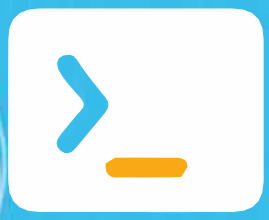


Argentina programa

Programación Orientada a Objetos con JAVA

Guía III

Relaciones entre Clases

RELACIONES:

Un conjunto de objetos aislados tiene escasa capacidad para resolver un problema. En una aplicación real los objetos colaboran e intercambian información; existiendo distintos tipos de relaciones entre ellos.

Las relaciones se expresan frecuentemente utilizando verbos o frases con verbo del lenguaje natural tales como: tiene-un, está compuesto de, usa un, es un. Por ejemplo:

Un Círculo es una Figura.

Un Auto tiene 4 ruedas.

Matemático usa una Calculadora.

A nivel de diseño, podemos distinguir 5 tipos de relaciones básicas entre clases de objetos: dependencia, asociación, agregación, composición y herencia.

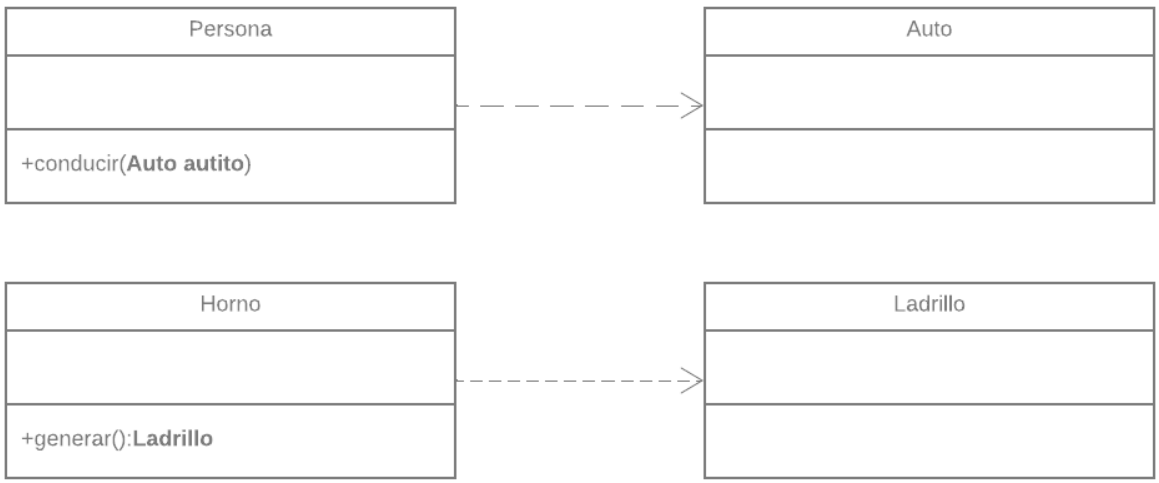
Cada relación tiene una representación gráfica en UML.

Dependencia:

La dependencia representa una relación del tipo "usa un", significa la necesidad de tener elementos acoplados en los cuales unos necesitan de otro para su funcionamiento.

Decimos que una clase depende de otra; cuando: uno de los parámetros o tipo de retorno de cualquiera de los métodos de la clase dependiente es del tipo de la clase independiente.

Por ejemplo:

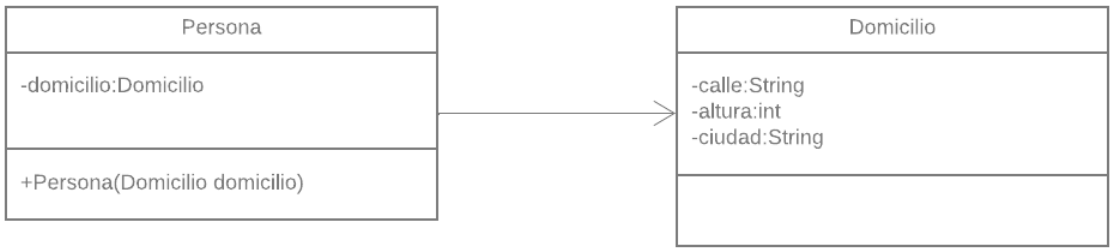


Asociación:

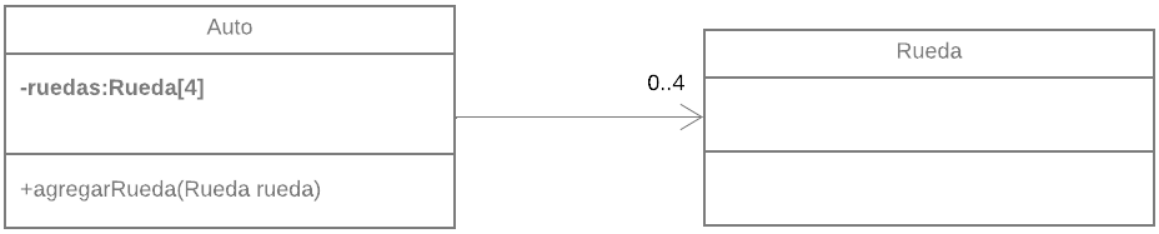
La asociación es la relación más importante y común. Refleja la relación entre dos clases independientes que se mantiene durante la vida de los objetos de dichas clases o al menos durante un tiempo prolongado. Representa una relación del tipo "tiene un".

Una asociación se implementa en Java, introduciendo **referencias** a objetos de una clase como atributos en la otra.

Normalmente la conexión entre los objetos se realiza recibiendo la referencia de uno de ellos en el constructor o una operación ordinaria (método) del otro.



Si la relación tiene una multiplicidad[1] superior a 1 (uno) entonces será necesario utilizar un array de referencias o alguna estructura de datos dinámica del paquete **java.util** como ArrayList ó Vector para almacenar las referencias. (Tema que desarrollaremos más adelante).

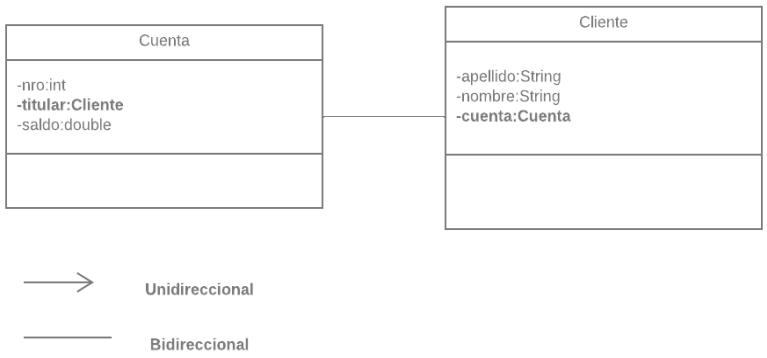


Las asociaciones son inherentemente **bidireccionales**, puede recorrerse en ambas direcciones. Por ejemplo:

Una Cuenta tiene un titular que es un Cliente.

Un Cliente tiene una única Cuenta.

[1] Multiplicidad: Es el número de objetos de un extremo de la asociación que están enlazados con un objeto del otro extremo. Por ejemplo: Una Empresa puede emplear a muchas Personas; la multiplicad de Empresa con Persona es "una a muchos": 1..*

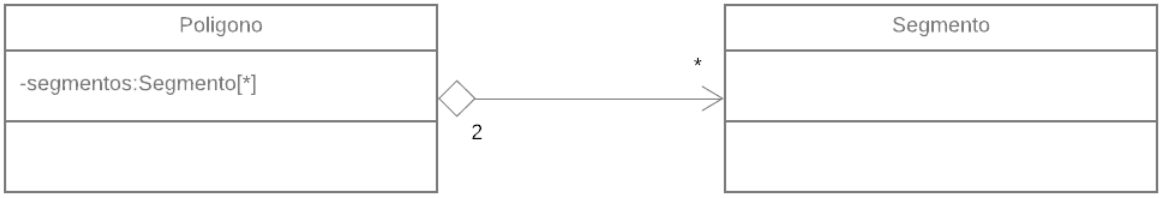


Agregación:

La agregación es un tipo especial de asociación donde se añade el matiz semántico de que la clase de donde parte la relación representa el "todo" y las clases asociadas "las partes".

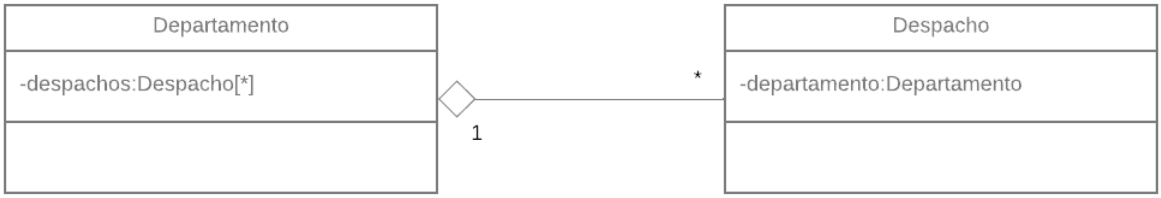
Realmente Java y la mayoría de los lenguajes orientados a objetos no disponen de una implementación especial para este tipo de relaciones. Básicamente se tratan como las asociaciones ordinarias.

Por ejemplo:



En este caso como queremos dar a entender que un Polígono puede estar formado por varios objetos de tipo Segmento; es decir, el Polígono es el "todo" formado por las partes de tipo "Segmento"; utilizamos un rombo del lado de la clase Polígono para indicar justamente eso. Como decíamos anteriormente, la forma de implementar esto en Java es como una simple asociación, es decir, la clase Polígono tendrá un atributo que permita referencias a varios objetos Segmento, por ejemplo con un arreglo.

Otro Ejemplo podría ser:

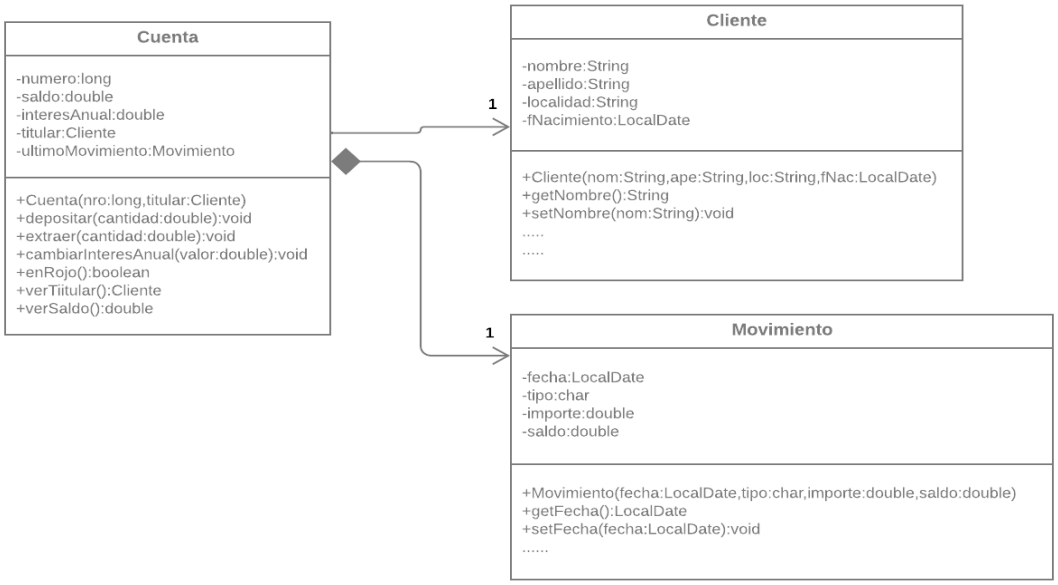


Un Departamento puede estar formado por muchos (*) Despachos y como en el diagrama está representada la relación como "bidireccional" leo también que un Despacho pertenece a un único Departamento. La clase Departamento es el "todo" y Despacho "las partes" que lo forman; no tendría sentido tener un Departamento si no tiene Despachos.

Composición (Agregación fuerte):

Es un tipo de agregación que añade el matiz de que la clase "todo" controla la existencia de las clases "parte". Es decir, normalmente la clase "todo" creará en algún momento las clases "parte" y luego al finalizar se encargará de su destrucción. Se representa con un rombo sombreado el extremo del "todo". Es importante aclarar, que un objeto sólo puede ser parte de una composición.

Analicemos el siguiente ejemplo:



Una Cuenta Bancaria posee: un *número de cuenta*, un *saldo actual*, un *interés anual*, el Cliente que es *titular* de dicha Cuenta y para ahorrar complejidad, en lugar de registrar todos los Movimientos sobre ella, sólo registraremos el *último Movimiento*.

Como podemos observar, la relación entre Cuenta y Cliente es del tipo "*tiene un*", es decir, una asociación ordinaria; y la forma que tendremos para vincular una Cuenta con un Clientes es a través del constructor de Cuenta que recibirá el Cliente titular. Ahora bien, con respecto a la relación Cuenta y Movimiento, también es una relación del tipo "tiene un", pero esta vez agregando el matiz de que Cuenta es el "todo" y Movimiento una "parte" de Cuenta; pero al ser una **composición**, será Cuenta la que creará cada Movimiento en alguno de sus métodos ya que no recibe según el UML un Movimiento como parámetro del constructor ni como parámetro de alguno de sus métodos. Piensen: en que métodos una Cuenta podría crear un objeto de tipo Movimiento y registrarlo en la variable atributo "*ultimoMovimiento*"?. Exacto!!!... los métodos depositar y extraer, serán los encargados de que cada vez que alguien le pida a un objeto de tipo

Cuenta hacer una extracción o un deposito, estos métodos tendrán la lógica necesaria para instanciar un Movimiento. Como un ejemplo dice más que mil palabras… verás a continuación una forma de implementar este modelo.

C:/Users/Usuario/Documents/NetBeansProjects/EjemploCompleto/src/ejemplocompleto/Cliente.java

```
 1 package ejemplocompleto;
 2 import java.time.LocalDate;
 3
 4 public class Cliente {
 5
 6     private String nombre;
 7     private String apellido;
 8     private String localidad;
 9     private LocalDate fNacimiento;
10
11     public Cliente(String nombre, String apellido, String localidad,
12             LocalDate fNacimiento) {
13         this.nombre = nombre;
14         this.apellido = apellido;
15         this.localidad = localidad;
16         this.fNacimiento = fNacimiento;
17     }
18
19     public String getNombre() {
20         return nombre;
21     }
22
23     public void setNombre(String nombre) {
24         this.nombre = nombre;
25     }
26
27     public String getApellido() {
28         return apellido;
29     }
30
31     public void setApellido(String apellido) {
32         this.apellido = apellido;
33     }
34
35     public String getLocalidad() {
36         return localidad;
37     }
38
39     public void setLocalidad(String localidad) {
40         this.localidad = localidad;
41     }
42     public LocalDate getfNacimiento() {
43         return fNacimiento;
44     }
45     public void setfNacimiento(LocalDate fNacimiento) {
46         this.fNacimiento = fNacimiento;
47     }
48      }
```

1.1 of 1 2022.12.11 13:33:37

```java
package ejemplocompleto;
import java.time.LocalDate;

public class Movimiento {
    private LocalDate fecha;
    private char tipo;
    private double importe;
    private double saldo;

    public Movimiento(LocalDate fecha, char tipo, double importe, double saldo){
        this.fecha = fecha;
        this.tipo = tipo;
        this.importe = importe;
        this.saldo = saldo;
    }

    public LocalDate getFecha() {
        return fecha;
    }

    public void setFecha(LocalDate fecha) {
        this.fecha = fecha;
    }

    public char getTipo() {
        return tipo;
    }

    public void setTipo(char tipo) {
        this.tipo = tipo;
    }

    public double getImporte() {
        return importe;
    }

    public void setImporte(double importe) {
        this.importe = importe;
    }

    public double getSaldo() {
        return saldo;
    }
    public void setSaldo(double saldo) {
        this.saldo = saldo;
    }
   }
```

```
 1 package ejemplocompleto;
 2 import java.time.LocalDate;
 3
 4 public class Cuenta {
 5     private long numero;
 6     private double saldo;
 7     private double interesAnual;
 8     private Cliente titular;
 9     private Movimiento ultimoMovimiento;
10
11     public Cuenta(long numero, Cliente titular) {
12         this.numero = numero;
13         this.titular = titular;
14     }
15     //------------------------
16     public void depositar(double cantidad){
17
18   this.saldo+=cantidad;
19   this.ultimoMovimiento=new Movimiento(LocalDate.now(),'D',cantidad,this.saldo);
20     }
21     //---------------------------------------
22     public void extraer(double cantidad){
23
24    this.saldo-=cantidad;
25   this.ultimoMovimiento=new Movimiento(LocalDate.now(),'E',cantidad,this.saldo);
26
27     }
28
29     public void cambiarInteresAnual(double interes){
30         this.interesAnual=interes;
31     }
32
33     public boolean enRojo(){
34
35         return this.saldo < 0;
36
37     }
38
39     public Cliente verTitular(){
40
41         return this.titular;
42     }
43     public double verSaldo(){
44
45         return this.saldo;
46     }
47      }
```

Ahora desde el método main de la clase principal del proyecto, crearemos una Cuenta obviamente con saldo 0 (cero) para un Cliente Juan López, nacido el 20 de abril de 1976, en la ciudad de San Luis; luego haremos un depósito de 1000 pesos en dicha Cuenta y por último pediremos ver el saldo.

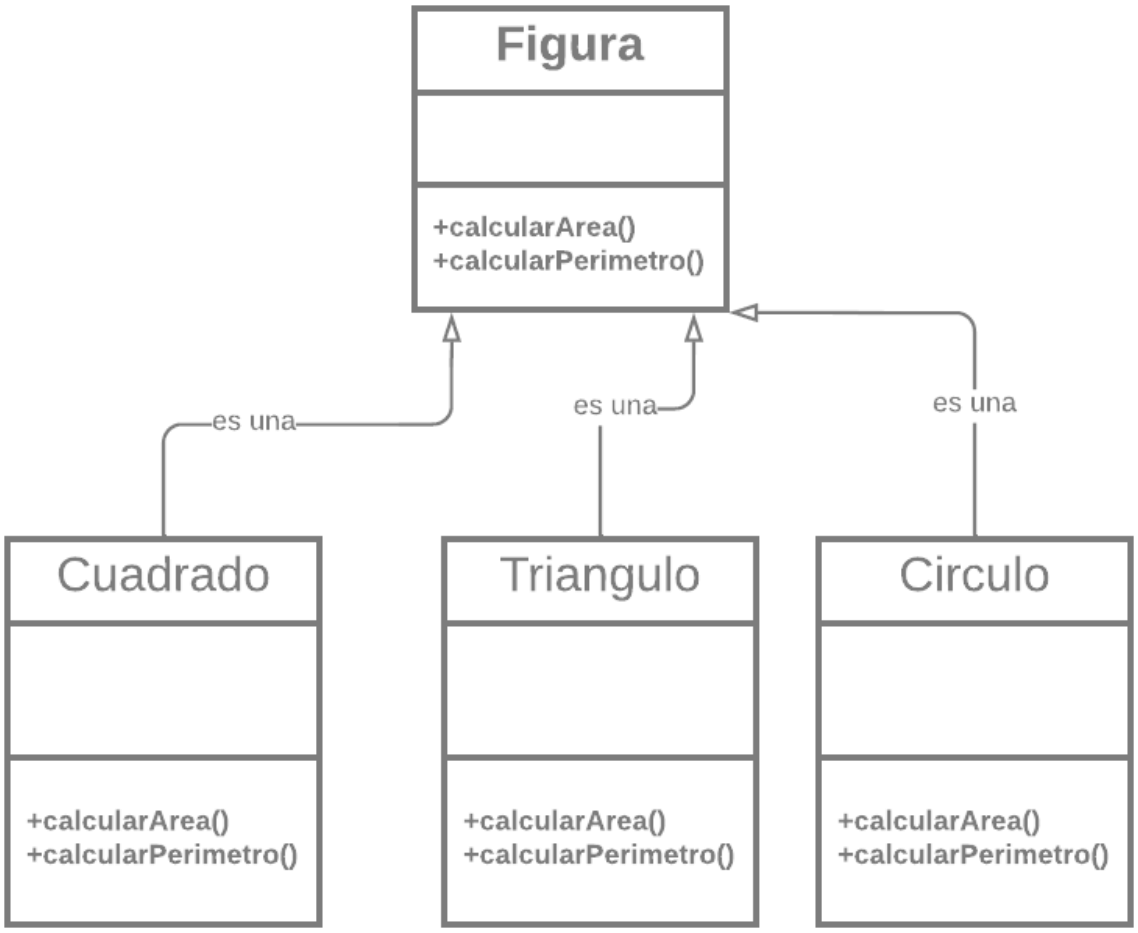
C:/Users/Usuario/Documents/NetBeansProjects/EjemploCompleto/src/ejemplocompleto/EjemploCompleto.java

```
1
2 package ejemplocompleto;
3
4 import java.time.LocalDate;
5 import java.time.Month;
6
7
8 public class EjemploCompleto {
9
10
11     public static void main(String[] args) {
12
13         //Crearemos un cliente
14         Cliente juan=new Cliente("Juan","López","San Luis",
15                 LocalDate.of(1976, Month.APRIL, 20));
16
17         Cuenta cuenta1=new Cuenta(101,juan);
18
19         cuenta1.depositar(1000);
20         System.out.println("Su saldo es: "+cuenta1.verSaldo());
21     }
22
23 }
```

Te propongo como desafío que modifiques el código de Cuenta para que puedas ver el último movimiento mostrando todos sus datos.

Herencia

El concepto de herencia se desarrollará en la siguiente guía. La herencia se representa con la siguiente flecha:



MANEJO DE FECHAS

Enumerados de mes y día de la semana

Existe en el API de java un enum en donde se definen los días de la semana: **java.time.DayOfWeek**

Por ejemplo:

Vamos a declarar una variable de tipo DayOfWeek y asignaremos a ella el día "Lunes".

DayOfWeek lunes = DayOfWeek.MONDAY;

Métodos que permiten manipular días hacia atrás y hacia adelante:

```
DayOfWeek lunes = DayOfWeek.MONDAY;
System.out.println("8 días será: "+lunes.plus(8));
System.out.println("2 días antes fue: "+lunes.minus(2));
```

Con el método **getDisplayName()** podemos obtener el nombre del día en un String dependiendo de la localización. Por ejemplo si queremos obtener el nombre del día en Argentina.

```
Locale arg=new Locale ("es", "AR");
DayOfWeek lunes=DayOfWeek.MONDAY;
System.out.println(lunes.getDisplayName(TextStyle.FULL, arg));
```

Para los meses, existe el enum **java.time.Month**

Por ejemplo:

```
Locale portugues=new Locale("pt");
Month mes=Month.MARCH;
System.out.println("2 meses más :" +mes.plus(2));
System.out.println("El mes anterior fue: "+mes.minus(1));
System.out.println("La cantidad de días de este mes es: "+mes.maxLength());
System.out.println("El nombre de este mes es: "+mes.getDisplayName(TextStyle.FULL,
portugués));
```

Clases de Fecha:

La clase de fecha **java.time.LocalDate** (java.time es el paquete en donde se encuentra esta clase), permite manejar fechas, pero a diferencia de java.util.Date, es que sólo trabaja fecha y no hora. Esto nos permite manipular fechas, para registrar fechas específicas.

Por ejemplo:

```
LocalDate fecha=LocalDate.of(2018,Month.MARCH,10); //10 de marzo de 2018.
DayOfWeek dia=fecha.getDayOfWeek();
System.out.println("La fecha es: "+fecha);
System.out.println("El día de la semana de fecha es: "+dia);
```

Si queremos obtener la fecha actual del sistema.

```
LocalDate fechaActual=LocalDate.now();
```

Si queremos convertir una cadena en una fecha:

```
String fechaCadena="20/04/2022";
```

Lo primero que podemos hacer es indicar cual es el patrón de la fecha que queremos convertir.

```
DateTimeFormatter forma=DateTimeFormatter.ofPattern("dd MM yyyy");
```

Luego convertimos (parseamos)

```
LocalDate fecha= LocalDate.parse(fechaCadena, forma);
```

CHRONOUNIT

Para obtener la diferencia entre dos fechas en una unidad de tiempo específica: días, meses o años, podemos utilizar el enum ChronoUnit que se encuentra en el paquete: java.time.temporal

Por ejemplo:

```
LocalDate fNacimiento=LocalDate.of(1998,Month.APRIL, 17);
LocalDate hoy=LocalDate.now();
System.out.println("Días vividos: "+ChronoUnit.DAYS.between(fNacimiento,hoy));
System.out.println("Meses vividos:"+ChronoUnit.MONTHS.between(fNacimiento,hoy));
System.out.println("Edad :"+ChronoUnit.YEARS.between(fNacimiento,hoy));
```

Ustedes pueden investigar acerca de las clases LocalTime y LocalDateTime.

BIBLIOGRAFIA:

Christopher Alexander, “A Pattern Language”, 1978

Erich Gamma, “Design Patterns: Elements of Reusable OO Software”

Martin Fowler, “Analysis Patterns: Reusable Object Models”, Addison Wesley,

1997

Kathy Sierra, “OCA/OCP JAVA SE 7 PROGRAMMER I & II STUDY GUIDE (EXAMS 1Z0-803 & 1Z0-8”, McGraw Hill, 2014