4 UML estructural

Tanto para AOO como para DOO se utilizan los conceptos y las notaciones, esencialmente gráficas, de UML (*Unified Modeling Language*). En una primera clasificación de la notación UML se puede dividir en varias vistas. Una vista es un subconjunto de construcciones de UML que representan un aspecto del sistema:

- UML estructural logico: describe la estructura lógica de los elementos del sistema y sus relaciones. Sus conceptos principales son las clases, los paquetes y los casos de uso. Esta vista incluye los diagramas de clases y los diagramas de casos de uso.

- UML Dinámico: describe las interacciones entre los objetos con el tiempo. Las vistas de comportamiento dinámico incluyen los diagramas de interacción, las máquinas de estado y los diagramas de actividades

- Implementación: describe la estructura física del SW en cuanto a los componentes de que consta y su ubicación. Está formada por los diagramas de componentes y de despliegue.

Este capítulo se centrará en el UML estructural lógico. Se llama así ya que muestra todas las relaciones posibles a lo largo del tiempo y no las que son válidas en un cierto momento. UML estructural lógico está constituido por:

- Diagramas de clases

- Diagrama de casos de uso

4.1 OMG y UML

OMG (*Object Managment Group*), creada en 1989, es una organización no lucrativa en la que participan más de 800 empresas de SW, HW, consultorías,... Su objetivo es la elaboración de estándares para la Programación Orientada a Objetos. Sólo se dedican a realizar documentos, no su implementación. Por ejemplo, CORBA (objetos distribuidos en la Red) es un estándar (documentos) de la OMG. Posteriormente, existen empresas que realizan su implementación. En el caso de CORBA, paquetes como MICO son componentes que dan soporte a los servicios establecidos en la documentación.

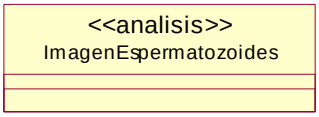
UML ha sido propuesto por OMG a ISO para que sea un estándar. UML es una cierta unificación de métodos anteriores como:

- OMT de Rumbaugh
- OOSE de Jacobson
- El método de Booch.

En 1997 aparece UML V1.0 presentado por la OMG.

4.2 Clases en UML

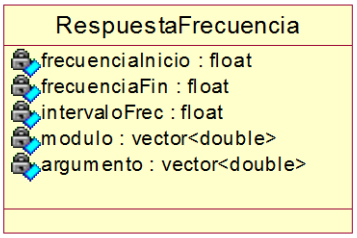
Las clases se representan por un rectángulo dividido en tres compartimentos: nombre de la clase, atributos y servicios. En el apartado del nombre, en la parte superior, se puede indicar un estereotipo, tales como <<análisis>>, <<diseño>>, ... El nombre de la clase será un sustantivo y empezará por mayúscula. Debajo del nombre se puede encontrar comentarios optativos entre llaves, { }.



Cada atribuyo tiene un nombre o identificador y un tipo. Un atributo se define de la siguiente forma:

Visibilidad nombre_atributo ':' tipo_atributo '=' valor inicial ´{`otras propiedades '}'

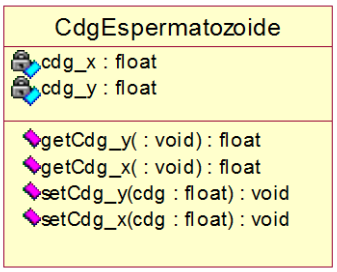
La visibilidad hace referencia a si el atributo es público, protegido o privado. UML emplea los símbolos **+**, **#** y – para indicar si es público, protegido o privado respectivamente[1]. El nombre del atributo es un sustantivo y empieza en minúsculas. Seguidamente aparecerá el tipo de atributo (float, char, int, ...). Opcionalmente puede aparecer el valor inicial del atributo y otras propiedades colocadas entre los signos de paréntesis.



Las especificaciones de las operaciones en UML tienen la siguiente sintaxis:

Visibilidad nombre_servicio '('lista de parámetros')':'tipo de retorno' {`otras propiedades´}`

El nombre del servicio empleará un verbo con un sustantivo. La primera letra se escribirá en minúscula. Entre paréntesis aparecerán los parámetros del servicio, siguiendo para cada uno de ellos, la regla sobre los atributos. Después se especificará el tipo de retorno y opcionalmente otras propiedades entre llaves. El usuario puede crear otros compartimentos para dar información adicional como excepciones, requisitos, etc.



Ejemplo 4.1

Realizar la implementación en C++ de la siguiente descripción UML referente a la clase Esfera.


```
#ifndef _INC_ESFERA_
#define _INC_ESFERA_

class Esfera
{
private:
        float radio_esfera;

public:
        Esfera()
                {this->radio_esfera = 2.0f;}
        float getRadio()
                {return (this->radio_esfera);}
        void setRadio(float radio)
                {this->radio_esfera=radio;}
};

#endif
```

4.2.1 Variantes en el concepto de clases

UML soporta diferentes tipos de clases que pueden ser implementadas o no por el lenguaje de programación.

[1] *Rational Rose* emplea los siguientes iconos para señalar la visibilidad del atributo o del servicio:  . El candado significa privado, la llave es protegido y la goma que es público.

4.2.1.1 Clases parametrizadas o contenedores

Las clases parametrizadas son unas clases que son empleadas para crear una familia de otra clase. Estas clases son un tipo de contenedor, son conocidas también como *templete*. No todos los lenguajes soportan los *templetes*. Por ejemplo, en ANSI C++ existe el paquete STL (*Standar Templete Library*), mientras que JAVA no soporta este tipo de clases. La biblioteca contenedora STL permite a los programadores desarrollar aplicaciones con contenedores estándar, tales como pilas, listas, colas de espera, así como manipular el contenido de dichos contenedores de diversas maneras. De esta forma, los desarrolladores hacen uso de servicios de alta calidad sobre componentes muy utilizados en casi todas las aplicaciones. Por ejemplo, la necesidad de mantener una lista dinámica de objetos es algo muy habitual. Al emplear los *templetes*, los desarrolladores no deben de implementar dichos servicios, sólo deben de saber utilizarlos. Por tanto, una clase parametrizada permite reutilizar el código.

Las clases parametrizadas son sólo plantillas de contenedores (vector, lista, árboles, ...). A las clases que definen un contenedor de un tipo específico, se las llama clases instanciadas, esto es, las clases instanciadas son instancias de clases parametrizadas. UML utiliza los signos de desigualdad, <>, para definir el tipo específico acompañado con el nombre del contenedor.

Ejemplo 4.2

Se desea realizar una aplicación para un *pocket* sobre los pasajeros de un vuelo. Plantéese bajo los *frameworks* de ANSI C++.

En jerarquía a dos niveles se plantea las siguientes características:

1. Mantener una lista de pasajeros asociados a un vuelo

    a. El sistema debe de tener de cada pasajero, el nombre, el número de pasaporte y el asiento.

    b. El sistema debe dar el número total de pasajeros, de asientos ocupados y asientos libres.

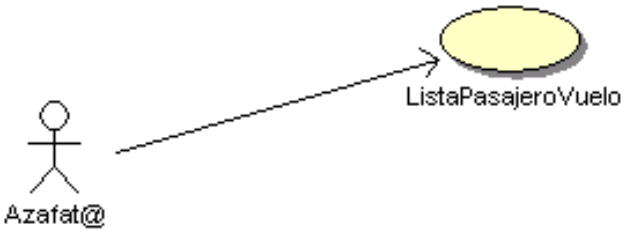
    c. El sistema debe de listar los datos de todos los pasajeros.

    d. El sistema debe de añadir datos sobre los pasajeros.

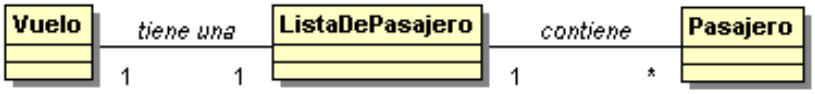
Los términos para el glosario serían: Pasajero, Vuelo, Asiento, Pasaporte, Nombre,.... La lista de evento-actor-objetivo estaría constituida por:

Evento	Actor	Objetivo
Introducir datos pasajero	Azafat@	Formar la base de datos del pasaje
Visualizar datos pasajero	Azafat@	Verificar los datos de un pasajero
Ocupación del vuelo	Azafat@	Obtener datos de ocupación del vuelo

El caso de uso sería la "ListaPasajeroVuelo". Se podría considerar que los datos pudieran venir de una base de datos y ser cargadas a través de algún tipo de conexión al pocket.

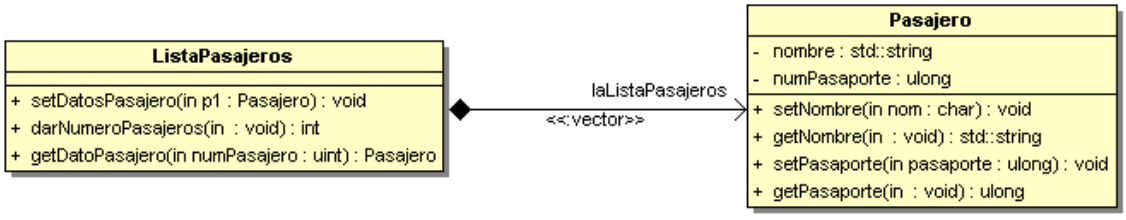


La primera tarea del AOO sería la construcción del modelo del dominio, éste podría ser:



Se deja al lector que haga el DSS y algún contrato de operación.

En un primer diseño se puede emplear ANSI C++. Se emplearán las *std::string* para los atributos de tiras de caracteres y las STL *std::vector* para el contenedor requerido. Para una mejor compresión de las clases parametrizadas sólo se muestra la solución lógica de una parte de la lista de Pasajeros. Un primer esbozo queda reflejado en el siguiente diagrama de clase de diseño, DCD:



Cuya implementación en C++ será:

```
#ifndef _PASAJERO_INC_
#define _PASAJERO_INC_

#include <string>

class Pasajero
{
public:
        void setNombre(const char *nom)
        {nombre = nom;}
        std::string & getNombre( void )
                {return  nombre;}
        void setPasaporte(unsigned long
        pasaporte)
        { numPasaporte =DNI;}
        unsigned long getDNI( void ) const
        {return (numPasaporte);}

private:
        std::string nombre;
        unsigned long numPasaporte;
};


#endif
```

```
#ifndef _INC_LISTA_PASAJEROS
#define _INC_LISTA_PASAJEROS
#include <vector>
#include "Pasajero.h";


class ListaPasajeros
{
public:
        void setDatosPasajero (const Pasajero p1)
        {laListaPasajeros.push_back(p1);}
        int darNumeroPasajeros ( void ) const
        { return laListaPasajeros.size(); }
        void iniciarLista ( void )
        { iteradorPasaje = laListaPasajeros.begin();}
        Pasajero  getDatoPasajero( unsigned numPasajero )
        { return (*(iteradorPasaje + numPasajero)); }

private:
        std::vector<Pasajero> laListaPasajeros;
        std::vector<Pasajero>::iterator iteradorPasaje;
};

#endif
```

Nótese los tres tipos de clase mostrados: conceptuales, de diseño y de implementación. En la página web de la asignatura encontrará las fuentes de este ejemplo.

4.2.1.2 Interfaces

Una interfaz especifica ciertas operaciones de algunos elementos del paquete que son visibles fuera del mismo. No necesita especificar todas las operaciones que soporta el paquete, por lo que el paquete podría incluir varias interfaces diferentes.

Una interfaz se define en un diagrama de clases utilizando un rectángulo, como el de icono de clase, pero con el estereotipo de <<interface>> en la división del nombre de la clase. No tiene atributos, por tanto, el icono sólo tiene dos divisiones. En Rational Rose también se representa por un círculo.

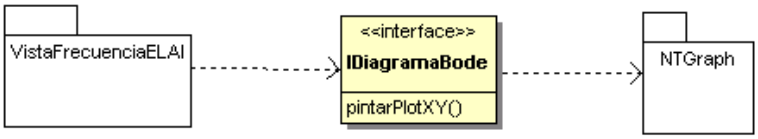
El sentido de las interfaces está relacionada con el concepto de cliente – servidor. Si un paquete está especializado en algunas tareas o servicios, se le dota de un interfaz para que los clientes pidan ese trabajo. Las modificaciones internas dentro del paquete no serán transmitidas a los clientes de estos servicios. Estos aspectos se verán con mayor detenimiento en el capítulo de diseño con patrones.

En C++ se emplea las clases abstractas para implementar los interfaces. En Java y en C# existe la palabra clave “interface”.

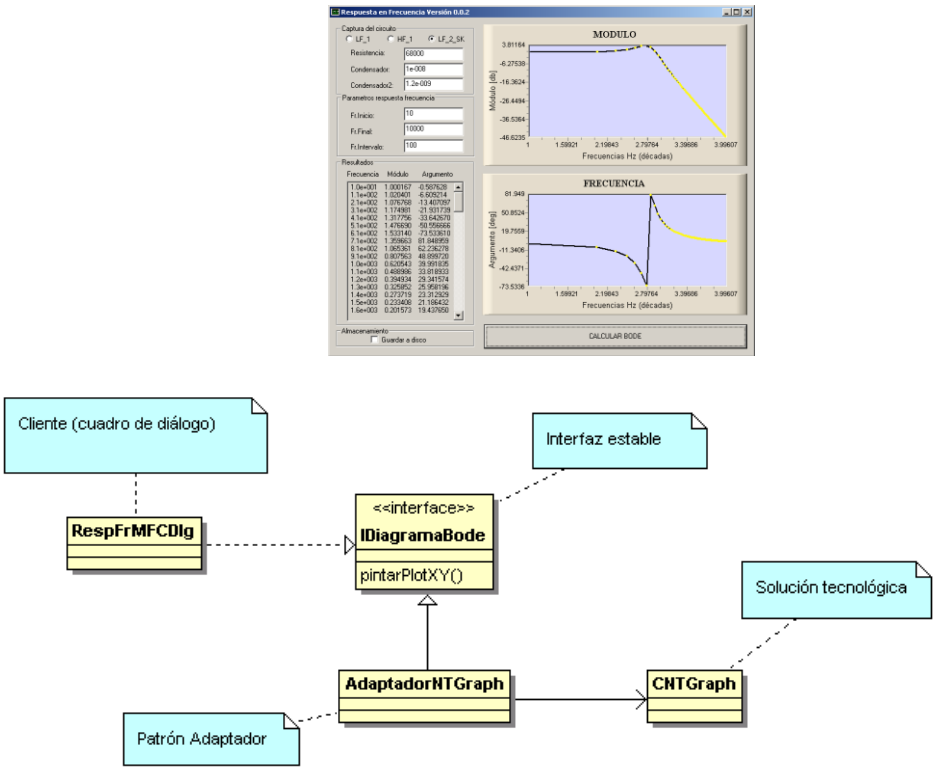
Ejemplo 4.3

Definir un interfaz para la visualización de los diagramas de Bode en la aplicación de respuesta en frecuencia de los filtros lineales.

Un buen diseño debería de independizar la aplicación de la visualización del diagrama de Bode. En el mercado existen distintos componentes para realizar un ploteado de una gráfica X-Y. Se ha localizado un componente gratuito llamado NTGRAPH[2]. Sin embargo, en el futuro se podría optar por otro tipo de componente. Para evitar las fluctuaciones e inestabilidades de este servicio, se define un interfaz estable a este servicio. El diagrama de paquetes quedaría:



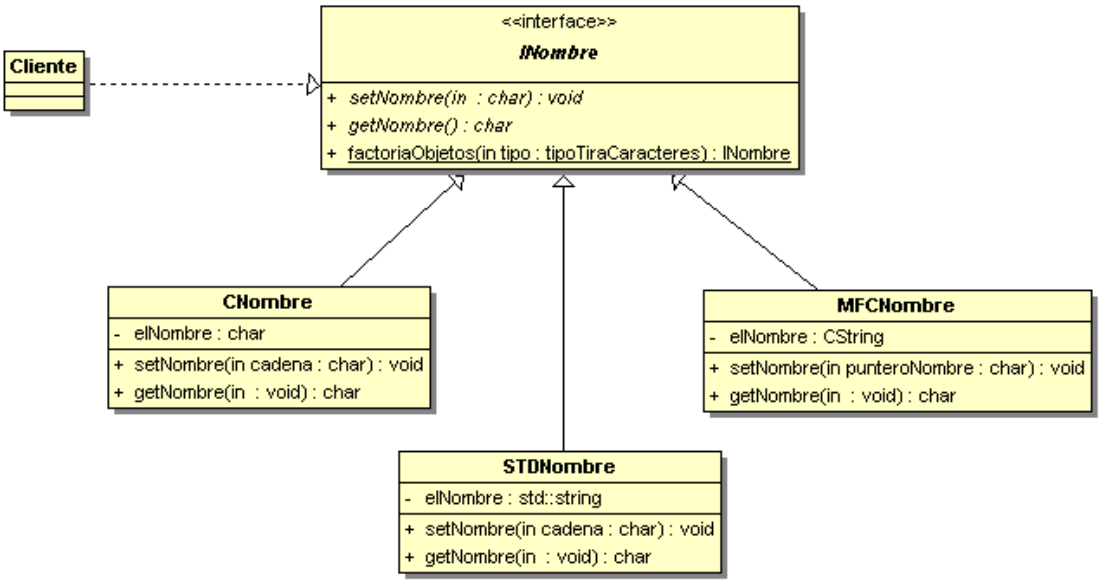
El DCD resultante de aplicar diversos patrones GoF muestra la solución lógica del problema:



[2] www.codeproject.com

Ejemplo 4.4

Realizar un paquete que sea capaz de ocultar el uso de framework para el manejo de tiras de caracteres. Empléese las MFC y ANSI C++.



```
#ifndef _INOMBRE_INC_
#define _INOMBRE_INC_

enum Plataforma{ESTANDAR_STL, ESTILO_C,
CADENA_MFC};

class INombre {
public:
  virtual void setNombre (const char *) = 0;
  virtual const char * getNombre () = 0;
  //Factoria de objetos
  static INombre *factoriaObjetos
       (enum  Plataforma);
};
#endif
```

```
#ifndef _INC_CNOMBRE_
#define _INC_CNOMBRE_

#include <string>
#include "INombre.h"

class CNombre : public INombre
{
public:
 virtual void setNombre(const char *cadena)
         { strcpy (elNombre, cadena); }
 virtual const char * getNombre (void)
         { return (elNombre);}
private:
        char elNombre[80];
};
#endif
```

```
#ifndef _INC_MFCNOMBRE_
#define _INC_MFCNOMBRE_

#include <afx.h>
#include "INombre.h"

class MFCNombre : public INombre
{
public:
 virtual void setNombre(const char *cadena)
         { elNombre=cadena; }
 virtual const char * getNombre (void)
         { return (elNombre);}
private:
        CString elNombre;
};
#endif
```

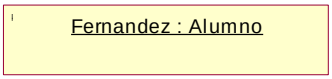
```
#ifndef _INC_STDNOMBRE_
#define _INC_STDNOMBRE_

#include <string>
#include "INombre.h"

class STDNombre : public INombre
{
public:
 virtual void setNombre(const char *cadena)
         { elNombre = cadena; }
 virtual const char * getNombre (void)
         { return (elNombre.c_str());}
private:
        std::string elNombre;
};
#endif
```

```
#include <iostream>
#include "../includes/STDNombre.h"
#include "../includes/CNombre.h"
#include "../includes/MFCNombre.h"

//Método único para producir los objetos nombres
INombre* INombre::factoriaObjetos(enum Plataforma tipo)
{
  if(tipo == ESTANDAR_STL) return new STDNombre;
  else if(tipo == ESTILO_C) return new CNombre;
  else if(tipo == CADENA_MFC) return new MFCNombre;
  else return NULL;

}
using namespace std;
int main ( void )
{
        INombre *pNombre1 = INombre::factoriaObjetos(ESTANDAR_STL);
        INombre *pNombre2 = INombre::factoriaObjetos(ESTILO_C);
        INombre *pNombre3 = INombre::factoriaObjetos(CADENA_MFC);

        pNombre1->setNombre("Manolo Gonzalez");
        pNombre2->setNombre("Pedro Lopez");
        pNombre3->setNombre("Ana Rodriguez");

        cout << pNombre1->getNombre() << endl;
        cout << pNombre2->getNombre() << endl;
        cout << pNombre3->getNombre() << endl;


        delete pNombre1, pNombre2, pNombre3;
        return 0;
}
```

Se recuerda que las clases abstractas en C++ carecen de instancias.

4.3 Representación de los objetos

Los objetos se representa parecido a una clase, indicando los valores instanciados de los atributos. El nombre del objeto, el cual es opcional, va seguido de ":" y el nombre de la clase, todo ello subrayado. En general, se suele omitir el tipo de los atributos, así como el comportamiento de los servicios, porque ambos se conocen gracias a la especificación de la clase.



Los objetos suelen aparecer en los diagramas de interacción. Estos artefactos describen la dinámica de la aplicación. Se verán en el UML dinámico y se analizarán con mayor detalle en el siguiente capítulo.

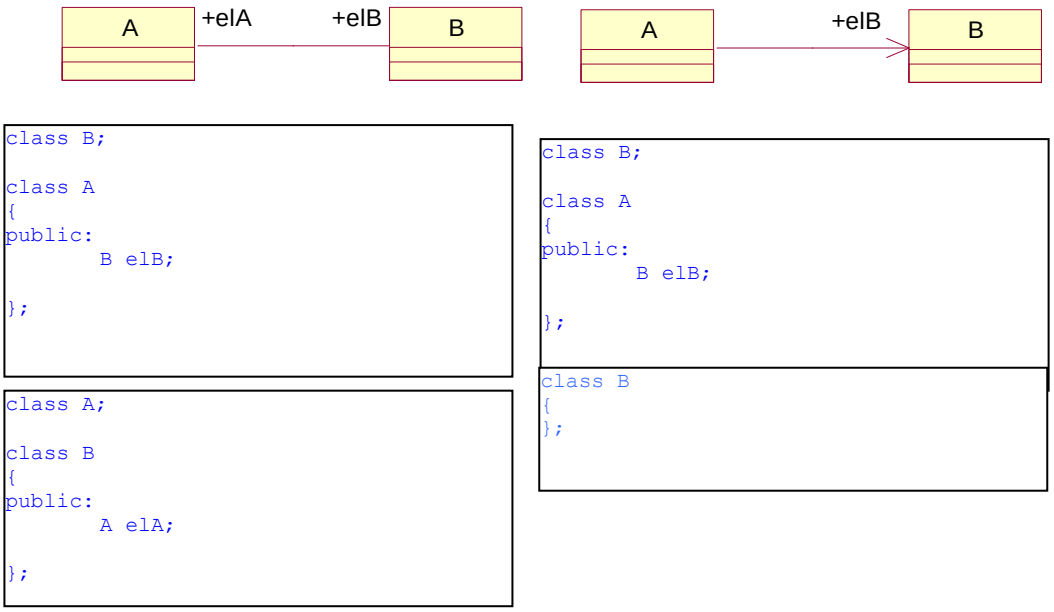
4.4 Tipos de relaciones

Hay varios tipos de relaciones en UML entre las clases, de las que se destacan: asociación, generalización y varios tipos de dependencia.

4.4.1 Asociaciones

Las asociaciones son conexiones semánticas entre clases. Cuando una asociación conecta a dos clases, cada clase puede mandar un mensaje a la otra. Una asociación permite a una clase conocer los atributos y las operaciones públicas de la otra clase. La asociación se representa por una línea continua. Esta opción indica que las clases unidas por la asociación tienen dependencia cíclica. Esta presentación es válida en el AOO y en su artefacto de Modelo del Dominio. Sin embargo, en el diseño se verá que esta forma de actuar es impropia con una buena organización de las responsabilidades entre las clases. Por este motivo, en las asociaciones de diseño suelen aparecer reflejadas unas flechas que indican el sentido de la navegabilidad. Este concepto indica que la asociación entre las clases es de carácter unidireccional. La fecha refleja que la clase a la que apunta puede ser empleada por la clase que emana la asociación, pero no en sentido contrario.

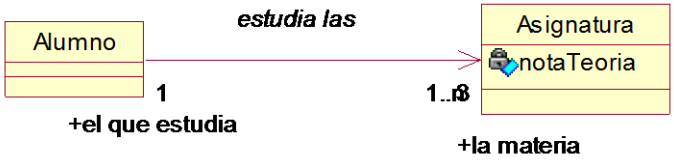
Si se pone un ejemplo de asociación entre la clase A y B de forma que sea bidireccional o unidireccional, véase el resultado de su implementación en C++:



Cada conexión de una asociación a una clase puede tener nombre (el rol que desempeña en la asociación), visibilidad (si se pueden ver los atributos y servicios) y la propiedad más importante que es la multiplicidad: cuantas instancias de una clase se puede relacionar con una instancia de otra clase. Si la asociación, además del nombre, tiene atributos se dice que es una clase de asociación, la cual será tratada más adelante.

La clase A está asociada con la clase B cuando:

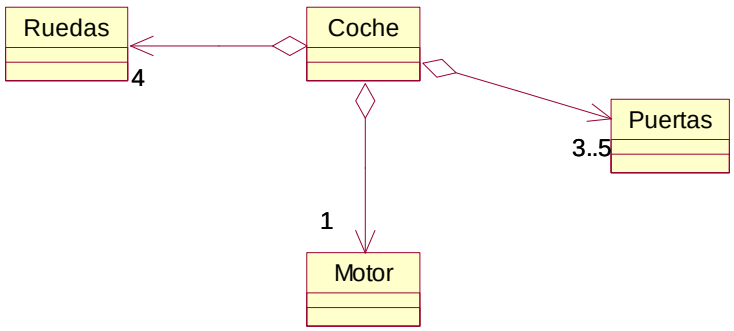
- Un objeto de la clase A usa un servicio de un objeto de la clase B.
- Un objeto de la clase A crea un objeto de la clase B.
- Un objeto de la clase A tiene un atributo cuyos valores son objetos de la clase B o colecciones de objetos de la clase B.
- Un objeto de la clase A recibe un mensaje con un objeto de la clase B pasado como argumento.



En resumen, si un objeto de la clase A tiene que saber algo de un objeto de la clase B aparece la asociación, i.e. se establece la relación "necesito-conocer".

4.4.1.1 Agregación y composición

Una agregación es una forma de asociación. Es un tipo de asociación del todo con las partes. Se represente con una línea continua, con un diamante en la clase que representa el todo y una flecha en la parte.



Un caso particular de la agregación es la composición o agregación por valor. En una composición, la vida del todo es la vida de las partes, i.e. que cuando se destruye el objeto compuesto también se destruyen las partes. Esta particularidad no se cumple en la agregación. Además, un objeto "parte" sólo puede ser de un objeto compuesto y no puede pasar de un objeto compuesto a otro.

En una composición las partes son atributos del todo. Se representa en UML mediante el diamante relleno.

Considere la agregación cuando:

- Existe un ensamblaje obvio del todo con las parte.
- Alguna propiedad del compuesto se propaga a las partes.
- Las operaciones que se aplican sobre el compuesto se propagan a las partes, como la destrucción, movimiento o grabación.
- El tiempo de vida de la parte está unido al tiempo de vida del compuesto (existe una dependencia de creación-eliminación de la parte con el todo).

Las prestaciones de la agregación se dan en el paso del análisis al diseño, por lo que no es muy significativa su inclusión en el Modelo del Dominio. Los beneficios son:

- Ayuda en la identificación de un creador utilizando el patrón GRASP creador.

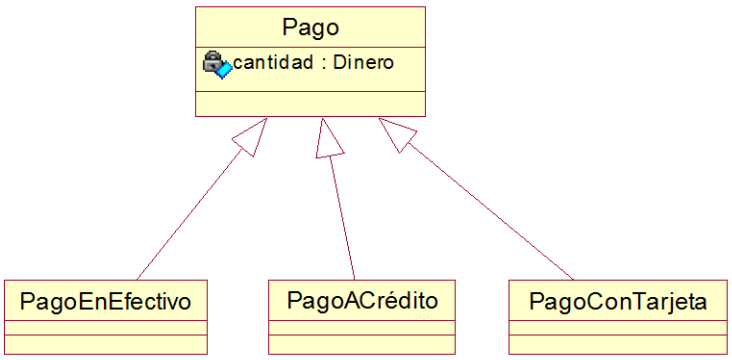
- Las operaciones –como la copia y la eliminación– que se aplican al todo a menudo se propagan a las partes.

- Como recomendación, en caso de duda hay que descartarla. Los noveles del AOO/D utilizan la agregación y la composición demasiado a menudo. Recuerde que ambos tipos son una asociación. En caso de duda utilice una simple asociación.

4.4.2 Generalización

La generalización y la especialización son conceptos fundamentales en el modelado del dominio que favorece una economía en las palabras; más aún, las jerarquías de clases conceptuales a menudo constituyen la fuente de inspiración para las jerarquías de clases SW que se aprovechan de la herencia y reducen la duplicación del código.

En el ejemplo:



La clase *Pago* representa la generalización y es la superclase, mientras las otras son de especialización y son subclases. En UML, la generalización se representa por una línea continua con una flecha hueca que apunta a la superclase.

Es una forma de construir clasificaciones taxonómicas entre los conceptos que son representados en una jerarquía de clases. Para la realización de estas generalizaciones deben de cumplir dos reglas:

A) Regla del 100%: La definición de la superclase se debe poder aplicar a la subclase. La subclase debe ajustarse al 100% de los atributos y asociaciones de la superclase.

B) Regla Es-Un: En el lenguaje natural debe comprobarse que la subclase Es Un tipo de la superclase (p.ej: *PagoACrédito* es un tipo de *Pago*).

Una subclase potencial debería de estar de acuerdo con la regla del 100% y de la regla Es-un. Se creará una subclase de una superclase cuando:

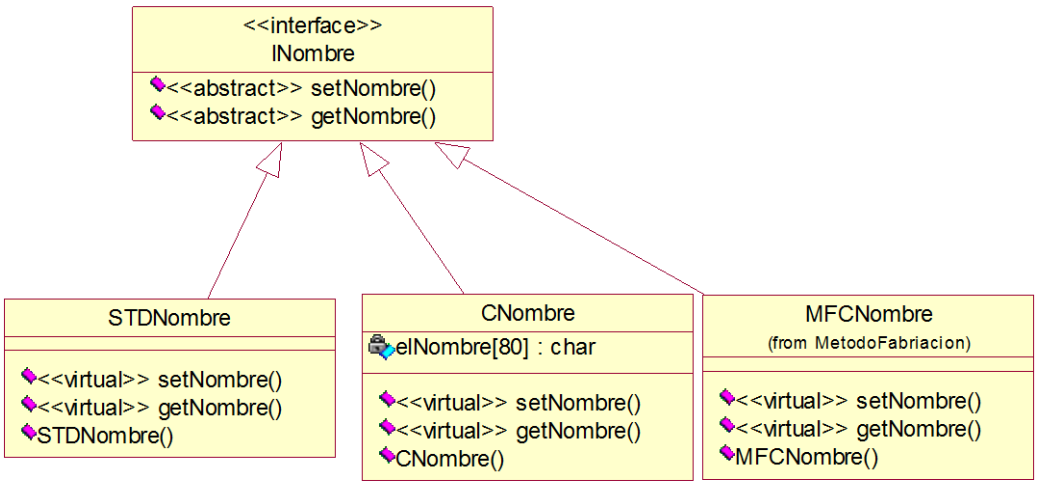
- La subclase tiene atributos adicionales.
- La subclase tiene asociaciones adicionales.
- La subclase funciona de manera diferente e interesante a la superclase o a otras subclases.

Se creará una superclase conceptual en una relación de generalización de subclases cuando:

1. Cuando las subclases potenciales representen variaciones de un concepto similar.
2. Las subclases se ajustarán a las reglas del 100% y Es-un.
3. Todas las subclases tienen el mismo atributo que se puede factorizar y expresarlo en la superclase.
4. Todas las subclases tienen la misma asociación que se puede factorizar y relacionar con la superclase.

4.4.2.1 Clases abstractas

Son aquellas que carecen de instancia y que son instanciadas desde una clase derivada (subclase). Recuérdese el ejemplo utilizado en las interfases.



En UML las clases abstractas se ponen su nombre en cursiva o se le añade el estereotipo de <<interface>> si éstas son empleadas para tal fin.

4.4.2.2 Herencia

Es una manera de implementar la generalización. La herencia no es la gran solución. Tiene el problema de la dependencia de la subclase con la superclase. De manera que si se modifica la superclase puede afectar a la subclase. Por este motivo sólo se empleará la herencia de clases cuando proceda de una relación de generalización conceptual. La composición es más robusta que la herencia.

Ejemplo 4.5

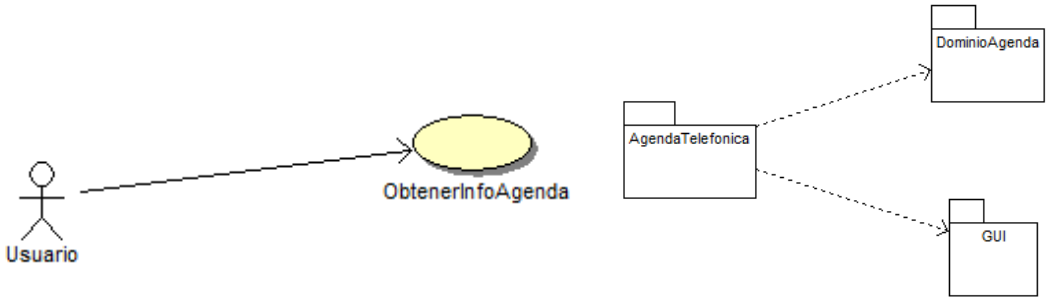
Realizar una agenda telefónica

Las características de la aplicación, en jerarquía a dos niveles, son:

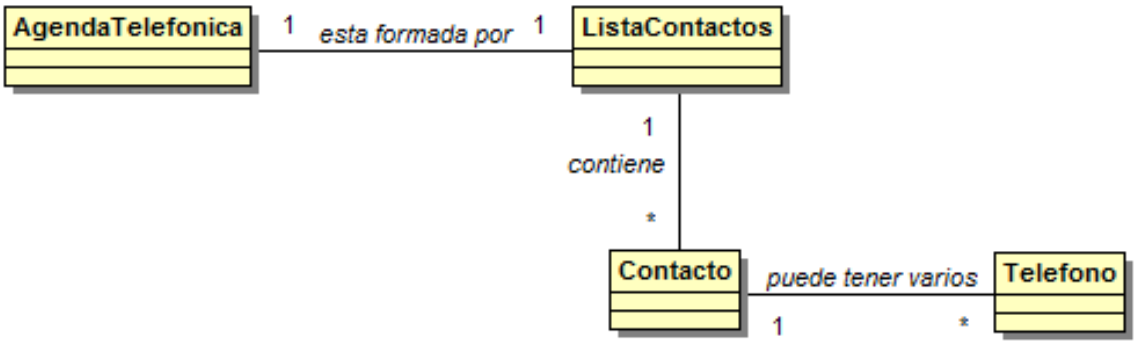
1. La agenda telefónica debe contener y gestionar la lista de teléfonos de los contactos:

1.a. Debe permitir añadir, eliminar, listar un contacto

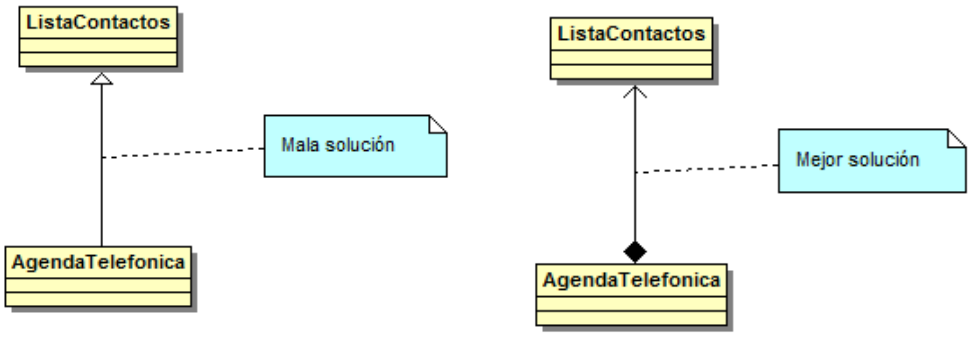
El glosario estaría formado por: listín telefónico, contacto, nombre, teléfono, agenda telefónica,… El caso de uso sería *ObtenerInfoAgenda*. La arquitectura básica estaría formada por un paquete del dominio y otro de visualización.



El modelo del dominio tiene las siguientes clases conceptuales:



La solución lógica podría ser de forma jerárquica o de composición. Sol. 1: Hacer *Agenda* sucbclase y que sea la *ListaUsuarios* de superclase. Esta solución no es buena, ya que la jerarquía es no conceptual. Sol. 2: Agenda tiene una composición con la clase instanciada de *ListaUsuarios*. Es mejor está solución; cualquier variación en la *ListaUsuarios* queda acotada por la asociación de agregación por valor. En el caso de que hubiese sido una subclase, las posibles variaciones en el futuro *ListaUsuarios* afectarían en la clase *Agenda*.



Ejemplo 4.6

Realizar un programa para la gestión de las fichas de libros y revista en una biblioteca.

La lista de características del sistema a dos niveles sería:

1. Gestionar las fichas de una biblioteca

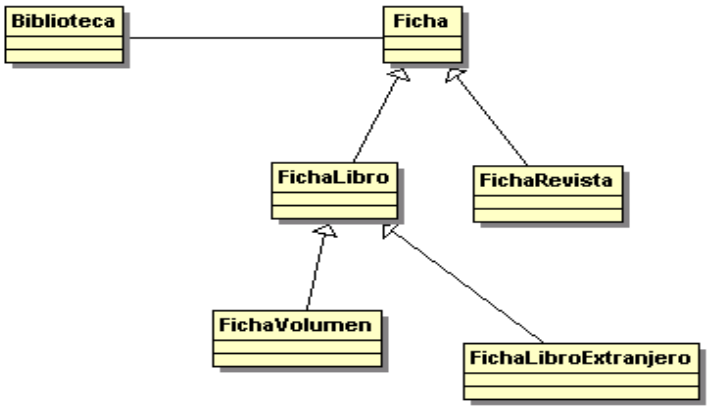
1.a. Insertar, eliminar y listar las fichas.

1.b. Las fichas pueden ser de libros o de revistas. Los libros pueden estar constituidos por uno o más volúmenes. Los libros pueden ser en lengua extranjera.

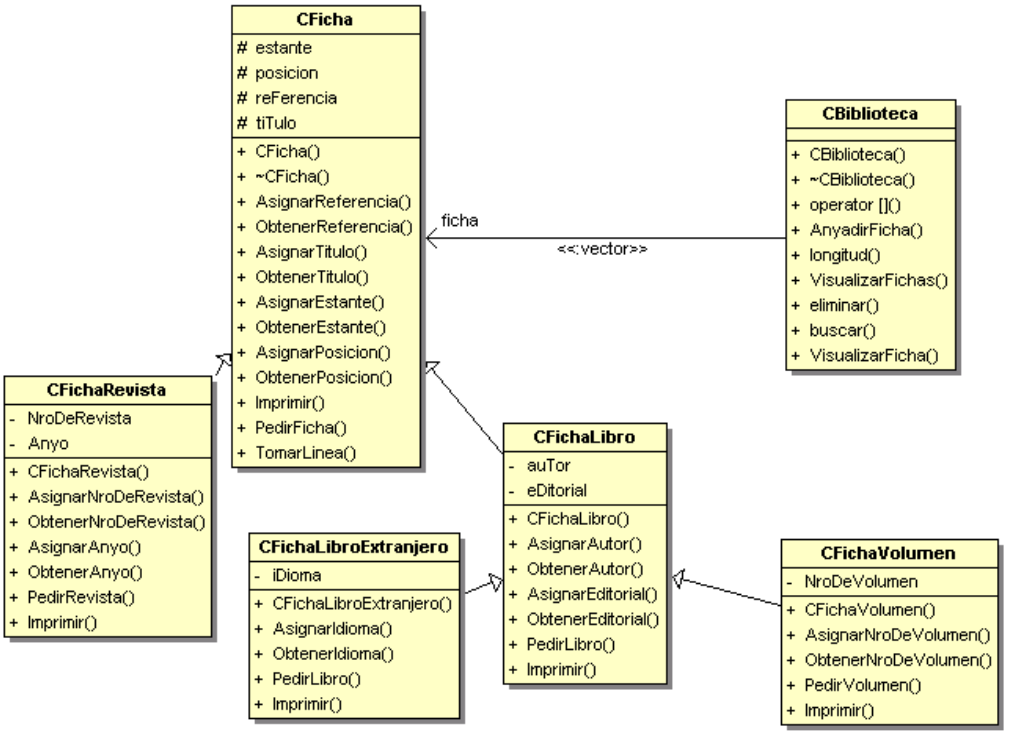
1.c. Las fichas de libros contendrán los campos de referencia, estante, posición, título, autor, editorial y volumen; mientras para las revistas deberán estar formados por el año y el número de la revista, junto con la referencia, estante. posición y el título.

El glosario estaría constituido por las definiciones de: Ficha, Biblioteca, Libros, Revista, Referencia, Título,...

El modelo del dominio podría ser del tipo:



Este ejemplo se ha utilizado como prácticas de la asignatura. Sobre el código entregado se ha realizado ingeniería inversa con el siguiente diagrama de clases:



En Java no existe la multiherencia.

4.4.3 Dependencia

Una dependencia es un tipo de relación entre dos o más elementos del modelo. La dependencia relaciona los elementos del modelo sin la necesidad de tener un conjunto de instancias para su significado. Indica una situación tal que un cambio en el elemento servidor puede requerir un cambio en el elemento cliente.

Una relación de dependencia muestra que una clase hace referencia a otra clase. Cuando hay dependencia entre dos clases, no se añaden atributos a la otra clase. Este aspecto lo diferencia respecto a la asociación. Cuando no hay una relación de asociación o de generalización se utiliza normalmente la dependencia para el resto de relaciones.

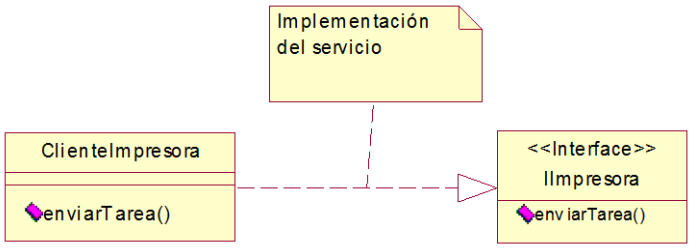
Las dependencias se suelen emplear con los paquetes.

En UML se representan mediante una línea a trazos acabada en flecha abierta.

4.4.4 Realización

La realización es un tipo de dependencia. La relación de realización conecta un elemento del modelo, tal como una clase, con otro elemento, como un interfaz, especificando su comportamiento pero no su estructura o implementación. Presenta un tipo de relación típica entre cliente con el servidor. El cliente debe tener por herencia o por declaración directa alguna de las operaciones públicas que tenga el proveedor.

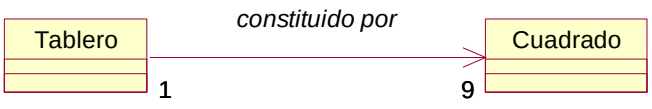
La realización se indica con una flecha de línea discontinua con una punta de flecha hueca cerrada. Es similar al símbolo de generalización pero con una línea discontinua, para indicar que es similar a un tipo de herencia con relación de dependencia.

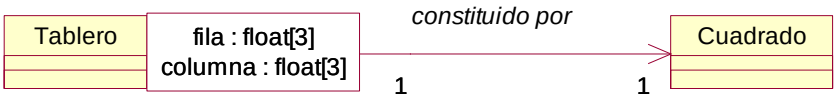


4.4.5 Otras asociaciones

De vez en cuando es útil dar más detalles de los que se tiene sobre una asociación. En estos casos se puede emplear un calificador. Un calificador es un valor que selecciona un objeto único del conjunto de objetos relacionados a través de la asociación. Los calificadores son importantes para modelar nombres y códigos de identificación.

Véase el modelado del tablero del juego de tres en rayas.

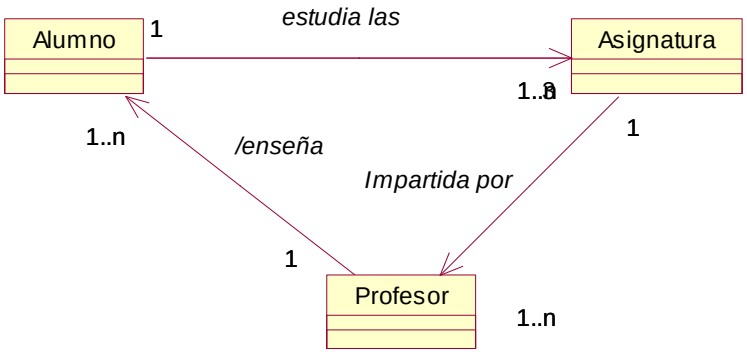




Se puede combinar la notación de asociación con calificador con otros adornos de las asociaciones.

4.4.5.1 Asociaciones derivadas

Una asociación derivada es una asociación redundante que se puede obtener como combinación de otras relaciones del modelo.



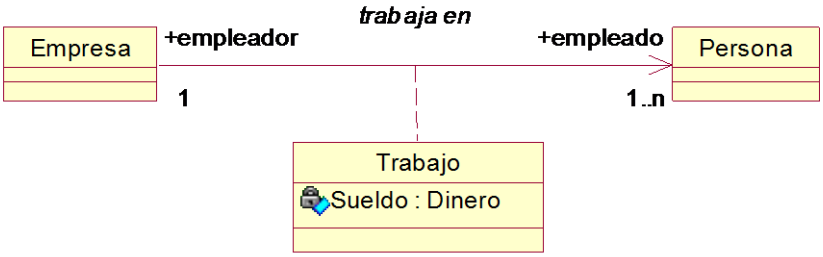
Un elemento derivado puede ser determinado a partir de otros. Los atributos y las asociaciones son los elementos comunes para ser derivados.

En UML un elemento derivado se representa anteponiendo una "/" al nombre del elemento.

4.4.5.2 Clases asociativas

En principio, una asociación no es una clase; no necesita tener atributos ni operaciones. No obstante, si una asociación debe tener atributos y operaciones propias o bien uno de los dos, entonces es preciso que se defina como clase. En este caso se habla de clase asociativa.

Una clase asociativa se representa como una clase colgada del símbolo de la asociación por medio de una línea discontinua.

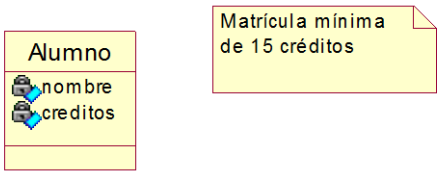


Los indicios de que se podría incluir clases de asociaciones en el diagrama de clases son:

- Un atributo está relacionado con una asociación.
- El tiempo de vida de las instancias de la clase asociación depende de la asociación.
- Existe una asociación de muchos a muchos entre dos conceptos e información asociada con la propia asociación.

4.5 Comentarios

Un comentario es una anotación unida a un elemento, carece de significado directo, pero muestra una información significativa para el que ha realizado el modelado. En UML, el comentario se pone dentro de un rectángulo con un vértice doblado, enlazado con una línea discontinua al cual se refiere.



4.6 Paquetes

Un paquete (*package*) es sólo una caja o contenedor que contiene elementos como clases, objetos, diagramas u otros paquetes. Si en el paralelismo con los sistemas biológicos, las clases son la analogía a los tipos de célula; los paquetes son a los órganos biológicos. Los paquetes agrupan las clases altamente cohesivas en la organización con un objetivo particular.

En un paquete puede aparecer tanto elementos del modelo como diagramas. El propósito general es la organización en grupos. Un paquete es usado para agrupar cosas que tengan algo en común. Los paquetes se pueden anidar dentro de otros paquetes. Gráficamente, un paquete es representado por UML como una carpeta.

Hay algunos elementos de un paquete que deben tener visibilidad para que puedan ser reconocidos desde otros paquetes. Los paquetes se configuran como un espacio de nombres de propósito general. Cada elemento del paquete es visible y reconocido dentro del paquete donde se ha declarado y su nombre no puede estar repetido dentro del paquete.

El nombre del elemento tiene que estar codificado junto con el identificador del paquete; siendo la regla el nombre del paquete más dos puntos (:) y el nombre del elemento.

Paquete : Elemento

Véase cómo este concepto se aplica en C++ con el espacio de nombres, utilizando el operador ámbito (::) :

```
#include <iostream>

int main()
{
        //Uso del paquete estándar
        //y sintaxis de C++ entre
        //el paquete y el elemento
        std::cout<< "Hola Mundo\n";

}
```

```
#include <iostream>

//Definición del espacio de nombres
using namespace std;

int main()
{
        cout<< "Hola Mundo\n";

}
```

4.6.1.1 Cómo dividir el modelo del sistema en paquetes

Ponga los elementos juntos dentro de un paquete sí:

- Se encuentran en la misma área de interés –estrechamente relacionados por conceptos u objetivos-.
- Están juntos en una jerarquía de clases
- Participan en los mismos casos de uso.
- Están fuertemente asociados.

Resulta útil que todos los elementos relacionados con el modelo del dominio tenga como raíz un paquete denominado *Dominio*. Todos los conceptos básicos o comunes se definan en un paquete que se pueda llamar *Elementos Básicos* o *Conceptos Comunes*.

Una forma de incrementar la estabilidad de los paquetes es reducir la dependencia de clases concretas de otros paquetes. El diseño empujará a realizar paquetes lo más autónomos posibles. Los paquetes estarán formados por clases funcionalmente cohesivas (estos aspectos serán tratados con mayor detalle en el capítulo del diseño orientado a objetos).

Hay que evitar la aparición de dependencias circulares, i.e. un paquete depende de otro y a su vez, éste depende del anterior. En el caso de necesidad de dependencia cíclica, se romperá el ciclo mediante la intersección de una interfaz estable.

Si los paquetes más responsables (de los que más se depende) son inestables, existe un mayor riesgo de extender el impacto de los cambios a quienes depende de estos paquetes. Hay que poner mucha atención al paquete que más se emplea. Sus constantes revisiones pueden afectar al resto. Hay que saber aislar para producir un diseño robusto.

En C++ se es más sensible a las dependencias que en Java. Un cambio en una clase C++ tiene un fuerte impacto en las clases que dependan de ésta.

El acoplamiento interno del paquete o cohesión relacional puede identificarse mediante:

$$CR = \frac{Número\ de\ relaciones\ \text{int}\ ernas}{Número\ de\ tipos}$$

Donde el número de relaciones internas incluye las relaciones de atributos y parámetros, herencia e implementaciones de interfaces. Un CR muy bajo indica falta de cohesión.

En resumen, las formas de incrementar la estabilidad en un paquete son:

- Contiene sólo o en su mayor parte interfaces y clases abstractas.
- No depende de otro paquete o de otros muy estables.
- Contiene código relativamente estable y se refinó antes de lanzar la versión.
- Es obligatorio una planificación a largo plazo de los cambios futuros.

Los paquetes son normalmente la unidad básica del trabajo de desarrollo y de versiones.

4.6.2 Vista de gestión del modelo

La gestión del modelo describe la organización de los propios modelos en unidades jerárquicas. El paquete es la unidad genérica de organización para los modelos. Esta vista cruza las otras vistas de UML y las organiza para el trabajo de desarrollo y el control de configuraciones.

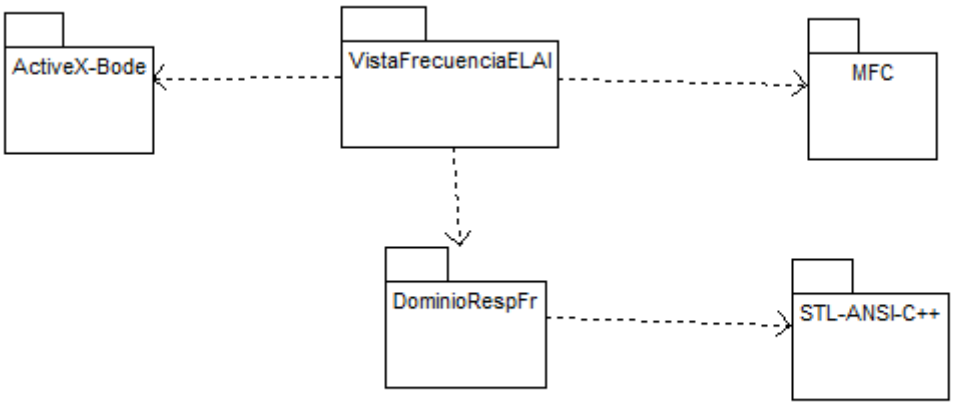
Los paquetes son unidades para manipular el contenido de un modelo. Cada elemento del modelo pertenece a un paquete. El modelo es una descripción completa del sistema desde un punto de vista, con una determinada precisión. Puede haber varios modelos de un sistema desde distintos puntos de vista; por ejemplo, un modelo de análisis y un modelo de diseño.

La información de gestión del modelo se representa en diagramas de clase, donde aparecen los paquetes y sus dependencias.

Ejemplo 4.10

Vista de gestión del modelo o diseño arquitectónico de la aplicación de respuesta en frecuencia.

Se propone realizar una división en paquetes de la aplicación según muestra el diagrama de paquetes siguiente:



4.7 Diagramas de casos de uso

Los diagramas de casos de uso sirven para mostrar las funciones de un sistema SW desde el punto de vista de sus interacciones con el exterior y sin entrar ni en la descripción detallada ni en la implementación de estas funciones. Reparte la funcionalidad del sistema en mensajes significativos entre los actores y el sistema. Un caso de uso es una descripción lógica de una parte de funcionalidad del sistema. El propósito de un caso de uso es definir una pieza de comportamiento coherente, sin revelar la estructura interna. Se utilizan tanto en la fase de requisitos como de análisis.

No obstante, las especificaciones de los requisitos se hacen escribiendo los documentos, no dibujando los casos de uso que es un paso opcional. El diagrama es sólo para ayudar a comprenderlos o reducir duplicaciones.

4.7.1 Relaciones entre casos de uso

Cuando se implementa un caso de uso, su funcionamiento se basa en la colaboración entre clases. Una clase puede participar en múltiples colaboraciones y por tanto, en múltiples casos de uso, dando paso a relaciones entre los casos de uso.

Aunque cada instancia de un caso de uso es independiente, la descripción de un caso de uso se puede descomponer en otros casos de uso más simple, apareciendo la relación de inclusión. Se utilizará el estereotipo *include* cuando una actividad se está repitiendo en dos o más casos de uso separados y se quiere evitar las repeticiones,

haciendo una factorización. Por tanto, la inclusión de casos de usos es esencialmente una forma de reutilización. Habrá que factorizar algunas subfunciones de forma separada y por tanto utilizar la relación *include* cuando:

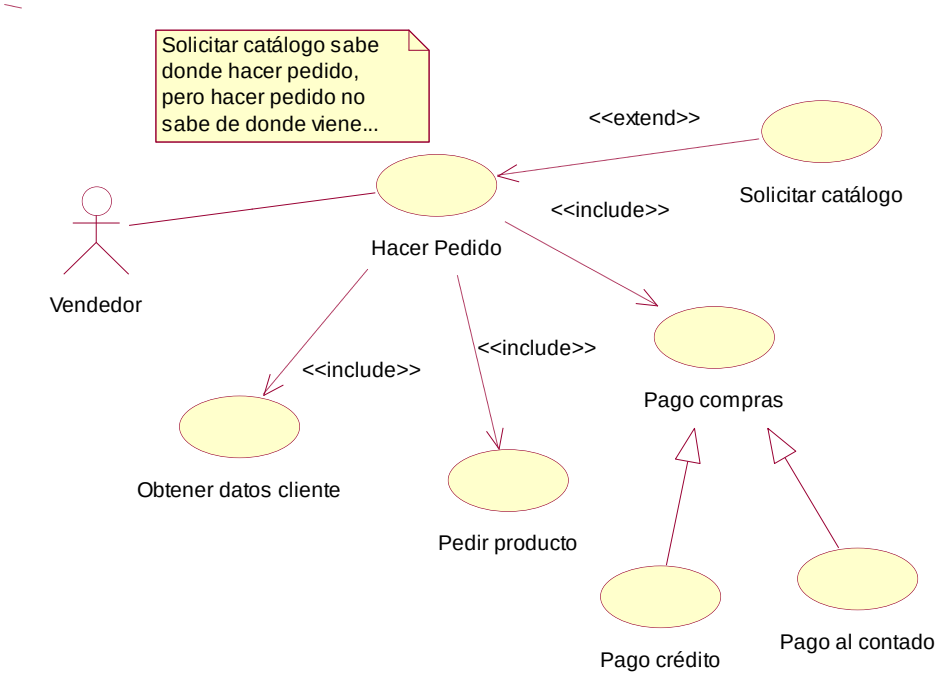
- Estén duplicados en otros casos de uso
- Un caso de uso es muy complejo y largo y separarlo en subfunciones facilita la compresión.

Un caso de uso se puede también definir como una extensión incremental de un caso de uso base. A esta relación se le llama de extensión y se utiliza el estereotipo de *extend*. Se dice que el caso de uso *A* extiende al *B* si dentro de *B* se ejecuta *A* cuando se cumple una condición determinada. Un caso de uso *A* extendido tiene que ser también ejecutable de forma separada de *B.*

Un caso de uso también se puede especializar en uno o más casos de uso. Esta relación es la de generalización/especialización de casos de uso. Un caso de uso *A* es una especialización de un caso de uso de *B*, cuando el *A* es un proceso más específico que el de *B.*

En UML, la relación entre los actores y los casos de uso se representarán como una asociación, indicando la comunicación bidireccional entre usuarios y sistema. Las relaciones de inclusión y extensión se dibujan como flechas de líneas discontinuas con la palabra clave <<*include*>> y <<*extend*>>, respectivamente. La relación de inclusión apunta al caso de uso a ser incluido; la relación de extensión señala al caso de uso que se extenderá. Las generalizaciones de casos de uso serán dibujadas mediante la flecha de relaciones de herencia, la punta señalará al caso de uso de generalización.

En la figura de abajo se muestra un ejemplo de diagrama de casos de uso sobre la gestión de un catálogo de ventas. En este diagrama aparecen todas las posibles relaciones anteriormente comentadas.



Se recuerda que lo importante son los documentos de los casos de uso. No hay que perder mucho tiempo en el diagrama de casos de uso. Es típico de los desarrolladores noveles dedicarse a dibujar casos de uso más o menos sofisticados, buscando relaciones y nuevas inclusiones, cuando lo importante es la captura de los requisitos funcionales (ver capítulo dedicado a la recogida de documentación y requisitos).

4.8 Problemas

1. Notación de UML sobre las clases.

2. Uso de las clases parametrizadas/instanciadas. Hágase un ejemplo sobre las pinturas de una galería de arte.

3. Defina un paquete sobre objetos geométricos, tales como cuadrado, rectángulo, triángulo,... Defina una interfaz.

4. Defina una jerarquía de clases para los objetos geométricos de la anterior pregunta. Presente las superclases y las subclases.

5. Cuando emplear una relación de asociación, agregación, composición, generalización y dependencia.

6. Cómo dividir una aplicación en paquetes.

7. Qué es la vista de gestión del modelo.

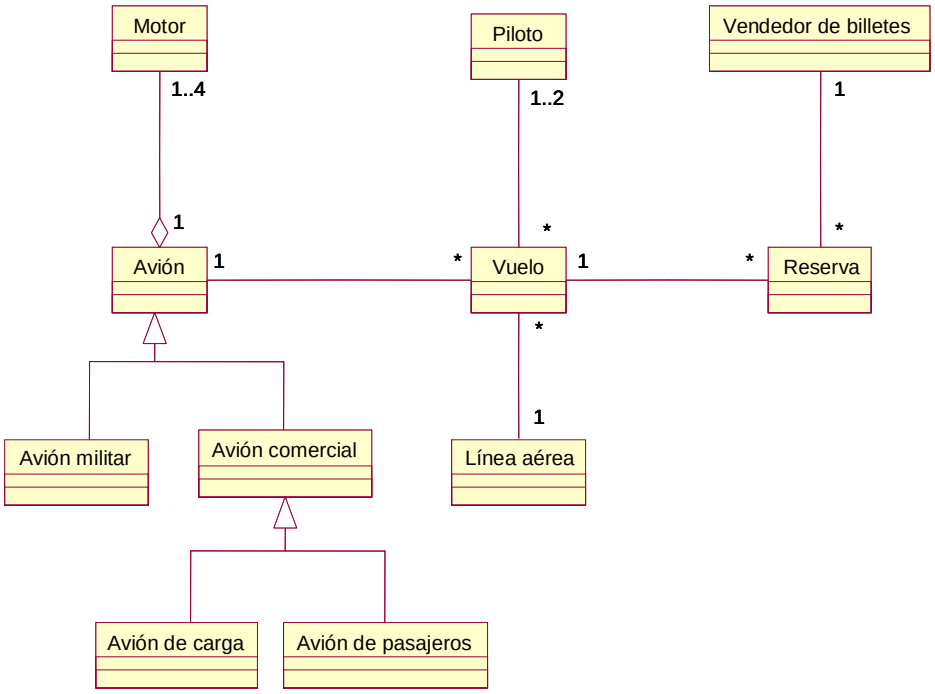
8. Relaciones en los diagramas de caso de uso.

Problema 4.1

Implementar en C++ la siguiente clase Dinero definida en UML según figura

Problema 4.2

Realizar una descripción textual del siguiente diagrama de clases.



Problema 4.3

Realizar una aplicación gráfica en cuyo mundo esté definida una CAJA que contiene un número indeterminado de ESFERAS. Se debe simular el rebote entre las propias ESFERAS y éstas con los SEGMENTOS de la CAJA.

1. Lista de características.
2. Modelo del dominio
3. Vista de gestión.
4. Diagrama de clases de diseño.

La lista de características del sistema a dos niveles sería:

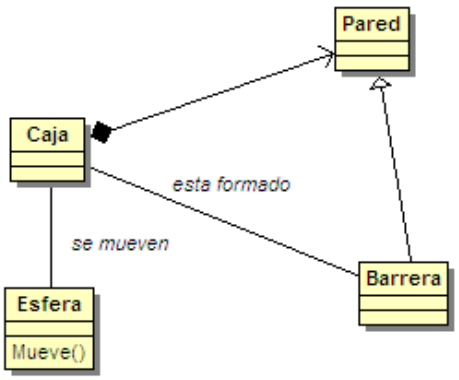
1. Simulación de esferas que rebotan dentro de una caja

1.a. Las esferas rebotan contra las paredes y barreras de la caja y con ellas mismas.

1.b. La caja es cerrada y no se pueden escapar las esferas.

1.c. La caja está formada por cuatro paredes y un número indeterminado de barreras.

Modelo del dominio:



La vista de gestión:

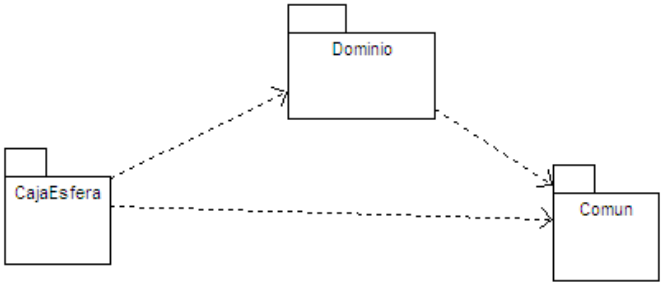


Diagrama de clase de diseño: