

PROGRAMACIÓN II



SEMANA 10 - 2

RELCIÓN ENTRE CLASES

Eric Gustavo Coronel Castillo
ecoronel@uch.edu.pe
gcoronelc.blogspot.com

Relación entre Clases

Todos los sistemas se construyen a partir de muchas clases y objetos cuya colaboración permite lograr el comportamiento deseado en el sistema.

La colaboración entre objetos imponen la existencia de relaciones entre ellos: dependencia, generalización y asociaciones.

Índice

| | | |
|----------|--------------------------------------------------------|-----------|
| 1 | INTRODUCCIÓN | 4 |
| 2 | DEPENDENCIA | 6 |
| 3 | GENERALIZACIÓN | 9 |
| 3.1 | DEFINICIÓN | 9 |
| 3.2 | SOBRE-ESCRITURA | 10 |
| 3.3 | CLASE ABSTRACTA Y CLASE CONCRETA..... | 11 |
| 3.4 | POLIMORFISMO | 12 |
| 3.5 | CLASES Y MÉTODOS FINALES | 13 |
| 3.6 | RESUMEN | 14 |
| 4 | ASOCIACIÓN | 18 |
| 4.1 | ASOCIACIÓN BINARIA..... | 18 |
| 4.1.1 | <i>Nombre de la asociación (association name).....</i> | <i>18</i> |
| 4.1.2 | <i>Nombre de rol (rolename).....</i> | <i>19</i> |
| 4.1.3 | <i>Multiplicidad (multiplicity)</i> | <i>19</i> |
| 4.1.4 | <i>Ordenación (ordering).....</i> | <i>21</i> |
| 4.1.5 | <i>Modificabilidad (change ability).....</i> | <i>21</i> |
| 4.1.6 | <i>Navegabilidad (navigability)</i> | <i>21</i> |
| 4.1.7 | <i>Visibilidad (visibility)</i> | <i>22</i> |
| 4.2 | AGREGACIÓN Y COMPOSICIÓN | 25 |
| 4.3 | ASOCIACIÓN N-ARIA | 27 |
| 5 | PROYECTOS RESUELTOS | 33 |
| 5.1 | EL BUEN SABOR | 33 |
| 5.1.1 | <i>Requerimiento de Software.....</i> | <i>33</i> |
| 5.1.2 | <i>Abstracción.....</i> | <i>34</i> |
| 5.1.3 | <i>Diagrama de Clases</i> | <i>34</i> |
| 5.1.4 | <i>Estructura del Proyecto en NetBeans</i> | <i>35</i> |
| 5.1.5 | <i>Codificación</i> | <i>35</i> |
| 5.1.6 | <i>Ejecución del Proyecto</i> | <i>39</i> |
| 5.2 | ELECTRONICA STORE..... | 40 |
| 5.2.1 | <i>Requerimiento de Software.....</i> | <i>40</i> |
| 5.2.2 | <i>Abstracción.....</i> | <i>40</i> |
| 5.2.3 | <i>Estructura del Proyecto en NetBeans</i> | <i>42</i> |
| 5.2.4 | <i>Codificación</i> | <i>43</i> |
| 5.2.5 | <i>Ejecución del Proyecto</i> | <i>49</i> |

1 Introducción

Las relaciones existentes entre las distintas clases de un sistema nos indican cómo se comunican los objetos de estas clases entre sí.

Los mensajes "navegan" por las relaciones existentes entre las distintas clases.

Existen 3 tipos de relaciones:

- Dependencia
- Generalización
- Asociación

Para hacer una representación gráfica de las clases se utilizan los diagramas de clases; donde, las dependencias se representan mediante una línea discontinua terminada en una punta de flecha, la generalización está representada por una línea continua terminada en un triángulo blanco, y la asociación es representada por una línea continua simple.

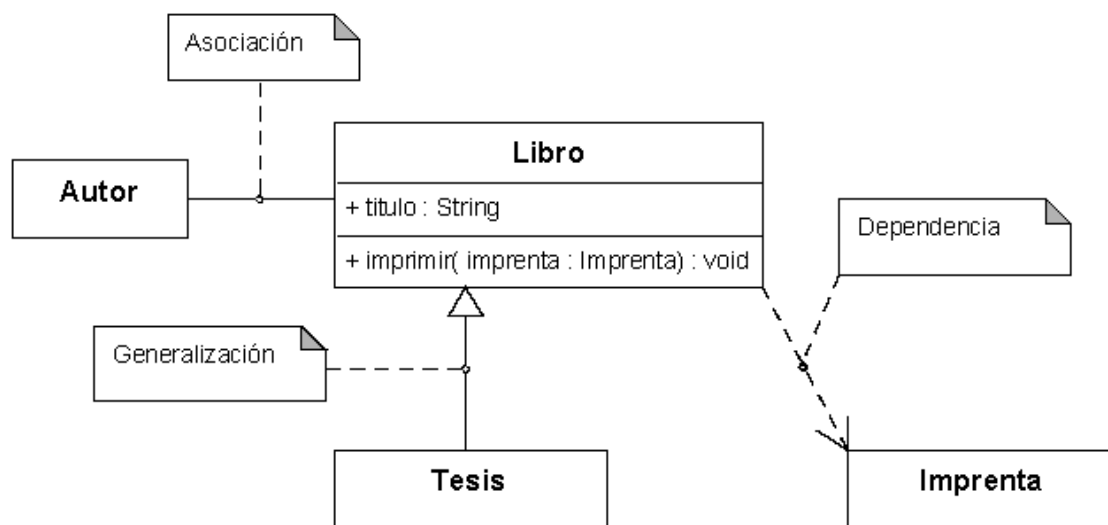


Figura 1 Diagrama de clases con sus tres relaciones básicas.

En la Figura 1 podemos ver un sencillo diagrama de clases con cuatro clases y tres relaciones. La clase **Libro** contiene el atributo `titulo` y la operación `imprimir()`, y tiene relaciones de asociación, generalización y dependencia con las clases **Autor**, **Tesis** e **Imprenta** respectivamente. La relación de asociación significa que las instancias de las clases **Autor** y **Libro** están relacionadas; la relación de generalización (o especialización, si se lee en sentido inverso) significa que el conjunto de instancias de la clase **Tesis** es un subconjunto del conjunto de instancias de la clase **Libro**; la relación de dependencia significa que la clase **Libro** depende de alguna manera de la clase **Imprenta**, por ejemplo,

por que la operación `imprimir()` requiere la especificación de una instancia de **Imprenta** como parámetro.

Un objeto representa una instancia particular de una clase, y tiene identidad y valores concretos para los atributos de su clase. La notación de los objetos se diferencia de la notación de las clases, mediante el empleo de subrayado. Un objeto se dibuja como un rectángulo con dos secciones. La primera sección contiene el nombre del objeto y de su clase, separados por el símbolo ":" y subrayados ambos, mientras que la segunda división, opcional, contiene la lista de atributos de la clase con los valores concretos que tiene en ese objeto.

Un enlace es una instancia de una asociación, del mismo modo que un objeto es una instancia de una clase. Un enlace se representa mediante una línea continua que une los objetos enlazados. **Un diagrama de objetos** es un grafo de instancias, es decir, objetos, valores de datos y enlaces. Un diagrama de objetos es una instancia de un diagrama de clases; muestra una especie de fotografía del estado del sistema en un sencillo instante de tiempo. En la Figura 2 podemos ver un sencillo ejemplo de un diagrama de objetos correspondiente al diagrama de clases de la Figura 1.

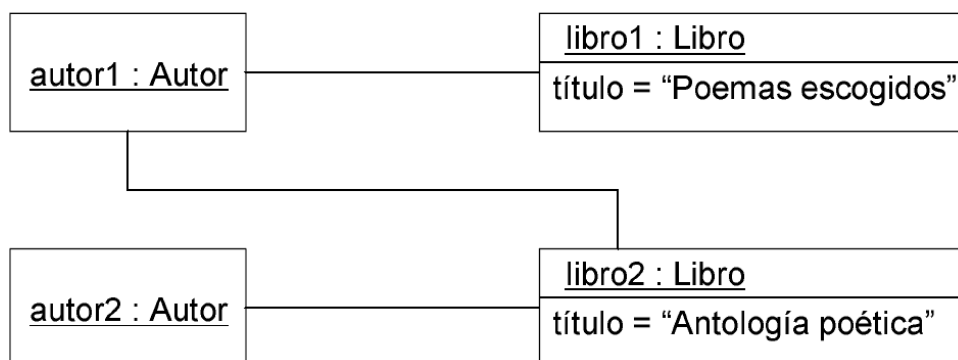


Figura 2 Diagrama de Objetos que muestra objetos, valores y enlaces.

En este diagrama podemos ver dos instancias de la clase **Autor**, dos instancias de la clase **Libro**, y tres instancias (es decir, enlaces) de la asociación **Autor-Libro**; estos enlaces significan que **autor1** es autor de dos libros, mientras que **autor2** sólo es autor de un libro; a su vez, **libro1** tiene un autor y **libro2** tiene dos autores.

Después de esta introducción pasaremos a ver cada uno de los tipos de relación.

2 Dependencia

Es una relación de uso, es decir una clase (dependiente) usa a otra clase (independiente) para ejecutar algún proceso. Se representa con una flecha discontinua que va desde la clase dependiente a la clase independiente.

Con la dependencia mostramos que un cambio en la clase independiente puede afectar al funcionamiento de la clase dependiente, pero no al contrario.

Ejemplo 1

En este ejemplo haremos el diseño de las clases para resolver una ecuación de segundo grado:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

En la Figura 3 se muestra la relación de dependencia entre las clases **Ecuación** y **Math**.

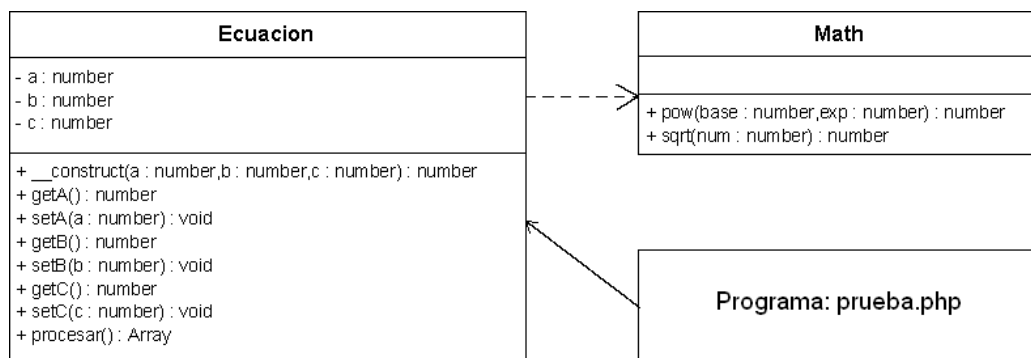


Figura 3 Relación de Dependencia.

La clase **Ecuacion** depende de la clase **Math** porque necesita de los métodos `pow` y `sqrt` para realizar el cálculo de las raíces de la ecuación, por lo tanto la clase **Math** es la clase independiente y la clase **Ecuacion** es la clase dependiente. También podemos afirmar que existe una dependencia entre el programa **prueba** y la clase **Ecuacion**, para este caso el programa **prueba** depende de la clase **Ecuación**.

La codificación de la clase **Math** es la siguiente:

Archivo: Math.php

```
<?php
class Math {

    public static function pow( $base, $exp ) {
        $rpta = pow($base, $exp);
        return $rpta;
    }

    public static function sqrt( $num ) {
        $rpta = sqrt( $num );
        return $rpta;
    }
}
?>
```

La codificación de la clase **Ecuacion** es el siguiente:

Archivo: Ecuacion.php

```
<?php
require_once 'Math.php';

class Ecuacion {

    private $a;
    private $b;
    private $c;

    function __construct($a = 0, $b = 0, $c = 0) {
        $this->a = $a;
        $this->b = $b;
        $this->c = $c;
    }

    public function getA() {
        return $this->a;
    }

    public function setA($a) {
        $this->a = $a;
    }
}
```

```
}

public function getB() {
    return $this->b;
}

public function setB($b) {
    $this->b = $b;
}

public function getC() {
    return $this->c;
}

public function setC($c) {
    $this->c = $c;
}

public function resolver() {
    $disc = Math::pow($this->getB(), 2) - 4 * $this->getA() * $this->getC();
    $temp = Math::sqrt($disc);
    $raiz[] = (-$this->getB() + $temp) / ( 2 * $this->getA() );
    $raiz[] = (-$this->getB() - $temp) / ( 2 * $this->getA() );
    return $raiz;
}

}

?>
```

La codificación del programa [prueba.php](#) es el siguiente:

Archivo: prueba.php

```
<?php

require_once 'Ecuacion.php';

$ecua = new Ecuacion(1,-4,3);
$raiz = $ecua->resolver();

echo("Raiz 1: {$raiz[0]}<br>");
echo("Raiz 2: {$raiz[1]}<br>");

?>
```

Si ejecutamos el programa **prueba.php** obtenemos el resultado que se muestra a continuación:

A : 1
B : -4
C : 3
Raiz 1: 3
Raiz 2: 1

3 Generalización

Muchos autores prefieren denominarla como **Generalización/Especialización**, porque se refieren a dos técnicas que nos llevan a obtener el mismo resultado, la herencia.

3.1 Definición

La **Generalización** consiste en factorizar las propiedades comunes de un conjunto de clases (clases hijas) en una clase más general (clase padre).

La **Especialización** es el proceso inverso a la **Generalización**; a partir de una clase denominada superclase se crean subclases que representan refinamientos de la superclase. Se añaden subclases para especializar el comportamiento de clases ya existentes.

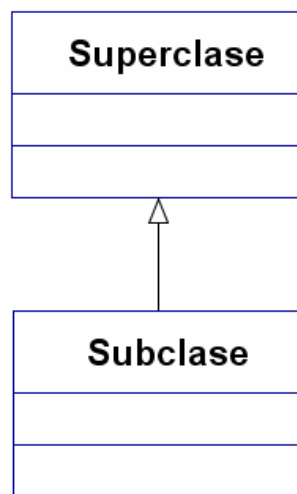


Figura 4 Notación UML para representar la herencia.

En la Figura 4 tenemos el diagrama UML para representar la herencia. El mecanismo usado para implementar la generalización ó especialización es la herencia, y los nombres usados para nombrar las clases son: superclase - subclase, clase padre - clase hija, clase base - clase derivada.

Las clases hijas heredan atributos y operaciones que están disponibles en sus clases padres.

En general, la herencia (Generalización o Especialización) es una técnica muy eficaz para la extensión y reutilización de código.

La implementación se realiza utilizando la palabra clave `extends`, a continuación tenemos el script de la implementación del grafico que se muestra en la Figura 4.

```
class Superclase {  
    // Implementación  
}  
  
class Subclase extends Superclase {  
    // Implementación  
}
```

Debemos tener en cuenta que los constructores de la clase padre no son llamados implícitamente si la clase hija define un constructor. Para poder ejecutar el constructor de la clase padre, se debe hacer una llamada a `parent::__construct()` dentro del constructor de la clase hija.

Al igual que los constructores, los destructores de la clase padre no serán llamados explícitamente por el compilador. Para ejecutar un destructor de la clase padre, se debe tener una llamada explícita a `parent::__destruct()` en el cuerpo del destructor.

Intentar generar una excepción desde un destructor la finalización del script produce un error fatal.

3.2 Sobre-Escritura

Consiste en redefinir un método de la clase padre en la clase hija con la misma firma, las razones pueden ser:

- Mejorar el algoritmo.
- Cambios en las reglas de negocio.
- etc.

Si por alguna razón queremos acceder a la implementación realizada en la clase padre debemos utilizar la palabra clave `parent` de la siguiente manera: `parent::nombreMétodo()`.

3.3 Clase Abstracta y Clase Concreta

Cuando pensamos en una clase como un tipo, asumimos que los programas crearán objetos de ese tipo. Sin embargo, hay casos en que es útil definir clases para las cuales no se desea instanciar objetos. Tales clases son llamadas **clases abstractas**. Debido a que normalmente son utilizadas como base en jerarquías de clases, nos referiremos a ellas como **clases bases abstractas**. Las clases abstractas no sirven para instanciar objetos porque están incompletas, siendo sus clases derivadas las que deberán definir las partes faltantes.

El propósito de una clase abstracta es proveer una clase base apropiada desde la cual otras clases hereden.

Una clase abstracta pueden tener implementación parcial, esto quiere decir que puede tener métodos abstractos y métodos concretos.

Las clases desde las cuales se pueden instanciar objetos se llaman **clases concretas**. Tales clases proveen implementaciones de cada método que definen y los métodos abstractos que hereda.

Las clases abstractas normalmente contienen uno o más métodos abstractos, los cuales no proveen implementación. Las clases derivadas deben reemplazar los métodos abstractos heredados para permitir la instanciación de objetos.

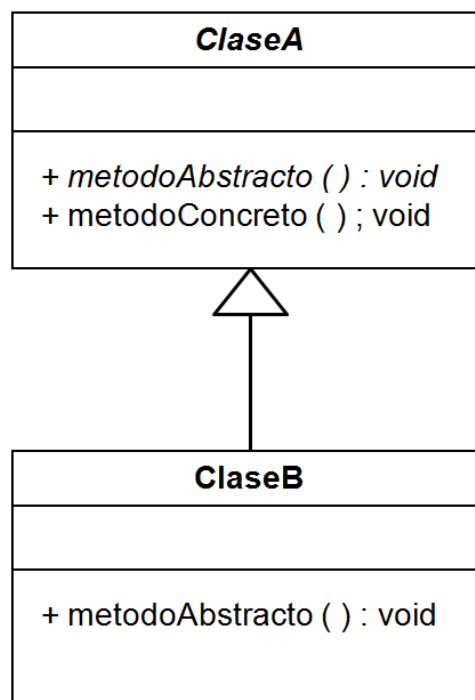


Figura 5 Representación UML de una clase abstracta y una clase concreta.

Para definir clases y métodos abstractos se debe utilizar el modificador **abstract**, tal como se ilustra en el siguiente script:

```
abstract class ClaseA {  
  
    public abstract function metodoAbstracto();  
  
    public function metodoConcreto() {  
        echo("Hola desde metodoConcreto en ClaseA.");  
    }  
  
}
```

Si tenemos una clase que herede de la clase [ClaseA](#) debe implementar el método [metodoAbstracto\(\)](#), tal como se ilustra en el siguiente script:

```
class ClaseB extends ClaseA {  
  
    public function metodoAbstracto() {  
        echo("Hola desde metodoAbstracto en ClaseB");  
    }  
  
}
```

3.4 Polimorfismo

El polimorfismo es la capacidad que tienen objetos de diferentes tipos de responder al mismo mensaje, entiéndase como mensaje la llamada a un método.

Por ejemplo, el mensaje si tenemos un objeto de la clase **Persona** y un objeto de la clase **Ave**, y ambas clases implementan el método **comer()**, entonces podemos afirmar que ambos objetos responden al mensaje **comer()** y de manera diferente, porque es obvio que un persona come de manera diferente a un ave.

Una manera de implementar el polimorfismo es redefiniendo un método abstracto de una clase padre en varias clases hijas, tal como se ilustra en la Figura 6.

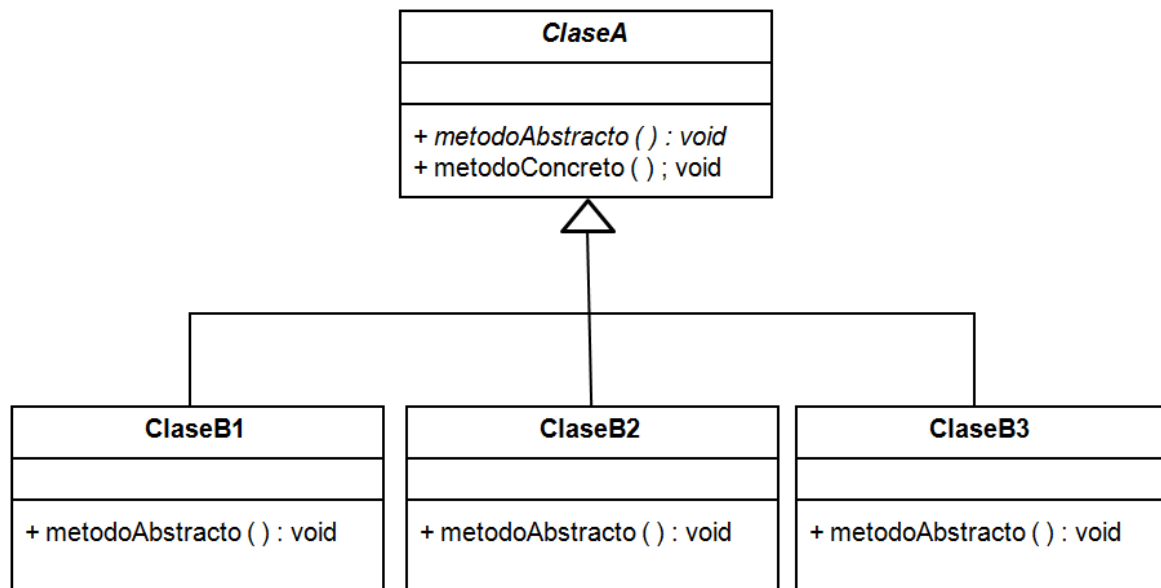


Figura 6 Implementación del polimorfismo.

3.5 Clases y Métodos Finales

Cuando definimos una clase como final, estamos indicando que dicha clase no podrá ser extendida, en otras palabras, no podrá tener clases hijas.

Nota:

No tiene sentido definir una clase como abstracta y final a la vez.

Si definimos un método como final no podrá ser sobre-escrito en las clases hijas.

Para implementar clases y métodos finales debemos utilizar el modificador *final*.

Implementación de una clase final:

```

final class NombreClase {

    // Implementación

}
    
```

Implementación de un método final:

```
class NombreClase {  
  
    public final function nombreMetodo ( ... ) {  
        // Implementación  
    }  
  
}
```

3.6 Resumen

El siguiente cuadro muestra un resumen del comportamiento de las clases abstractas y concretas.

| | Clase Concreta | Clase Abstracta |
|---------------------|---------------------|---------------------|
| Herencia | extends (simple) | extends (simple) |
| Instanciable | Si | No |
| Implementa | Métodos | Algunos métodos |
| Datos | Se permite | Se permite |

Ejemplo 2

En este ejemplo veremos un caso práctico, muy ilustrativo de como se puede aplicar la herencia. Tenemos una clase base abstracta de nombre [CalculadoraBase](#), y una subclase de nombre [Calculadora1](#) que hereda de [CalculadoraBase](#).

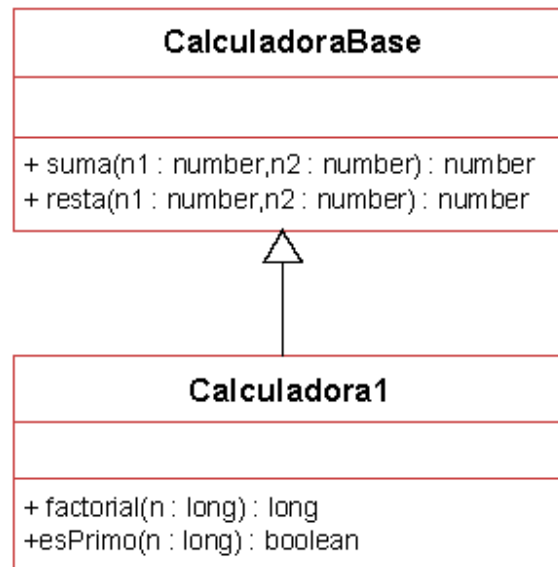


Figura 7 Representación UML de Herencia entre Clases.

En la Figura 7 se puede apreciar que la subclase **Calculadora1** hereda los métodos `suma()` y `resta()`, e implementa los métodos `factorial()` y `esPrimo()`.

Una instancia de la clase **CalculadoraBase** solo tendrá los métodos `suma()` y `resta()`, mientras que una instancia de la clase **Calculadora1** tendrá acceso a los métodos heredados `suma()` y `resta()`, y además sus métodos propios `factorial()` y `esPrimo()`.

A continuación tenemos la implementación de la clase **CalculadoraBase**:

Archivo: CalculadoraBase.php

```

<?php
class CalculadoraBase {

    public function suma( $n1, $n2 ) {
        $rpta = $n1 + $n2;
        return $rpta;
    }

    public function resta( $n1, $n2 ) {
        $rpta = $n1 - $n2;
        return $rpta;
    }

}

?>
    
```

A continuación tenemos la implementación de la clase [Calculadora1](#), la palabra [extends](#) en la definición de la clase indica que está heredando de la clase [CalculadoraBase](#), y puede usted verificar que solo se está implementando los nuevos métodos.

Archivo: [Calculadora1.php](#)

```
<?php

require_once 'CalculadoraBase.php';

class Calculadora1 extends CalculadoraBase {

    public function factorial( $n ) {
        $f = 1;
        while( $n > 1 ) {
            $f *= $n--;
        }
        return $f;
    }

    public function esPrimo( $n ) {
        $primo = TRUE;
        $k = 1;
        while( ++$k < $n ) {
            if( $n % $k == 0 ) {
                $primo = FALSE;
                break;
            }
        }
        return $primo;
    }
}

?>
```

A continuación tenemos el programa [prueba.php](#) que nos permite probar la clase [Calculadora1](#):

Archivo: prueba.php

```
<?php

require_once 'Calculadora1.php';

// Datos
$n1 = 5;
$n2 = 11;

// Proceso
$obj = new Calculadora1();
$suma = $obj->suma($n1, $n2);
$fact = $obj->factorial($n1);
$primo = $obj->esPrimo($n2);

// Reporte
echo("$n1 + $n2 = $suma<br>");
echo("Factorial de $n1 es $fact<br>");
echo("$n2 es primo: " . ($primo?"Si":"No"));

?>
```

Note usted que desde un objeto de la clase [Calculadora1](#) podemos acceder a los métodos definidos en la clase [CalculadoraBase](#) y a los suyos propios.

A continuación podemos ver el resultado de la ejecución del programa [prueba](#).

```
5 + 11 = 16
Factorial de 5 es 120
11 es primo: Si
```


4 Asociación

La asociación se define como una relación semántica entre dos o más clases que especifica conexiones entre las instancias de estas clases.

Una instancia, a menudo es usada como sinónimo de objeto, es una entidad que tiene identidad única, un conjunto de operaciones que se le pueden aplicar, y un estado que almacena los efectos de las operaciones, y una clase es la descripción de un conjunto de objetos que comparten los mismos atributos, operaciones, métodos, relaciones y semántica.

En un sistema informático orientado a objetos, los objetos existentes en un momento dado están conectados entre sí y estas conexiones son descritas en el nivel abstracto de las clases mediante asociaciones.

4.1 Asociación Binaria

Una asociación binaria es una asociación entre dos clases. Se representa mediante una línea continua que conecta las dos clases asociadas: la línea puede ser horizontal, vertical, oblicua, curva, o estar formada por dos o más trazos unidos. El cruce de dos asociaciones que no están conectadas se puede mostrar con un pequeño semicírculo (como en los diagramas de circuitos eléctricos). La asociación en si misma puede distinguirse de sus extremos (association ends), mediante los cuales se conecta a las dos clases asociadas.

Las propiedades relevantes de una asociación pueden pertenecer a la asociación como tal, o a uno de sus extremos. En cada caso estas propiedades se representan mediante adornos gráficos (graphical adornments) situados en el centro de la asociación o en el extremo correspondiente (de tal forma que al mover o modificar la forma de una asociación en una herramienta CASE, el adorno debe desplazarse solidariamente con la asociación). Los adornos que podemos encontrar en una asociación binaria son:

4.1.1 Nombre de la asociación (association name)

Es opcional, y se representa mediante una cadena de caracteres situada junto al centro de la asociación, suficientemente separada de los extremos como para no ser confundida con un nombre de rol.

El nombre de la asociación puede contener un pequeño triángulo negro, denominado “dirección del nombre” (name direction), que apunta en la dirección en la que se debe leer la asociación, tal como se aprecia en la Figura 8.

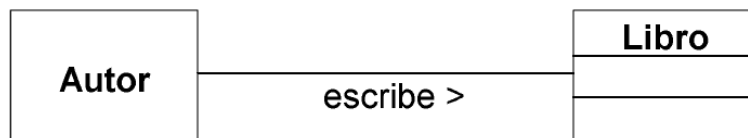


Figura 8 Ejemplo de asociación con nombre y dirección de nombre.

4.1.2 Nombre de rol (rolename)

Se representa mediante una cadena de caracteres situada junto a un extremo de la asociación, como se puede ver en la Figura 9. Es opcional, pero si se especifica en el modelo ya no puede ser omitido en las vistas. Indica el rol que juega en la asociación la clase unida a ese extremo.

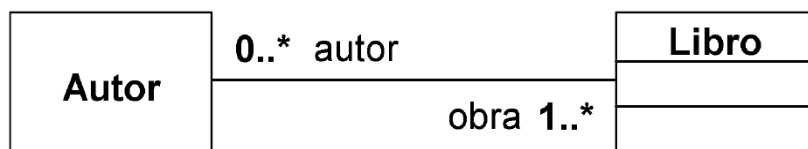


Figura 9 Ejemplo de asociación con nombres de rol y multiplicidades

Tanto el nombre de rol como la multiplicidad deben situarse cerca del extremo de la asociación, para que no se confundan con otra asociación distinta. Se pueden poner en cualquier lado de la línea: es tentador especificar que siempre se sitúen en el mismo lado (a la mano derecha o a la mano izquierda, según se mira desde la clase hacia la asociación), pero en un diagrama repleto de símbolos debe prevalecer la claridad. El nombre de rol y la multiplicidad pueden situarse en lados contrarios del mismo extremo de asociación, o juntos en el mismo lado.

4.1.3 Multiplicidad (multiplicity)

La multiplicidad de una relación determina cuántos objetos de cada clase (tipo de clase) intervienen en la asociación.

Cada asociación tiene dos multiplicidades, una a cada extremo de la asociación.

Para indicar la multiplicidad de una asociación hay que indicar la multiplicidad mínima y máxima utilizando el siguiente formato:

limite_inferior..limite_superior

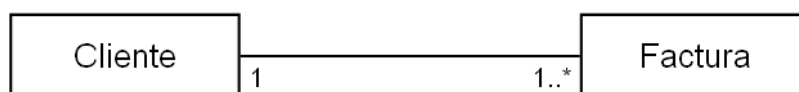
Donde *limite_inferior* y *limite_superior* son valores enteros literales que especifican un intervalo cerrado de enteros. Además, el asterisco (*) se puede usar como limite superior para denotar un valor ilimitado (muchos).

Si se especifica un único valor, entonces el rango contiene sólo ese valor. Si la multiplicidad consiste en solo un asterisco, entonces denota el rango completo de los enteros no negativos, es decir, equivale a 0..* (cero o más). La multiplicidad 0..0 no tiene sentido, ya que indicaría que no puede haber ninguna instancia.

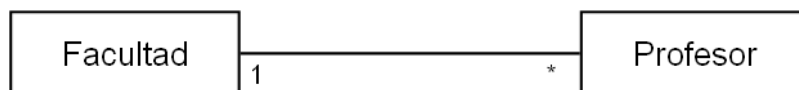
El siguiente cuadro muestra los diferentes tipos de multiplicidad de una asociación:

| Multiplicidad | Significado |
|---------------|-----------------------------|
| 0..1 | Cero o uno. |
| 1 | Uno y sólo uno. |
| 0..* | Cero o muchos. |
| * | Cero o muchos. |
| 1..* | Uno o muchos (al menos uno) |
| N..M | De N hasta M. |

En la Figura 10 tenemos dos ejemplos de cardinalidad.



Un cliente tiene por lo menos una factura.
Una factura pertenece sólo a un cliente.



Una facultad tiene muchos profesores.
Un profesor pertenece a sólo una facultad.

Figura 10 Ejemplos de multiplicidad de una asociación.

4.1.4 Ordenación (ordering)

Si la multiplicidad es mayor que uno, entonces el conjunto de elementos relacionados puede estar ordenado o no ordenado (que es la opción por defecto si no se indica nada). La ordenación se especifica mediante la propiedad `{ordered}` en el extremo correspondiente.

La declaración de que un conjunto es ordenado no especifica cómo se establece o mantiene la ordenación, y en todo caso no se permiten elementos duplicados. Las operaciones que insertan nuevos elementos son responsables de especificar su posición, ya sea implícitamente (por ejemplo, al final) o explícitamente. La estructura de datos y el algoritmo empleados para la ordenación son detalles de implementación y decisiones de diseño que no añaden nueva semántica a la asociación.

En la Figura 11 se ilustra un ejemplo.

4.1.5 Modificabilidad (change ability)

Si los enlaces son modificables en un extremo (pueden ser añadidos, borrados o cambiados), entonces no se necesita ninguna indicación especial. Por el contrario, la propiedad `{frozen}` indica que no se pueden añadir, borrar ni cambiar enlaces de un objeto una vez que ha sido creado e inicializado. La propiedad `{addOnly}` indica que se pueden añadir enlaces, pero no se pueden borrar ni modificar los existentes. La Figura 11 ilustra un ejemplo estas propiedades.



Figura 11 Ejemplo de asociación con especificación de ordenación y modificabilidad.

4.1.6 Navegabilidad (navigability)

Una punta de flecha en el extremo de una asociación indica que es posible la navegación hacia la clase unida a dicho extremo (ver Figura 12). Se pueden poner flechas en uno, dos o ninguno de los extremos. Para ser totalmente explícito, se puede mostrar la flecha en todos los lugares donde la navegación es posible: en la práctica, no obstante, a menudo conviene suprimir algunas de las flechas y mostrar sólo las situaciones excepcionales. Si no se muestra la flecha en un extremo, no se puede inferir que la asociación no sea navegable en esa dirección: se trata simplemente de información omitida.

Conviene adoptar un estilo uniforme en la presentación de las flechas de navegabilidad en los diagramas: el estilo adoptado seguramente variará a lo largo del tiempo en función del tipo de diagrama que se trate. Algunos estilos posibles son:

- **Primer estilo:** mostrar todas las flechas. La ausencia de una flecha indica que la navegación no es posible.
- **Segundo estilo:** suprimir todas las flechas. No se puede inferir nada sobre la navegabilidad de las asociaciones, de igual modo que en otras situaciones hay determinada información que se suprime en una vista por conveniencia, no porque la información no exista.
- **Tercer estilo:** suprimir las flechas con navegabilidad en ambas direcciones, mostrar las flechas sólo para asociaciones con navegabilidad en una dirección. En este caso, no se puede distinguir la asociación navegable en las dos direcciones de la asociación que no es navegable en ninguna dirección, aunque este último caso es ciertamente muy raro en la práctica.

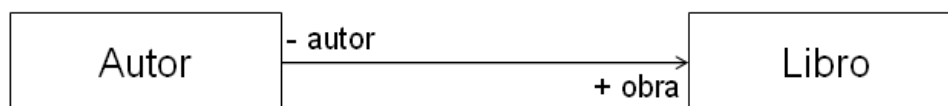


Figura 12 Ejemplo de asociación con especificación de navegabilidad y visibilidad

4.1.7 Visibilidad (visibility)

La visibilidad de un extremo de la asociación se indica mediante un signo especial o con un nombre de propiedad explícito delante del nombre de rol (ver Figura 10), y especifica la visibilidad de la asociación al transitarla en dirección hacia ese nombre de rol. Las posibles visibilidades son:

- (+) public
- (#) protected
- (-) private

La indicación de visibilidad puede ser suprimida, sin que eso signifique que la visibilidad está indefinida o es pública, simplemente, no se desea mostrar en la vista actual.

Ejemplo 3

En el siguiente ejemplo ilustraremos el uso de asociación, para este caso contamos con dos clases: **Article** y **Articles**, tal como se muestra en el diagrama de clases de la Figura 13.

Un objeto de la clase **Articles** contiene una lista de objetos de la clase **Article**,

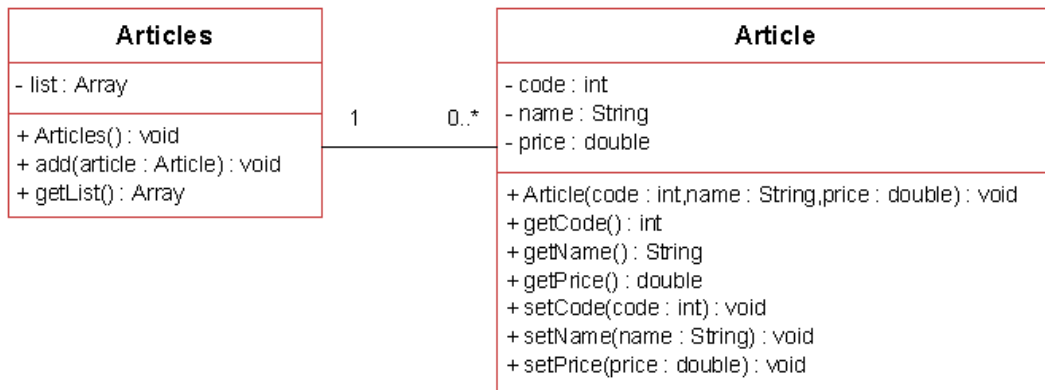


Figura 13 Diagrama de clases de la asociación entre **Articles** y **Article**.

A continuación tenemos la implementación de la clase **Article**:

Archivo: **Article.php**

```

<?php

class Article {

    private $code = 0;
    private $name = null;
    private $price = 0.0;

    function __construct($code, $name, $price) {
        $this->code = $code;
        $this->name = $name;
        $this->price = $price;
    }

    public function getCode() {
        return $this->code;
    }

    public function setCode($code) {
        $this->code = $code;
    }

    public function getName() {
        return $this->name;
    }
}
    
```

```
}

public function setName($name) {
    $this->name = $name;
}

public function getPrice() {
    return $this->price;
}

public function setPrice($price) {
    $this->price = $price;
}

}

?>
```

En la clase [Articles](#) estamos usando un arreglo. Un arreglo puede contener una lista de objetos de cualquier tipo de clase, pero en esta oportunidad esta parametrizado para que los objetos sean de tipo [Article](#). A continuación tenemos la implementación de la clase [Articles](#):

Archivo: Articles.php

```
<?php

require_once 'Article.php';

class Articles {

    private $list = null;

    function __construct() {
        $this->list = array();
    }

    public function add( Article $article){
        $this->list[] = $article;
    }

    public function getList(){
        return $this->list;
    }

}

?>
```

Para ilustrar el uso de la clase `Articles` estamos creando un programa `prueba.php`. El script es el siguiente:

Archivo: `prueba.php`

```
<?php

require_once 'Articles.php';

$articles = new Articles();

$articles->add(new Article(10,"Impresora",150.00));
$articles->add(new Article(20,"Monitor",320.00));
$articles->add(new Article(30,"Teclado",25.00));

foreach ( $articles->getList() as $obj ) {
    echo("{ $obj->getCode()} { $obj->getName()} { $obj->getPrice()}<br>");
}
?>
```

El resultado de su ejecución es:

```
10  Impresora 150
20  Monitor   320
30  Teclado   25
```

4.2 Agregación y Composición

La agregación es un tipo especial de asociación que se emplea para representar la relación entre un todo y sus partes. Se indica mediante un rombo blanco en el extremo de la asociación unido a la clase que representa el “todo”, también llamado “agregado” (aggregate) (ver Figura 14). Evidentemente, no se puede poner el rombo de agregación en los dos extremos de una asociación, ya que se trata de una relación asimétrica. Si una asociación se define como agregación en el modelo, el símbolo de la agregación no se puede omitir en las vistas.

Dos o más agregaciones con el mismo “todo” se pueden dibujar en forma de árbol, juntando los extremos en uno solo, cuando coinciden las demás propiedades en el extremo agregado (multiplicidad, navegabilidad, etc.). La representación como árbol es meramente una opción de presentación, y no conlleva ninguna semántica especial.

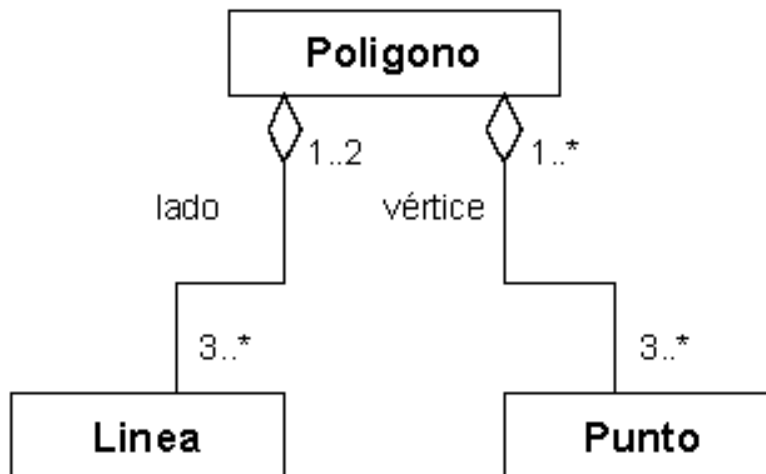


Figura 14 Ejemplo de dos agregaciones que no pueden dibujarse como un árbol porque no tienen la misma multiplicidad en el extremo agregado.

La composición es un tipo de agregación más fuerte, representada por un rombo negro en lugar de blanco, que se sitúa igualmente en el extremo unido al todo o “compuesto” (composite) (ver Figura 15). La composición implica que cada instancia-parte está incluida en un momento dado como máximo en una instancia-todo (compuesto), y que el compuesto tiene la responsabilidad exclusiva de la disposición de sus partes. La multiplicidad en el extremo compuesto no puede exceder de uno, es decir, las partes no se comparten entre distintos todos.

En lugar de usar líneas terminadas en rombos negros en la forma convencional, la composición se puede representar mediante el anidamiento de los símbolos de las partes dentro del símbolo del todo. La multiplicidad de la parte respecto al compuesto se muestra en la esquina superior derecha de la clase anidada que representa la parte, si se omite, se asume que es “muchos” por defecto. El nombre de rol de la parte respecto al todo se escribe delante del nombre de la clase anidada, separado por “:” (ver Figura 15). Nótese que la relación entre una clase y sus atributos es en la práctica una relación de composición.

Las partes de una composición pueden ser tanto clases como asociaciones. Una asociación como parte de un compuesto significa que tanto los objetos conectados como el propio enlace pertenecen todos al mismo compuesto. En la notación convencional, será necesario representar la asociación-parte como clase-asociación, para poder dibujar la composición con el todo mediante una línea terminada en rombo negro. Usando la notación anidada, una asociación entre dos partes dibujada dentro de los límites de la clase que representa el compuesto se considera que es parte de la composición. Por el contrario, si la asociación cruza el borde del compuesto, no será parte de la composición, y sus enlaces pueden establecerse entre partes de diferentes objetos compuestos.

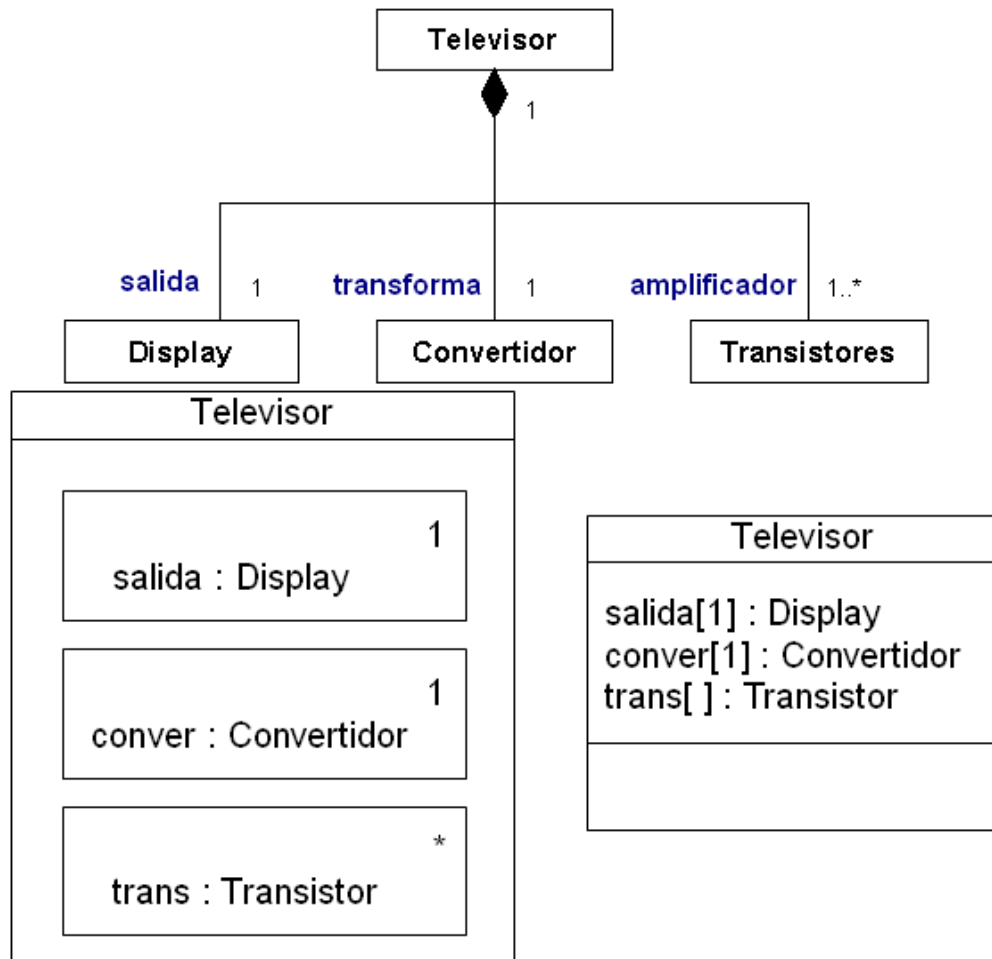


Figura 15 Ejemplo de composición representada en forma convencional (mediante asociaciones), en forma anidada, y en forma de atributos.

4.3 Asociación n-aria

Una asociación n-aria es una asociación entre tres o más clases, alguna de las clases puede participar más de una vez en la misma asociación, pero con roles distintos. Cada instancia de la asociación es una n-tupla de valores de las respectivas clases. Una asociación binaria se puede considerar que es un caso particular de asociación n-aria, con su propia notación.

Una asociación n-aria se representa mediante un rombo unido con una línea a cada clase participante (ver Figura 16). El nombre de la asociación se muestra junto al rombo. Cada extremo de la asociación puede tener adornos, como en una asociación binaria. En particular, se puede indicar la multiplicidad, pero ningún extremo puede tener cualificación ni agregación.

Se puede unir un símbolo de clase al rombo mediante una línea discontinua para indicar que se trata de una clase-asociación con atributos, operaciones o asociaciones con otras clases.

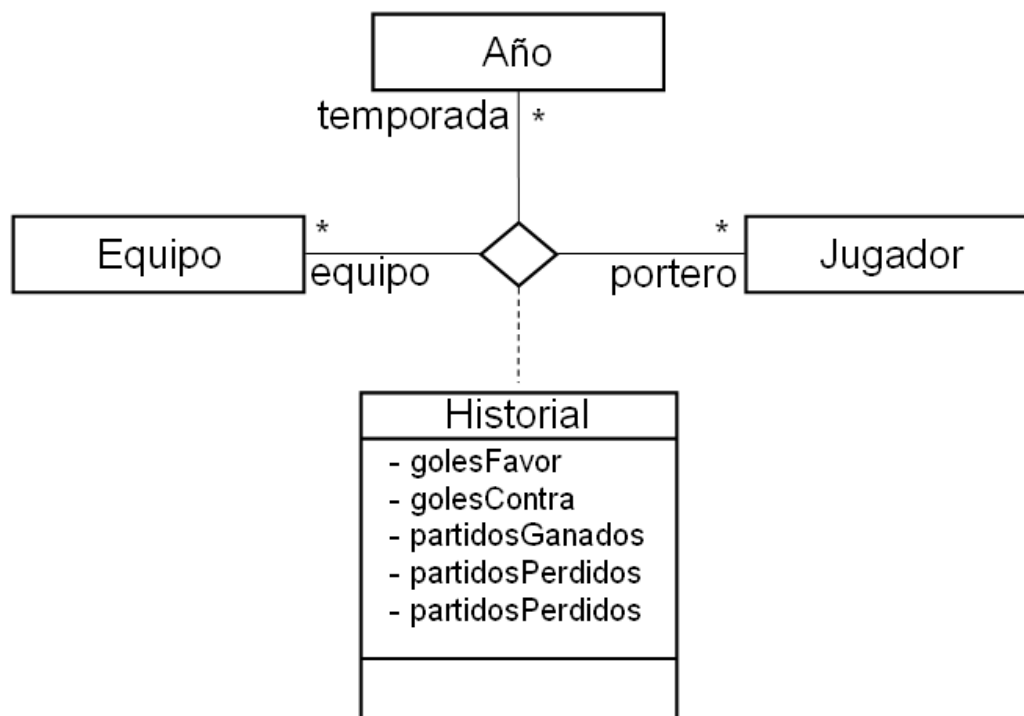


Figura 16 Una asociación ternaria que también es clase-asociación. En este ejemplo se muestra el historial de un equipo en cada temporada con un portero concreto. Se supone que el portero puede ser intercambiado durante la temporada y por tanto puede aparecer en varios equipos

Un enlace n-ario (instancia de asociación n-aria) se representa del mismo modo, mediante un rombo unido a cada una de las instancias participantes.

La multiplicidad se puede especificar en las asociaciones n-arias, pero su significado es menos obvio que la multiplicidad binaria. La multiplicidad de un rol representa el potencial número de instancias en la asociación cuando se fijan los otros N-1 valores.

Ejemplo 4

En este ejemplo veremos cómo implementar la agregación entre dos clases, para tal ilustración tomaremos como ejemplo las clases **Banco** y **Cliente** cuya relación se puede ver en la Figura 17.

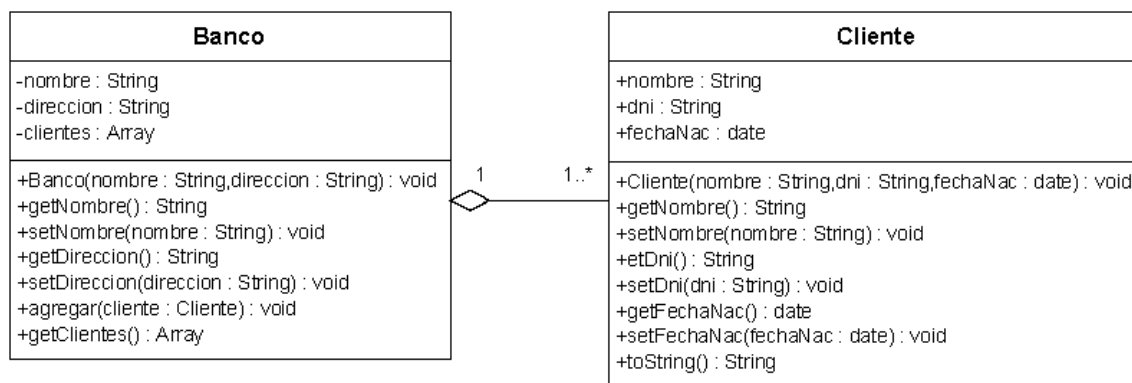


Figura 17 Diagrama de clases donde se ilustra la agregación.

Tanto el banco como el cliente tienen existencia independiente, pero el banco está conformado por clientes. Un cliente puede serlo no solo de un banco, sino de varios bancos, y si un objeto de tipo **Banco** deja de existir los objetos de tipo **Cliente** no tienen que hacerlo, en eso radica su independencia.

A continuación tenemos el script de la clase **Cliente**:

Archivo: Cliente.php

```

<?php

class Cliente {

    private $nombre = null;
    private $dni = null;
    private $fechaNac = null;

    public function __construct($nombre, $dni, $fechaNac) {
        $this->nombre = $nombre;
        $this->dni = $dni;
        $this->fechaNac = $fechaNac;
    }

    public function getNombre() {
        return $this->nombre;
    }

    public function setNombre($nombre) {

```

```
$this->nombre = $nombre;
}

public function getDni() {
    return $this->dni;
}

public function setDni($dni) {
    $this->dni = $dni;
}

public function getFechaNac() {
    return $this->fechaNac;
}

public function setFechaNac($fechaNac) {
    $this->fechaNac = $fechaNac;
}

public function toString() {
    $rpta = "{$this->nombre} - {$this->dni} - {$this->fechaNac}";
    return $rpta;
}
}

?>
```

A continuación tenemos el script de la clase [Banco](#):

Archivo: Banco.php

```
<?php

require_once 'Cliente.php';

class Banco {

    private $nombre = null;
    private $direccion = null;
    private $clientes = array();

    public function __construct($nombre, $direccion) {
        $this->nombre = $nombre;
        $this->direccion = $direccion;
    }

    public function agregar(Cliente $cliente){
        $this->clientes[] = $cliente;
    }
}
```

```
public function getNombre() {  
    return $this->nombre;  
}  
  
public function setNombre($nombre) {  
    $this->nombre = $nombre;  
}  
  
public function getDireccion() {  
    return $this->direccion;  
}  
  
public function setDireccion($direccion) {  
    $this->direccion = $direccion;  
}  
  
public function getClientes() {  
    return $this->clientes;  
}  
}  
?>
```

A continuación tenemos el script del programa prueba.php:

Archivo: prueba.php

```
<?php  
  
require_once 'Banco.php';  
  
// Creamos objetos de tipo Cliente  
$clie01 = new Cliente("Gustavo","06914897","06-Abr-1964");  
$clie02 = new Cliente("Claudia","08154643","15-Jul-1968");  
$clie03 = new Cliente("Sergio","01167437","02-Mar-1960");  
$clie04 = new Cliente("Fabiola","11659812","03-Mar-1990");  
  
// Creamos objetos de tipo Banco  
$banco01 = new Banco("Credito","Perú");  
$banco02 = new Banco("Santander","España");  
  
// Agregamos clientes a los bancos  
$banco01->agregar($clie01);  
$banco01->agregar($clie03);  
$banco01->agregar($clie04);  
  
$banco02->agregar($clie01);
```

```
$banco02->agregar($clie02);  
$banco02->agregar($clie04);  
  
// Imprimimos los datos del banco 01  
echo("Banco: {$banco01->getNombre()} <br>");  
foreach ($banco01->getClientes() as $banco) {  
    echo("{$banco->toString()} <br>");  
}  
  
// Imprimimos los datos del banco 02  
echo("Banco: {$banco02->getNombre()} <br>");  
foreach ($banco02->getClientes() as $banco) {  
    echo("{$banco->toString()} <br>");  
}  
  
?>
```

A continuación tenemos el resultado de su ejecución:

```
Banco: Credito  
Gustavo - 06914897 - 06-Abr-1964  
Sergio - 01167437 - 02-Mar-1960  
Fabiola - 11659812 - 03-Mat-1990  
  
Banco: Santander  
Gustavo - 06914897 - 06-Abr-1964  
Claudia - 08154643 - 15-Jul-1968  
Fabiola - 11659812 - 03-Mat-1990
```

Puede usted comprobar que el objeto `clie01` forma parte del objeto `banco01` y `banco02`, si el objeto `banco01` dejase de existir los objetos de tipo `Cliente` no lo tienen que hacer, porque el tiempo de vida de los objetos incluidos es independiente de objeto que lo incluye.

Si quisiéramos imprimir los datos de todos los clientes podríamos utilizar el siguiente script, sin necesidad de pasar por algún objeto de tipo Banco:

```
echo("{$clie01->toString()} <br>");  
echo("{$clie02->toString()} <br>");  
echo("{$clie03->toString()} <br>");  
echo("{$clie04->toString()} <br>");
```

Finalmente, si la relación fuese de tipo composición las cosas cambiarían, porque en una relación de este tipo el tiempo de vida del objeto incluido va a depender del tiempo de vida del objeto que lo incluye.

5 Proyectos Resueltos

5.1 El Buen Sabor

5.1.1 Requerimiento de Software

El restaurante "**El Buen Sabor**" necesita implementar un programa en PHP que permita a sus empleados calcular los datos que se deben registrar en el comprobante de pago.

Los conceptos que se manejan cuando se trata de una factura son los siguientes:

- Consumo 100.00
- Impuesto 19.00
- Total 119.00
- Servicio (10%) 11.90
- Total General 130.90

Cuando se trata de una boleta son los siguientes:

- Total 119.00
- Servicio (10%) 11.90
- Total General 130.90

Diseñe y desarrolle un programa en PHP que automatice el requerimiento solicitado por el restaurante.

Se sabe que el dato que debe proporcionar el empleado es el **Total**.

5.1.2 Abstracción

La solución planteada considera una clase abstracta de nombre **Comprobante** que debe ser implementada por las hijas **Factura** y **Boleta**.

La clase **Comprobante** tiene los siguientes métodos:

| Método | Descripción |
|-------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| procesar() | Se trata de un método abstracto que debe ser implementado por las clases que deriven de la clase Comprobante . |
| getComprobante() | Se trata de un método con alcance de clase que retorna un objeto de algún tipo de comprobante, para nuestro caso puede ser Factura o Boleta . La implementación de este método hace que la clase Comprobante asuma la función de fábrica de objetos de sus clases hijas. |

Las clases **Factura** y **Boleta** sólo implementan el método **procesar()**, por lo tanto estaremos implementando el polimorfismo.

5.1.3 Diagrama de Clases

En este caso tenemos un programa PHP de nombre **IUComprobantea.php** que nos servirá para ilustrar la funcionalidad del diseño propuesto, este programa se encuentra en la carpeta **view**.

El programa **IUComprobantea.php** hace uso de la clase **Comprobante** para obtener una instancia de la clase **Factura** o **Boleta**, gráficamente lo representamos de la siguiente manera:

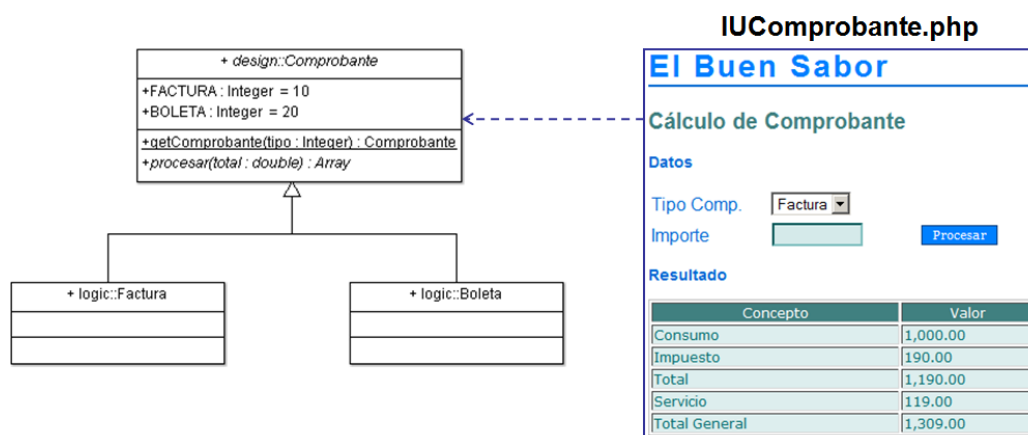
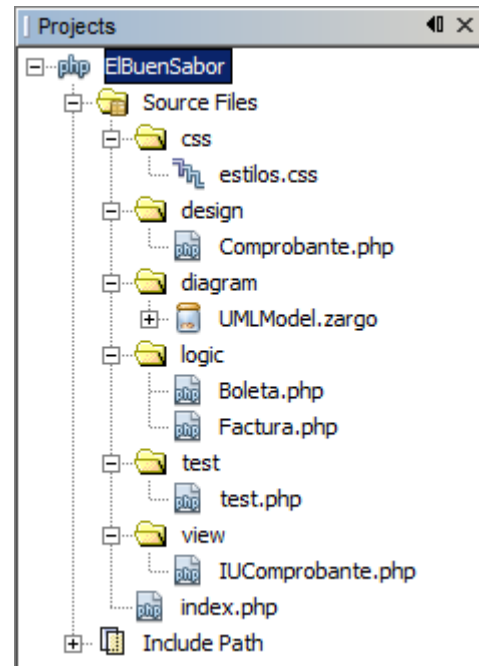


Figura 18 Diagrama de Clases de la Solución

5.1.4 Estructura del Proyecto en NetBeans

Como se puede apreciar en el gráfico de la derecha, tenemos seis carpetas, en la carpeta **css** tenemos la hoja de estilos, en la carpeta **design** ubicamos la clase abstracta **Comprobante**, en el carpeta **diagram** encontramos el diagrama en UML realizado con ArgoUML, en la carpeta **logic** encontramos las clases **Boleta** y **Factura**, en la carpeta **test** tenemos un programa de prueba y en la carpeta **view** el programa **IUComprobante.php**.

En la raíz del proyecto tenemos el programa **index.php**, éste es el que inicia la ejecución del proyecto, y en este caso es hacer un redireccionamiento al programa **IUComprobante.php**.



5.1.5 Codificación

Archivo: ElBuenSabor\design\Comprobante.php

```
<?php

require_once '../logic/Factura.php';
require_once '../logic/Boleta.php';

abstract class Comprobante {

    // Constantes
    const FACTURA = 10;
    const BOLETA = 20;

    // Implementación Parcial
    public static function getComprobante( $tipo ) {
        $obj = null;
        switch ( $tipo ) {
            case self::FACTURA:
                $obj = new Factura();
                break;
            case self::BOLETA:
                $obj = new Boleta();
                break;
        }
    }
}
```

```
        return $obj;  
    }  
  
    // Métodos Abstractos  
    public abstract function procesar( $total );  
}  
?>
```

Archivo: ElBuenSabor\logic\Boleta.php

```
<?php  
  
require_once '../design/Comprobante.php';  
  
class Boleta extends Comprobante {  
  
    public function procesar( $total ){  
        $servicio = $total * 0.10;  
        $totalGeneral = $total + $servicio;  
        $resultado[] = array("Total",$total);  
        $resultado[] = array("Servicio",$servicio);  
        $resultado[] = array("Total General",$totalGeneral);  
        return $resultado;  
    }  
}  
?  
>
```

Archivo: ElBuenSabor\logic\Factura.php

```
<?php  
  
require_once '../design/Comprobante.php';  
  
class Factura extends Comprobante {  
  
    public function procesar( $total ){  
        $consumo = $total / 1.19;  
        $impuesto = $total - $consumo;  
        $servicio = $total * 0.10;  
        $totalGeneral = $total + $servicio;  
        $resultado[] = array("Consumo", number_format($consumo,2));  
        $resultado[] = array("Impuesto",number_format($impuesto,2));  
        $resultado[] = array("Total",number_format($total,2));  
        $resultado[] = array("Servicio",number_format($servicio,2));  
        $resultado[] = array("Total General",number_format($totalGeneral,2));  
    }  
}
```

```
        return $resultado;
    }

}

?>
```

Archivo: ElBuenSabor\test\test.php

```
<?php

require_once '../design/Comprobante.php';

$obj = Comprobante::getComprobante(Comprobante::BOLETA);
$resultado = $obj->procesar(1000);
foreach ($resultado as $rec) {
    echo( "{$rec[0]} ==> {$rec[1]} <br>" );
}

?>
```

Archivo: ElBuenSabor\view\IUComprobante.php

```
<?php

require_once '../design/Comprobante.php';

if( isset( $_POST["procesar"] ) ) {
    $tipo = $_POST["tipo"];
    $importe = $_POST["importe"];
    if($tipo == "A") {
        $obj = Comprobante::getComprobante(Comprobante::FACTURA);
    }else {
        $obj = Comprobante::getComprobante(Comprobante::BOLETA);
    }
    $resultado = $obj->procesar($importe);
}

?>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
    <head>
        <meta http-equiv="Expires" content="Mon, 06 Jan 1990 00:00:01 GMT">
        <meta http-equiv="Cache-Control" content="no-cache, must-revalidate">
        <meta http-equiv="Pragma" content="no-cache">
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
        <link rel="stylesheet" type="text/css" href="../css/estilos.css"/>
        <title>EL BUEN SABOR</title>
```

```

</head>
<body>
  <h1>El Buen Sabor</h1>
  <h2>Cálculo de Comprobante</h2>
  <div id="datos">
    <h3>Datos</h3>
    <form name="form1" method="post"
    action="<?php echo($_SERVER["PHP_SELF"]); ?>">
      <table width="400">
        <tr>
          <td width="103" height="27">Tipo Comp. </td>
          <td width="132"><select name="tipo" id="tipo">
            <option value="A">Factura</option>
            <option value="B">Boleta</option>
          </select>
          </td>
          <td width="149">&nbsp;</td>
        </tr>
        <tr>
          <td>Importe</td>
          <td>
            <input name="importe" type="text" class="campoEdicion"
            id="importe" size="10" maxlength="10">
          </td>
          <td>
            <input name="procesar" type="submit" class="boton"
            id="procesar" value="Procesar">
          </td>
        </tr>
      </table>
    </form>
  </div>
  <?php if( $_REQUEST ) { ?>
  <div id="resultado">
    <h3>Resultado</h3>
    <table width="352" border="1">
      <tr class="tablaTitulo">
        <td width="220">Concepto</td>
        <td width="116">Valor</td>
      </tr>
      <?php foreach ($resultado as $rec) { ?>
      <tr class="TablaDato">
        <td><?php echo( $rec[0] ) ?></td>
        <td><?php echo( $rec[1] ) ?></td>
      </tr>
      <?php } ?>
    </table>
  </div>
  <?php } ?>
</body>
</html>

```

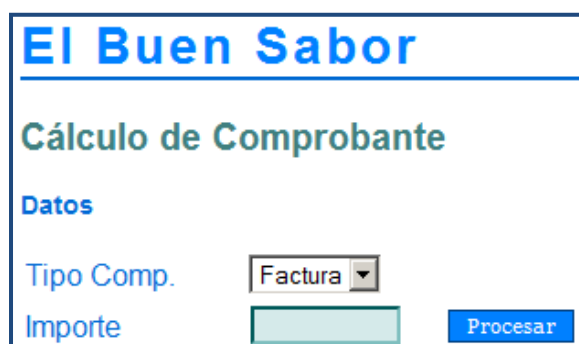
Archivo: ElBuenSabor\index.php

```
<?php
    header("location: view/IUComprobante.php");
?>
```

El archivo css lo puede revisar en el CD que viene junto con este libro.

5.1.6 Ejecución del Proyecto

Cuando ejecutamos el proyecto inicialmente tenemos la siguiente interfaz:



El Buen Sabor

Cálculo de Comprobante

Datos

Tipo Comp.

Importe

Procesar

Después de ingresar los datos y hacer clic en el botón **Procesar** obtenemos el siguiente resultado:



El Buen Sabor

Cálculo de Comprobante

Datos

Tipo Comp.

Importe

Procesar

Resultado

| Concepto | Valor |
|---------------|----------|
| Consumo | 1,000.00 |
| Impuesto | 190.00 |
| Total | 1,190.00 |
| Servicio | 119.00 |
| Total General | 1,309.00 |

Luego usted puede ingresar otros valores para hacer otra operación.

5.2 Electronica Store

5.2.1 Requerimiento de Software

La empresa **Electrónica Store** necesita una aplicación para que sus vendedores puedan registrar sus ventas del día.

Al finalizar el día la aplicación debe permitir obtener un listado de todas las ventas realizadas.

5.2.2 Abstracción

La solución planteada considera una clase denominada **Venta** para registrar una venta, y una clase de nombre **Ventas** que está compuesta por todas las ventas del día.

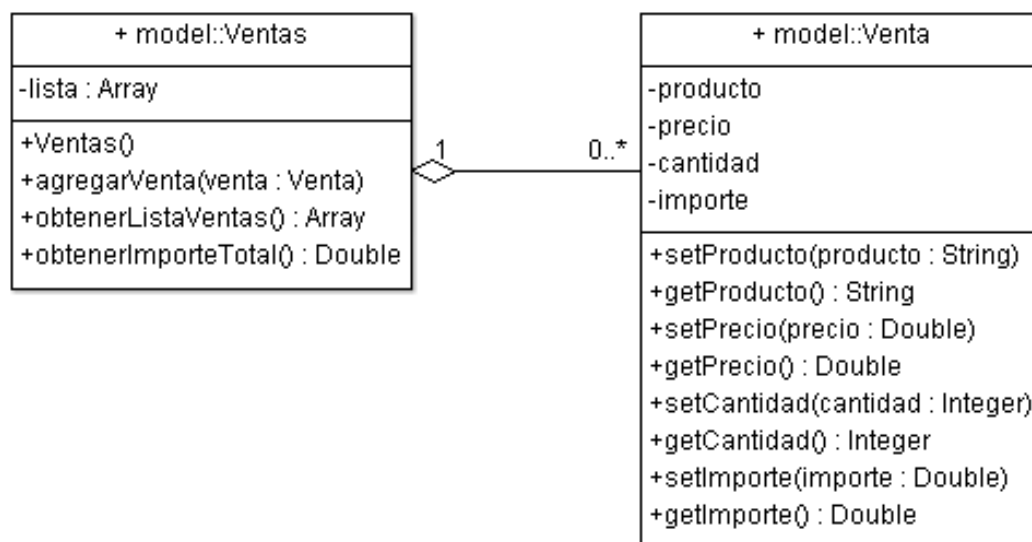


Figura 19 Abstracción de la solución.

Es necesario guardar una instancia de la clase venta en sesión.

También tenemos el diseño de la interfaz de usuario para el registro de ventas:



The interface shows a navigation bar with links: Inicio, Registrar Venta, Listado Ventas, and Salir. The main heading is 'Registrar Venta'. Below it are three input fields labeled 'Producto', 'Precio', and 'Cantidad'. At the bottom are two buttons: 'Grabar' (Save) and 'Cancelar' (Cancel).

Figura 20 Interfaz de usuario para el registro de las ventas.

Y el diseño de la interfaz de usuario para el listado de todas las ventas del día.



The interface shows a navigation bar with links: Inicio, Registrar Venta, Listado Ventas, and Salir. The main heading is 'Ventas del Día'. Below it is a table with 4 columns: Producto, Precio, Cant, and Importe. The table contains 3 rows of data. Below the table, it shows 'Importe Total: 2466'.

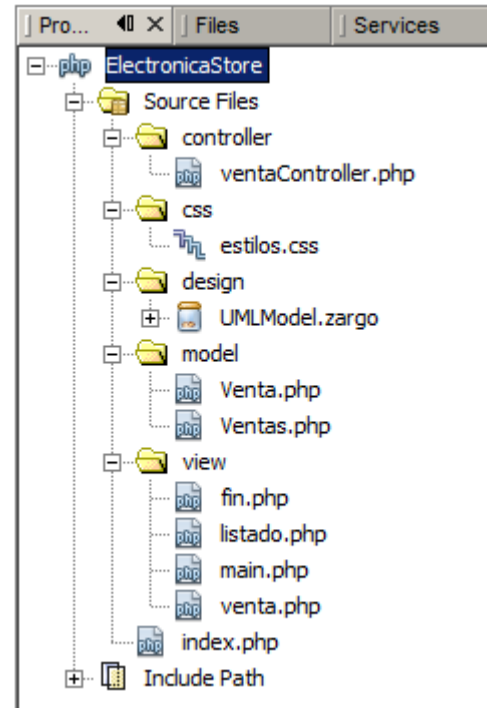
| Producto | Precio | Cant | Importe |
|----------------|--------|------|---------|
| Disco Duro | 250.0 | 5 | 1250 |
| Impresora | 240 | 3 | 720 |
| Memoria de 4GB | 124.0 | 4 | 496 |

Importe Total: 2466

5.2.3 Estructura del Proyecto en NetBeans

Como se puede apreciar en el gráfico de la derecha, tenemos cinco carpetas, en la carpeta **css** tenemos la hoja de estilos, en la carpeta **model** las clases **Venta** y **Ventas**, en el carpeta **design** encontramos el diagrama en UML realizado con ArgoUML, en la carpeta **control** encontramos el programa **ventaController.php** que se encarga de llevar el control del registro de las ventas en una instancia de la clase **Ventas** que se guarda en sesión y en la carpeta **view** tenemos los programas que corresponden a las interfaces de usuario.

En la raíz del proyecto tenemos el programa **index.php**, éste es el que inicia la ejecución del proyecto, y en este caso es hacer un redireccionamiento al programa **main.php**.



El siguiente grafico ilustra el funcionamiento de programa.

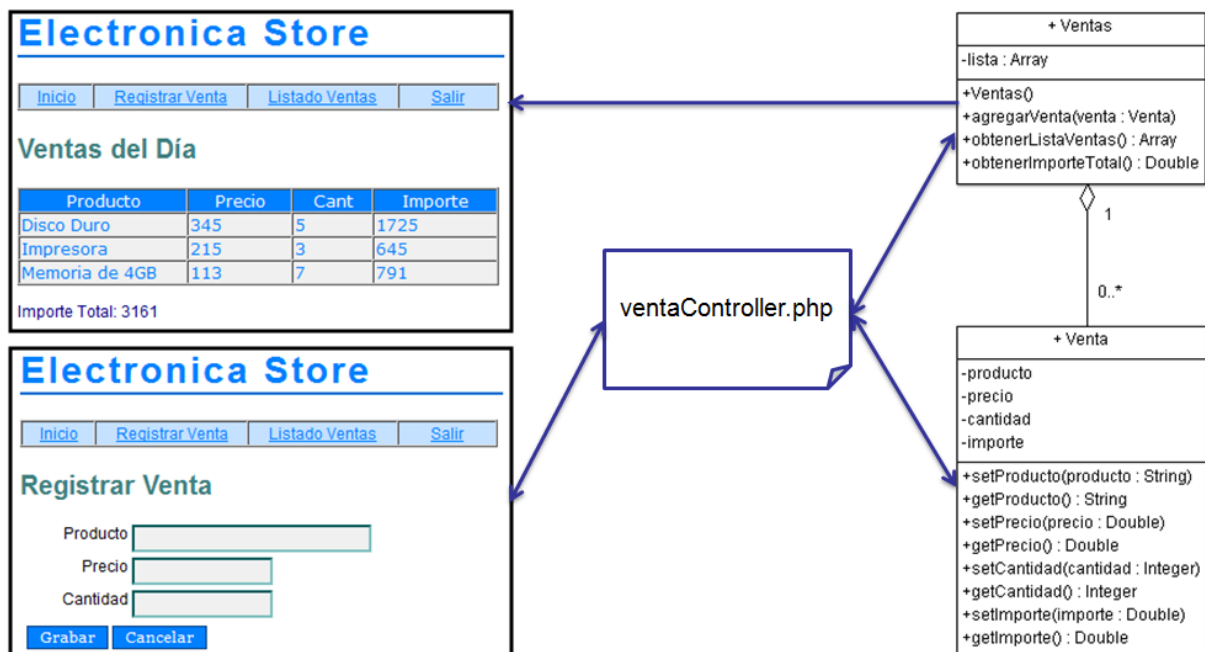


Figura 21 Esquema de funcionamiento del programa.

5.2.4 Codificación

Archivo: ElectronicaStore\model\Venta.php

```
<?php
class Venta {

    private $producto;
    private $precio;
    private $cantidad;
    private $importe;

    function __construct () {
        $this->setProducto(null);
        $this->setPrecio(null);
        $this->setCantidad(null);
        $this->setImporte(null);
    }

    public function getProducto() {
        return $this->producto;
    }

    public function setProducto($producto) {
        $this->producto = $producto;
    }

    public function getPrecio() {
        return $this->precio;
    }

    public function setPrecio($precio) {
        $this->precio = $precio;
    }

    public function getCantidad() {
        return $this->cantidad;
    }

    public function setCantidad($cantidad) {
        $this->cantidad = $cantidad;
    }

    public function getImporte() {
        return $this->importe;
    }

    public function setImporte($importe) {
        $this->importe = $importe;
    }
}
```

```
}  
  
}  
?>
```

Archivo: ElectronicaStore\model\Ventas.php

```
<?php  
class Ventas {  
  
    private $lista; // Arreglo para almacenar objetos de tipo Venta  
  
    function __construct() {  
        $this->lista = array ();  
    }  
  
    public function agregarVenta(Venta $venta) {  
        $this->lista[] = $venta;  
    }  
  
    public function obtenerListaVentas() {  
        return $this->lista;  
    }  
  
    public function obtenerImporteTotal() {  
        $total = 0.0;  
        foreach ($this->lista as $venta) {  
            $total += $venta->getImporte();  
        }  
        return $total;  
    }  
}  
?>
```

Archivo: ElectronicaStore\controller\ventaController.php

```
<?php  
session_start();  
  
require_once '../model/Venta.php';  
require_once '../model/Ventas.php';  
  
if( isset( $_REQUEST["btnGrabar"] ) ) {  
  
    // Datos  
    $venta = new Venta();  
    $venta->setProducto($_REQUEST["producto"]);
```

```
$venta->setPrecio($_REQUEST["precio"]);
$venta->setCantidad($_REQUEST["cantidad"]);
$venta->setImporte($venta->getPrecio()*$venta->getCantidad());

// Acceso al objeto Ventas
if( isset($_SESSION["ventas"]) ) {
    $ventas = unserialize($_SESSION["ventas"]);
} else {
    $ventas = new Ventas();
}

// Registrar la venta
$ventas->agregarVenta($venta);
// Registrar el objeto ventas en sesión
$_SESSION["ventas"] = serialize($ventas);

// Documento destino
$destino = "../view/venta.php?msg=Proceso ok.";

} elseif( isset( $_REQUEST["btnCancelar"] ) ) {

    // Documento destino
    $destino = "../view/main.php";

}

header("location: $destino");

?>
```

Archivo: **ElectronicaStore\view\main.php**

```
<?php
session_start();
?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <link rel="stylesheet" type="text/css" href="../css/estilos.css">
        <title>Electrónica Store</title>
    </head>
    <body>
        <h1>Electronica Store</h1>
        <table border="1" cellspacing="0">
            <tr class="menu01">
                <td width="52"><a href="main.php">Inicio</a></td>
                <td width="112"><a href="venta.php">Registrar Venta</a></td>
                <td width="110"><a href="listado.php">Listado Ventas</a></td>
```

```
<td width="71"><a href="fin.php">Salir</a></td>
</tr>
</table>
</body>
</html>
```

Archivo: ElectronicaStore\view\venta.php

```
<?php
session_start();
if( isset($_REQUEST["msg"]) ) {
    $msg = $_REQUEST["msg"];
}
?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<link rel="stylesheet" type="text/css" href="../css/estilos.css">
<title>Electrónica Store</title>
</head>
<body>
<h1>Electronica Store</h1>
<table border="1" cellspacing="0">
<tr class="menu01">
<td width="52"><a href="main.php">Inicio</a></td>
<td width="112"><a href="venta.php">Registrar Venta</a></td>
<td width="110"><a href="listado.php">Listado Ventas</a></td>
<td width="71"><a href="fin.php">Salir</a></td>
</tr>
</table>
<h2>Registrar Venta</h2>
<form name="form1" method="post" action="../controller/ventaController.php">
<div class="divCampo">
<label for="producto" class="etiqueta">Producto</label>
<input name="producto" type="text" class="campoEdicion"
id="producto" size="30" maxlength="30">
</div>
<div class="divCampo">
<label for="precio" class="etiqueta">Precio</label>
<input name="precio" type="text" class="campoEdicion"
id="precio" size="15" maxlength="15">
</div>
<div class="divCampo">
<label for="cantidad" class="etiqueta">Cantidad</label>
<input name="cantidad" type="text" class="campoEdicion"
id="cantidad" size="15" maxlength="15">
</div>
<div class="divBotones">
```

```

        <input name="btnGrabar" type="submit" class="boton"
        id="btnGrabar" value="Grabar">
        <input name="btnCancelar" type="submit" class="boton"
        id="btnCancelar" value="Cancelar">
    </div>
</form>
<?php if( isset($msg) ) {?>
<p><?php echo($msg); ?></p>
<?php } ?>
</body>
</html>

```

Archivo: ElectronicaStore\view\listado.php

```

<?php
session_start();
require_once '../model/Venta.php';
require_once '../model/Ventas.php';

if( isset($_SESSION["ventas"]) ) {
    $ventas = unserialize($_SESSION["ventas"]);
    $listado = $ventas->obtenerListaVentas();
    $total = $ventas->obtenerImporteTotal();
} else {
    $msg = "No existe ninguna venta.";
}
?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <link rel="stylesheet" type="text/css" href="../css/estilos.css">
        <title>Electrónica Store</title>
    </head>
    <body>
        <h1>Electronica Store</h1>
        <table border="1" cellspacing="0">
            <tr class="menu01">
                <td width="52"><a href="main.php">Inicio</a></td>
                <td width="112"><a href="venta.php">Registrar Venta</a></td>
                <td width="110"><a href="listado.php">Listado Ventas</a></td>
                <td width="71"><a href="fin.php">Salir</a></td>
            </tr>
        </table>
        <h2>Ventas del Día</h2>
        <?php if( isset( $listado ) ) { ?>
        <table width="363" border="1" cellspacing="0">
            <tr class="tablaTitulo">
                <td width="123">Producto</td>

```

```
<td width="75">Precio</td>
<td width="57">Cant</td>
<td width="90">Importe</td>
</tr>
<?php foreach ($listado as $venta) { ?>
<tr class="TablaDato">
    <td><?php echo($venta->getProducto()); ?></td>
    <td><?php echo($venta->getPrecio()); ?></td>
    <td><?php echo($venta->getCantidad()); ?></td>
    <td><?php echo($venta->getImporte()); ?></td>
</tr>
<?php } ?>
</table>
<p>Importe Total: <?php echo($total); ?></p>
<?php } ?>
<?php if( isset( $msg ) ) { ?>
<p><?php echo($msg); ?></p>
<?php }?>
</body>
</html>
```

Archivo: **ElectronicaStore\view\fin.php**

```
<?php
session_start();
session_unset();
session_destroy();
?>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <link rel="stylesheet" type="text/css" href="../css/estilos.css">
        <title>Electrónica Store</title>
    </head>
    <body>
        <h1>Electronica Store</h1>
        <p>Sesión Finalizada</p>
    </body>
</html>
```

Archivo: **ElectronicaStore\index.php**

```
<?php
    header("location: view/main.php");
?>
```

5.2.5 Ejecución del Proyecto

Cuando ejecutamos el proyecto inicialmente tenemos la siguiente interfaz:



The screenshot shows the main menu of the 'Electrónica Store' application. It has a title bar 'Electrónica Store' and a navigation bar with four buttons: 'Inicio', 'Registrar Venta', 'Listado Ventas', and 'Salir'.

Para registrar las ventas del día ingresamos a la opción **Registrar Venta**, obtenemos la siguiente interfaz:



The screenshot shows the 'Registrar Venta' form. It has a title bar 'Electrónica Store' and a navigation bar with four buttons: 'Inicio', 'Registrar Venta', 'Listado Ventas', and 'Salir'. Below the navigation bar is the title 'Registrar Venta'. There are three input fields: 'Producto', 'Precio', and 'Cantidad'. At the bottom are two buttons: 'Grabar' and 'Cancelar'.

Para visualizar el listado de las ventas del día ingresamos a la opción **Listado Ventas**, obtenemos la siguiente interfaz:



The screenshot shows the 'Listado Ventas' interface. It has a title bar 'Electrónica Store' and a navigation bar with four buttons: 'Inicio', 'Registrar Venta', 'Listado Ventas', and 'Salir'. Below the navigation bar is the title 'Ventas del Día'. There is a table with the following data:

| Producto | Precio | Cant | Importe |
|----------------|--------|------|---------|
| Disco Duro | 345 | 5 | 1725 |
| Impresora | 215 | 3 | 645 |
| Memoria de 4GB | 113 | 7 | 791 |

Below the table, it says 'Importe Total: 3161'.

Para regresar al menú inicial ingresamos a la opción **Inicio**.

Para salir del sistema y borrar todos los datos de sesión ingresamos a la opción **Salir**.