 6.2)

Una vez tengamos las rutas necesarias comenzará la verificación de los elementos pulsando sobre el botón . Si por el contrario deseamos salir de la aplicación pulsaremos el botón .

Al seleccionar “Comenzar la inspección del código”, aparecerá la siguiente pantalla con los posibles elementos a verificar:

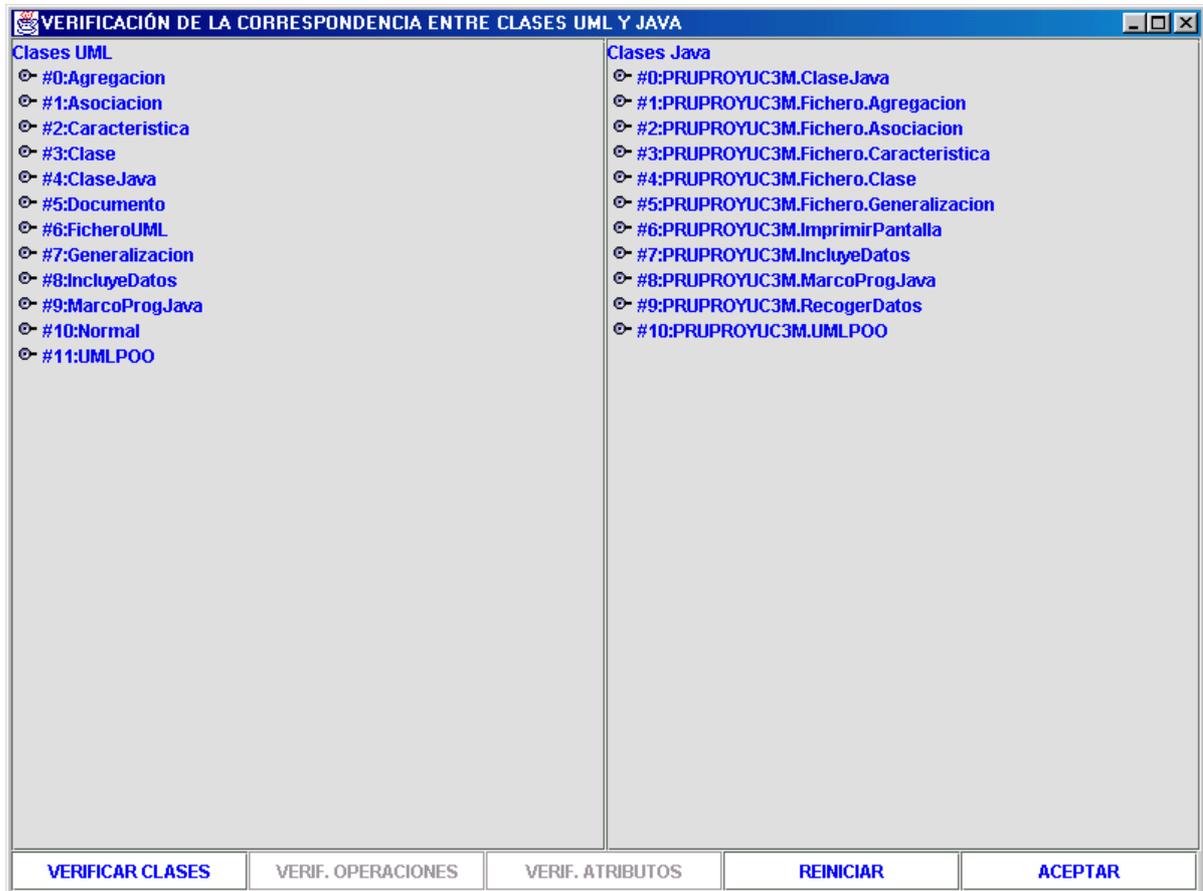


(Figura 6.3)

Pulsando sobre los distintos elementos se presentarán las siguientes pantallas:

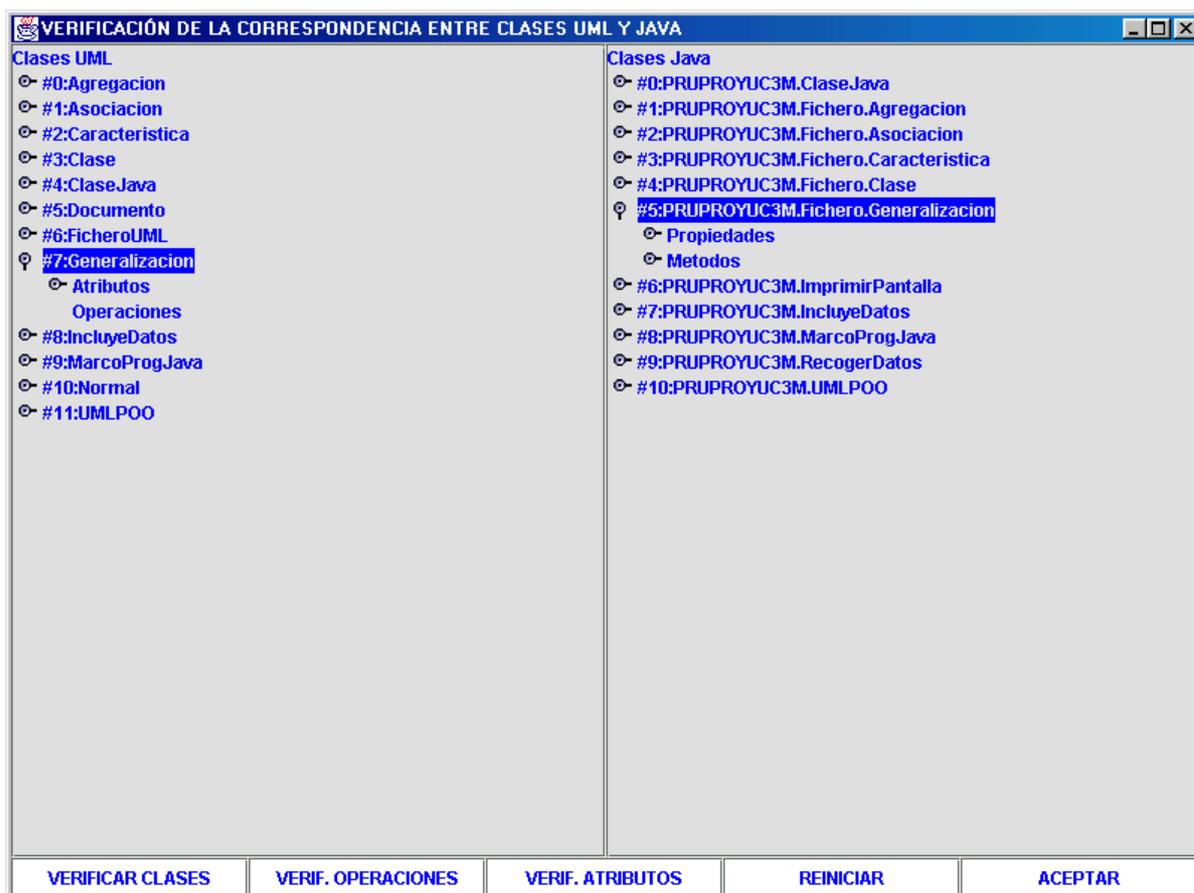
## 6.1. Verificación de Clases

Aparecerá una pantalla como la siguiente:



(Figura 6.4)

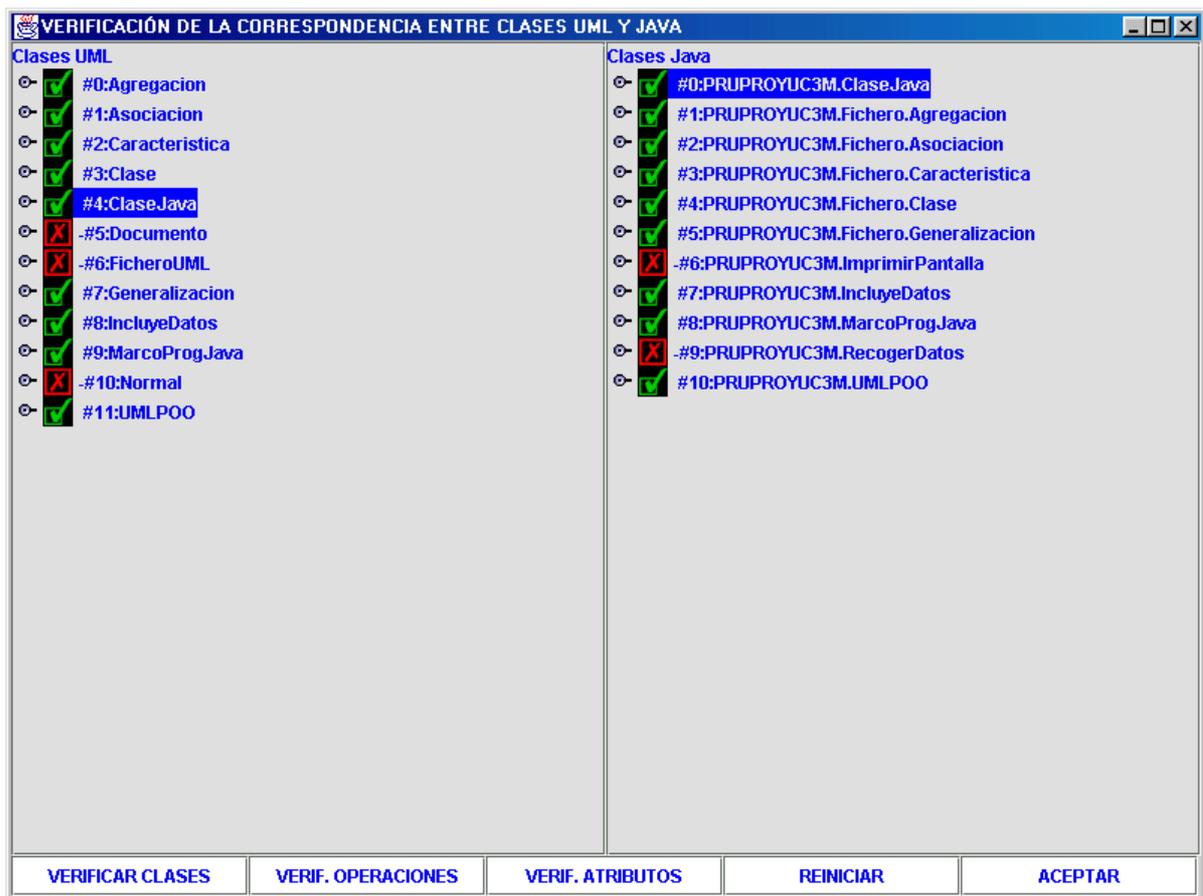
Por un lado tenemos las clases UML encontradas en el diagrama de clases creado en Rational Rose y por otro las encontradas en la implementación Java. Al seleccionar cualquier clase, ésta aparecerá sombreada al igual que su correspondiente al otro lado:



(Figura 6.5)

Si pulsamos sobre una clase, ésta debe aparecer seleccionada también en el lado contrario, si no es así es porque no se ha encontrado su homóloga con lo que aparecerá con un aspa roja al realizar la verificación seleccionando “VERIFICAR CLASES”. Como se puede observar, el botón de “VERIFICAR OPERACIONES” y el de “VERIFICAR ATRIBUTOS” se habilita cuando pulsemos en un elemento válido (clase encontrada en ambos lados: diagrama UML e implementación Java) como es este caso.

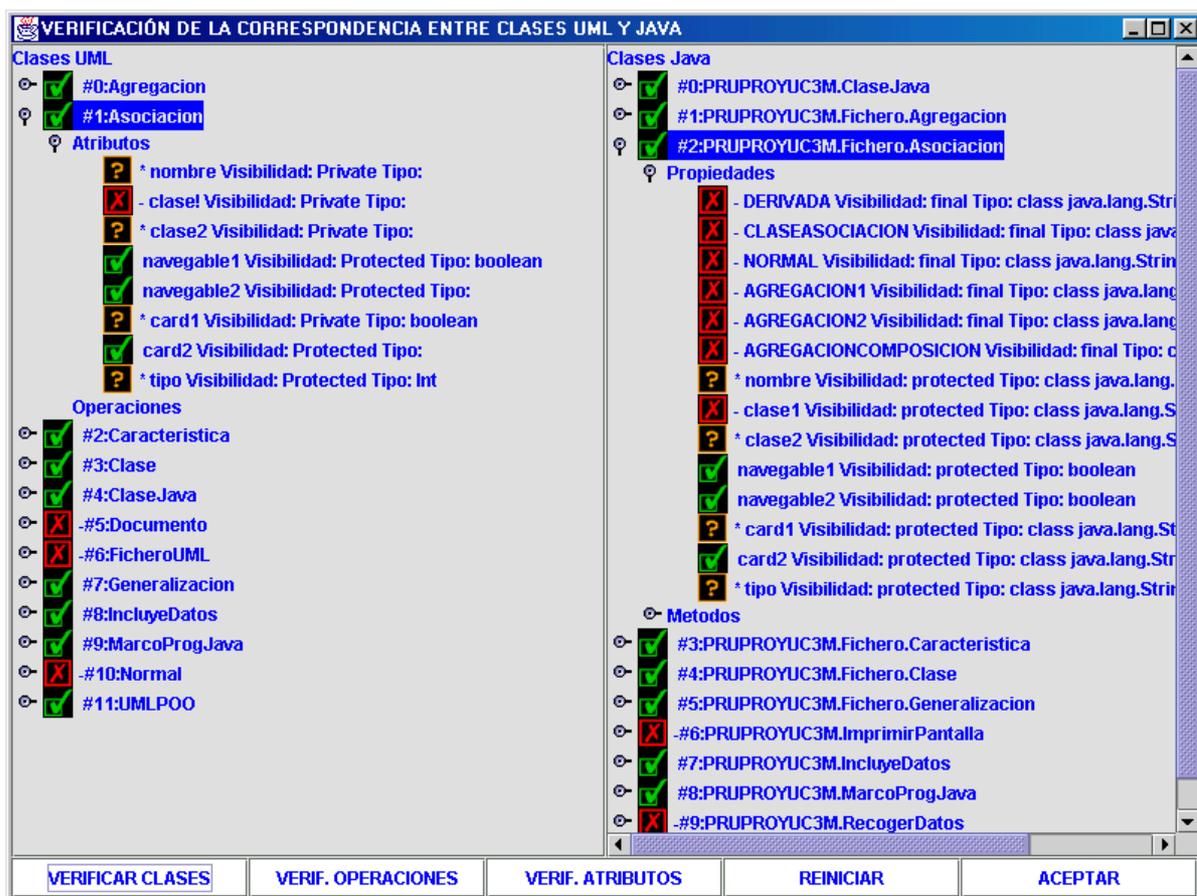
Una de las opciones que presenta esta pantalla es la de “VERIFICAR CLASES”, donde al seleccionarla aparecerá el resultado de la verificación de este modo:



(Figura 6.6)

Aparecerán entonces los distintos iconos que indican el resultado de la verificación de clases, dónde el aspa en rojo significa fallo en la verificación y el tick verde significa verificación conseguida. Para más detalle consultar el documento de diseño de la herramienta.

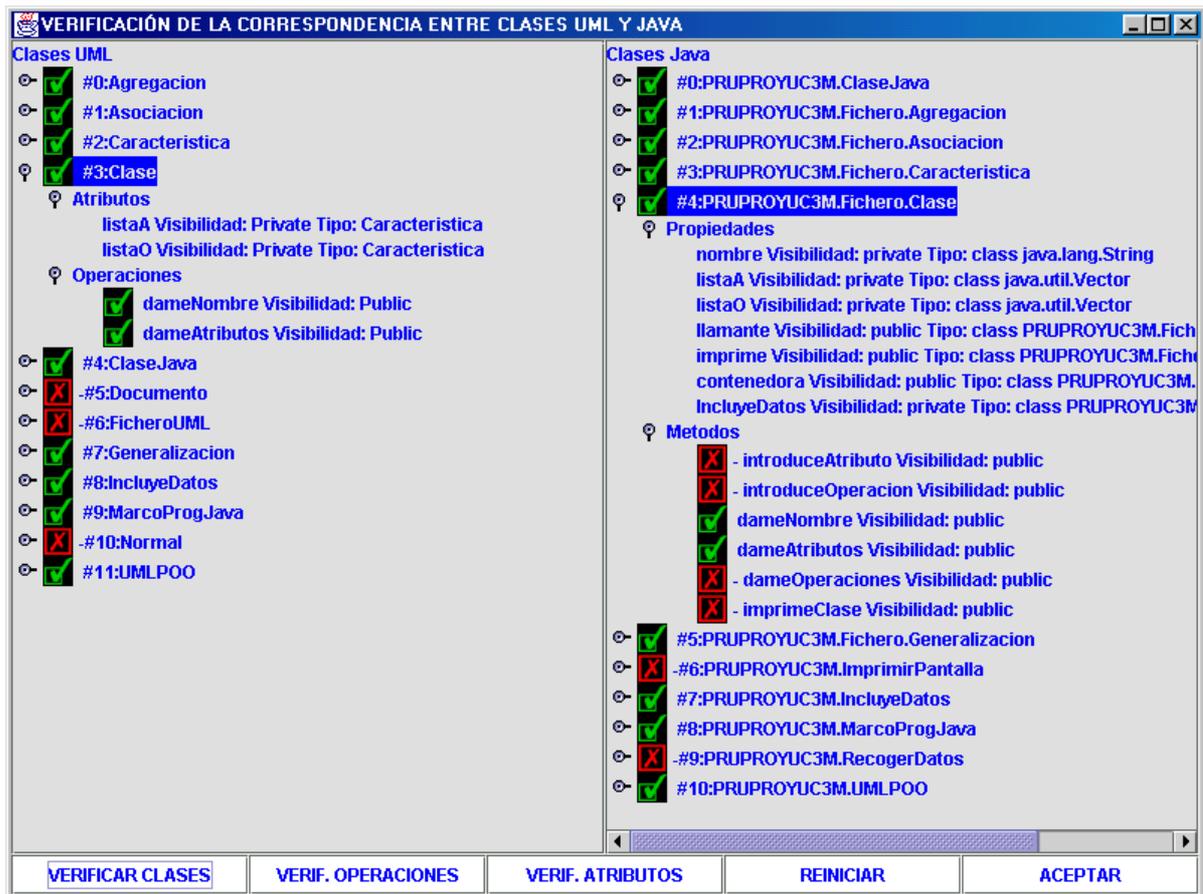
Si una vez seleccionada una clase pulsamos en “VERIFICAR OPERACIONES”, se realizará una comprobación entre las operaciones UML y los métodos Java de esa clase:



(Figura 6.7)

Aquí volvemos a encontrarnos los iconos que indican el resultado de la operación, pero en este caso también aparece el icono con interrogación naranja que indica verificación parcial. Para más detalle consultar el documento de diseño de la herramienta.

Lo mismo ocurre en la verificación de atributos o propiedades:



(Figura 6.8)

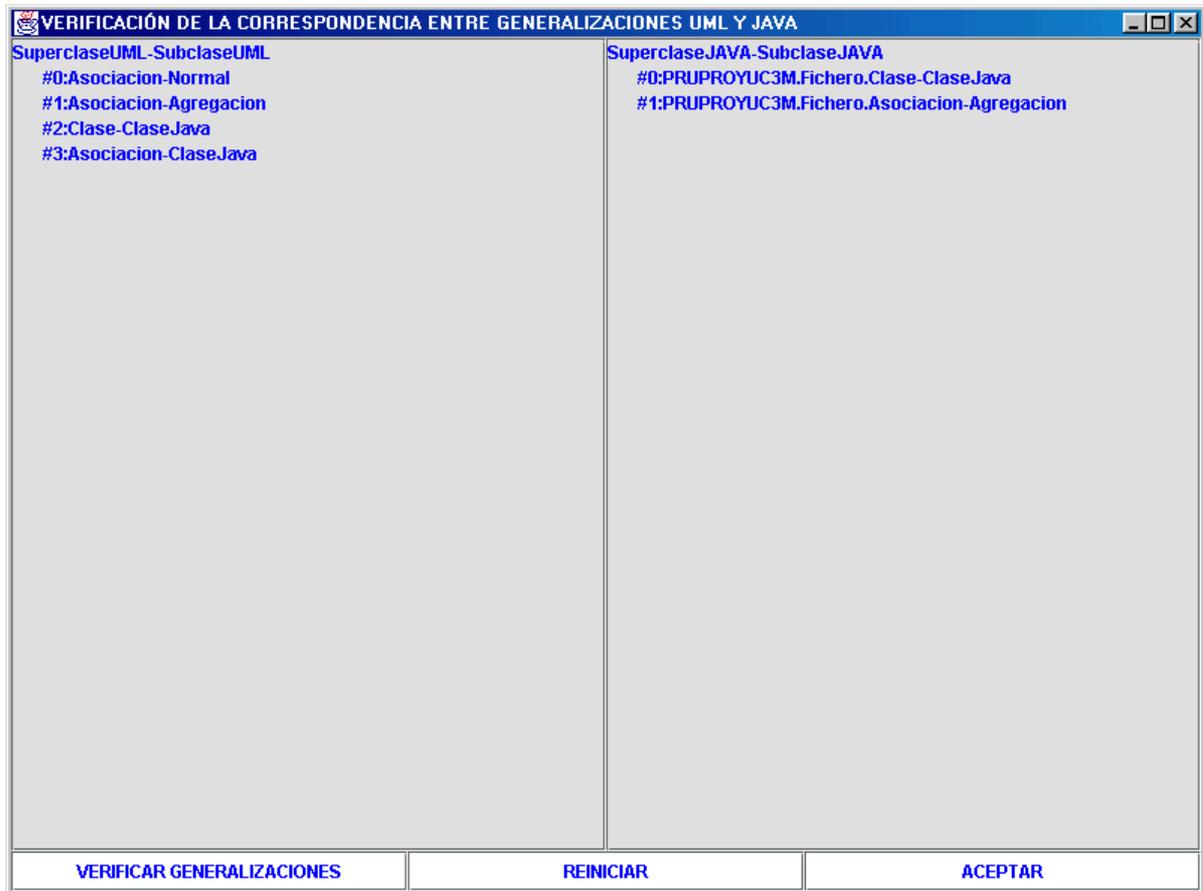
Como vemos, vuelven a aparecer los distintos iconos que identifican el resultado de la verificación de cada atributo.

En ésta pantalla también se puede pulsar el botón de “REINICIAR” que limpia la pantalla y la deja en el estado inicial (figura 6.4).

Por último, al pulsar sobre el botón “ACEPTAR” se cerrará esta pantalla y volveremos a la principal (figura 6.3).

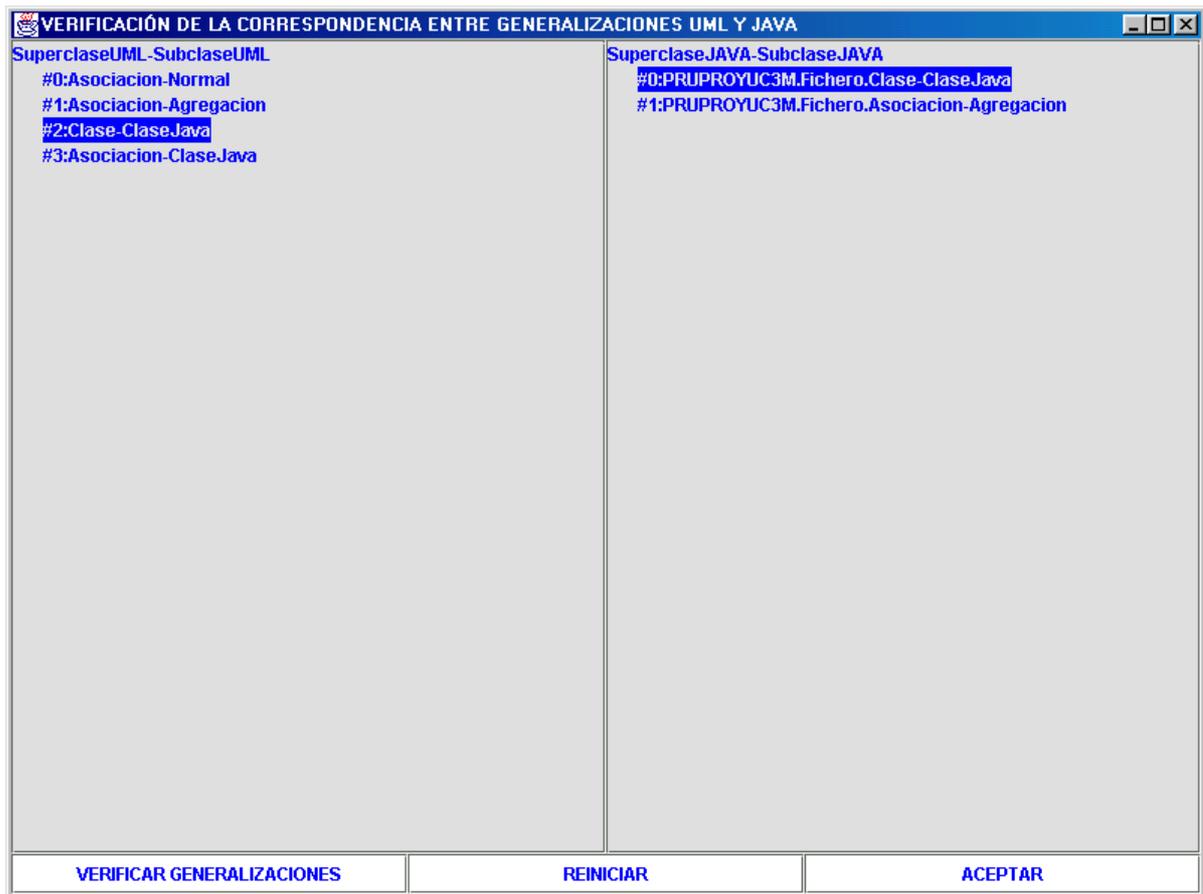
## 6.2. Verificación de Generalizaciones

Si en la pantalla principal (figura 6.3) seleccionamos la opción de “GENERALIZACIÓN”, tendremos lo siguiente:



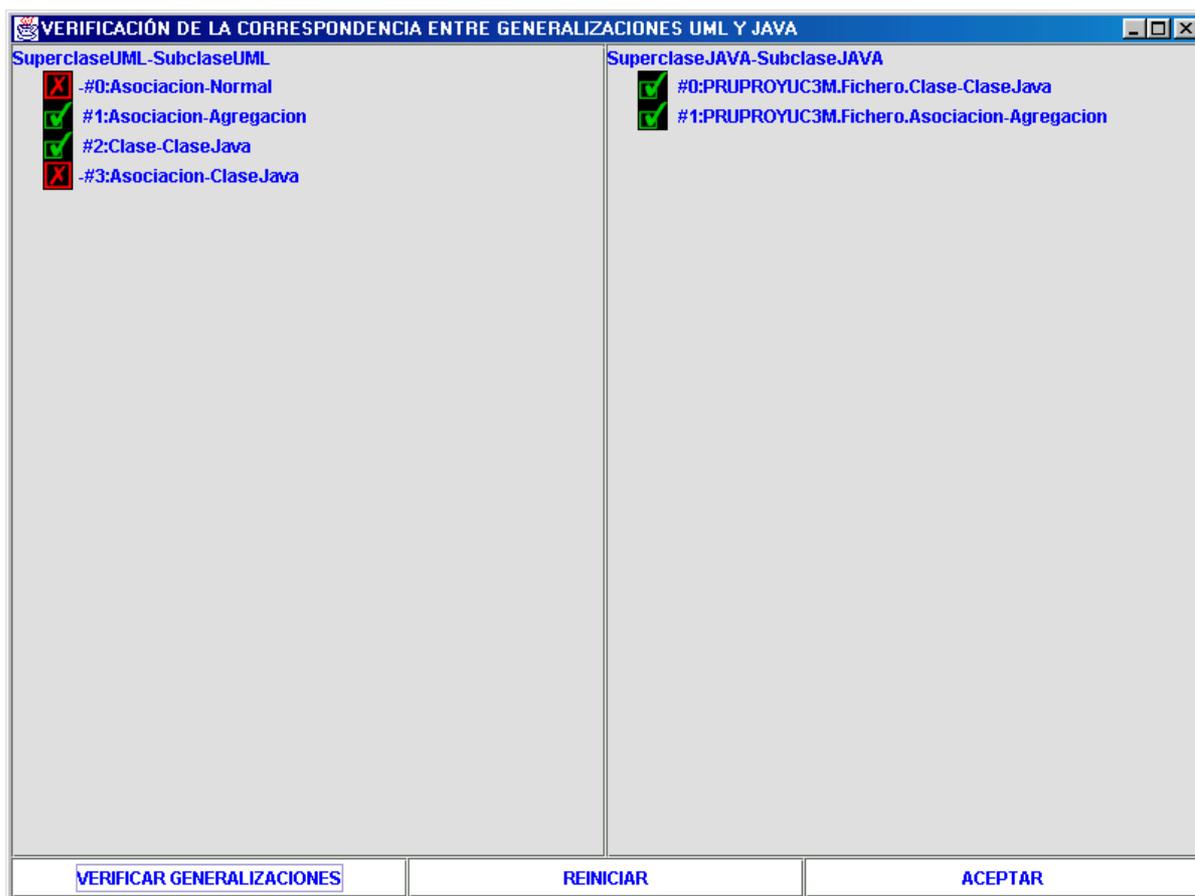
(Figura 6.9)

Encontramos por un lado las generalizaciones localizadas en el diagrama de clases y a la derecha las encontradas en la implementación Java. Al igual que en las clases, al pulsar sobre cualquier generalización aparecerá sombreada ésta y su correspondiente en el lado contrario:



(Figura 6.10)

Dentro de esta pantalla podemos pulsar el botón “VERIFICAR GENERALIZACIONES” y obtendremos el resultado de la verificación:



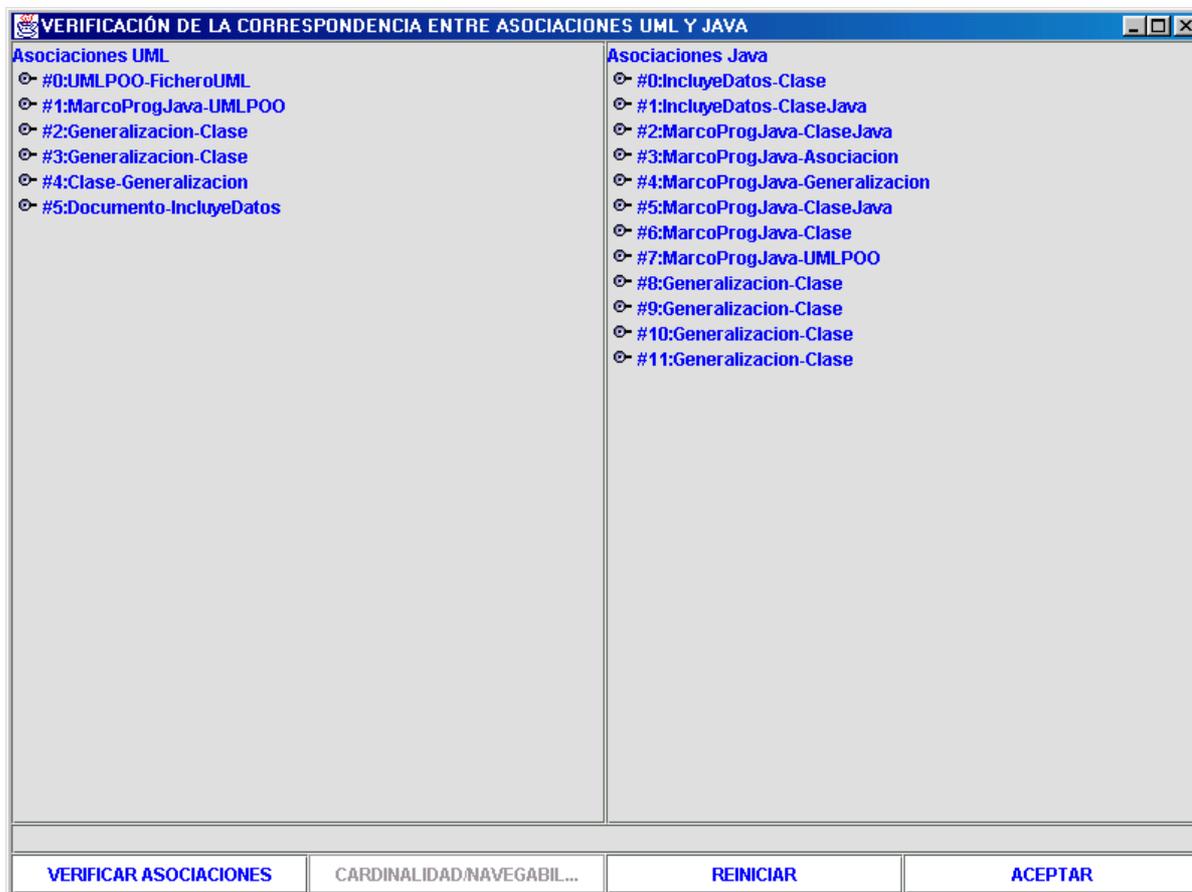
(Figura 6.11)

En esta pantalla no es posible expandir ningún árbol, puesto que la única información que necesitamos sobre la generalización es el nombre de la superclase y el de la subclase.

Si queremos que desaparezca el resultado de la verificación sólo hay que pulsar en “REINICIAR” y volveremos a la pantalla inicial de generalización (figura 6.9). Si lo que queremos es salir de esta pantalla y volver a la pantalla principal (figura 6.3) debemos pulsar sobre el botón “ACEPTAR”.

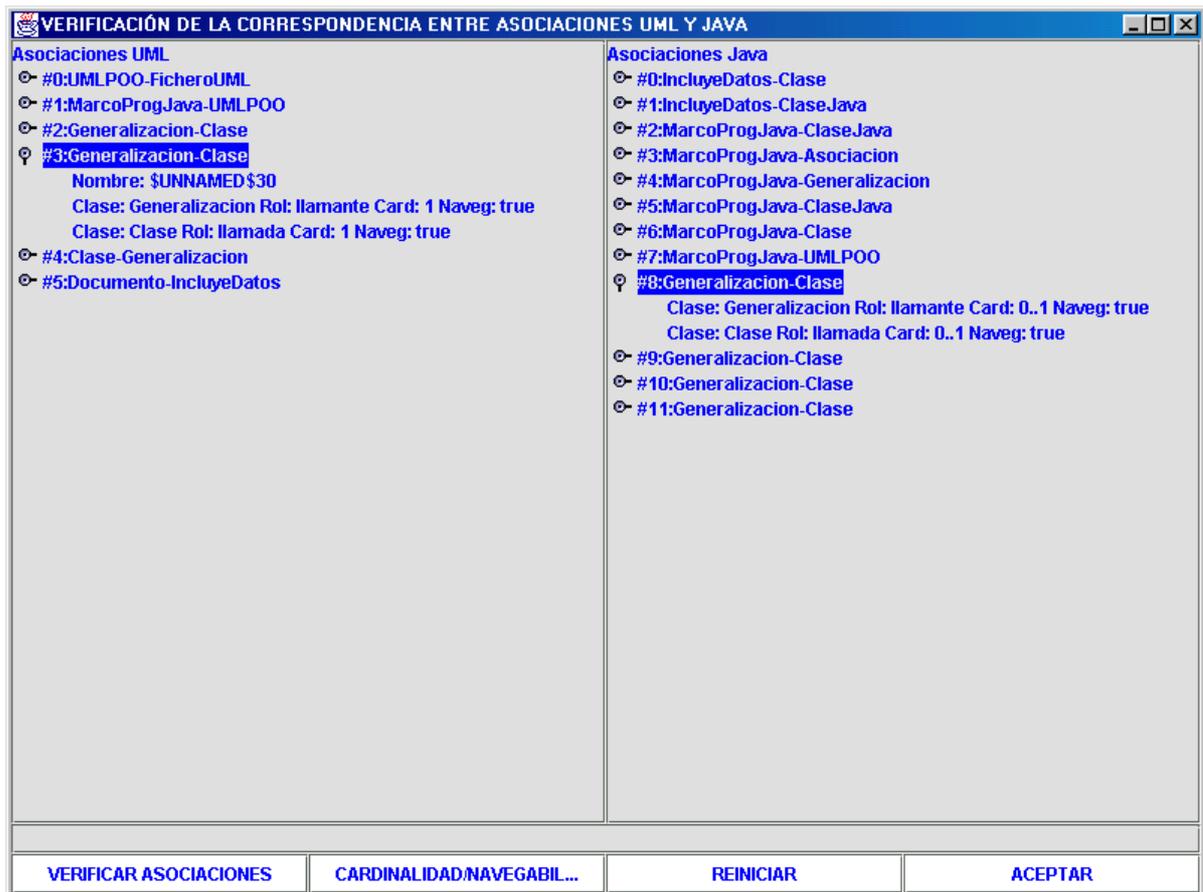
### 6.3. Verificación de Asociaciones

Al seleccionar la opción de “ASOCIACIÓN” en la pantalla principal (figura 6.3), tendremos lo siguiente:



(Figura 6.12)

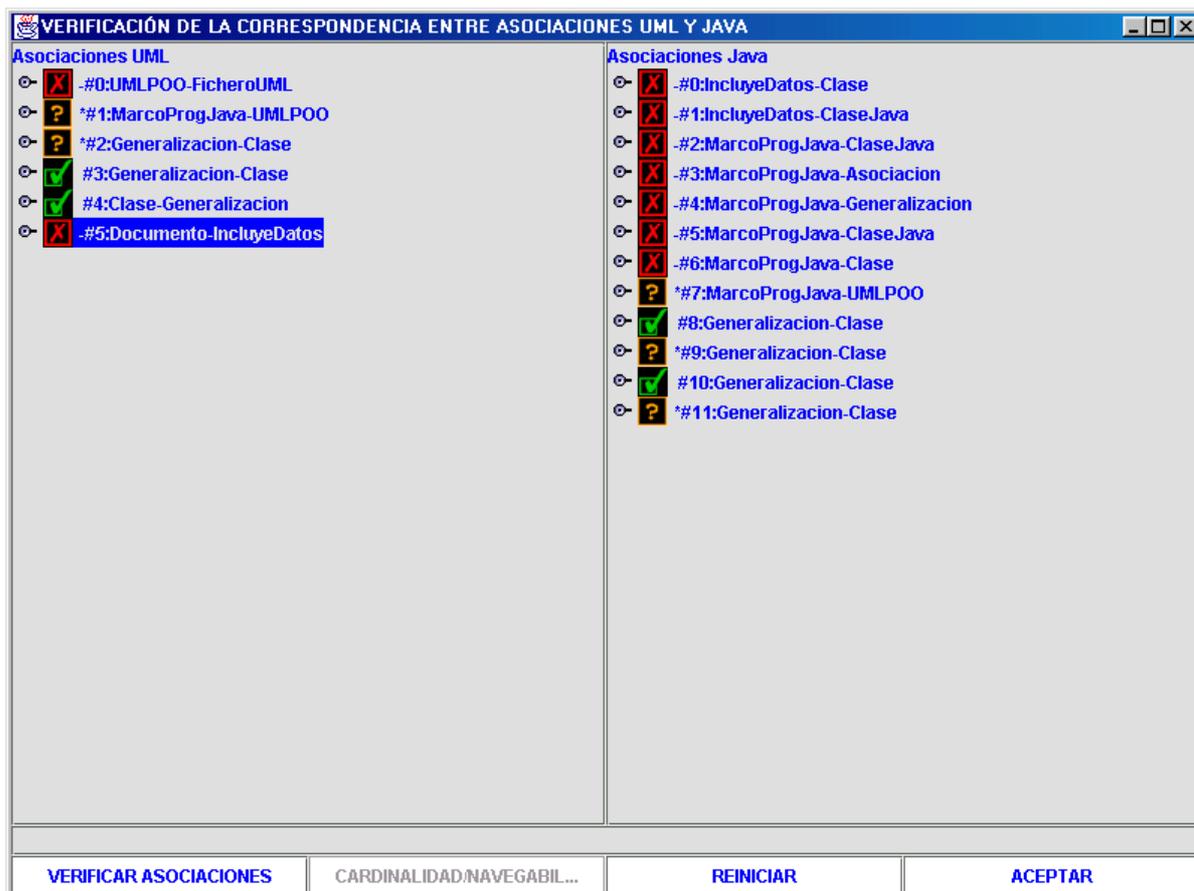
Volvemos a encontrarnos por un lado las asociaciones UML y por otro las encontradas en la implementación Java. Al igual que en las clases, al pulsar sobre cualquier asociación aparece sombreada ésta y su correspondiente al otro lado, pero en este caso también podemos encontrarnos con asociaciones indeterminadas (representadas en la verificación con el icono con símbolo “?” en naranja). Este tipo de asociaciones no tendrá su correspondencia al otro lado de la selección hasta que no se realice la verificación de asociaciones, puesto que es posible que esa asociación tenga una o varias posibles correspondencias al otro lado, que hasta que no realicemos la verificación no podremos detectar. Por lo tanto, a este nivel (antes de verificar), sólo aparecerá seleccionada la correspondencia al pinchar sobre una asociación totalmente correcta como en esta pantalla. Para más detalle consultar el documento de diseño de la herramienta.



(Figura 6.13)

Como vemos, el botón de verificación de “CARDINALIDAD/NAVEGABILIDAD” aparece habilitado una vez seleccionado un elemento válido.

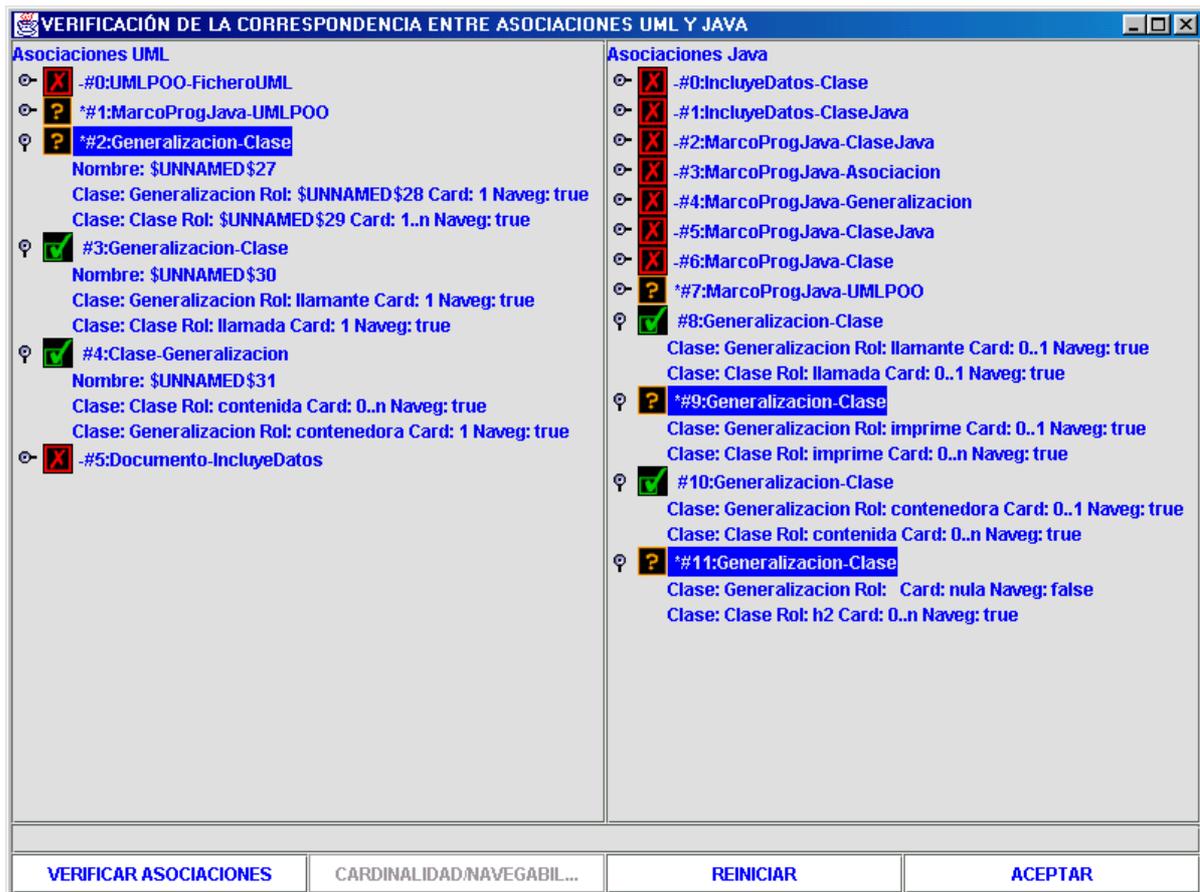
Tenemos el botón “VERIFICAR ASOCIACIONES”, donde al seleccionar aparecerá el resultado de la verificación de este modo:



(Figura 6.14)

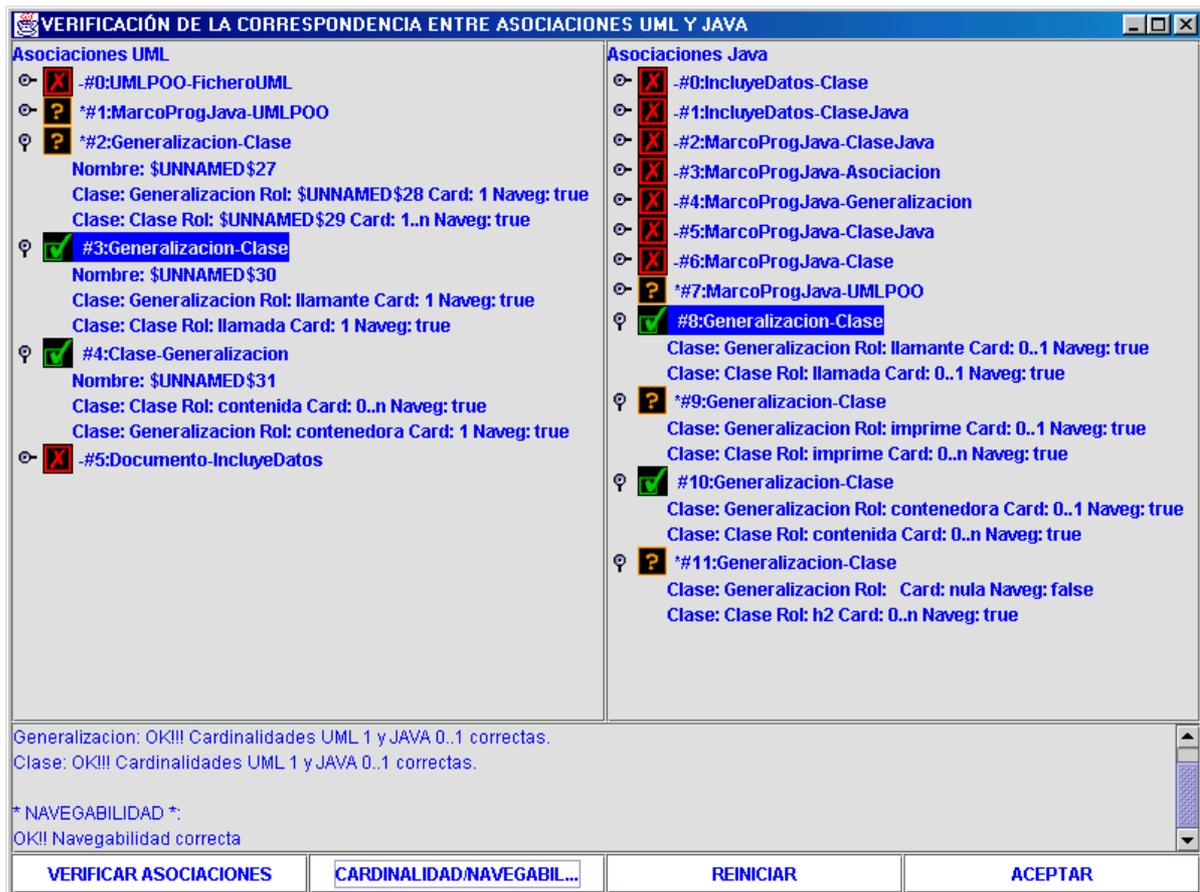
Como ya se ha explicado en apartados anteriores, aparecen los distintos iconos que identifican el resultado de la verificación de esa asociación.

Si nos encontramos un elemento con “?” en naranja, significa verificación parcial. Esto hace que al seleccionar una asociación con este icono, puede que al otro lado nos aparezcan varias selecciones con este mismo icono, que representan las posibles asociaciones que podrían corresponderse con la seleccionada inicialmente:



(Figura 6.15)

Dentro de esta pantalla también podemos pulsar el botón de “CARDINALIDAD/NAVEGABILIDAD”, si antes hemos seleccionado una asociación válida. Inmediatamente nos aparecerá el mensaje de comprobación:



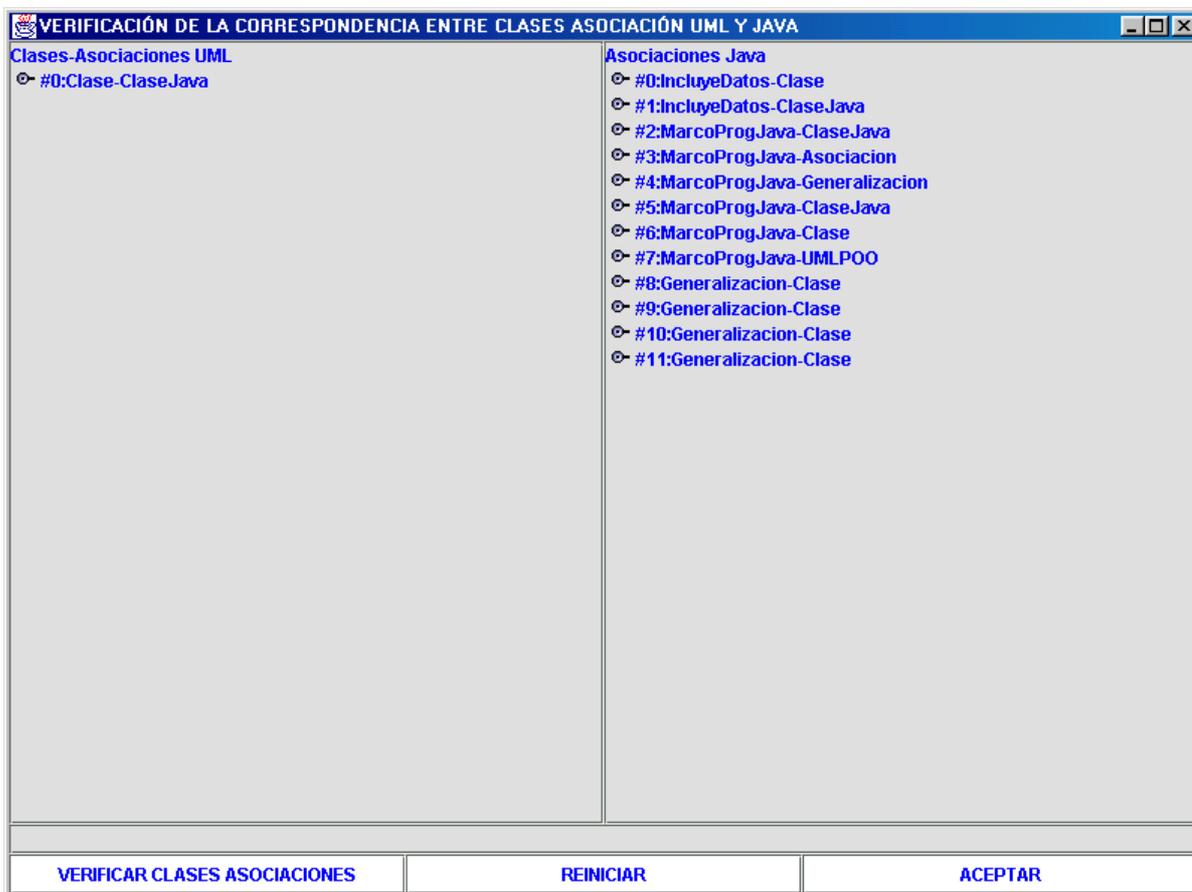
(Figura 6.16)

El botón “REINICIAR” limpiará la pantalla quedando en el estado inicial (figura 6.12).

Al pulsar sobre el botón “ACEPTAR” se cerrará esta pantalla para volver a la ventana principal.

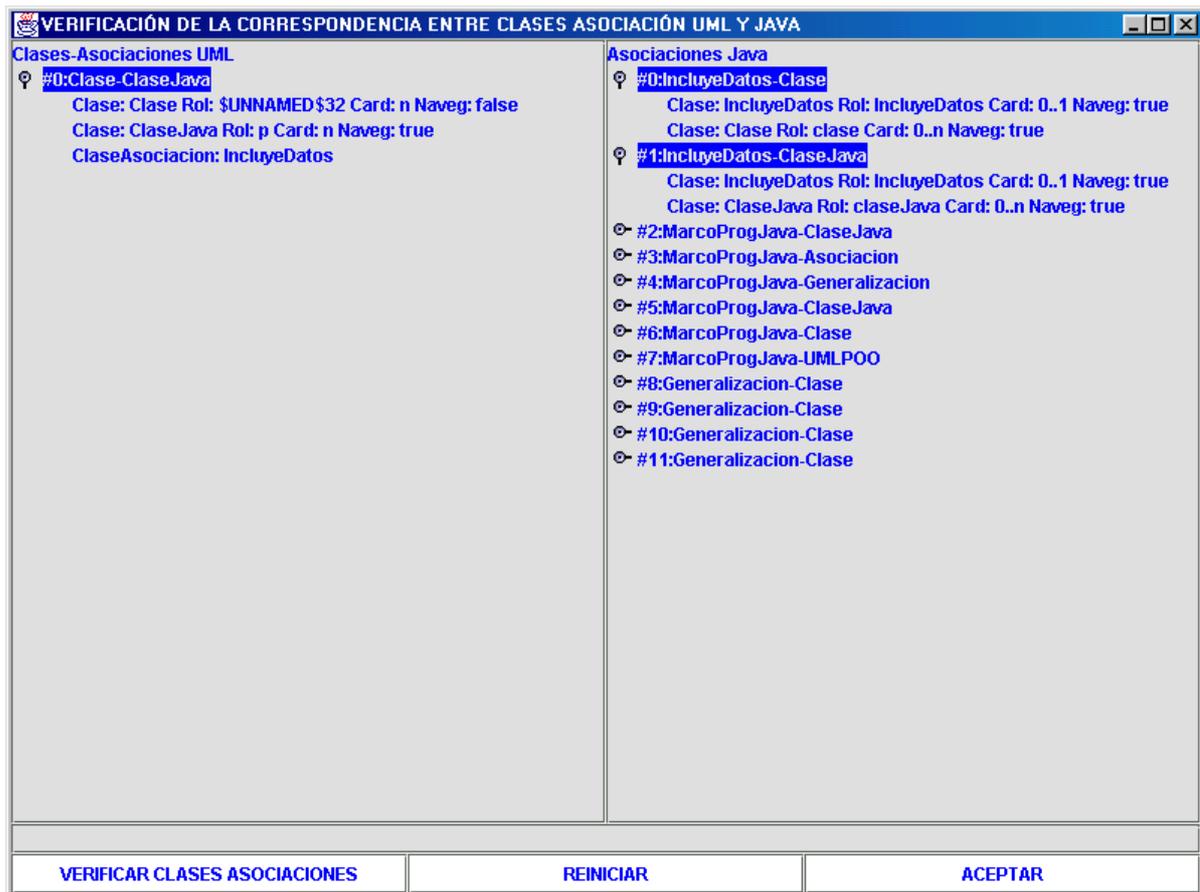
#### 6.4. Verificación de Clases Asociaciones

Otra opción dentro del menú principal (figura 6.3) es “CLASE ASOCIACIÓN”:



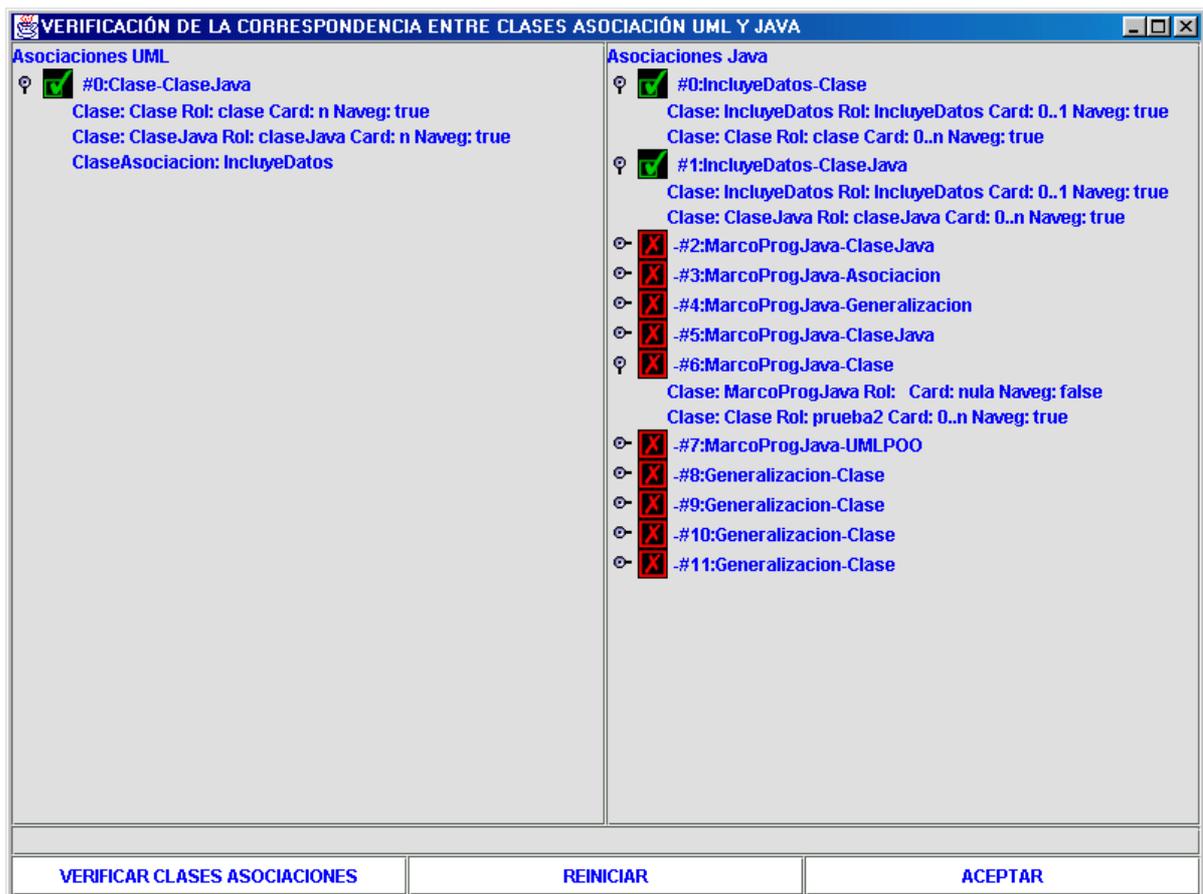
(Figura 6.17)

En esta pantalla, al igual que en el resto, podemos pulsar sobre un elemento y si éste existe aparecerá sombreado en los dos lados, pero en este caso tendremos que una clase-asociación UML se corresponde con dos asociaciones Java:



(Figura 6.18)

Para encontrar la clase asociación se busca entre las asociaciones Java las dos asociaciones que tengan como nombre de clases asociadas el nombre de clase asociación y cada una de las clases extremos de la clase asociación. Además deben cumplir que ambas asociaciones sean navegables en ambos sentidos y su cardinalidad la representada en el elemento UML. Es por esto que en esta pantalla sólo aparece el botón de “VERIFICAR CLASES ASOCIACIONES” puesto que la cardinalidad y navegabilidad ya se han verificado para poder encontrar la clase-asociación correspondiente.



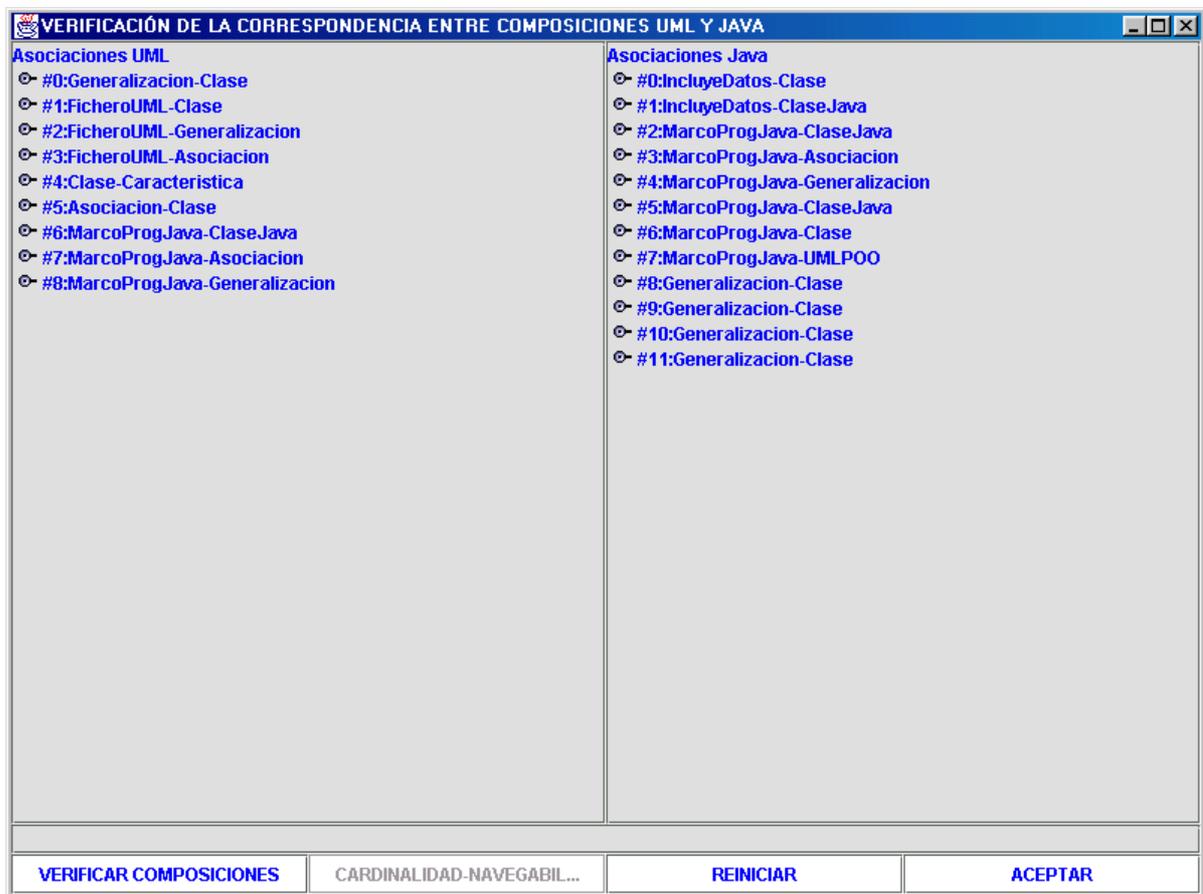
(Figura 6.19)

Como se puede observar, la mayoría de las asociaciones Java aparecen con aspa roja. Esto se debe a que sólo tenemos una clase asociación UML y por lo tanto entre todas las asociaciones encontradas en Java sólo dos serán las que formen este elemento. El resto podrán ser asociaciones comunes o de tipo composición, pero no forman parte de ninguna clase asociación.

En esta pantalla podremos pulsar en “REINICIAR” y se limpiará la pantalla quedando en el estado inicial (figura 6.18) ó podremos seleccionar “ACEPTAR” para cerrar esta pantalla y volver a la principal (figura 6.3).

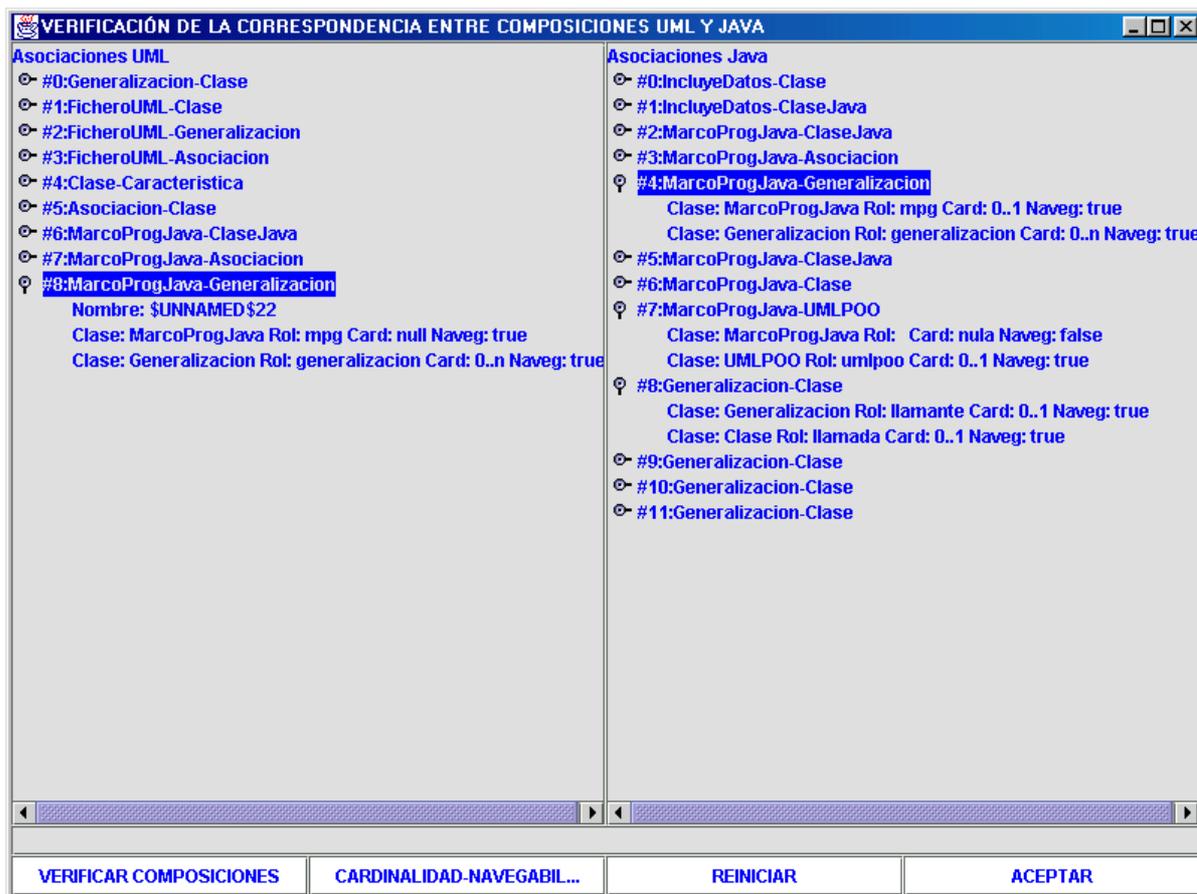
## 6.5. Verificación de Composición

Al pulsar en la opción “COMPOSICIÓN” en la pantalla principal (figura 6.3) tendremos una serie de pantallas equivalentes a las que aparecen en la opción de “ASOCIACIONES”:



(Figura 6.20)

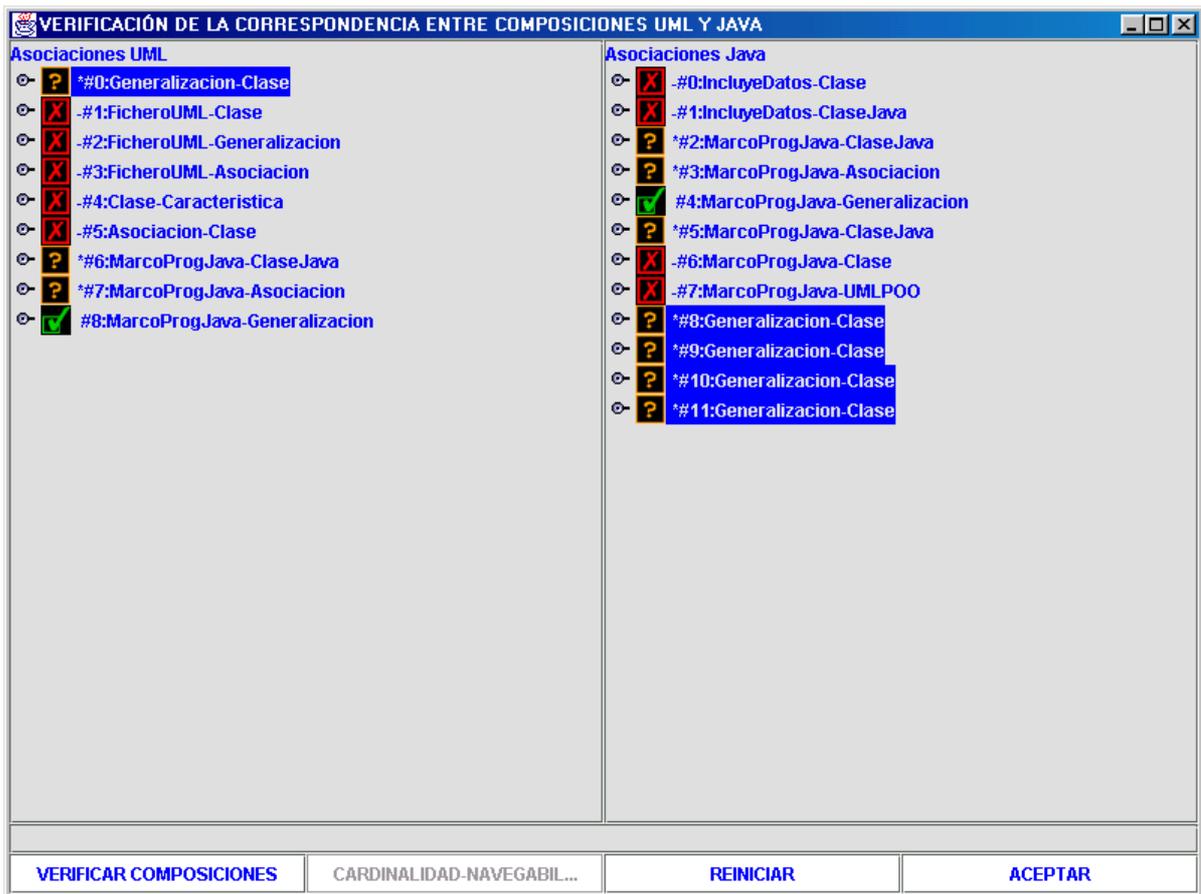
También podemos seleccionar un elemento dentro del diagrama UML o la implementación Java:



(Figura 6.21)

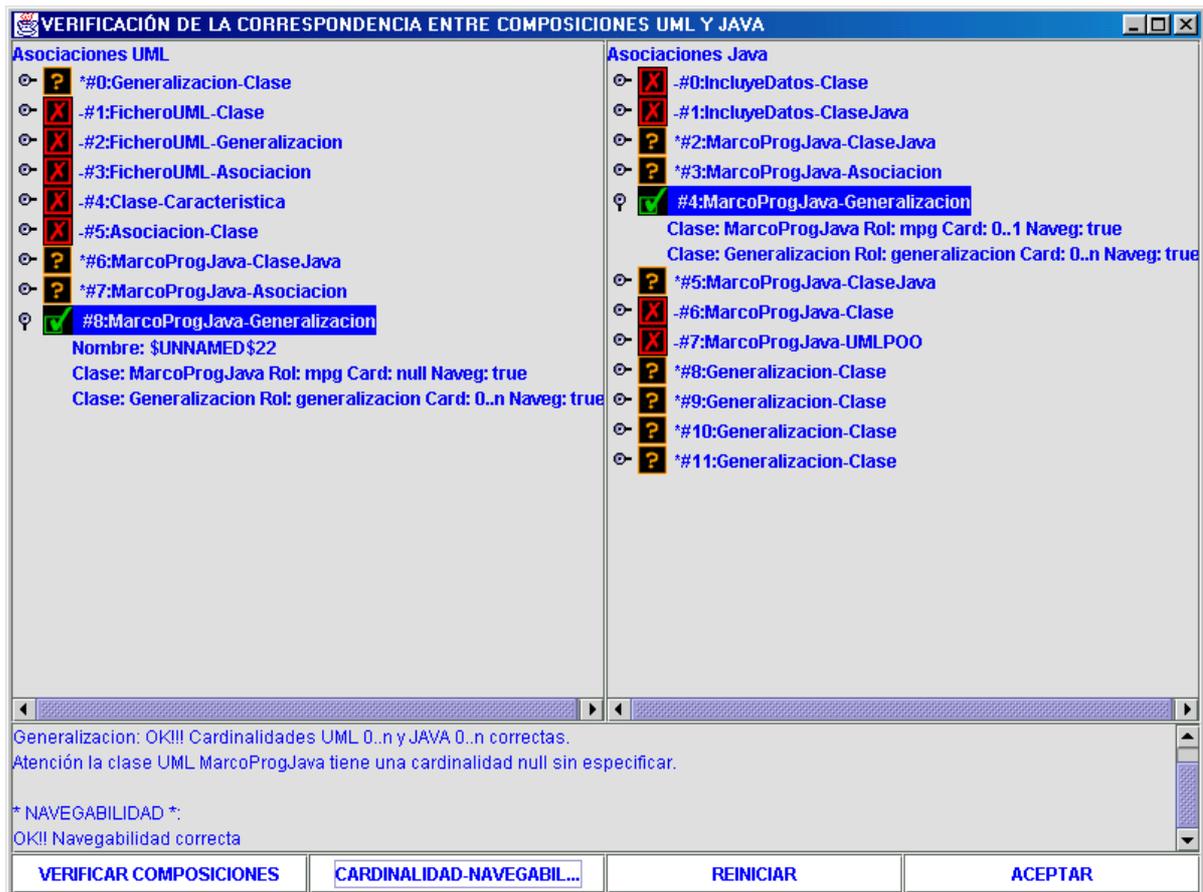
Volvemos a encontrarnos por un lado las composiciones UML y por otro las asociaciones encontradas en la implementación Java. Al igual que en ventana de asociaciones, al pulsar sobre cualquier composición aparece sombreada ésta y su correspondiente al otro lado, pudiéndonos encontrar composiciones indeterminadas. Este tipo de composiciones no tendrá su correspondencia al otro lado de la selección hasta que no se realice la verificación de composiciones, puesto que es posible que esa composición tenga una o varias posibles correspondencias al otro lado, que hasta que no realicemos la verificación no podremos detectar. Por lo tanto, a este nivel (antes de verificar), sólo aparecerá seleccionada la correspondencia al seleccionar una composición totalmente correcta como en esta pantalla.

Si pulsamos en “VERIFICAR COMPOSICIONES” tenemos:



(Figura 6.22)

Si pulsamos en “CARDINALIDAD/NAVEGABILIDAD” nos aparecerá el mensaje de comprobación:



(Figura 6.23)

También podemos reiniciar la pantalla o salir con el botón de “ACEPTAR”.

## 7. RESULTADOS, CONCLUSIONES Y PROPUESTAS

Después de realizar este estudio sobre la “Verificación de la Implementación de Elementos UML en Java” se ha obtenido como resultado, que realmente no existe demasiada información que podamos verificar sobre los elementos de un diagrama de clases, que se implementan en un lenguaje de programación orientada a objetos como es Java. Se ha podido comprobar que existen ciertos elementos ó características del diagrama de clases, que realmente es imposible localizar en un programa Java, lo que ha limitado bastante la herramienta realizada para este proyecto. Entre estos elementos tenemos:

- Asociación. No existe como tal dentro de una implementación en Java como ocurre con el caso de las clases, aunque realmente los atributos de clases emparejados dos a dos son los que permiten conocer verificar la correspondencia (apartado 4.3 Asociación).
- Nombre de asociación. Al no existir propiamente asociaciones en Java tampoco podemos obtener esta característica de la misma.
- Cardinalidad mínima de un atributo de asociación. Como se ha explicado en el apartado “Complejidad de las cardinalidades”, las cardinalidades se obtienen a partir de las propiedades de clases y en estas solo podemos especificar el valor máximo de elementos que podemos contener, pero no el mínimo.
- Cardinalidad máxima como valor determinado. Debido a los tipos de datos que utiliza Java (“Complejidad de las cardinalidades”) no es posible identificar de forma exacta un valor máximo de cardinalidad. Lo único que podemos obtener entonces es si la cardinalidad máxima es 1 o es n.
- Multiasociaciones entre dos clases: Al no existir propiamente asociaciones en la implementación Java, si existen varias asociaciones entre dos clases no podemos saber qué rol es el utilizado para cada asociación detectada. Se llega a una solución donde se asocian según el orden en que se encuentran los atributos o propiedades dentro de la clase asociada y esto normalmente da lugar a asociaciones indeterminadas (representadas con un icono con interrogación naranja), si no encontramos los roles seleccionados en las asociaciones localizadas en el diagrama de clases UML (apartado “Múltiples Asociaciones”).
- Composición: Al igual que ocurre con las asociaciones, tampoco tenemos elementos de este tipo dentro de una implementación y además es imposible saber si una asociación encontrada en la implementación es realmente una asociación o una composición, puesto que como ya se ha explicado en el apartado 4.4 “Composición”, se trata de una diferencia semántica que es posible recoger en UML pero no en Java.
- Clase-Asociación: Tampoco podemos implementar directamente este tipo de elemento UML en un programa Java (“Clase asociación”). Para este problema, en la herramienta se ha decidido utilizar una clase intermedia navegable, dónde el equivalente Java de este elemento UML son dos asociaciones navegables en ambos sentidos entre la clase asociación y cada una de las clases extremos de la clase asociación. (apartado 4.3.6 Clase Asociación)
- Asociación-Derivada: Tenemos el mismo problema que para los anteriores elementos, al ser un tipo de elemento que ofrece una semántica específica cuya implementación tampoco facilita el lenguaje (apartado “Asociación Derivada”).

Con todo esto llegamos a la conclusión de que el problema real es la forma en que se tratan las asociaciones en Java ó en cualquier lenguaje de POO y es a la utilización de punteros.

Como ya se ha comentado en el apartado “Integridad”, la forma de relacionar objetos a través de punteros crea un problema de coherencia entre pares de atributos Java correspondientes a la misma asociación UML, dónde la única información exacta que se puede obtener son los nombres de las clases que forman la asociación. Esto hace que no se tenga control sobre los atributos asociados, pudiendo encontrar atributos de asociación que no se apuntan entre sí para formar la asociación correctamente.

También hemos podido comprobar que una implementación en Java tampoco se adapta al análisis realizado con un diagrama de clases, puesto que no existen conceptos como agregación-composición ó asociación derivada, ni tampoco una forma exacta de implementar estos elementos.

Una propuesta que se presenta en este trabajo podría ser la creación de una capa intermedia entre el diagrama de clases UML y la implementación en un lenguaje de programación orientado a objetos, que garantice que realmente existe una asociación entre dos clases. Esto significaría controlar que cada objeto asociado (perteneciente a cada una de las clases de la asociación) cumpla con los requisitos de la misma, como son las cardinalidades. Debería obligar a que si existe una asociación entre dos clases con cardinalidad 1:n, realmente tengamos un objeto de tipo clase1 que puede estar asociado a varios objetos de la clase2, pero cada objeto creado en ésta última está asociado únicamente a un objeto de la clase1.

## 8. BIBLIOGRAFÍA

- [1] “The unified modeling language user guide”.  
Grady Booch, Ivar Jacobson y Jim Rumbaugh  
Editorial: Addison-Wesley, 1999
  
- [2] “Modelado de objetos con UML”  
Pierre-Alain MULLER  
Editorial: Eyrolles, 1997
  
- [3] “Programación en Java 2”  
John Zukowski  
Editorial: Anaya, 1999
  
- [4] Forum UML Cafe  
<http://cafe.rational.com/HyperNews/get/hn/umlcafe.html>