

ACTIVIDAD PRIMER Y SEGUNDO PARCIAL

PATRONES DE DISEÑO DE SOFTWARE

PEDRO DAVID ARNEDO ROMERO

**GRUPO 01
DEIVIS DE JESÚS MARTÍNEZ ACOSTA**

**UNIVERSIDAD POPULAR DEL CESAR
FACULTAD DE INGENIERIAS Y TECNOLOGIAS
PROGRAMA DE INGENIERÍA DE SISTEMAS**

**VALLEDUPAR - CESAR
2022**

ACTIVIDAD PARCIAL PRIMER Y SEGUNDO CORTE

1. Describa la diferencia e importancia de cada uno de los pilares de la programación orientada a objetos demostrando cada caso con un ejemplo práctico (descrito y en código).

R//: Ahora bien, pasare a hacer una descripción sobre los pilares de la programación orientada a objetos según lo dado en clases e investigación, y con ejemplos.

1) Abstracción

Principalmente tenemos que la abstracción cumple un papel fundamental en la programación orientada a objetos puesto que los programas suelen ser muy grandes y muchos objetos se suelen estar comunicando entre sí, por eso:

- Al implementarlo facilita el mantenimiento del código.
- Nos centramos en abstraer las características más importantes.
- Centrarnos en lo que hace antes de implementar.
- No entrar en detalles, basarnos en cosas simples para algo complejo.
- Separamos los datos que luego serán el molde (clase).
- Este es una extensión de la encapsulación.

2) Herencia

Tenemos que es una de las herramientas más poderosas de la programación orientada a objetos esta nos permite crear nuevas clases a partir de otras que ya existen, haciendo relaciones jerárquicas entre clases, por esto:

- Se obtienen nuevas clases derivadas a partir de una base.
- Se relacionan varias clases yendo de una general a otras más específicas.
- Las clases hijas heredaran los atributos y métodos de la clase padre.
- Evita repetir código.
- Acelera el desarrollo y ayuda la reutilización de código.

3) Encapsulamiento

Tenemos que la encapsulación, cada objeto es responsable de su información y de su estado, para que este pueda ser modificado es mediante los mismos métodos del objeto.

Por lo tanto, vemos que estos atributos internos del objeto no deben estar accesibles desde fuera, pudiéndose modificar sólo llamando a las funciones correspondientes.

- Esto mejora y simplifica la programación.
- Permite reutilizar código
- Todo lo del objeto está aislado y se tiene control de lo que sucede dentro de la misma clase
- Se protege la funcionalidad de la clase, desde adentro y afuera
- Aumenta la cohesión

4) Polimorfismo

Este concepto es un poco más técnico, aunque no tan difícil de explicar, el polimorfismo consiste en diseñar objetos para compartir comportamientos, lo que nos permite procesar objetos de diferentes maneras.

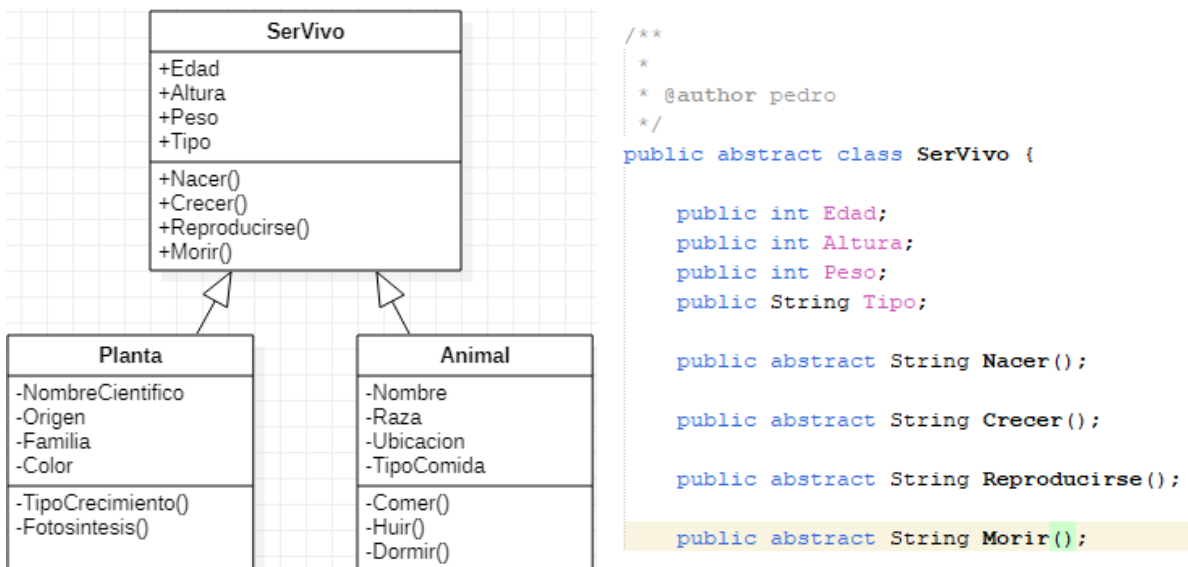
- Se utiliza para crear métodos con el mismo nombre, pero con diferente comportamiento.
- Evita el uso excesivo de condicionales.
- Simplifica la programación y da consistencia.
- varios objetos de diferentes clases, pero con una base común, se pueden usar de manera indistinta
- Esta muy ligado a la herencia
- Convierte cosas complejas en estructuras simples reproducibles.
- Evita la duplicación de código.

Diferencias

Las diferencias más notorias es en cuanto a, la abstracción, puesto que se encarga de obtener lo más importante y que será mostrado al usuario sin tanto detalle, y está relacionado con la encapsulación, a diferencia de la herencia y polimorfismo que la herencia se basa en heredar unos atributos o métodos a unas clases hijas, y el polimorfismo suele hacer lo mismo tener una clase con métodos los cuales serán implementados en las subclases, solo que en esta clase no se especifica que va a realizar los métodos, pero si en las subclases que la implementen.

Ejemplo

He hecho un ejemplo en UML, un diagrama para representar lo que es los 4 pilares, seguido de la misma implementación en código.



Lo que realice fue, abstracción de las características o atributos y métodos más importantes y las incluí en una clase padre.

Al tener dos clases que también incluyen estos mismos atributos aplicamos el concepto de herencia, estos incluyen y pueden hacer uso de los atributos y métodos.

Al mismo tiempo se realizó una encapsulación al proteger los atributos y algunos métodos para solo la clase tenga el control de ellos.

Y para finalizar el polimorfismo se usa al momento que las subclases implementan un método de una superclase, pero esta super clase no se especifica que va hacer, mientras que en las subclases es donde se va a especificar que va hacer concretamente.

```
*
* @author pedro
*/
public class Planta extends SerVivo {

    private String NombreCientifico;
    private String Origen;
    private String Familia;
    private String Color;

    private void TipoCrecimiento() {}

    private void Fotosintesis() {}

    @Override
    public String Nacer() {
        return "Nace por semilla" + "Tiene poca raiz" + "Es acuatica";
    }

    @Override
    public String Crecer() {
        return "Desarrolla sistema de defensa" + "Produce mas hojas";
    }

    @Override
    public String Reproducirse() {
        return "Tiene capacidad de producir muchas semillas";
    }

    @Override
    public String Morir() {
        return "Murio luego de ser comida de un animal";
    }
}
```

```
*
* @author pedro
*/
public class Animal extends SerVivo {

    private String Nombre;
    private String Raza;
    private String Ubicacion;
    private String TipoComida;

    private void Comer() {}

    private void Huir() {}

    private void Ubicacion() {}

    @Override
    public String Nacer() {
        return "Nace por huevo" + "Nace sano" + "Tiene cola" + "Es mamifero";
    }

    @Override
    public String Crecer() {
        return "El animal se adapta a la vida salvaje y aprende a cazar";
    }

    @Override
    public String Reproducirse() {
        return "No logro reproducirse";
    }

    @Override
    public String Morir() {
        return "Muerio al completar su ciclo de vida";
    }
}
```

3. Describa tres situaciones del mundo real donde puede aplicar el patrón de diseño de software SINGLETON.

R//: A continuación, procedo a describir 3 o más situaciones donde se aplique el patrón de diseño de SINGLETON.

- 1) Para este primer caso, tomemos un ejemplo bastante sencillo, estamos en una casa con varias personas, poniendo el caso que uno de ellos se compre un carro, dicho carro puede ser utilizado por cualquiera en la casa que sepa conducir, para salir en el carro solo se necesita que este esté parqueado en el garaje básicamente que esté disponible, dicho carro puede ser usado por cualquier persona de la familia y en este caso si implicamos el patrón de singleton, esto se vería de forma que cualquier persona puede realizar la acción de conducir carro, pero no es necesario crear instancias de más, ya que solo se necesita un objeto carro, siempre que alguien utilice el carro, se tomara el objeto que ya fue instanciado.
- 2) Para este segundo caso tomamos el ejemplo de una empresa, esta empresa posee un grupo de trabajadores, dichos trabajadores van a utilizar un objeto impresora para x o y motivos necesarios dependiendo su labor, teniendo en cuenta que el patrón singleton lo usamos para crear o instanciar un objeto una sola vez, en este caso los empleados solo necesitan una impresora, aplicándolo sería, tal que, al momento de codificar antes de realizar cualquier impresión utilizando la impresora, ya debe existir o haberse creado la instancia del objeto impresora, con singleton lo que hacemos es validar que solo se cree una vez este objeto impresora, y este mismo sea utilizado varias veces sin tener que crear otra vez la impresora.
- 3) En este caso nos pondremos en la situación de una panadería, entrando en contexto es bien sabido que una panadería normalmente cuenta con varios empleados, nos enfocamos en una que tenga varios panaderos, al tener varios trabajando al mismo tiempo o por turnos, ellos necesariamente para producir el producto necesitan unas herramientas, la más importante e indispensable es el horno, todos necesitan el horno para producir los panes, todos los chefs van a utilizar el horno, haciendo la comparación esto es singleton, al momento de llevarlo a la programación tenemos que en la vida real solo se necesita un horno para hacer cocinar panes, y un mismo horno para ser usado por varias personas, en este caso solo se crearía una sola vez el objeto horno, que será utilizado en más ocasiones y no es necesario estar creándolo porque el mismo cumple las funciones, no es necesario crearlo de nuevo el mismo horno.
- 4) Para este caso pondremos un ejemplo común como es el caso de un computador en una biblioteca pública, vemos que para dicho sistema se tendrán número indefinido de usuarios que pueden utilizar este recurso, muchos usuarios en momentos diferentes siempre van a estar haciendo consultas y entrando al sistema, sabiendo esto no es necesario cambiar o comprar un nuevo computador por cada usuario, para este caso también se implementa el singleton, con este validamos que en todo momento solo exista y este creada una única vez la instancia computador, y cada vez que un usuario realizara cualquier acción solo sería llamar el mismo objeto, no hace falta estar instanciándolo de nuevo.

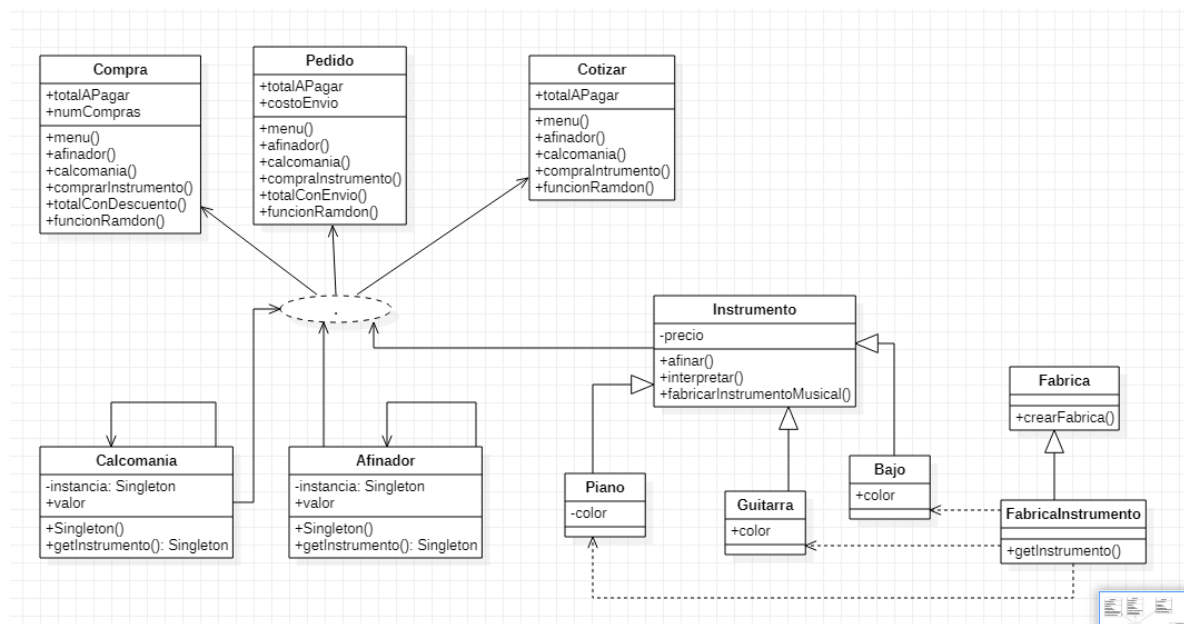
2. Usted hace parte de un equipo de desarrollo de software que está realizando la sistematización de una empresa encargada de producir instrumentos musicales (Guitarra, Bajo y Piano) con la acción de afinar y la de interpretar para ellos, y estos se pueden crear de forma directa en los módulos de pedido, orden de compra, facturación y cotizaciones, pero se presenta la necesidad de que puedan crear de forma aleatoria, también se pueden crear de forma específica según el usuario que haga el requerimiento, adicionalmente se estarán produciendo elementos complementarios a estos instrumentos que son accesorios como calcomanías y afinador de instancia única, se presenta la necesidad de que se puedan tener estos de tipo eléctricos y no eléctricos.

Plantea el o los patrones que puedes aplicar entre SINGLETON y FACTORY.

Elabora el diagrama de clases para la solución e impleméntalo en el lenguaje de programación de tu preferencia.

Documenta el código y donde apliques cualquier pilar de la programación orientada a objetos destácalo.

DIAGRAMA DE CLASES



DOCUMENTACION DEL CODIGO

Ahora pasare a realizar la explicación y documentación del código

- FABRICA

```

Source History
1 package pedro.gestioninstrumentos;
2
3 /**
4  *
5  * @author pedro
6  */
7
8 public class Fabrica {
9
10     public Instrumentos getInstrumento(String tipo) {
11         if (tipo.toUpperCase().equals("GUITARRA")) {
12             return new Guitarra();
13         } else {
14             if (tipo.toUpperCase().equals("BAJO")) {
15                 return new Bajo();
16             } else {
17                 if (tipo.toUpperCase().equals("PIANO")) {
18                     return new Piano();
19                 }
20             }
21         }
22         return null;
23     }
24 }

```

Lo primero será analizar la fábrica, con esta es que estaremos creando nuestros instrumentos, ya que no sabemos en qué momento o de qué tipo se requiere, las que tendremos serán los instrumentos guitarra, bajo y piano, y estos retornan una clase de ese tipo, por eso implementamos una fábrica que es una solución muy buena.

- Tipos Instrumentos

Fabrica.java	Piano.java	Bajo.java	Guitarra.java
<pre> 1 package pedro.gestioninstrumentos; 2 3 /** 4 * 5 * @author pedro 6 */ 7 8 public class Guitarra extends Instrumentos { 9 10 private String color = "NEGRA"; 11 12 @Override 13 public String afinar() { 14 return ""; 15 } 16 17 @Override 18 public String interpretar() { 19 return ""; 20 } 21 22 @Override 23 public String fabricarInstrumentoMusical() { 24 setPrecio(150000); 25 return "GUITARRA - " + color; 26 } 27 } </pre>	<pre> 1 package pedro.gestioninstrumentos; 2 3 /** 4 * 5 * @author pedro 6 */ 7 8 public class Bajo extends Instrumentos { 9 10 private String color = "NEGRO"; 11 12 @Override 13 public String afinar() { 14 return ""; 15 } 16 17 @Override 18 public String interpretar() { 19 return ""; 20 } 21 22 @Override 23 public String fabricarInstrumentoMusical() { 24 setPrecio(500000); 25 return "BAJO - " + color; 26 } 27 } </pre>	<pre> 1 package pedro.gestioninstrumentos; 2 3 /** 4 * 5 * @author pedro 6 */ 7 8 public class Piano extends Instrumentos { 9 10 private String color = "BLANCO"; 11 12 @Override 13 public String afinar() { 14 return ""; 15 } 16 17 @Override 18 public String interpretar() { 19 return ""; 20 } 21 22 @Override 23 public String fabricarInstrumentoMusical() { 24 setPrecio(100000); 25 return "PIANO - " + color; 26 } 27 } </pre>	<pre> 1 package pedro.gestioninstrumentos; 2 3 /** 4 * 5 * @author pedro 6 */ 7 8 public class Instrumentos { 9 10 private String color = "BLANCO"; 11 12 @Override 13 public String afinar() { 14 return ""; 15 } 16 17 @Override 18 public String interpretar() { 19 return ""; 20 } 21 22 @Override 23 public String fabricarInstrumentoMusical() { 24 return ""; 25 } 26 } </pre>

```

1 package pedro.gestioninstrumentos;
2
3 /**
4  *
5  * @author pedro
6  */
7
8 public class Piano extends Iinstrumentos {
9
10     private String color = "NEGRO";
11
12     @Override
13     public String afinar() {
14         return "";
15     }
16
17     @Override
18     public String interpretar() {
19         return "";
20     }
21
22     @Override
23     public String fabricarInstrumentoMusical() {
24         setPrecio(750000);
25         return "PIANO - " + color;
26     }
27
28 }

```

Se creo una clase por cada tipo de instrumento, estos heredaron de una clase padre, esta posee métodos que también son implementados en las clases hijas ya que estas son abstractas.

Tiene un atributo y con el método de fabricar instrumento es donde se le asigna el precio de ese instrumento, cada una con sus valores diferentes

- Clase abstracta instrumentos.

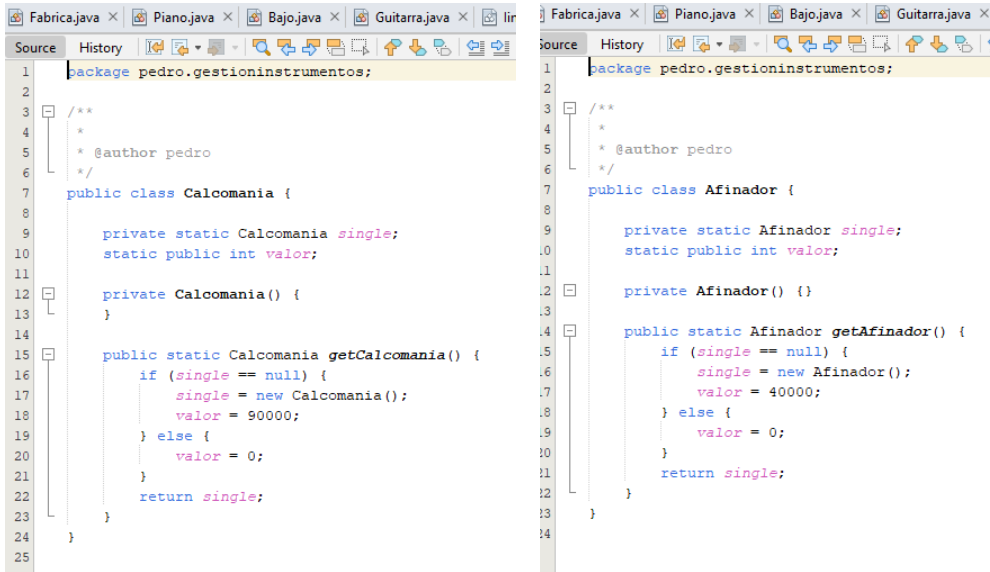
```

1 package pedro.gestioninstrumentos;
2
3 /**
4  *
5  * @author pedro
6  */
7
8 public abstract class Iinstrumentos {
9
10     private int precio;
11
12     public abstract String afinar();
13
14     public abstract String interpretar();
15
16     public abstract String fabricarInstrumentoMusical();
17
18     public int getPrecio() {
19         return precio;
20     }
21
22     public void setPrecio(int precio) {
23         this.precio = precio;
24     }
25
26 }

```


En esta clase es donde se encuentran los métodos abstractos, dichos métodos que las clases hijas van a implementar.

- Clases de calcomanía y afinador



The image shows two side-by-side screenshots of a Java IDE. The left screenshot displays the `Calcomania` class in the `pedro.gestioninstrumentos` package. It features a private static `Calcomania` instance named `single`, a static `int` variable `valor`, a private constructor, and a public static `getCalcomania()` method that implements the Singleton pattern. The right screenshot displays the `Afinador` class in the same package. It has a private static `Afinador` instance named `single`, a static `int` variable `valor`, a private constructor, and a public static `getAfinador()` method that also implements the Singleton pattern.

```
package pedro.gestioninstrumentos;

/**
 *
 * @author pedro
 */
public class Calcomania {

    private static Calcomania single;
    static public int valor;

    private Calcomania() {
    }

    public static Calcomania getCalcomania() {
        if (single == null) {
            single = new Calcomania();
            valor = 90000;
        } else {
            valor = 0;
        }
        return single;
    }
}

package pedro.gestioninstrumentos;

/**
 *
 * @author pedro
 */
public class Afinador {

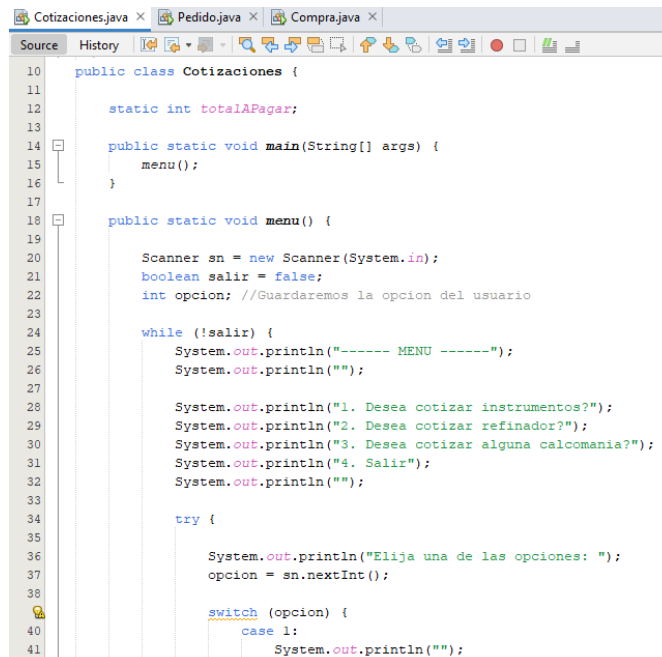
    private static Afinador single;
    static public int valor;

    private Afinador() {}

    public static Afinador getAfinador() {
        if (single == null) {
            single = new Afinador();
            valor = 40000;
        } else {
            valor = 0;
        }
        return single;
    }
}
```

En estas clases van definido los parámetros y los métodos que tiene cada uno, se le asigna por defecto el valor a cada uno de estos y este retornara dependiendo que este necesitando el usuario. Algo importante a destacar que estas están hechas con el patrón de diseño de Singleton, porque lo que se busca es que solo se cree una vez la instancia de cada uno de estos, tanto afinador como calcomanía.

- Módulos principales



The image shows a screenshot of a Java IDE with the `Cotizaciones.java` file open. The code defines a `Cotizaciones` class with a static `totalAPagar` variable, a `main` method that calls `menu()`, and a `menu()` method. The `menu()` method uses a `Scanner` to get user input, prints a menu with four options (cotizar instrumentos, cotizar refinador, cotizar calcomanía, and salir), and uses a `switch` statement to handle the user's choice.

```
public class Cotizaciones {

    static int totalAPagar;

    public static void main(String[] args) {
        menu();
    }

    public static void menu() {

        Scanner sn = new Scanner(System.in);
        boolean salir = false;
        int opcion; //Guardaremos la opcion del usuario

        while (!salir) {
            System.out.println("----- MENU -----");
            System.out.println("");

            System.out.println("1. Desea cotizar instrumentos?");
            System.out.println("2. Desea cotizar refinador?");
            System.out.println("3. Desea cotizar alguna calcomania?");
            System.out.println("4. Salir");
            System.out.println("");

            try {
                System.out.println("Elija una de las opciones: ");
                opcion = sn.nextInt();

                switch (opcion) {
                    case 1:
                        System.out.println("");

```

En este módulo se creó un menú de operaciones, tenemos que esta funcionalidad es aplicada a los demás módulos.

Que se hace en este menú, cuando llega un comprador, o que haga un pedido, o que se quiera cotizar, el sistema va a preguntar, que opción desea.

Lo primero pregunta si desea comprar instrumentos, y luego da la opción de elegir cuales son de su preferencia o si son escogidos de manera aleatoria.

```
izaciones.java x Pedido.java x Compra.java x
History
case 1:
    System.out.println("");
    System.out.println("Cuantos instrumentos que va a comprar");
    int cantidad = sn.nextInt();
    for (int i = 0; i < cantidad; i++) {
        System.out.println("");
        System.out.println("Elige 1 guitarra");
        System.out.println("Elige 2 bajo");
        System.out.println("Elige 3 piano");
        System.out.println("Elige 4 aleatorio");
        int tipoInstrumento = sn.nextInt();

        totalAPagar = totalAPagar + comprarInstrumento(tipoInstrumento);
    }
    System.out.println("Total a pagar por los instrumentos es: " + totalAPagar);
    System.out.println("");
    break;
case 2:
    Afinador();
    break;
case 3:
    Calcomania();
    break;
case 4:
    salir = true;
    break;
default:
    System.out.println("Solo números entre 1 y 4");
}
} catch (InputMismatchException e) {
    System.out.println("Debes insertar un número");
    sn.next();
}
```

```
izaciones.java x Pedido.java x Compra.java x
History
}
}
}

public static void Afinador() {
    Afinador afinador = Afinador.getAfinador();
    int valorAfinador = afinador.valor;
    totalAPagar = totalAPagar + valorAfinador;

    System.out.println("");
    if (valorAfinador == 0) {
        System.out.println("SOLO PUEDE LLEVAR UN AFINADOR");
    }
    System.out.println("Valor del afinador: " + valorAfinador);
    System.out.println("Total a pagar: " + totalAPagar);
    System.out.println("");
}

public static void Calcomania() {
    Calcomania calcomania = Calcomania.getCalcomania();
    int valorCalcomania = calcomania.valor;
    totalAPagar = totalAPagar + valorCalcomania;

    System.out.println("");
    if (valorCalcomania == 0) {
        System.out.println("SOLO PUEDE LLEVAR UNA CALCOMANIA");
    }
    System.out.println("Valor de la calcomania: " + valorCalcomania);
    System.out.println("Total a pagar: " + totalAPagar);
    System.out.println("");
}
}
```

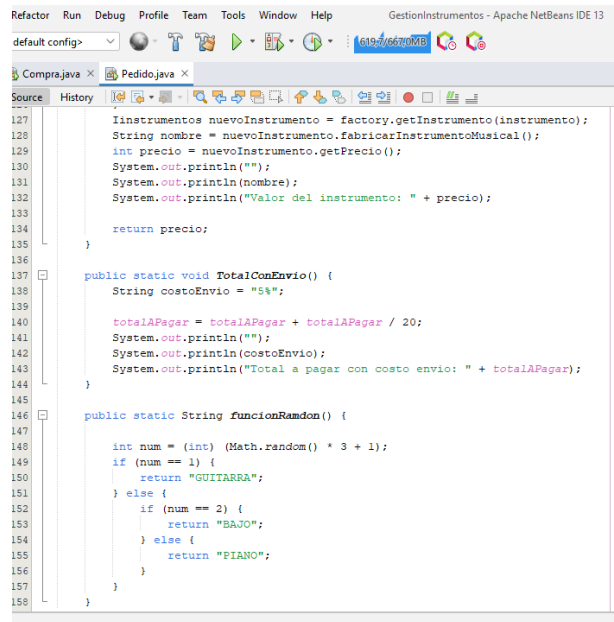
También tenemos el método afinador y calcomanía, estos están implementados con el patrón singleton, valida tal que solo se cree una única instancia.

```
Cotizaciones.java x Pedido.java x Compra.java x
Source History
11 System.out.println("");
12 }
13
14 public static int comprarInstrumento(int tipo) {
15     String instrumento = "";
16     int valorAPagar = 0;
17
18     Fabrica factory = new Fabrica();
19
20     if (tipo == 1) {
21         instrumento = "guitarra";
22     } else {
23         if (tipo == 2) {
24             instrumento = "bajo";
25         } else {
26             if (tipo == 3) {
27                 instrumento = "piano";
28             } else {
29                 instrumento = funcionRamdon();
30             }
31         }
32     }
33
34     Instrumentos nuevoInstrumento = factory.getInstrumento(instrumento);
35     String nombre = nuevoInstrumento.fabrica:InstrumentoMusical();
36     int precio = nuevoInstrumento.getPrecio();
37     System.out.println("");
38     System.out.println(nombre);
39     System.out.println("Valor del instrumento: " + precio);
40
41     return precio;
42 }
43
44 public static String funcionRamdon() {
```

Ahora tenemos el método comprar instrumento que se encarga de seleccionarme que tipo de instrumento es que se esta comprando en el momento, si se va la opción de elegirlo manualmente o de manera aleatoria.

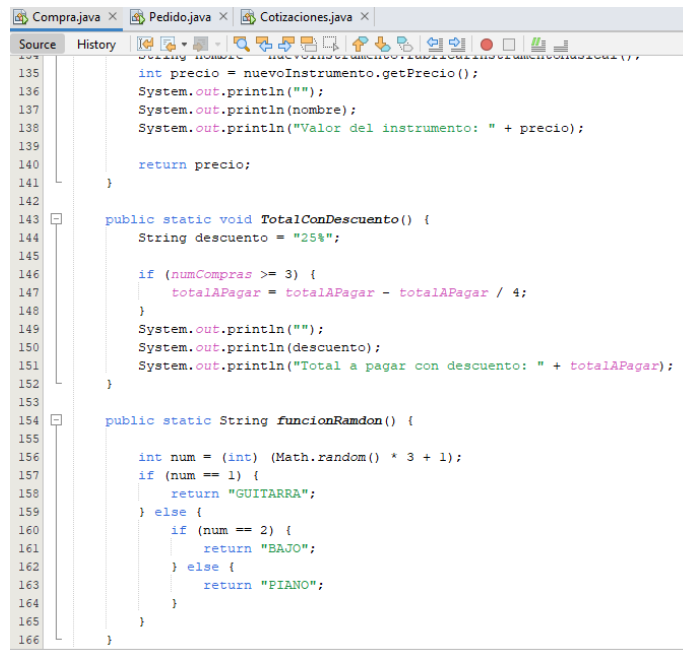
```
Cotizaciones.java x Pedido.java x Compra.java x
Source History
126 System.out.println("");
127 System.out.println(nombre);
128 System.out.println("Valor del instrumento: " + precio);
129
130 return precio;
131 }
132
133 public static String funcionRamdon() {
134
135     int num = (int) (Math.random() * 3 + 1);
136     if (num == 1) {
137         return "GUITARRA";
138     } else {
139         if (num == 2) {
140             return "BAJO";
141         } else {
142             return "PIANO";
143         }
144     }
145 }
146
147 }
```

Con esta función creo un numero aleatorio para elegir un instrumento si así lo requiere el comprador.



```
Refactor Run Debug Profile Team Tools Window Help GestionInstrumentos - Apache NetBeans IDE 13
default config>
Comprajava x Pedido.java x
Source History
127 Instrumentos nuevoInstrumento = factory.getInstrumento(instrumento);
128 String nombre = nuevoInstrumento.fabricarInstrumentoMusical();
129 int precio = nuevoInstrumento.getPrecio();
130 System.out.println("");
131 System.out.println(nombre);
132 System.out.println("Valor del instrumento: " + precio);
133
134 return precio;
135 }
136
137 public static void TotalConEnvio() {
138     String costoEnvio = "5%";
139
140     totalAPagar = totalAPagar + totalAPagar / 20;
141     System.out.println("");
142     System.out.println(costoEnvio);
143     System.out.println("Total a pagar con costo envio: " + totalAPagar);
144 }
145
146 public static String funcionRandom() {
147
148     int num = (int) (Math.random() * 3 + 1);
149     if (num == 1) {
150         return "GUITARRA";
151     } else {
152         if (num == 2) {
153             return "BAJO";
154         } else {
155             return "PIANO";
156         }
157     }
158 }
```

Este método es añadido para el modulo de pedido, en pedidos se realizo lo mismo, que cotizaciones solo que le puse un pequeño recargo por el envío.



```
Comprajava x Pedido.java x Cotizaciones.java x
Source History
135 int precio = nuevoInstrumento.getPrecio();
136 System.out.println("");
137 System.out.println(nombre);
138 System.out.println("Valor del instrumento: " + precio);
139
140 return precio;
141 }
142
143 public static void TotalConDescuento() {
144     String descuento = "25%";
145
146     if (numCompras >= 3) {
147         totalAPagar = totalAPagar - totalAPagar / 4;
148     }
149     System.out.println("");
150     System.out.println(descuento);
151     System.out.println("Total a pagar con descuento: " + totalAPagar);
152 }
153
154 public static String funcionRamdon() {
155
156     int num = (int) (Math.random() * 3 + 1);
157     if (num == 1) {
158         return "GUITARRA";
159     } else {
160         if (num == 2) {
161             return "BAJO";
162         } else {
163             return "PIANO";
164         }
165     }
166 }
```

Para el modulo de compra, le añadí un método de descuento, como bono si se llega a realizar la compra de varios instrumentos.