



Editorial de la Universidad
Tecnológica Nacional

Proyectos UML Diagramas de clases y aplicaciones JAVA en NetBeans 6.9.1

Ubaldo José Bonaparte

Cátedra Paradigmas de Programación

Facultad Regional Tucumán
Universidad Tecnológica Nacional – U.T.N.
Argentina

2012

Editorial de la Universidad Tecnológica Nacional – edUTecNe

<http://www.edutecne.utn.edu.ar>

edutecne@utn.edu.ar

© [Copyright] La Editorial de la U.T.N. recuerda que las obras publicadas en su sitio web son *de libre acceso para fines académicos y como un medio de difundir el conocimiento generado por autores universitarios*, pero que los mismos y edUTecNe se reservan el derecho de autoría a todos los fines que correspondan.

UML (Lenguaje de Modelado Unificado)

Introducción

En general al producirse un requerimiento de software, surge una idea. Por ejemplo un administrador general de un negocio que compra y vende productos, observa que utilizando la informática puede mejorar sustancialmente su administración. Entonces, teniendo una idea bastante clara de su necesidad, acude a especialistas en desarrollo de software. Después de varias entrevistas, los especialistas determinan que deben cumplir con las siguientes etapas de trabajo para generar el software adecuado a los requerimientos de su cliente:

- a) Relevamiento
- b) Análisis
- c) Diseño
- d) Desarrollo
- e) Capacitación
- f) Mantenimiento

El relevamiento consiste en un dialogo permanente de los especialistas y el cliente (puede incluir al personal de diferentes sectores del negocio) con el fin que los primeros identifiquen todos y cada uno de los componentes de dicho negocio y como interactúan. En definitiva, los especialistas deben comprender aquella idea detalladamente y mantenerla mientras se produce el software.

Para esto, los especialistas pueden hacer uso del Lenguajes Unificado de Modelado ya que les ayudará a capturar la idea del sistema requerido, para luego comunicarla a los involucrados en el proyecto. Esta tarea se lleva a cabo en las etapas de análisis y diseño, utilizando simbología y diagramas UML con el objeto de modelar el sistema.

Modelar el sistema utilizando los diagramas de UML, significara en definitiva contar con documentos que plasman el trabajo de capturar la idea para la posterior evolución del proyecto. El cliente podrá entender el plan de trabajo de los especialistas y señalar cambios si no se capto correctamente alguna necesidad; o bien, indicar cambios sobre la marcha del proyecto. A su vez, los especialistas encargados del desarrollo generalmente trabajaran en equipo, por lo que cada uno de ellos podrá identificar su trabajo particular y el general a partir de los diagramas UML.

UML proporciona las herramientas para organizar un diseño solido y claro, que comprendan los especialistas involucrados en las distintas etapas de la evolución del proyecto, y por que no para documentar un anteproyecto que será entregado al cliente.

Historia de UML

UML respaldado por el OMG (Object Management Group), es un lenguaje de modelado de sistemas de software. Diseñado como una herramienta gráfica donde se puede construir, especificar, visualizar y documentar sistemas.

Permite representar el modelo de un escenario, donde se describen las entidades intervinientes y sus relaciones. También podemos al describir cada entidad, especificar las propiedades y el comportamientos de las mismas.

Rational Software Corporation contrato en 1994 a James Rumbaugh y la compañía se convirtió en la fuente de los dos esquemas de modelado orientado a objetos más populares de la época:

- OMT (Object-modeling technique) de Rumbaugh, que era mejor para análisis orientado a objetos.
- Método Booch de Grady Booch, que era mejor para el diseño orientado a objetos.

Poco después se les une Ivar Jacobson, el creador del método de ingeniería de software orientado a objetos. Jacobson se unió a Rational en 1995, después de que su compañía Objectory AB fuera comprada por Rational.

En 1996 Rational concluyó que la abundancia de lenguajes de modelado estaba alentando la adopción de la tecnología de objetos, y para orientarse hacia un método unificado, encargaron a estos especialistas que desarrollaran un Lenguaje Unificado de Modelado abierto.

Se organizó en 1996 un consorcio internacional llamado UML Partners, para completar las especificaciones del *Lenguaje Unificado de Modelado (UML)*, y para proponerlo como una respuesta al OMG RFP. El borrador de la especificación UML 1.0 de UML Partners fue propuesto a la OMG en enero de 1997. Durante el mismo mes la UML Partners formó una Fuerza de Tarea Semántica, encabezada por Cris Kobryn y administrada por Ed Eykholt, para finalizar las semánticas de la especificación y para integrarla con otros esfuerzos de estandarización. El resultado de este trabajo, el UML 1.1, fue presentado ante la OMG en agosto de 1997 y adoptado por la OMG en noviembre de 1997.

UML desde 1995, es un estándar aprobado por la ISO como ISO/IEC 19501:2005 Information technology — Open Distributed Processing — Unified Modeling Language (UML) Version 1.4.2.

Diagramas de UML

UML está compuesto por diversos elementos gráficos que se combinan para conformar diagramas. Al ser UML un lenguaje, existen reglas para combinar dichos elementos. En conjunto, los diagramas UML brindan diversas perspectivas de un sistema, por ende el modelo. Ahora bien, el modelo UML describe lo que hará el sistema y no como será implementado.

Diagramas de clases



Si observamos a nuestro alrededor, veremos una serie de cosas (objetos), los cuales tienen atributos (propiedades) y nos damos cuenta que algunos realizan acciones (métodos). Esas cosas, naturalmente se agrupan en categorías (automóviles, viviendas, etc). Una clase es una categoría de cosas u objetos que poseen atributos y acciones similares.

Por ejemplo: la clase lavadora tiene las propiedades fabricante, número de serie y realiza las acciones de remojo, lavado, enjuague y centrifugado.

Las clases las representamos en un rectángulo compuesto por tres secciones 1) nombre de la clase 2) propiedades y 3) acciones.

Los diagramas de clases representan las clases intervinientes en el sistema, destacando con que otras clases se relacionan y como lo hacen.

Diagramas de casos de uso

Describen las acciones de un sistema desde el punto de vista del usuario. Si la finalidad es crear un sistema que pueda ser usado por la gente en general, es importante este diagrama, ya que permite a los desarrolladores (programadores) obtener los requerimientos desde el punto de vista del usuario.

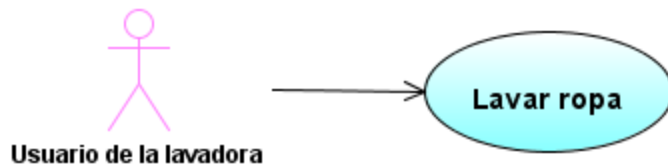


Diagrama de estados

Muestra las transiciones de un objeto en sus cambios de estados.

Por ejemplo: una persona es recién nacida, niño, adolescente o adulto.

Una lavadora puede estar en las fases de remojo, lavado, enjuague, centrifugado o apagada. Un elevador se puede mover hacia abajo, hacia arriba o estar en esta de reposo.

El símbolo de la parte superior indica el estado inicial y el de la parte inferior el estado final (apagado).

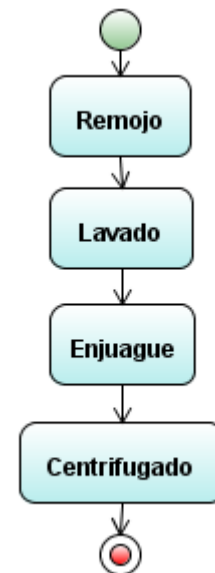


Diagrama de secuencias

Representan información dinámica ya que los objetos interactúan entre si mientras el tiempo transcurre. En definitiva, los diagramas de secuencias, visualizan la mecánica de interacciones entre objetos con base en tiempos.

Sobre nuestro ejemplo de la lavadora, encontramos los componentes manguera, tambor y drenaje como objetos que interactúan mientras transcurre el tiempo de funcionamiento.

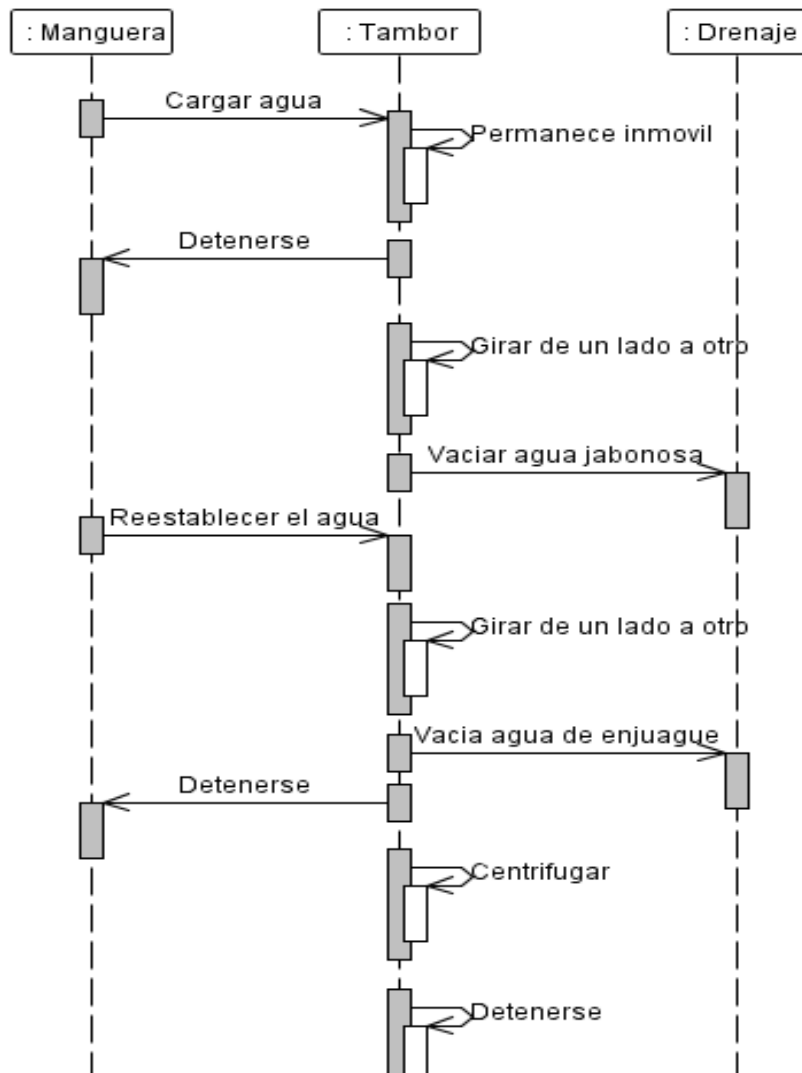
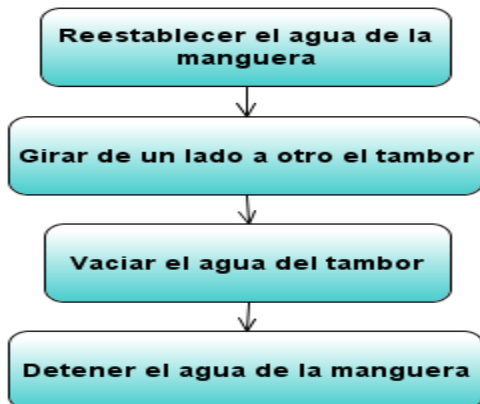


Diagrama de actividades



En un caso de uso como en el comportamiento de objetos en un sistema, siempre hay actividades que generalmente son secuenciales. Sin importar el tiempo, podemos reflejar en el diagrama de actividades, la secuencia de acciones que desarrollan los objetos.

Para el ejemplo de la lavadora aquí reflejamos la secuencia de las acciones 7 al 10 vistas en el diagrama de secuencias.

Diagrama de clases

Un diagrama de clases representa en un esquema gráfico, las clases u objetos intervinientes y como se relacionan en su escenario, sistema o entorno. Con estos diagramas, se logra diseñar el sistema a ser desarrollado en un lenguaje de programación, generalmente orientado a objetos. Estos diagramas los incorporan algunos entornos de desarrollo, tal es el caso de Eclipse con el plugin Papyrus o Netbeans con su respectivo plugin UML. Es un buen hábito generar proyectos UML con sus respectivos diagramas de clases para luego automáticamente obtener código fuente que nos colabore en el desarrollo del sistema o software.

-Conceptos básicos

Previo al desarrollo, en etapas de análisis y diseño de sistemas los diagramas de clases juegan un papel muy importante ya que permiten visualizar a partir de las clases y sus vínculos, como los objetos interactúan en el entorno propuesto.

Una *clase* va a representar a los objetos que se produzcan a partir de haberla instanciado, indicando claramente las propiedades y métodos que poseen. Si la clase es abstracta no podrá ser instanciada sino a partir de sus clases derivadas.

Una *relación* representa el detalle del vínculo entre dos clases, destacando el tipo (cual es la relación), la aridad o multiplicidad (cantidad de objetos de una y otra clase) y la navegabilidad (que objeto puede observar a otro). Ante un diseño orientado a objetos, es importante conocer la diversidad de relaciones que se pueden producir, necesitar o establecer entre clases.

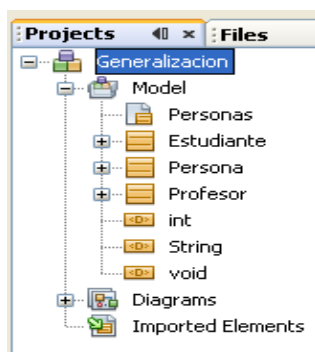
-Tipos de relaciones

Aprovecharemos la descripción de las relaciones para orientar a nuestros estudiantes hacia el código Java que involucran. Para esto describiremos en cada tipo de relación:

- Conceptos involucrados.
- Ejemplos de proyectos UML sobre el IDE Netbeans.
- Generación automática de código fuente en proyectos de escritorio Java sobre el IDE Netbeans.
- Agregados al código fuente para reflejar las relaciones.
- Compilación y ejecución de los proyectos de escritorio Java.

-Relación de generalización: se basa en los elementos comunes encontrados en dos o mas clases que permiten, reunidos ser generalizados hacia una clases superior. Con este concepto, al ser instanciada una clase derivada, se heredan propiedades y métodos de la clase superior. Las clases superiores pueden ser abstractas, con lo que podremos aprovechar el concepto de métodos polimórficos.

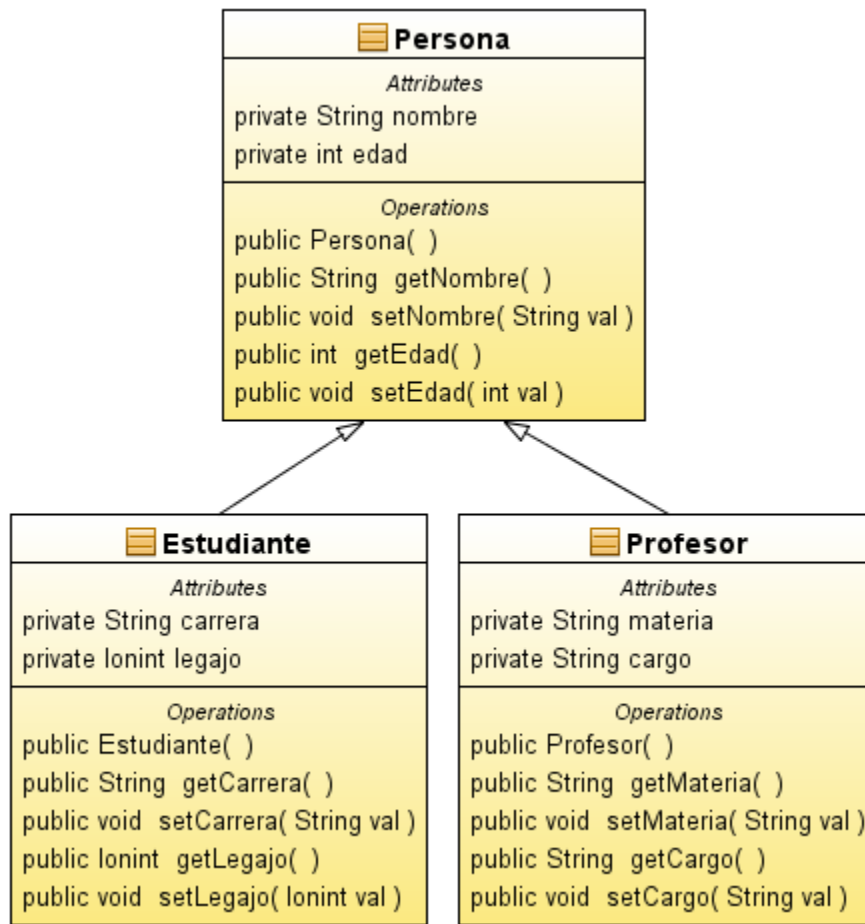
Ejemplo: proyecto UML Netbeans Generalizacion



Hemos creado un proyecto UML sobre el IDE Netbeans de nombre Generalizacion. El cual se presenta en el explorador de proyectos como se observa en el grafico. En la carpeta Model se ha creado un diagrama de clases llamado Personas que contiene como consecuencia de lo trabajado, las clases: Persona, Estudiante y Profesor y los tipos de datos involucrados por las propiedades y retornos de métodos int, String y void.



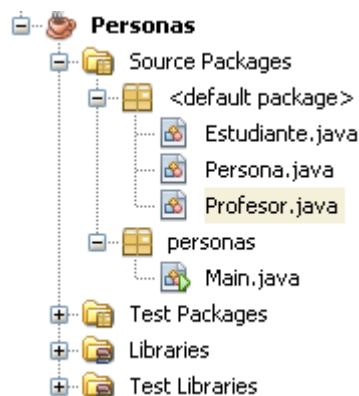
Símbolo que representa la generalización en el lenguaje UML. Persona generaliza a Estudiante y Profesor.

Detalle del diagrama Personas del proyecto Generalizacion

Observamos que se generalizó hacia la clase Persona ya que las clases Estudiante y Profesor poseen propiedades comunes como lo son nombre y edad. Por lo que Persona es la clase base y Estudiante y Profesor son clases derivadas. Que significa esto?. Que si instanciamos a las clases derivadas ellas heredaran de la clase superior. Entonces que tipos de objetos podemos tener a partir de este modelo? Objetos de tipo Estudiante y Profesor que heredan de Persona y objetos de tipo Persona que pueden ser visitas en nuestra facultad.

Código fuente Java generado automáticamente

Después de haber creado un proyecto de escritorio Java, que denominamos Personas e indicando al proyecto UML que genere el código fuente obtenemos en el proyecto Personas lo siguiente:



En el paquete <default package>, que se genera automáticamente, nos crea el código de las clases involucradas en el diagrama Personas. Un archivo .java por cada clase.

Detalle del código generado automáticamente

Veremos las líneas de código de cada una de las clases Java generadas en archivos .java con algunos comentarios de líneas agregados por nosotros, que los identificará por la doble barra (//).

Persona.java (contiene)

```
public class Persona {
    private String nombre; // propiedad
    private int edad; // propiedad

    public Persona () { // constructor
    }
    public int getEdad () { // asesor a edad
        return edad;
    }
    public void setEdad (int val) { // mutador de edad
        this.edad = val;
    }
    public String getNombre () { // asesor a nombre
        return nombre;
    }
    public void setNombre (String val) { // mutador de nombre
        this.nombre = val;
    }
}
```

Estudiante.java (contiene)

```
public class Estudiante extends Persona {

    private String carrera; // propiedad
    private int legajo; // propiedad

    public Estudiante () { // constructor
    }
    public String getCarrera () { // asesor a carrera
        return carrera;
    }
    public void setCarrera (String val) { // mutador de carrera
        this.carrera = val;
    }
    public int getLegajo () { // asesor a legajo
        return legajo;
    }
    public void setLegajo (int val) { // mutador de legajo
        this.legajo = val;
    }
}
```


Profesor.java (contiene)

```
public class Profesor extends Persona {

    private String materia; // propiedad
    private String cargo; // propiedad

    public Profesor () { // constructor
    }

    public String getCargo () { // asesor de cargo
        return cargo;
    }

    public void setCargo (String val) { // mutador de cargo
        this.cargo = val;
    }

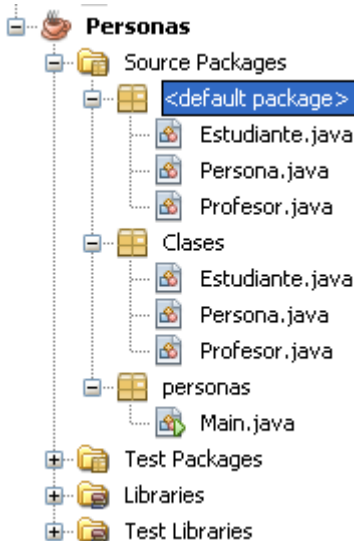
    public String getMateria () { // asesor de materia
        return materia;
    }

    public void setMateria (String val) { // mutador de materia
        this.materia = val;
    }

}
```

Observamos que las clases Estudiante y Profesor en sus prototipos de clases, a partir de la palabra reservada extends, indican que heredan o bien son extensiones de la clase Persona.

Que hacer para que nuestro proyecto Personas genere objetos de las tres clases, mute sus propiedades y las muestre por la consola de salida?



- 1) Debemos informarles al archivo Main.java donde están las Clases Persona, Estudiante y Profesor importándolas. Para poder importar los archivos .java hacia Main.java creamos el paquete Clases y copiamos los archivos .java. desde <default package>.
- 2) En Main.java agregamos las líneas que importan las clases necesarias.

```
import Clases.Persona;
import Clases.Estudiante;
import Clases.Profesor;
```

Con estas directivas estaremos preparados para crear objetos mutar sus propiedades y mostrarlas en el método main() de la clase Main.

El archivo Main.java quedaría con las siguientes líneas de código Java:

```

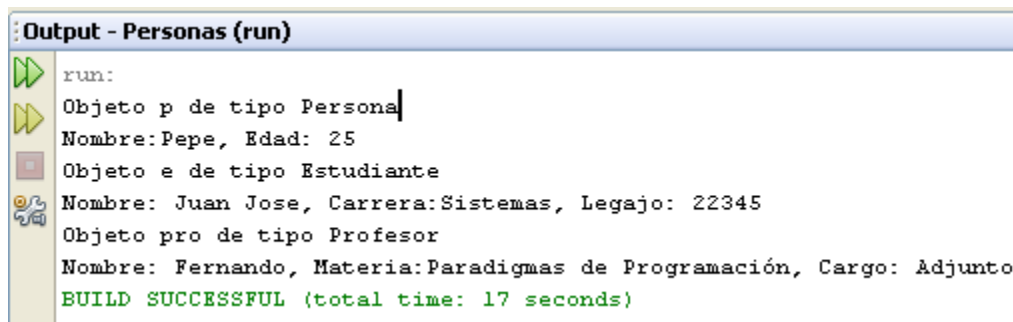
package personas;
import Clases.*;

public class Main {

    public static void main(String[] args) {
        Persona p = new Persona();
        p.setNombre("Pepe");
        p.setEdad(25);
        Estudiante e = new Estudiante();
        e.setNombre("Juan Jose");
        e.setLegajo(22345);
        e.setCarrera("Sistemas");
        Profesor pro = new Profesor();
        pro.setNombre("Fernando");
        pro.setMateria("Paradigmas de Programación");
        pro.setCargo("Adjunto");
        System.out.println("Objeto p de tipo Persona");
        System.out.printf("Nombre:%s, Edad: %d \n", p.getNombre(), p.getEdad());
        System.out.println("Objeto e de tipo Estudiante");
        System.out.printf("Nombre: %s, Carrera:%s, Legajo: %d \n", e.getNombre(),
            e.getCarrera(), e.getLegajo());
        System.out.println("Objeto pro de tipo Profesor");
        System.out.printf("Nombre: %s, Materia:%s, Cargo: %s \n", pro.getNombre(),
            pro.getMateria(), pro.getCargo());
    }
}

```

Si compilamos y ejecutamos el proyecto Personas, observaremos la siguiente salida por la consola estándar.



```

run:
Objeto p de tipo Persona
Nombre:Pepe, Edad: 25
Objeto e de tipo Estudiante
Nombre: Juan Jose, Carrera:Sistemas, Legajo: 22345
Objeto pro de tipo Profesor
Nombre: Fernando, Materia:Paradigmas de Programación, Cargo: Adjunto
BUILD SUCCESSFUL (total time: 17 seconds)

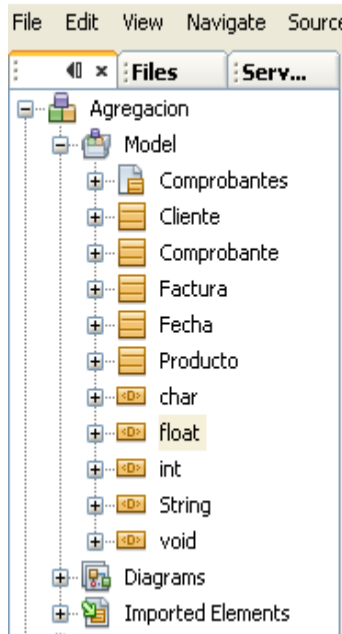
```

Esta relación de generalización ejemplificada, esta diseñada de modo tal que cuando queremos identificar a una persona que no es estudiante ni profesor, nos permite instanciar la clase Persona y trabajar con dicho objeto. Puede ocurrir que generalicemos solo métodos comunes a dos o mas clases con lo que tendríamos una superclase abstracta. En este caso no se podrá instanciar la superclase sino a través de una clase derivada.

-Relación de asociación: es una relación estructural que describe una conexión entre objetos. Dos o más clases pueden estar asociadas de diferentes modos:

-Relación de asociación agregación: si una clase posee una propiedad de otra clase y al ser instanciada recibe una copia de dicho objeto como parámetro, decimos que lo agrega a la clase. Con esto podemos expresar que el objeto agregado persiste si se encuentra el fin de ámbito del objeto que lo agrego.

Ejemplo: proyecto UML Netbeans Agregacion



Se puede observar el proyecto UML denominado Agregacion en el explorador de proyectos del IDE Netbeans. Dicho proyecto tiene como finalidad modelar el subsistema que permite la conformación del comprobante tipo factura de venta de productos.

Toda factura es un comprobante de venta, que debe poseer una fecha, el tipo de comprobante, un número, datos del cliente, los productos involucrados y un importe total.

En base al precio de los n productos que posee la factura se calcula el total.

Diagrama de clases Comprobantes

Se observa en el diagrama que Comprobantes generaliza a Factura. Todo comprobante posee un tipo "F" para el caso de las facturas y por ejemplo "R" para los remitos, un número correlativo y una fecha de confección. Las fechas las trataremos como objetos, de modo que podemos agregar la fecha del día a todo comprobante que se confeccione con la relación de agregación sin navegabilidad. El cliente y los productos los agregamos con una relación de gregación

con navegabilidad y una determinada multiplicidad; observe que no se visualizan las propiedades cliente y productos en la clase Factura.



Símbolo que representa la relación de agregación.



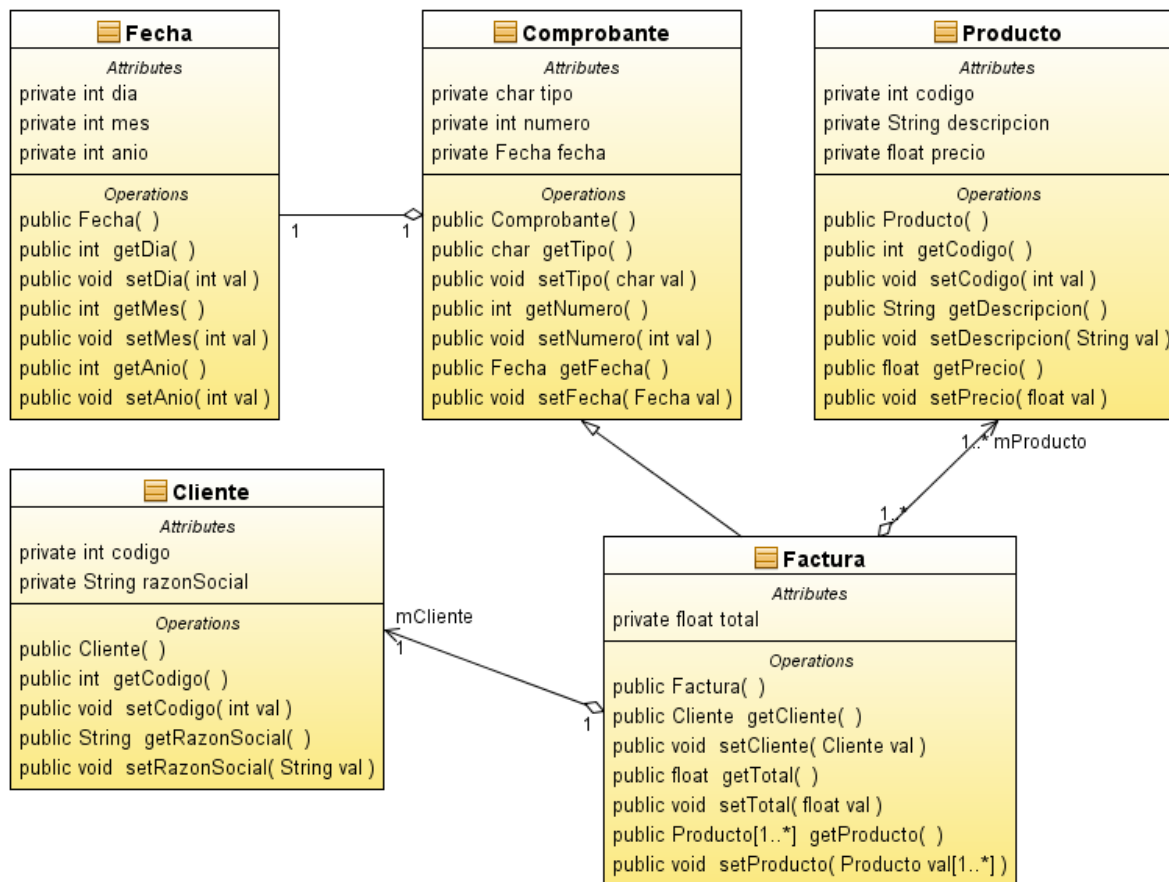
Símbolo que representa la relación de agregación con navegabilidad.

La diferencia entre agregación y agregación con navegabilidad se pone de manifiesto ya que si tiene navegabilidad no hace falta indicar la propiedad agregada en la clase contenedora y en el otro caso si se debe definir la propiedad.

Observe que en las relaciones de agregación entre Factura – Cliente y Factura – Producto nos coloca UML los identificadores mCliente y mProducto. Los cuales serán propiedades de la clase Factura al ser generado el código fuente correspondiente.

En cuando a la multiplicidad, podemos expresar que para la relación Factura – Cliente es 1 a 1, para una factura un cliente. Para la relación Factura – Producto es 1 a 1..*, en una factura pueden estar involucrados por lo menos un producto o bien n.

Detalle del diagrama Comprobantes del proyecto Agregacion



Veamos el código fuente Java que se genera automáticamente y lo trabajemos, de modo tal de hacer funcionar nuestro proyecto de escritorio Java.

Fecha.java (código fuente)

```

public class Fecha {

    private int dia;
    private int mes;
    private int anio;

    public Fecha () {
    }
    public int getAnio () {
        return anio;
    }
    public void setAnio (int val) {
        this.anio = val;
    }
    public int getDia () {
        return dia;
    }
    public void setDia (int val) {
        this.dia = val;
    }
    public int getMes () {
        return mes;
    }
    public void setMes (int val) {
        this.mes = val;
    }
}

```

Cliente.java (código fuente)

```

public class Cliente {

    private int codigo;
    private String razonSocial;

    public Cliente () {
    }
    public int getCodigo () {
        return codigo;
    }
    public void setCodigo (int val) {
        this.codigo = val;
    }
    public String getRazonSocial () {
        return razonSocial;
    }
    public void setRazonSocial (String val) {
        this.razonSocial = val;
    }
}

```

Comprobante.java

```

public class Comprobante {

    private char tipo;
    private int numero;
    private Fecha fecha;

    public Comprobante () {
    }
    public Fecha getFecha () {
        return fecha;
    }
    public void setFecha (Fecha val) {
        this.fecha = val;
    }
    public int getNumero () {
        return numero;
    }
    public void setNumero (int val) {
        this.numero = val;
    }
    public char getTipo () {
        return tipo;
    }
    public void setTipo (char val) {
        this.tipo = val;
    }
}

```

Producto.java

```

public class Producto {

    private int codigo;
    private String descripcion;
    private float precio;

    public Producto () {
    }
    public int getCodigo () {
        return codigo;
    }
    public void setCodigo (int val) {
        this.codigo = val;
    }
    public String getDescripcion () {
        return descripcion;
    }
    public void setDescripcion (String val) {
        this.descripcion = val;
    }
    public float getPrecio () {
        return precio;
    }
    public void setPrecio (float val) {
        this.precio = val;
    }
}

```

Factura.java

```

import java.util.ArrayList;

public class Factura extends Comprobante {

    private ArrayList<Producto> mProducto;
    private float total;
    private Cliente mCliente;

    public Factura () {
    }
    public Cliente getCliente () {
        return mCliente;
    }
    public void setCliente (Cliente val) {
        this.mCliente = val;
    }
    public float getTotal () {
        return total;
    }
    public void setTotal (float val) {
        this.total = val;
    }
    public ArrayList<Producto> getProducto () {
        return mProducto;
    }
    public void setProducto (ArrayList<Producto> val) {
        this.mProducto = val;
    }
}

```

Sobre el código fuente generado para la clase Factura observamos:

- La propiedad `ArrayList<Producto> mProducto`. Esto se debe a la multiplicidad 1..* en el extremo de la relación de agregación con la clase `Producto`. Con lo que por cada objeto `Factura` tendremos la posibilidad de agregar n productos.
- La propiedad `mCliente`. Debido a la multiplicidad 1 en el extremo de la relación de agregación con la clase `Cliente`. Por lo que para cada objeto `Factura` podremos agregar un objeto `Cliente`.

En las demás clases no observamos detalles nuevos. Lo destacable es que el generador automático de código fuente, no nos refleja las relaciones de agregación, por lo que tenemos que hacer las modificaciones adecuadas para que esto suceda en nuestro programa.

Pasos a seguir:

- Adecuar los constructores de todas las clases para poder instanciarlas desde el método `main()` y pasar los parámetros necesarios para sus propiedades.
- Escribir métodos en la clase `Factura` para:
 - Incorporar productos a la factura y recalcular el total.
 - Mostrar los datos de la factura
 - Mostrar los productos de la factura.
- Escribir el método `main()`.

Constructor de la clase `Fecha`

```
public Fecha(int d, int m, int a){
    setDia(d);
    setMes(m);
    setAnio(a);
}
```

Constructor de la clase `Cliente`

```
public Cliente(int c, String r){
    setCodigo(c);
    setRazonSocial(r);
}
```

Constructor de la clase `Producto`

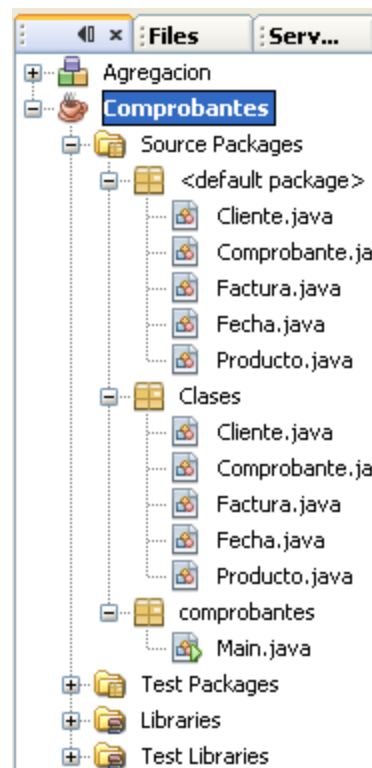
```
public Producto(int c, String d, float p){
    setCodigo(c);
    setDescripcion(d);
    setPrecio(p);
}
```

Constructor de la clase `Comprobante`

```
public Comprobante(int t, int n, Fecha f){
    setTipo(t);
    setNumero(n);
    setFecha(f);
}
```

Constructor de la clase `Factura`

```
Public Factura(int t, int n, Fecha f, Cliente cli){
    super(t,n,f);
    setCliente(cli);
}
```



Métodos de la clase Factura

```
public void agregarProducto(Producto p){
    mProducto.add(p);
    setTotal(getTotal() + p.getPrecio());
}

public void mostrarProductos(){
    Iterator<Producto> iter = mProducto.iterator();
    while (iter.hasNext()) {
        Producto p = iter.next();
        System.out.printf("Codigo: %d Descripcion: %s Precio: %5.2f \n",
            p.getCodigo(), p.getDescripcion(), p.getPrecio());
    }
}

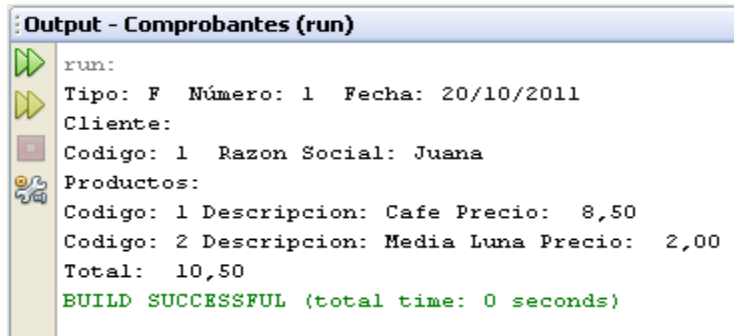
public void mostrar(){
    System.out.printf("Tipo: %c Número: %d Fecha: %d/%d/%d\n",
        getTipo(), getNumero(),
        getFecha().getDia(), getFecha().getMes(), getFecha().getAnio());
    System.out.printf("Cliente: \n");
    System.out.printf("Codigo: %d Razon Social: %s \n",
        mCliente.getCodigo(), mCliente.getRazonSocial());
    System.out.printf("Productos: \n");
    mostrarProductos();
    System.out.printf("Total: %6.2f \n",getTotal());
}
```

Código del método main() de la clase Main

```
package comprobantes;
import Clases.Fecha;
import Clases.Producto;
import Clases.Cliente;
import Clases.Factura;

public class Main {

    public static void main(String[] args) {
        Fecha hoy = new Fecha(20,10,2011);
        Producto pro1 = new Producto(1, "Cafe", (float) 8.5);
        Producto pro2 = new Producto(2, "Media Luna", 2);
        Cliente cliente = new Cliente(1, "Juana");
        Factura f1 = new Factura('F', 1, hoy, cliente);
        f1.agregarProducto(pro1);
        f1.agregarProducto(pro2);
        f1.mostrar();
    }
}
```

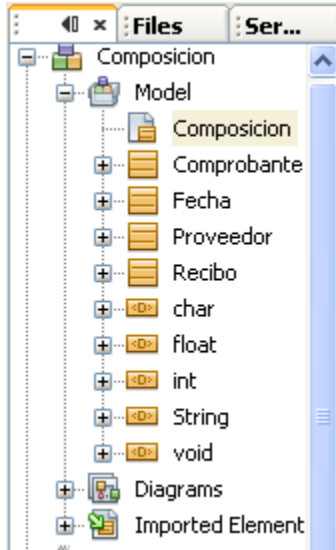


```
Output - Comprobantes (run)
run:
Tipo: F Número: 1 Fecha: 20/10/2011
Cliente:
Codigo: 1 Razon Social: Juana
Productos:
Codigo: 1 Descripcion: Cafe Precio: 8,50
Codigo: 2 Descripcion: Media Luna Precio: 2,00
Total: 10,50
BUILD SUCCESSFUL (total time: 0 seconds)
```

Ahora si se reflejan en el código fuente las relaciones de agregación. Lo podemos observar, siempre que instanciamos una clase y le pasamos algún parámetro que es un objeto de otra clase, en el método main() de la clase Main.

-Relación de asociación composición: si una clase posee una propiedad de otra clase y se instancia la clase de dicha propiedad en algún método de la clase, se dice que dicho objeto es parte del objeto contenedor. Por lo que al encontrar fin de ámbito el objeto contenedor, deja de persistir el objeto contenido.

Ejemplo: proyecto UML Netbeans Composicion



Este ejemplo pretende modelar el subsistema que va a confeccionar recibos a proveedores de servicios.

Un recibo se identifica por su tipo de comprobante, el número, la fecha, datos del proveedor, un detalle del servicio y el importe total del mismo.

Nuestra idea, de cómo confeccionar el recibo, es crear el objeto Recibo y dentro de él crear los objetos Fecha y Proveedor de modo tal que lo compongan.

Diagrama de clases Composicion

La clase Comprobante se compone por la clase Fecha y generaliza a la clase Recibo, quien se compone por la clase Proveedor.

El tipo de comprobante debe ser "R", su número correlativo y la fecha del día. El recibo contiene datos del proveedor, el importe y el detalle del pago a nuestro proveedor de servicios.

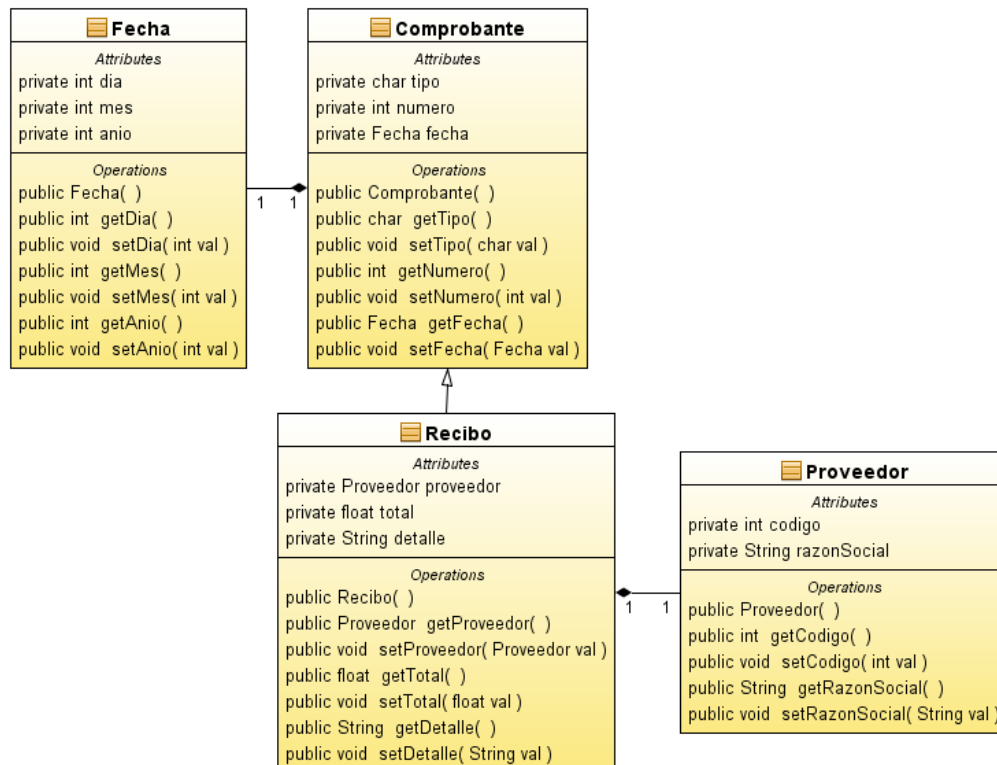


Símbolo que representa la relación de composición.



Símbolo que representa la relación de composición con navegabilidad.

Detalle del diagrama de clases Composicion del proyecto UML



Las relaciones de composición entre las clases Recibo – Proveedor y Comprobante – Fecha tienen una multiplicidad 1 a 1. Al ser instanciada la clase Recibo, esta instanciará la superclase Comprobante que a su vez instanciará a la clase Fecha. Luego la clase Recibo debe instanciar a la clase Proveedor. Esa sería nuestra secuencia de instanciaciones para que se cumplan las composiciones diagramadas.

Al no ser dirigidas las relaciones de composición, debemos crear los atributos de tipo Proveedor en Recibo y de tipo Fecha en Comprobante.

Desde luego, en el código que genere automáticamente Netbeans en base al proyecto UML, no veremos reflejadas las relaciones de composición. Las tenemos que construir.

Código fuente Java generado automáticamente

Fecha.java

```
public class Fecha {

    private int dia;
    private int mes;
    private int anio;

    public Fecha () {
    }
    public int getAnio () {
        return anio;
    }
    public void setAnio (int val) {
        this.anio = val;
    }
    public int getDia () {
        return dia;
    }
    public void setDia (int val) {
        this.dia = val;
    }
    public int getMes () {
        return mes;
    }
    public void setMes (int val) {
        this.mes = val;
    }
}
```

Comprobante.java

```
public class Comprobante {

    private char tipo;
    private int numero;
    private Fecha fecha;

    public Comprobante () {
    }
    public Fecha getFecha () {
        return fecha;
    }
    public void setFecha (Fecha val) {
        this.fecha = val;
    }
    public int getNumero () {
        return numero;
    }
    public void setNumero (int val) {
        this.numero = val;
    }
    public char getTipo () {
        return tipo;
    }
    public void setTipo (char val) {
        this.tipo = val;
    }
}
```

Recibo.java

```

public class Recibo extends Comprobante {

    private Proveedor proveedor;
    private float total;
    private String detalle;

    public Recibo () {
    }
    public String getDetalle () {
        return detalle;
    }
    public void setDetalle (String val) {
        this.detalle = val;
    }
    public Proveedor getProveedor () {
        return proveedor;
    }
    public void setProveedor (Proveedor val) {
        this.proveedor = val;
    }
    public float getTotal () {
        return total;
    }
    public void setTotal (float val) {
        this.total = val;
    }
}

```

Proveedor.java

```

public class Proveedor {

    private int codigo;
    private String razonSocial;

    public Proveedor () {
    }
    public int getCodigo () {
        return codigo;
    }
    public void setCodigo (int val) {
        this.codigo = val;
    }
    public String getRazonSocial () {
        return razonSocial;
    }
    public void setRazonSocial (String val) {
        this.razonSocial = val;
    }
}

```

Como observamos el código refleja la generalización de Recibo hacia Comprobante y ninguna otra relación de las establecidas en el diagrama de clases.

Trabajaremos el código sobre la base del siguiente método main() de la clase Main.

```

package recibos;

import Fuentes.*;

public class Main {
    public static void main(String[] args) {
        Recibo recibo = new Recibo(27,10,2011,"Limpituc SA",2023);
        recibo.setTipo('R');
        recibo.setNumero(1);
        recibo.setDetalle("Pago de servicio jardineria");
        recibo.setTotal(350);
        recibo.mostrar();
    }
}

```

Al ser instanciada la clase Recibo

```
Recibo recibo = new Recibo(27,10,2011,"Limpituc SA",2023);
```

Le pasamos como parámetros a su constructor, tres enteros 27, 10, 2011 para que al invocar al constructor de la superclase Comprobante le pase dichos parámetros y luego al ser instanciada la clase Fecha, desde el constructor de la clase Comprobante, también se pasen dichos parámetros a fin de inicializar las propiedades del objeto fecha compuesto en la clase Comprobante. El String "Limpituc SA" y el entero 2023 son para inicializar las propiedades del objeto proveedor que se instancia en el constructor de la clase Recibo.

Para que esto se cumpla, debemos preparar los constructores adecuadamente.

Constructor de la clase Recibo

```
public Recibo (int dia,int mes,int anio, String razonSocial, int codigo) {
    super(dia, mes, anio); // ejecuta el constructor de la superclase
    proveedor = new Proveedor(razonSocial, codigo);
}
```

Constructor de la clase Comprobante

```
public Comprobante (int dia, int mes, int anio) {
    fecha = new Fecha(dia, mes, anio);
}
```

Constructor de la clase Fecha

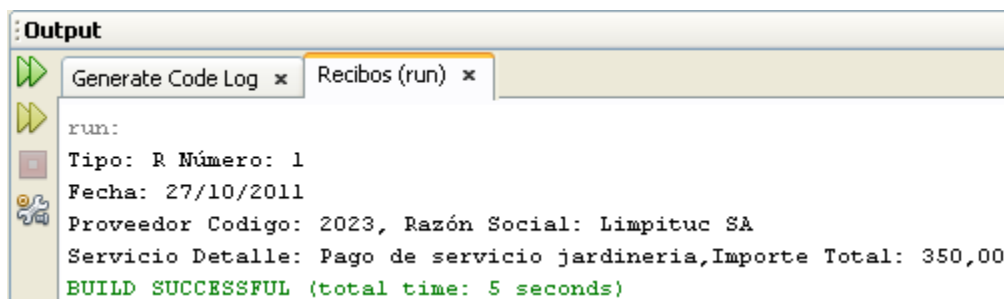
```
public Fecha (int dia, int mes, int anio) {
    setDia(dia);
    setMes(mes);
    setAnio(anio);
}
```

Constructor de la clase Proveedor

```
public Proveedor (String razonSocial, int codigo) {
    setRazonSocial(razonSocial);
    setCodigo(codigo);
}
```

El resto de líneas del método main() tienen la finalidad de inicializar las propiedades del objeto recibo y mostrar toda la información del recibo por la consola de salida.

Nuestro proyecto Recibos en ejecución, produce la siguiente salida:



```
Output
Generate Code Log x Recibos (run) x
run:
Tipo: R Número: 1
Fecha: 27/10/2011
ProveedorCodigo: 2023, Razón Social: Limpituc SA
Servicio Detalle: Pago de servicio jardineria,Importe Total: 350,00
BUILD SUCCESSFUL (total time: 5 seconds)
```

Es correcto pensar que al instanciar la clase Recibo, se pueden pasar todos los parámetros para que inicialicen todas las propiedades de los objetos involucrados. Del siguiente modo:

```
Recibo recibo = new Recibo('R', 1, 27, 10, 2011,  
                           "Limpituc SA", 2023, "Pago de servicio jardineria", 350 );
```

Con lo que debemos adecuar el constructor de la clase Recibo sobre los parámetros recibidos y agregar los set necesarios para inicializar propiedades. Sería bueno para ustedes hacer estas adaptaciones al proyecto.

Concluimos esta relación de composición, expresando que la base conceptual de la relación es que un objeto construye objetos en su interior. De modo que al finalizar su ámbito, se entrega al recolector de basura dicho objeto con todos los objetos que lo componen.

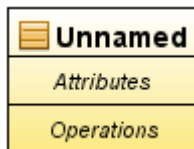
Conclusiones generales: es importante concluir que UML y sus diagramas de clases nos colaboran sobre la base de clases (objetos) intervinientes en nuestro sistema y sus relaciones. Desde luego que las relaciones de agregación y composición no se reflejan en el código generado a partir de los diagramas de clases, pero hemos aprendido a codificarlas adecuadamente. UML nos permite visualizar con claridad las clases y relaciones del modelo en tratamiento, esto no se logra observando solo código, o bien se torna muy engorroso.

Apéndice A

1- Componentes de un diagrama de clases

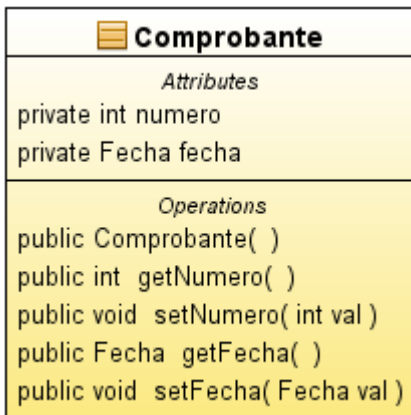
- a) Clase: representa un objeto o conjunto de estos indicando su nombre, propiedades y métodos.
- b) Relación: establecen el vínculo entre clases, indicando la navegabilidad y la multiplicidad.

2- Representación de clases en diagramas de clases, para proyectos UML sobre el IDE Netbeans 6.9.1



Clase: Unnamed representa el nombre o identificador de la clase.
 Attributes representa los atributos o propiedades.
 Operations representa los métodos o implementaciones.

Ejemplo: clase Comprobante



Clase de nombre Comprobante con los atributos número y fecha, el método constructor y los métodos asesores y mutadores de las propiedades.

3- Simbología que representa las relaciones entre clases en diagramas de clases, para proyectos UML sobre el IDE Netbeans 6.9.1

-Representación de relaciones entre clases.



Generalización. Nos representa que la clase superior generaliza a la clase derivada.



Implementación de interfaz o dependencia. Representa que una clase depende de otra.



Asociación. Representa que dos clases están asociadas entre si.



Asociación agregación. Representa que una clase agrega a otra.



Asociación composición. Representa que una clase esta compuesta por otra.



Asociación navegable.



Asociación agregación navegable.



Asociación composición navegable.



Asociación de clases.



Contenedor



Dependencia



Realización



Uso



Permiso



Abstracción



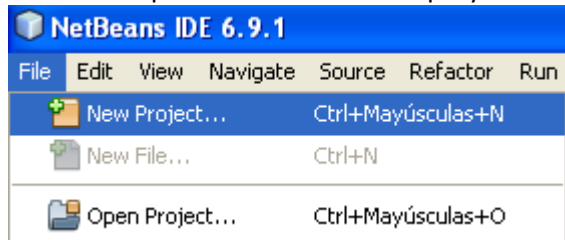
Comentario

4- Creación de proyectos UML en Netbeans 6.9.1

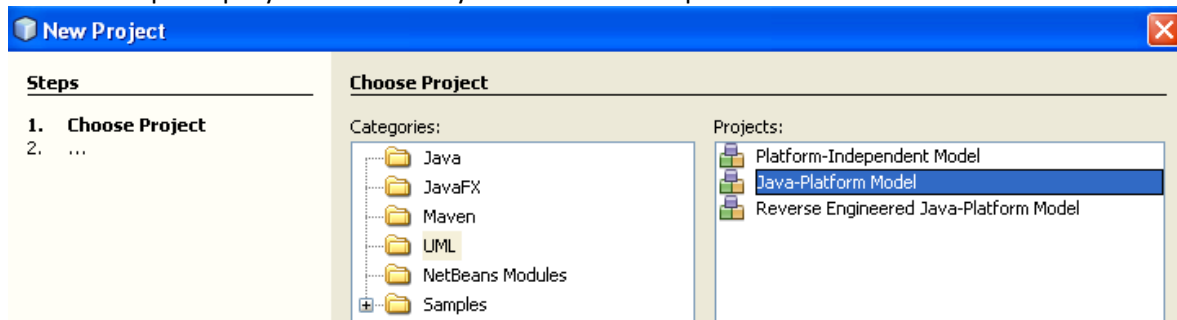
Para crear un proyecto UML en el IDE Netbeans debemos seleccionar de la opción del menú principal File (archivo), el ítem New Project (nuevo proyecto) para luego en el cuadro de dialogo que se visualice, seleccionar en Categories (categoría) a UML y en Projects (proyectos) Java Platform Model (Modelo de Plataforma Java) y seleccionamos el botón Next (siguiente). Aparecerá un cuadro de dialogo solicitándonos que especifiquemos el Name Project (nombre del proyecto) y la Location (localización en carpetas del disco) y seleccionamos el botón Finish (final). Aparecerá un cuadro de dialogo solicitándonos que indiquemos si queremos crear algún diagrama nuevo (New Diagram) y seleccionamos el botón Cancel (cancelar).

Paso a paso:

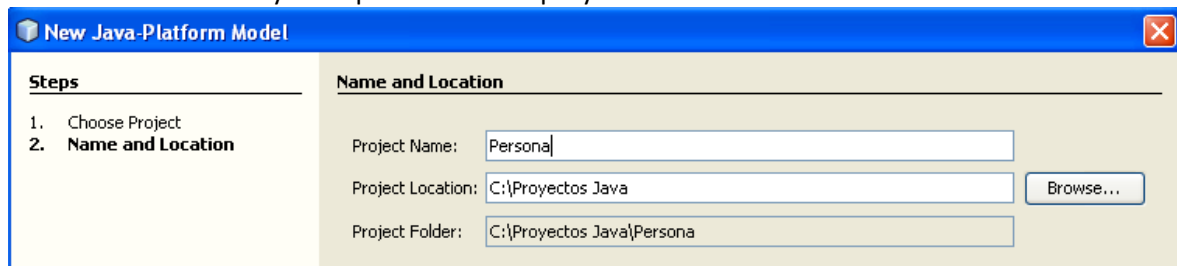
-Indicando que deseamos crear un proyecto nuevo.



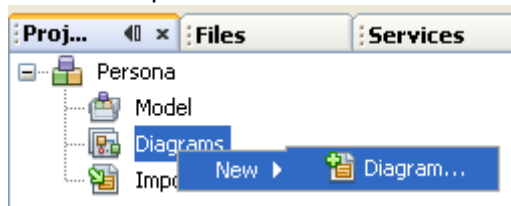
-Indicando que el proyecto será UML y un modelo sobre plataforma Java



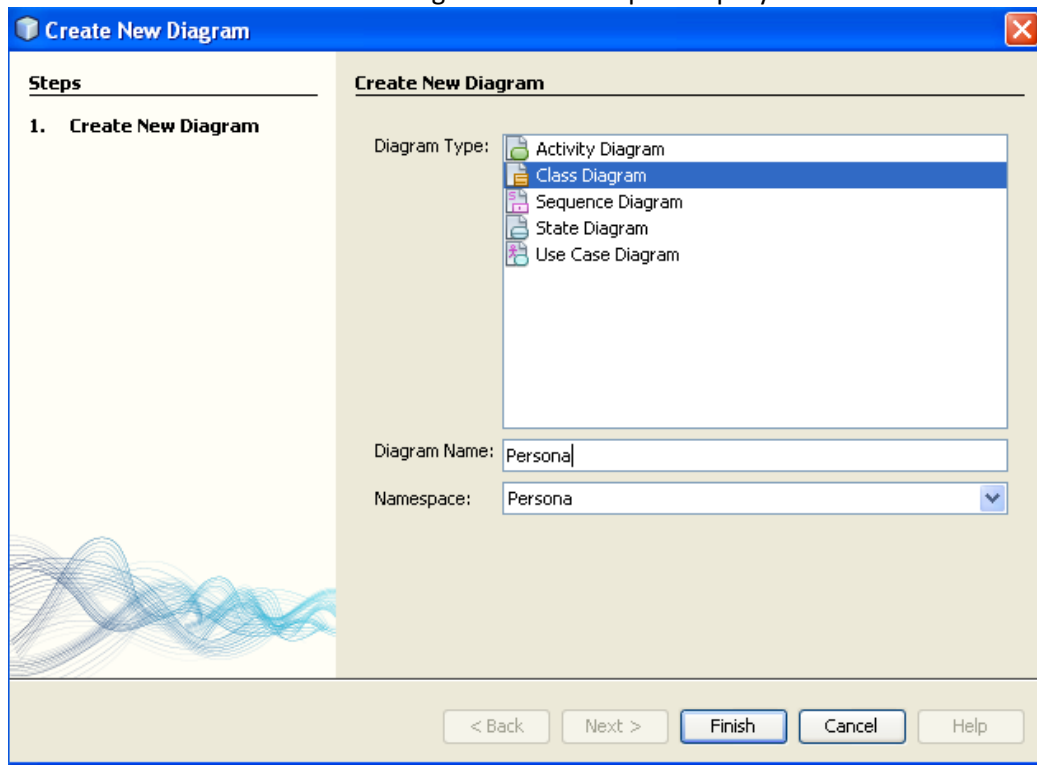
-Indicando el nombre y la carpeta del nuevo proyecto



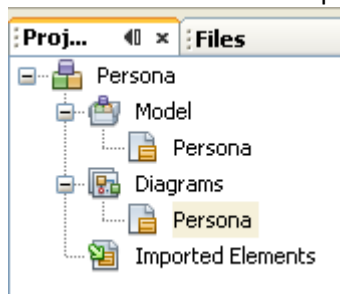
-Indicando que deseamos crear un nuevo diagrama para el proyecto UML



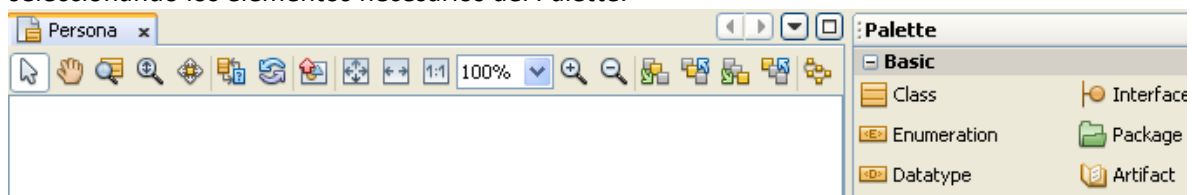
-Indicando el nombre del nuevo diagrama de clases para el proyecto



En el explorador de proyectos observamos que se a creado la base para construir el diagrama de clases Persona dentro del proyecto UML Persona.



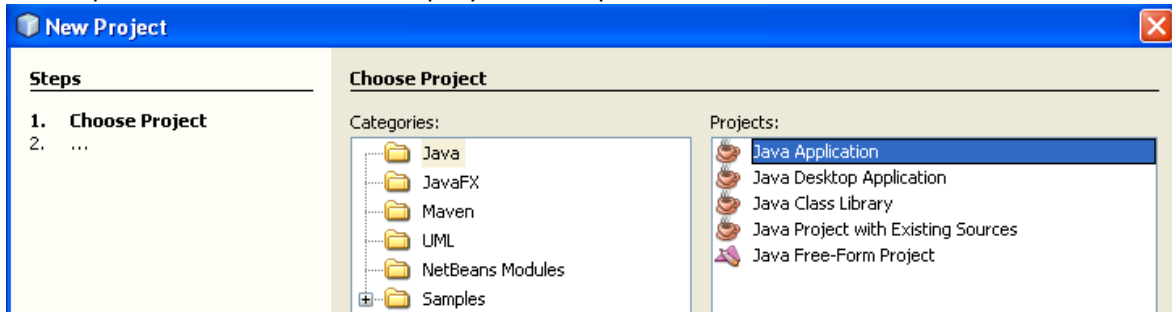
Y quedaríamos en la situación de poder comenzar con la construcción del diagrama de clases seleccionando los elementos necesarios del Palette.



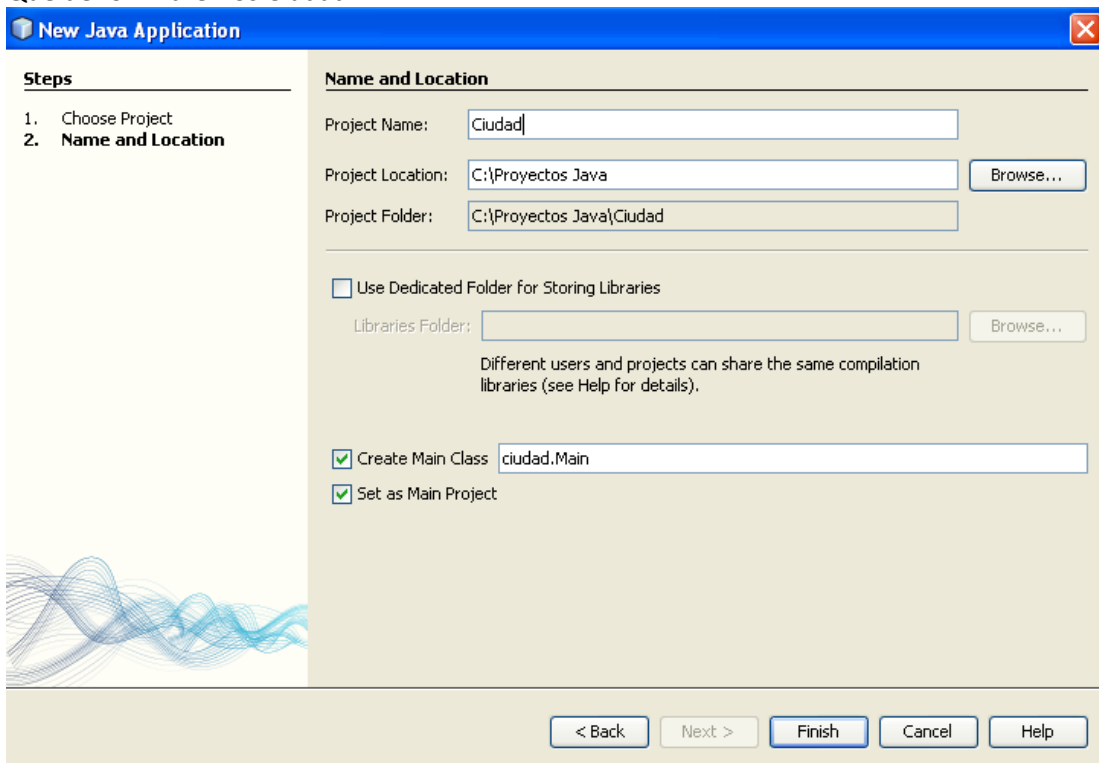
5- Creación de proyectos de aplicación Java en Netbeans 6.9.1

Para generar automáticamente el código Java correspondiente a un proyecto UML, Netbeans nos exige poseer un proyecto Java donde incorporarlo.

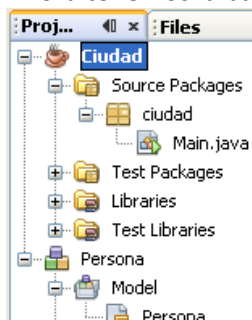
Por lo que vamos a crear un nuevo proyecto de aplicación Java.



Que denominaremos Ciudad



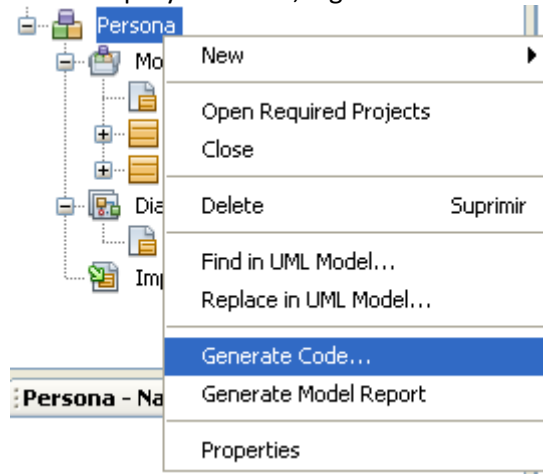
Ahora tenemos la base, para la construcción de un proyecto Java de aplicación



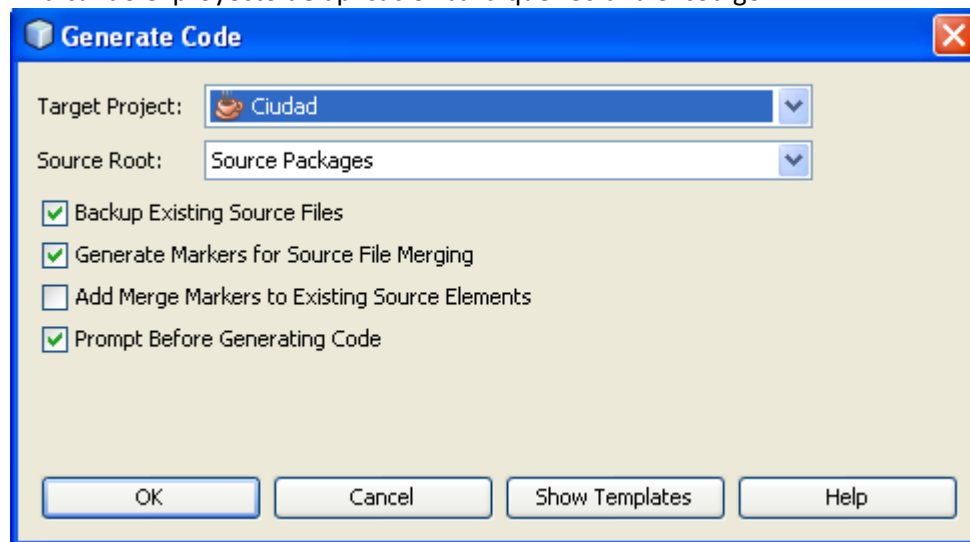
6- Generación del código Java

Una vez finalizado nuestro proyecto UML y sus diagramas de clases, podemos generar automáticamente el código Java correspondiente hacia algún proyecto de aplicación Java.

-Sobre el proyecto UML, digitamos botón derecho y seleccionamos la opción generate Code



-Indicando el proyecto de aplicación Java que recibirá el código.



Entonces en Source Package del proyecto Ciudad y dentro del paquete <default package>, encontraremos el código correspondiente a las clases que poseía el diagrama de clases Persona del proyecto UML Persona.