uts

Unidades Tecnológicas de Santander


EII
SEMILLERO DE
INVESTIGACIÓN
DE INGENIERÍA
DE SISTEMAS


GRUPO DE INVESTIGACIÓN
EN INGENIERÍA DE SOFTWARE
GRIIS

PATRONES DE DISEÑO DE SOFTWARE

¡APLICACIONES ORIENTADAS A OBJETOS!

Elaborado por

BRAYAN NICOLAS ACEROS GUERRERO
LENIN FABIÁN BILIVAR ROJAS
SERGIO IVÁN DÍAZ VEGA
DIEGO ALEXANDER GARCÍA ECHEVERRÍA
ALBERT STEPHEN GERENA CASTELLANOS
JHON MARIO MEZA RIOS
JOSE ANTONIO OTÁLORA PORRAS
SERGIO IVÁN VILLAMIZAR LUNA

Director proyecto de investigación

ELIECER MONTERO OJEDA PhD.
LÍDER SEMILLERO DE INVESTIGACIÓN SEIIS

Revisión

ERWIN MEZA VEGA MSc.

PROYECTO DE INVESTIGACIÓN
INGENIERÍA DE SISTEMAS
FACULTAD DE CIENCIAS
NATURALES E INGENIERÍAS

FICHA TECNICA				
Título	**Cartilla de actividades para la aplicación de los patrones de diseño de software básico para los estudiantes del programa de Ingeniería de Sistemas de las Unidades Tecnológicas de Santander.**			
Objetivo:	Apoyar la apropiación del conocimiento de los patrones de diseño de software y su aplicación en el diseño y construcción de aplicaciones orientada a objetos			
Fecha de inicio	Marzo de 2020			
Fecha de Terminación	En curso			
Investigador principal	Eliecer Montero Ojeda, Investigador y líder del semillero de Investigación de Ingeniería de Sistemas - SEIIS -			
Línea de investigación	Arquitectura y Diseño de Software			
Grupo de investigación	Grupo de Investigación en Ingeniería de Software – GRISS -			
Estudiantes participantes	Tecnología		Ingeniería	X
Sede UTS	Bucaramanga			
Programa Académico	Ingeniería de Sistemas			
Asignatura	Patrones de Diseño de Software			
Semestre	Décimo			
No. estudiantes participantes	8 (Ver Anexo No 1)			

Los Patrones de Diseño de software son soluciones para problemas típicos y recurrentes que nos podemos encontrar a la hora de desarrollar una aplicación orientada a objetos. Aunque nuestra aplicación sea única, tendrá cosas comunes con otras aplicaciones ya desarrolladas: Acceso a datos, creación de clases y objetos, operaciones entre diferentes sistemas, etc.

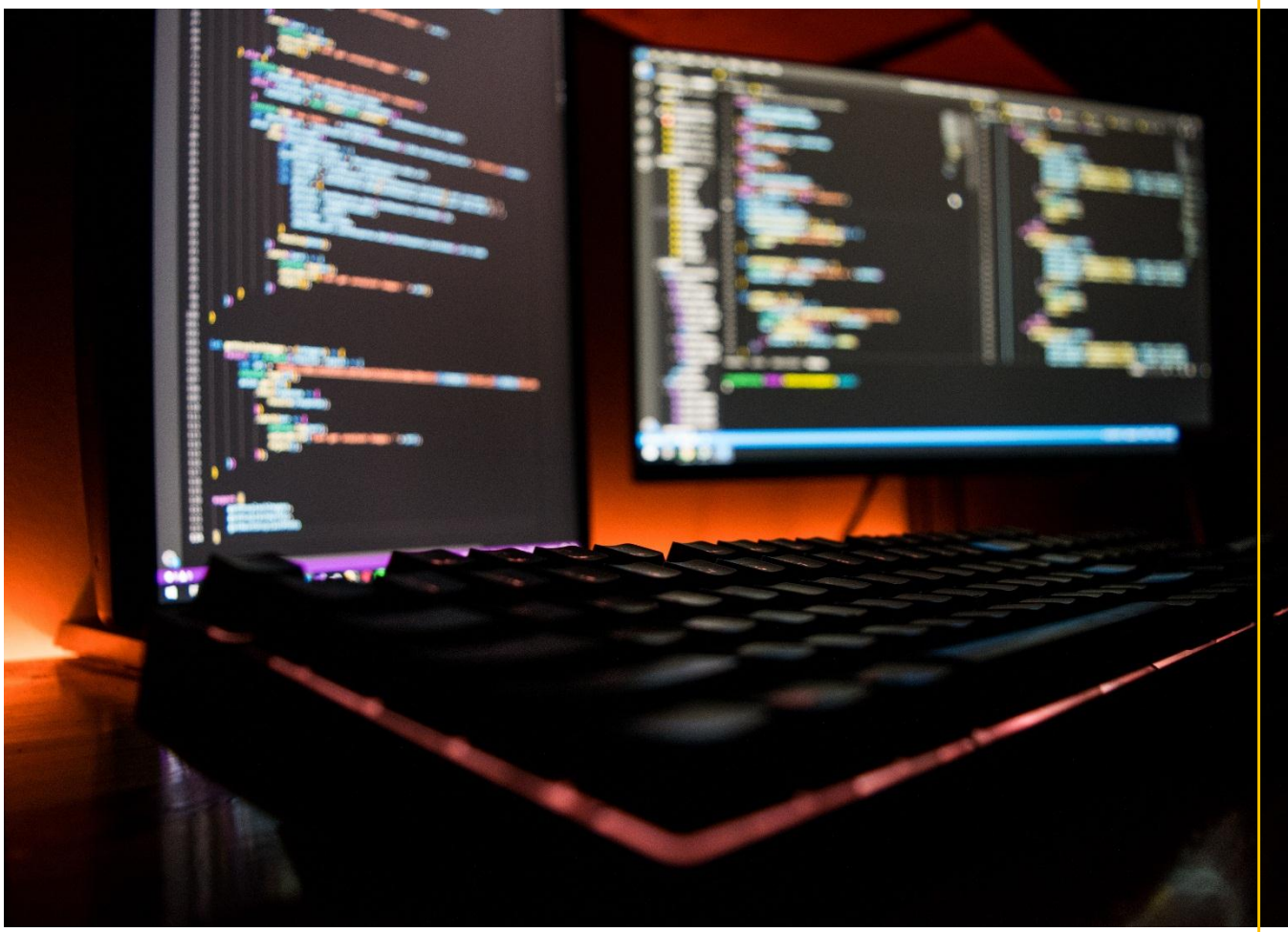


Photo by Fotis Fotopoulos on Unsplash

Los patrones de diseño son muy útiles por muchos motivos: Ahorran tiempo; te ayudan a estar seguro de la validez de su código; establecen un lenguaje común entre todos los miembros del equipo.

Los patrones de diseño se dividen en distintos grupos, según el tipo de problema que solucionan.

Photo by Markus Spiske on Unsplash

Creational Patterns

- Abstract Factory
- Builder
- Factory Methis
- Prototype
- Singleton

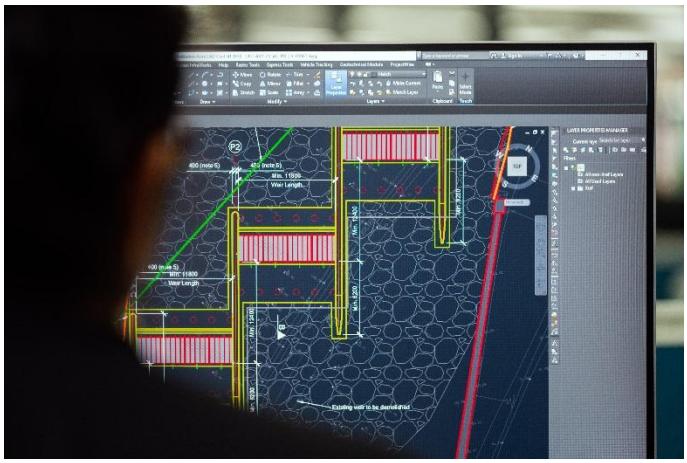


Photo by ThisisEngineering RAEng on Unsplash

Structural Patterns

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

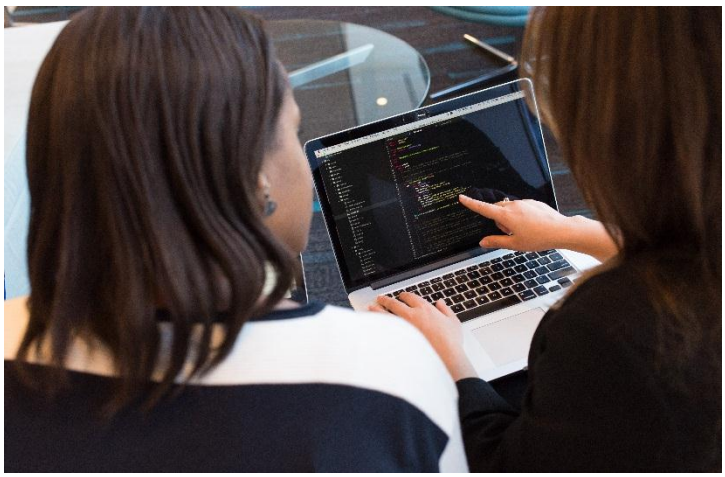
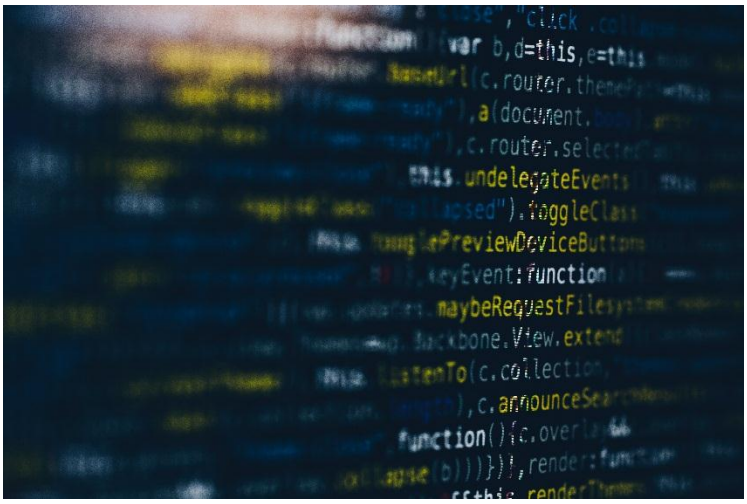


Photo by Christina @ wocintechchat.com on Unsplash

Behavioral Patterns

- Command
- Chain
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

CREATIONAL PATTERNS



Creational Patterns

- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton

Son los que facilitan la tarea de creación de nuevos objetos, de tal forma que el proceso de creación pueda ser desacoplado de la implementación del resto del sistema.

- **Creacional de la Clase**

  Los patrones creacionales de Clases usan la herencia como un mecanismo para lograr la instanciación de la Clase. Por ejemplo el Factory Method.

- **Creacional del Objeto**

  Los patrones creacionales de objetos son más escalables y dinámicos comparados de los patrones creacionales de Clases. Por ejemplo la Abstract Factory y el patrón Singleton.

ABSTRACT FACTORY

El objetivo de este patrón de diseño es la creación de objetos agrupados en familias; sin tener que conocer las clases concretas destinadas a la creación de los objetos.

DIAGRAMA DE CLASES

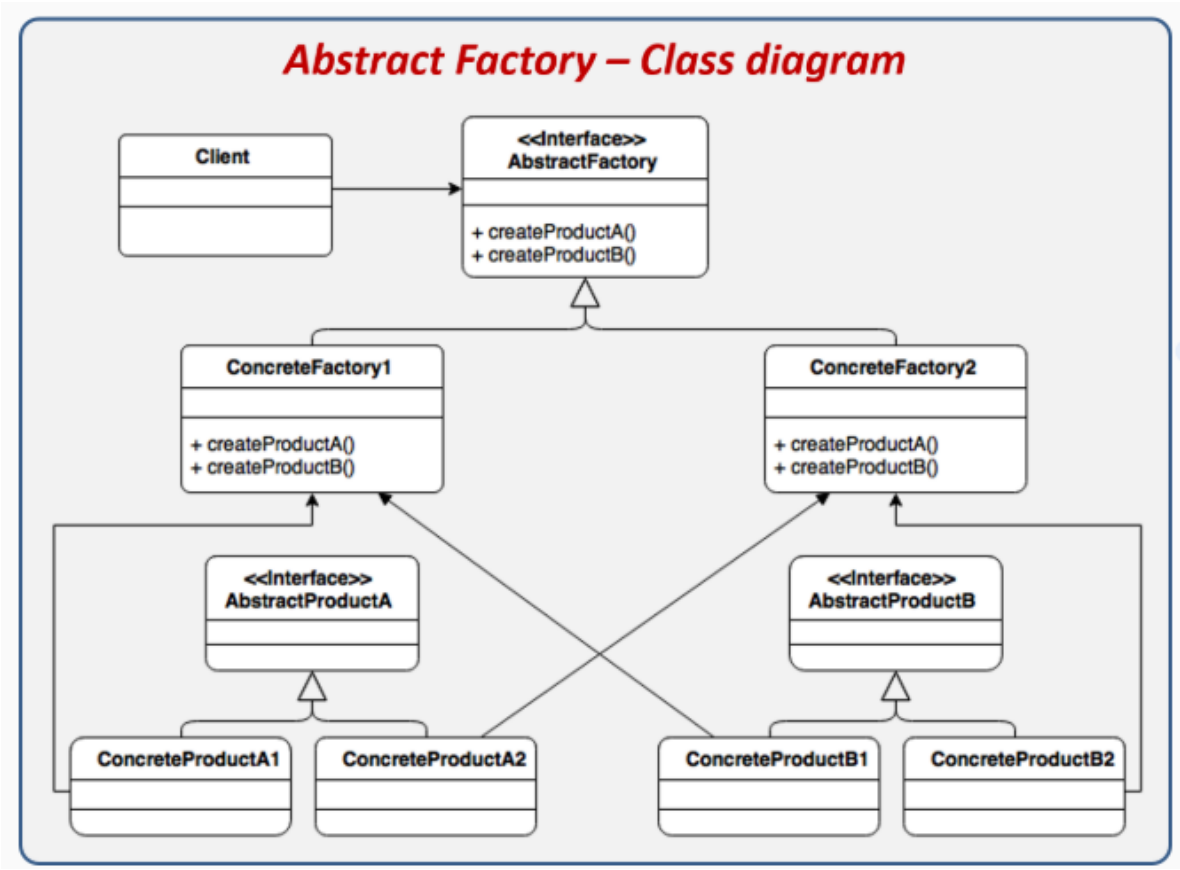


Diagrama de Clases Genérico

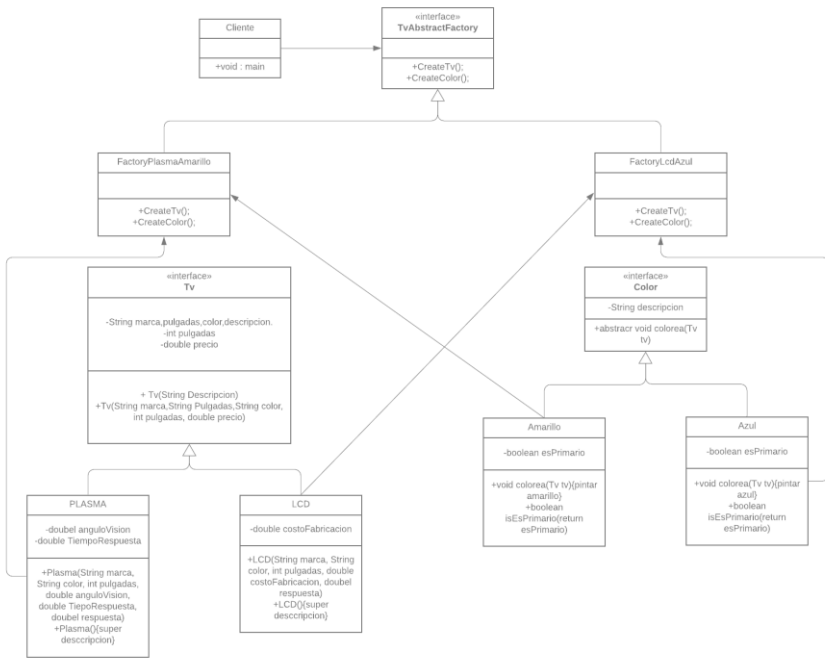
APLICACIÓN

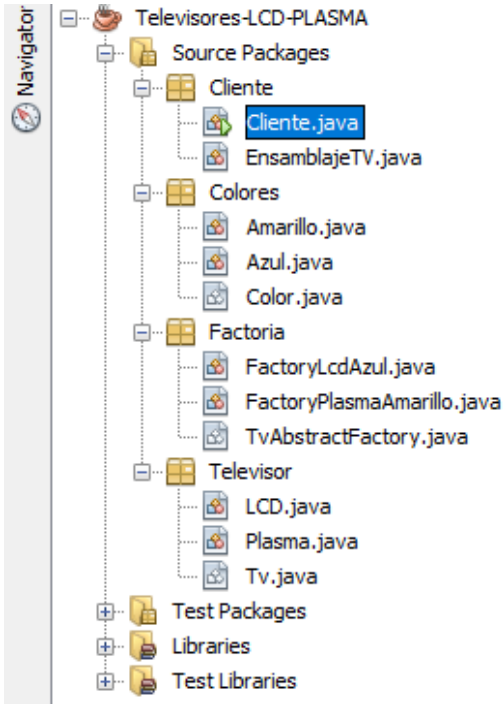
Hagamos de cuenta que tenemos dos familias de objetos:

1) La clase TV, que tiene dos hijas: Plasma y LCD.

2) La clase Color, que tiene dos hijas: Amarillo y Azul (¡los mejores colores, sin duda!).

Más allá de todos los atributos/métodos que puedan tener la clase Color y TV, lo importante aquí es destacar que Color define un método abstracto. Escenario: nuestra empresa se dedica a darle un formato estético específico a los televisores LCD y Plasma. Se ha decidido que todos los LCD que saldrán al mercado serán azules y los plasma serán amarillos. Ahora bien, una solución simple sería en la clase Azul colocar el LCD y en la clase Amarillo colocar el Plasma y todo funcionaría de maravillas. ¿Cuál sería el problema? Que esta todo hardcodeado. Esto quiere decir que el hecho de que los LCD sean azules y los plasmas amarillos es una decisión del negocio y, como tal, puede variar (y de hecho el negocio varía constantemente). Por ejemplo, que pasa si mañana se desea agrega otro color o me cambian el color del LCD o mucho peor, ¿qué pasa si se crea otro producto LED y también se lo quiere pintar de Azul?





Estructura de Carpetas

Clase Tv Abstract

```
package Televisor;

public  abstract class Tv {
    private String marca;
    private int pulgadas;
    private String color;
    private String descripcion;
    private double precio;

    public Tv(String descripcion) {
        this.descripcion = descripcion;
    }
    public Tv(String marca, int pulgadas, String color, double precio) {
        this.marca = marca;
        this.pulgadas = pulgadas;
        this.color = color;
        this.precio = precio;
    }

    public String getMarca() {...3 lines }
    public void setMarca(String marca) {...3 lines }
    public int getPulgadas() {...3 lines }
    public void setPulgadas(int pulgadas) {...3 lines }
    public String getColor() {...3 lines }
    public void setColor(String color) {...3 lines }
    public String getDescripcion() {...3 lines }
    public void setDescripcion(String descripcion) {...3 lines }
    public double getPrecio() {...3 lines }
    public void setPrecio(double precio) {...3 lines }
}
```

Clase Color Abstract

```
package Colores;

import Televisor.Tv;

public abstract class Color {
    private String descripcion;
    public abstract void colorea(Tv tv);

    public String getDescripcion() {
        return descripcion;
    }

    public void setDescripcion(String descripcion) {
        this.descripcion = descripcion;
    }
}
```

Clase Plasma

```
package Televisor;

public class Plasma extends Tv {
    private double anguloVision;
    private double tiempoRespuesta;
    public Plasma(double anguloVision, double tiempoRespuesta, String marca,
            int pulgadas, String color, double precio) {
        super(marca, pulgadas, color, precio);
        this.anguloVision = anguloVision;
        this.tiempoRespuesta = tiempoRespuesta;
    }
    public Plasma() {
        super("Plasma...Proximamente sera LED");
    }
    public double getAnguloVision() {...3 lines }
    public void setAnguloVision(double anguloVision) {...3 lines }
    public double getTiempoRespuesta() {...3 lines }
    public void setTiempoRespuesta(double tiempoRespuesta) {...3 lines }
}
```

Clase LCD

```
package Televisor;
public class LCD extends Tv{
    private double costoFabricacion;
    public LCD(double costoFabricacion, String marca, int pulgadas, String color, double precio) {
        super(marca, pulgadas, color, precio);
        this.costoFabricacion = costoFabricacion;
    }
    public LCD() {
        super("LCD");
    }
    public double getCostoFabricacion() {...3 lines }
    public void setCostoFabricacion(double costoFabricacion) {...3 lines }
}
```

Clase Azul

```
package Colores;

import Televisor.Tv;

public class Azul extends Color{
     boolean esPrimario;
    @Override
    public void colorea(Tv tv) {
        System.out.println("Pintando de azul el " + tv.getDescripcion());
    }
    public boolean isEsPrimario(){
        return esPrimario;
    }
    public void setEsPrimario(boolean esPrimario) {
        this.esPrimario = esPrimario;
    }
}
```

Clase Amarillo

```
package Colores;

import Televisor.Tv;

public class Amarillo extends Color {
    boolean esPrimario;
    @Override
    public void colorea(Tv tv) {
        System.out.println("Pintando de amarillo el " + tv.getDescripcion());
    }
    public boolean isEsPrimario(){
        return esPrimario;
    }
    public void setEsPrimario(boolean esPrimario) {
        this.esPrimario = esPrimario;
    }
```

Para evitar un dolor de cabeza conviene separar estas familias y utilizar el Abstract Factory:

Clase TvAbstractFactory Abstract

```
package Factoria;

import Colores.Color;
import Televisor.Tv;

public abstract class TvAbstractFactory {
    public abstract Tv createTV();
    public abstract Color createColor();
}
```

Y los Factory Concretos, que relacionan las familias:

Clase FactoryPlasmaAmarillo

```
package Factoria;

import Colores.Amarillo;
import Colores.Color;
import Televisor.Plasma;
import Televisor.Tv;

public class FactoryPlasmaAmarillo  extends  TvAbstractFactory{
    @Override
    public Tv createTV() {
        return new Plasma();
    }
    @Override
    public Color createColor() {
        return new Amarillo();
    }
}
```

Clase FactoryLcdAzul

```
package Factoria;
import Colores.Azul;
import Colores.Color;
import Televisor.LCD;
import Televisor.Tv;
public class FactoryLcdAzul extends TvAbstractFactory{
    @Override
    public Tv createTV() {
        return new LCD();
    }
    @Override
    public Color createColor() {
        return new Azul();
    }
}
```

Para ordenar un poco las cosas se va a crear un gestor de factorías

Clase EnsamblajeTV

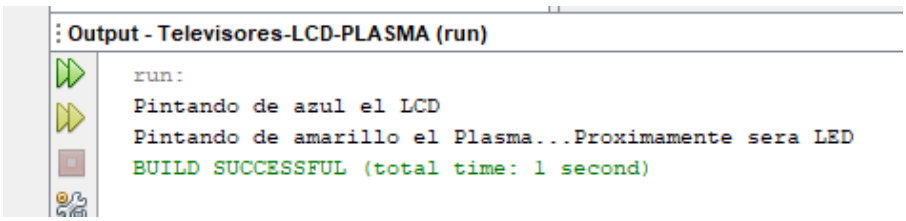
```
package Cliente;

import Colores.Color;
import Televisor.Tv;
import Factoria.TvAbstractFactory;

public class EnsamblajeTV {
    public EnsamblajeTV(TvAbstractFactory factory){
    Color color=factory.createColor();
    Tv tv = factory.createTV();
    color.colorea(tv);
    }
}
```

Y, por último, el main:

Clase Cliente (main)

```
package Cliente;
import Factoria.FactoryLcdAzul;
import Factoria.FactoryPlasmaAmarillo;
import Factoria.TvAbstractFactory;

public class Cliente {
    public static void main(String[] args) {
        //probando el factory LCD + Azul
        TvAbstractFactory f1 = new FactoryLcdAzul();
        EnsamblajeTV e= new EnsamblajeTV(f1);
        //probndo el factory plasma + Amarillo
        TvAbstractFactory f2= new FactoryPlasmaAmarillo();
        EnsamblajeTV e2= new EnsamblajeTV(f2);
    }
}
```

ALGUNAS PRUEBAS….



BUILDER

Este patrón tiene por objetivo facilitar la instanciación de objetos de una determinada clase o familia de clases, entregando objetos cuyos atributos posean valores definidos sin recurrir a su llenado mediante constructores ni métodos de tipo setter en otras palabras Este es un patrón que es muy simple pero muy útil, el cual nos permite crear objetos complejos a través de uno más simple.

DIAGRAMA DE CLASES

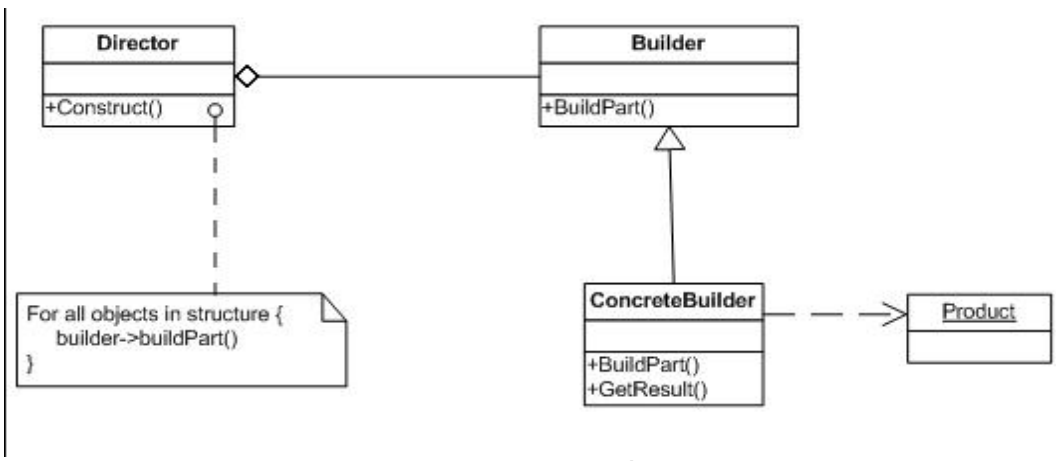
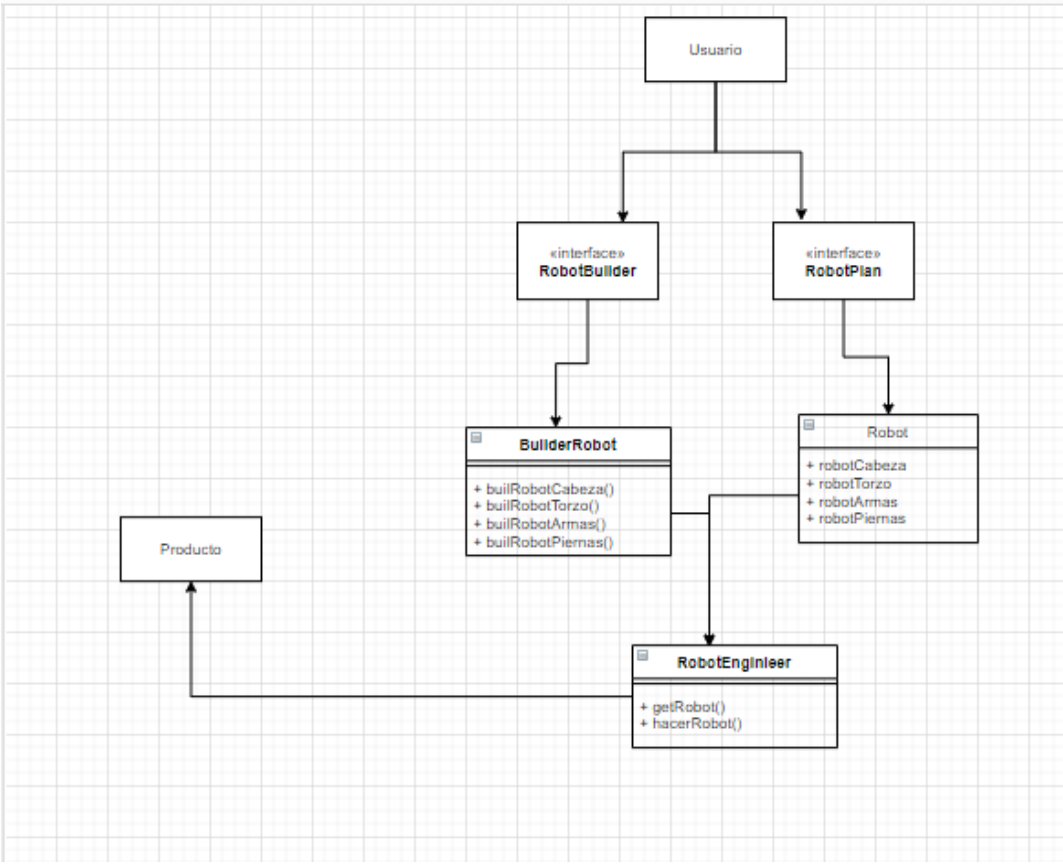


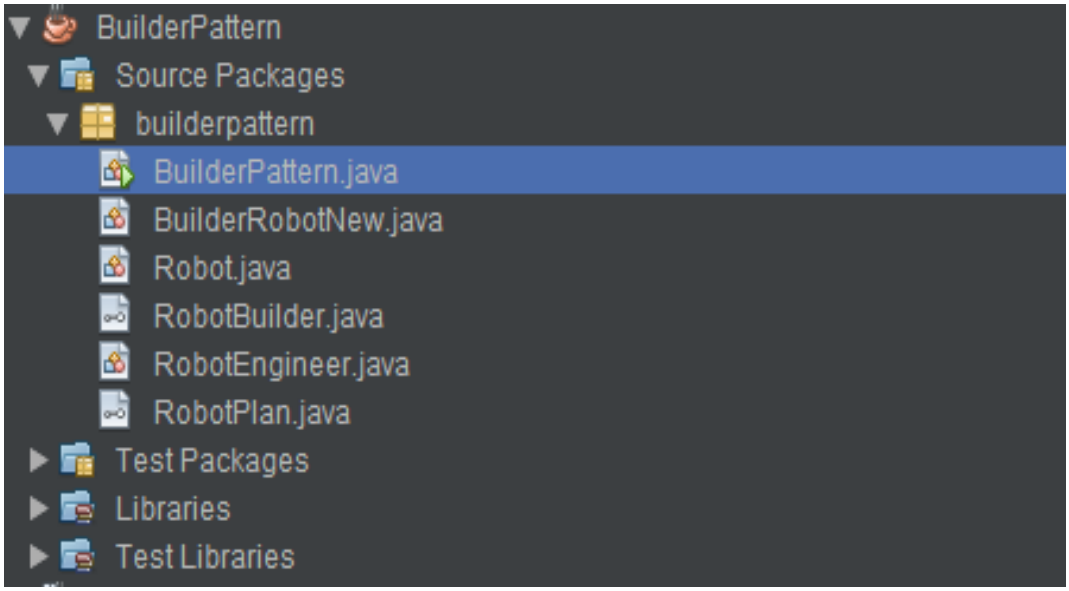
Diagrama de Clases Genérico

APLICACIÓN

Se requiere arma un robot con diferentes partes y poder ensamblar haciendo uso del patrón de diseño builder. Al implementar este patrón garantizamos que partes queremos ensamblar y en qué orden podemos ensamblar el robot



En este diagrama se representa como el usuario crea el robot, utilizando las interfaces constructor y interfaz plan que permite ordenar la construcción del robot



Estructura de Carpetas

Interface RobotPlan

```java
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package builderpattern;

/**
 *
 * @author LENOVO
 */
public interface RobotPlan {

    public void setRobotCabeza(String cabeza);

    public void setRobotTorso(String torso);

    public void setRobotArmas(String armas);

    public void setRobotPiernas(String piernas);

}
```

Clase RobotEngineer

```
public class RobotEngineer {

    private RobotBuilder robotBuilder;

    public RobotEngineer(RobotBuilder robotBuilder) {
        this.robotBuilder = robotBuilder;
    }

    public Robot getRobot() {
        return this.robotBuilder.getRobot();
    }

    public void hacerRobot() {

        this.robotBuilder.buildRobotCabeza();

        this.robotBuilder.buildRobotTorso();

        this.robotBuilder.buildRobotArmas();

        this.robotBuilder.buildRobotPiernas();

    }

}
```

Interface RobotBuilder

```
public interface RobotBuilder {

    public void buildRobotCabeza();

    public void buildRobotTorso();

    public void buildRobotArmas();

    public void buildRobotPiernas();

    public Robot getRobot();

}
```

Clase BuilderPattern

```
public class BuilderPattern {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        RobotBuilder antiguoStiloRobot = new BuilderRobotNew();

        // Pase la especificación BuilderRobotNew a la ingeniera
        RobotEngineer robotEngineer = new RobotEngineer(antiguoStiloRobot);

        // Se le dice al ingeniero que haga el robot usando las especificaciones
        // de la clase  BuilderRobotNew
        robotEngineer.hacerRobot();

        // El ingeniero devuelve el robot correcto basado en la especificación
        Robot firstRobot = robotEngineer.getRobot();

        System.out.println("Construcción Robot");

        System.out.println("Tipo de cabeza de robot: " + firstRobot.getRobotCabeza());

        System.out.println("Tipo de torso robot: " + firstRobot.getRobotTorso());

        System.out.println("Tipo de armas robot: " + firstRobot.getRobotArmas());

        System.out.println("Tipo de piernas robot: " + firstRobot.getRobotPiernas());

    }

}
```

Clase Robot

```
public class Robot implements RobotPlan {

    private String robotCabeza;

    private String robotTorso;

    private String robotArmas;

    private String robotPiernas;


    public void setRobotCabeza(String cabeza) {
        robotCabeza = cabeza;
    }


    public void setRobotTorso(String torso) {
        robotTorso = torso;
    }


    public void setRobotArmas(String armas) {
        robotArmas = armas;
    }


    public void setRobotPiernas(String piernas) {
        robotPiernas = piernas;
    }

    public String getRobotCabeza() {
        return robotCabeza;
    }

    public String getRobotTorso() {
        return robotTorso;
    }

    public String getRobotArmas() {
        return robotArmas;
    }

    public String getRobotPiernas() {
        return robotPiernas;
    }


}
```

Clase BuilderRobotNew

```
public class BuilderRobotNew implements RobotBuilder {

    private Robot robot;

    public BuilderRobotNew() {
        this.robot = new Robot();
    }

    @Override
    public void buildRobotCabeza() {
        this.robot.setRobotCabeza("CABEZA REDONDA");
    }

    @Override
    public void buildRobotTorso() {
        this.robot.setRobotTorso("TORZO ANCHO");
    }

    @Override
    public void buildRobotArmas() {
        this.robot.setRobotArmas("PISTOLAS Y CAÑONES");
    }

    @Override
    public void buildRobotPiernas() {
        this.robot.setRobotPiernas("PRIENAS LARGAS");
    }

    public Robot getRobot() {
        return this.robot;
    }

}
```

Clases BuilderPattern (main)

```
public class BuilderPattern {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        RobotBuilder antiguoStiloRobot = new BuilderRobotNew();

        // Pase la especificación BuilderRobotNew a la ingeniera
        RobotEngineer robotEngineer = new RobotEngineer(antiguoStiloRobot);

        // Se le dice al ingeniero que haga el robot usando las especificaciones
        // de la clase  BuilderRobotNew
        robotEngineer.hacerRobot();

        // El ingeniero devuelve el robot correcto basado en la especificación
        Robot firstRobot = robotEngineer.getRobot();

        System.out.println("Construcción Robot");

        System.out.println("Tipo de cabeza de robot: " + firstRobot.getRobotCabeza());

        System.out.println("Tipo de torso robot: " + firstRobot.getRobotTorso());

        System.out.println("Tipo de armas robot: " + firstRobot.getRobotArmas());

        System.out.println("Tipo de piernas robot: " + firstRobot.getRobotPiernas());

    }

}
```

ALGUNAS PRUEBAS….

```
run:
Construcción Robot
Tipo de cabeza de robot: CABEZA REDONDA
Tipo de torso robot: TORZO ANCHO
Tipo de armas robot: PISTOLAS Y CAÑONES
Tipo de piernas robot: PRIENAS LARGAS
BUILD SUCCESSFUL (total time: 0 seconds)
```

En este resultado podemos visualizar como se ha creado el robot con sus diferentes partes y los diferentes tipos de accesorios

FACTORY METHOD

Define una interfaz para crear un objeto, pero deja en manos de las subclases la decisión de qué clase concreta e instancie.

DIAGRAMA DE CLASES

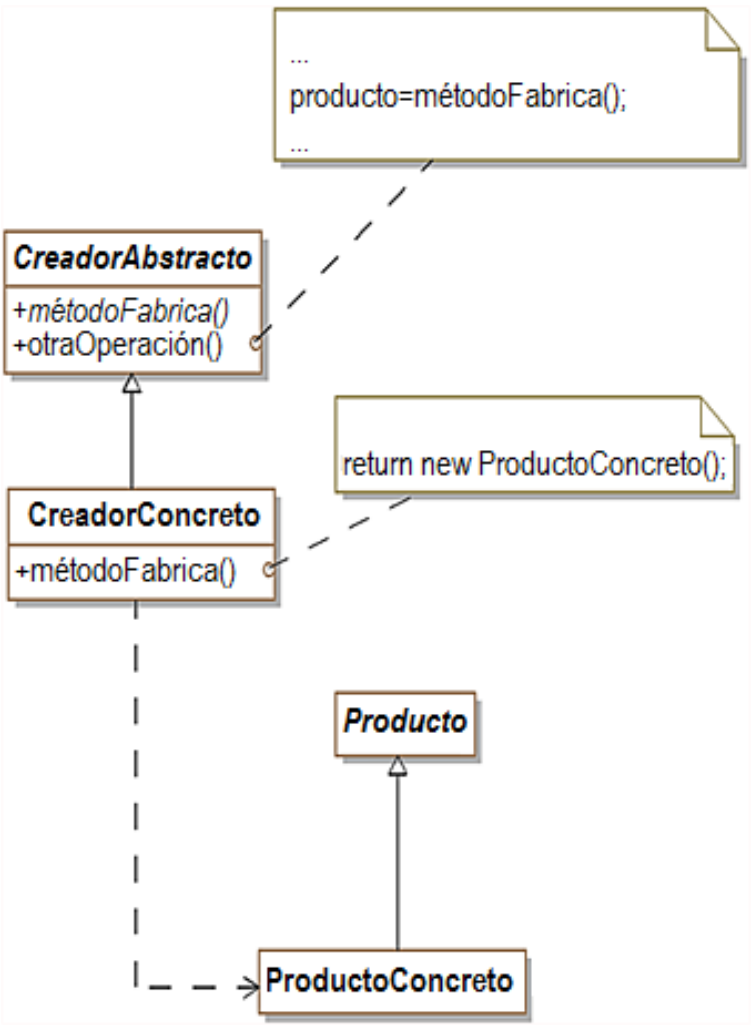
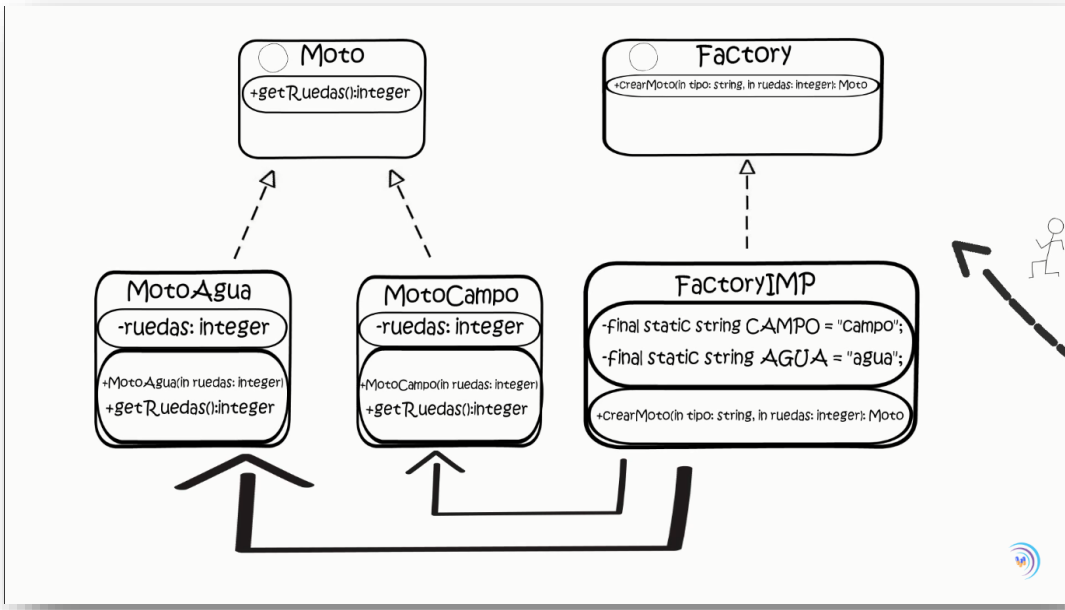
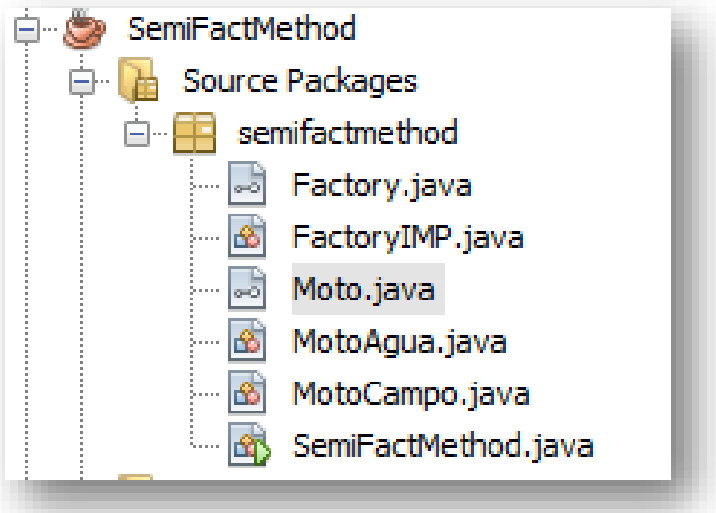


Diagrama de Clases Genérico

APLICACIÓN

Se desea ensamblar las piezas de una moto que permita su construcción para varios terrenos de acuerdo a las especificaciones del cliente.





Estructura de Carpetas

Clase MotoCampo

```
public class MotoCampo implements Moto{

      int ruedas;

    public MotoCampo(int ruedas){
        this.ruedas = ruedas;
    }

    public int getRuedas(){
        return this.ruedas;
    }
}
```

Clase MotoAgua

```
public class MotoAgua implements Moto {

     int ruedas;

    public MotoAgua(int ruedas){
        this.ruedas = ruedas;
    }

    public int getRuedas(){
        return this.ruedas;
    }
}
```

Interface Moto

```
public interface Moto {
    public int getRuedas();
}
```

Interface Factory

```
public interface Factory {
    public Moto creaMoto(String tipo, int ruedas);
}
```

Clase FactoryIMP

```
public class FactoryIMP implements Factory{

    public final static String AGUA = "agua";
    public final static String CAMPO = "campo";

    public Moto creaMoto(String tipo, int ruedas){
        switch(tipo){
            case AGUA:return new MotoAgua(ruedas);
            case CAMPO:return new MotoCampo(ruedas);
            default:return null;
        }
    }
}
```

Clase SemiFactMethod (main)

```
public class SemiFactMethod {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {

            String tipoMoto = "campo";
            int numRuedas= 4;
            Factory mifactoria = new FactoryIMP();
            Moto miMoto = mifactoria.creaMoto(tipoMoto, numRuedas);
            System.out.println("Es una moto de " + miMoto.getRuedas()+ " ruedas." );
    }
}
```

ALGUNAS PRUEBAS….

```
run:
Es una moto de 4 ruedas.
BUILD SUCCESSFUL (total time: 1 second)
```

PROTOTYPE

Prototipo es un patrón de diseño de software creacional cuyo primordial objetivo es crear a partir de un modelo; el funcionamiento de este patrón es simple ya que su propósito consiste en realizar una o varias copias exactas de otro objeto, permitiendo la utilización de atributos ya existentes sin requerir la creación de nuevos objetos. (Gamma, Vlissides, Johnson, & Helm, 1994)

DIAGRAMA DE CLASES

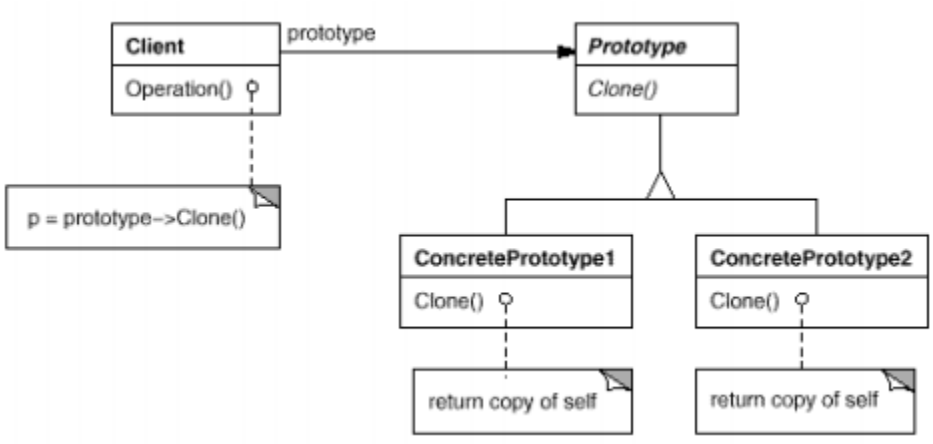
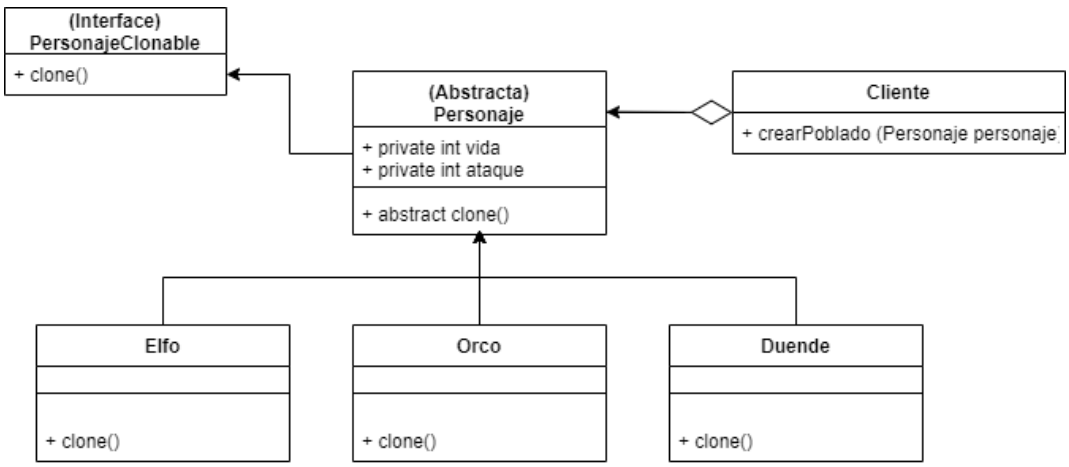
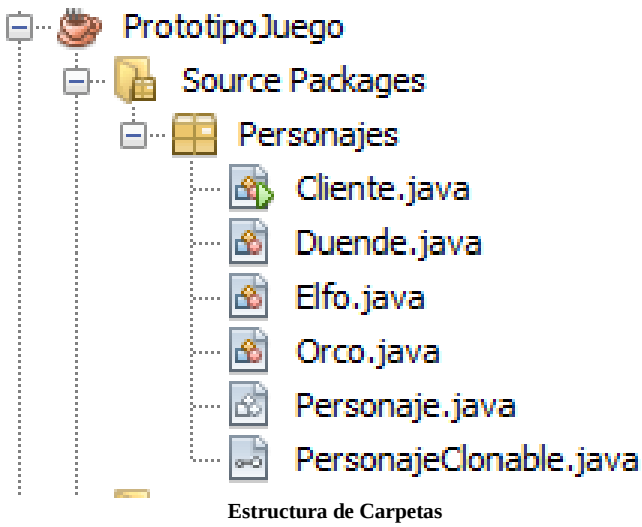


Diagrama de Clases Genérico

APLICACIÓN

Para un videojuego de roles es necesaria la creación de tres tipos de personajes (Elfos, Duendes y Orcos) donde cada rol tendrá un puntaje de vida y ataque particular.





Estructura de Carpetas

Clase Personaje

La clase Personaje es una clase abstracta que define los atributos vida y ataque que estarán presentes en todos los tipos de rol, en esta clase implementa la clase interfaz clonable (PersonajeClonabble) y la creación de todos los personajes se realizara a través de esta.

```java
package Personajes;

/**
 *
 * @author agerena
 */
public abstract class Personaje implements PersonajeClonable{

    public int vida;
    public int ataque;

    public Personaje(int vida, int ataque){

        this.vida = vida;
        this.ataque = ataque;

    }

    @Override
    public abstract PersonajeClonable clone ();

}
```

Clase PersonajeClonable

La clase PersonajeClonable es de tipo interfaz clonable y en ella se declara el método clon que se utilizara para la creación de los prototipos desde la clase Personajes.

```java
package Personajes;

/**
 *
 * @author agerena
 */
public interface PersonajeClonable extends Cloneable{

    public abstract PersonajeClonable clone();

}
```

Clase Orco

La clase Orco cuenta con los atributos propios del tipo de rol, esta clase es extendida a la clase Personajes y mediante el método clon realiza la creación de los personajes cuyo rol es Orco, asignando los puntajes de vida y ataque definidos,

```java
package Personajes;

/**
 *
 * @author agerena
 */
public class Orco extends Personaje{

    public static final int VIDA = 90;
    public static final int ATAQUE = 80;

    public Orco (){
        super(VIDA,ATAQUE);
    }

    @Override
    public PersonajeClonable clone (){
        Orco clon = new Orco();
        clon.vida = this.vida;
        clon.ataque = this.ataque;
        return clon;
    }

    @Override
    public String toString(){
        return "orco";
    }
}
```

Clase Elfo

La clase Elfo cuenta con los atributos propios del tipo de rol, esta clase es extendida a la clase Personajes y mediante el método clon realiza la creación de los personajes cuyo rol es Elfo, asignando los puntajes de vida y ataque definidos.

```java
package Personajes;

/**
 *
 * @author agerena
 */
public class Elfo extends Personaje{

    public static final int VIDA = 100;
    public static final int ATAQUE = 120;

    public Elfo (){
        super(VIDA,ATAQUE);
    }

    @Override
    public PersonajeClonable clone (){
        Elfo clon = new Elfo();
        clon.vida = this.vida;
        clon.ataque = this.ataque;
        return clon;
    }

    @Override
    public String toString(){
        return "elfo";
    }
}
```

Clase Duende

La clase Duende cuenta con los atributos propios del tipo de rol, esta clase es extendida a la clase Personajes y mediante el método clon realiza la creación de los personajes cuyo rol es Duende, asignando los puntajes de vida y ataque definidos.

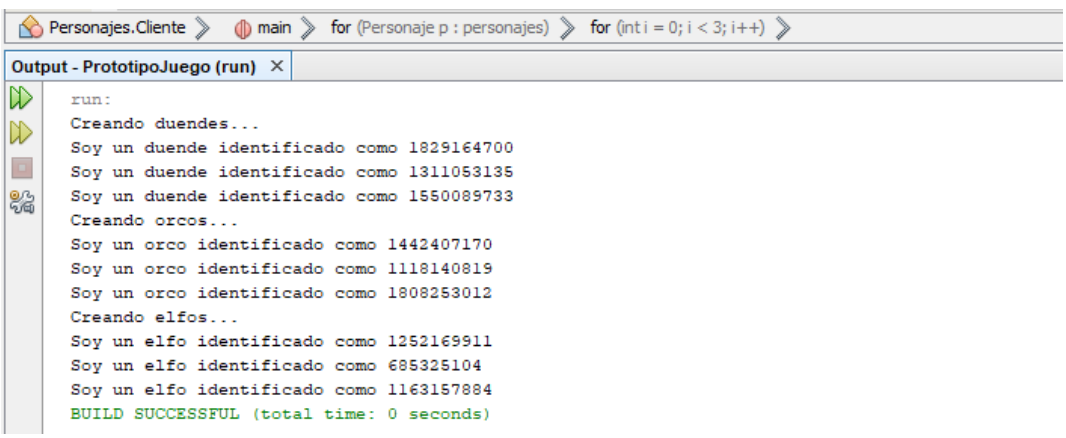
```java
package Personajes;

/**
 *
 * @author agerena
 */

public class Duende extends Personaje{

    public static final int VIDA = 70;
    public static final int ATAQUE = 50;

    public Duende (){
        super(VIDA,ATAQUE);
    }

    @Override
    public PersonajeClonable clone (){
        Duende clon = new Duende();
        clon.vida = this.vida;
        clon.ataque = this.ataque;
        return clon;
    }

    @Override
    public String toString(){
        return "duende";
    }
}
```

Clase Cliente (main)

A través de la clase cliente se realiza la creación del número de personajes definido por el usuario llamando el método de cada tipo de rol, para la creación de los personajes se agrega un código hash que permita identificar que aunque los personajes tienen igual nombre y puntajes cuentan con un identificador propio siendo una clon idéntico.

```
1  package Personajes;
2
3  /**
4   *
5   * @author agerena
6   */
7  public class Cliente {
8
9      public static void main(String[] args) {
10
11         Personaje[] personajes = new Personaje[3];
12         personajes[0] = new Duende();
13         personajes[1] = new Orco();
14         personajes[2] = new Elfo();
15
16         for (Personaje p : personajes) {
17             System.out.println("Creando " + p.toString() + "s...");
18             for (int i = 0; i < 3; i++) {
19                 System.out.println("Soy un " + p.clone() + " identificado como "
                   + System.identityHashCode(System.identityHashCode(p.clone())));
21             }
22
23         }
24     }
25
26 }
```

ALGUNAS PRUEBAS….



SINGLETON

El objetivo de este patrón es el de crear una única instancia de cada sistema, de esta manera evitando la duplicación de registros dentro de este, o ejecución múltiple de procedimientos del mismo, con ello logrando establecer un punto de acceso global a la clase, con el fin de evitar errores de confiabilidad y veracidad en los datos o registros.

DIAGRAMA DE CLASES

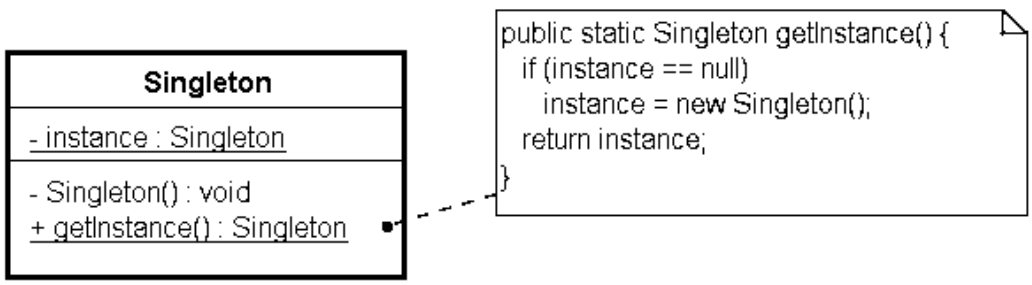
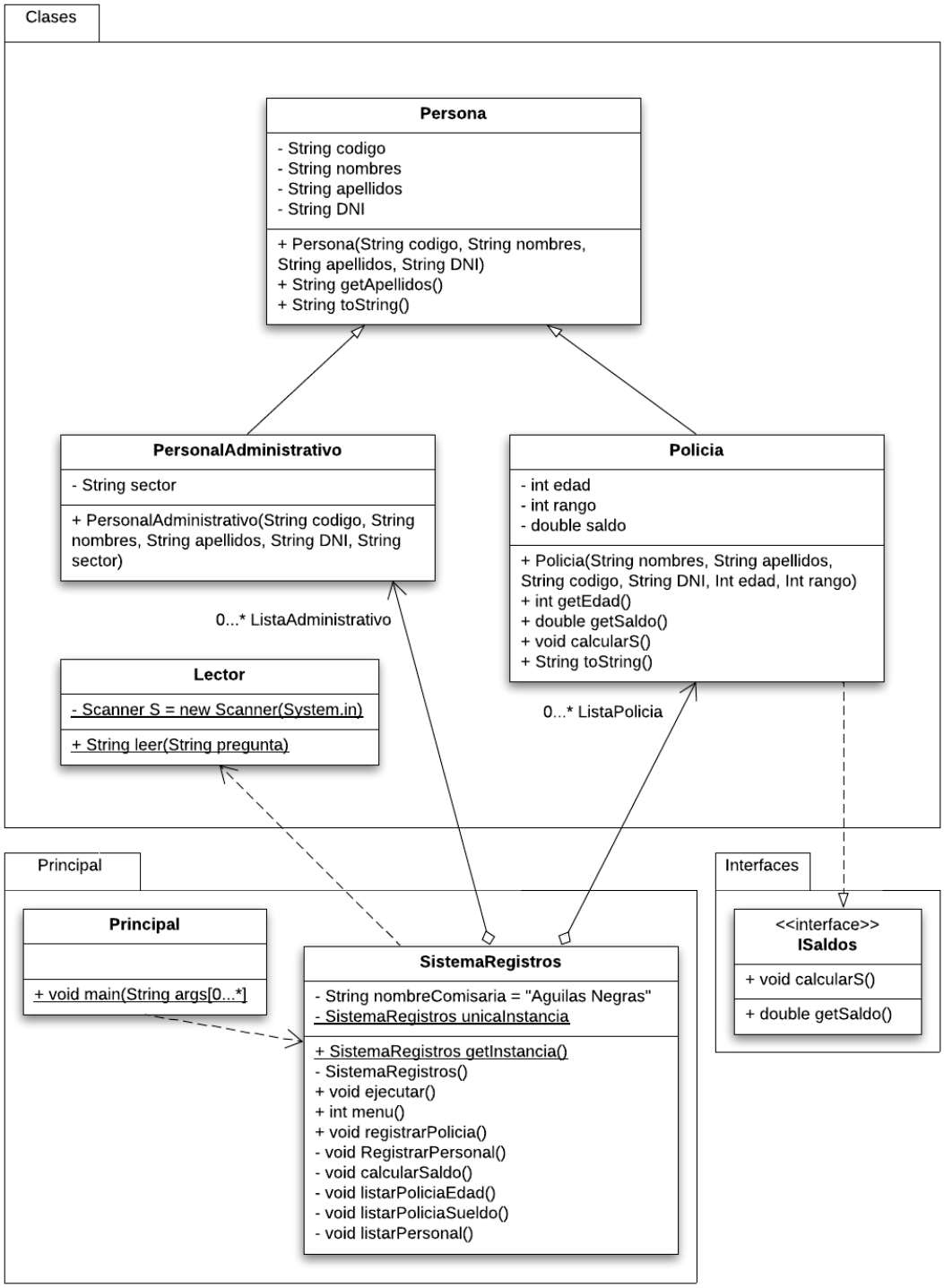
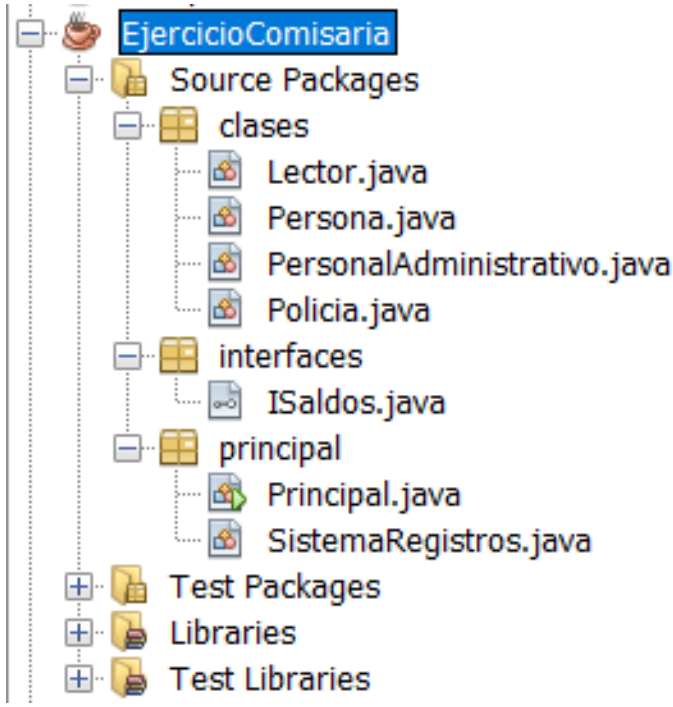


Diagrama de Clases Genérico

APLICACIÓN

Se requiere desarrollar un sistema en el cual se pueda llevar un registro del personal policial y administrativo, este debe tener la posibilidad de calcular los sueldo de los empleados (policías) y listar dichos registros, el fin de este patrón es evitar que hayan múltiples instancias de las clases ya que podrían haber conflictos, duplicación de registros, esto trae consigo afectación en la veracidad de la información, también ver que el patrón singleton puede ser aplicado en proyectos grandes como el de este ejemplo, si en este ejemplo no se aplicara el patrón singleton el problema que traería consigo seria la perdida de información, uno de estos errores seria cálculos errados en los sueldos de los policías.
Al implementar este patrón se garantiza el acceso global a la clase sin necesidad de instanciar de manera múltiple la clase, y de esta manera poder acceder a los métodos y variables dentro de la misma.


Clases
Persona
- String codigo
- String nombres
- String apellidos
- String DNI
+ Persona(String codigo, String nombres, String apellidos, String DNI)
+ String getApellidos()
+ String toString()
PersonalAdministrativo
- String sector
+ PersonalAdministrativo(String codigo, String nombres, String apellidos, String DNI, String sector)
Policia
- int edad
- int rango
- double saldo
+ Policia(String nombres, String apellidos, String codigo, String DNI, Int edad, Int rango)
+ int getEdad()
+ double getSaldo()
+ void calcularS()
+ String toString()
0...* ListaAdministrativo
0...* ListaPolicia
Lector
- Scanner S = new Scanner(System.in)
+ String leer(String pregunta)
Principal
Principal
+ void main(String args[0...*]
SistemaRegistros
- String nombreComisaria = "Aguilas Negras"
- SistemaRegistros unicaInstancia
+ SistemaRegistros getInstancia()
- SistemaRegistros()
+ void ejecutar()
+ int menu()
+ void registrarPolicia()
- void RegistrarPersonal()
- void calcularSaldo()
- void listarPoliciaEdad()
- void listarPoliciaSueldo()
- void listarPersonal()
Interfaces
<<interface>>
ISaldos
+ void calcularS()
+ double getSaldo()



Estructura de Carpetas

Clase Persona

```java
package clases;

public class Persona{
    private String codigo;
    private String nombres;
    private String apellidos;
    private String DNI;

    public Persona(String codigo, String nombres, String apellidos, String DNI) {
        this.codigo = codigo;
        this.nombres = nombres;
        this.apellidos = apellidos;
        this.DNI = DNI;
    }

    public String getApellidos() {
        return apellidos;
    }

    @Override
    public String toString() {
        return "\nCódigo: "+codigo
               +"\nNombres: "+nombres
               +"\nApellidos: "+apellidos
               +"\nDNI: "+DNI;
    }

}
```

Clase PersonalAdministrativo

```java
package clases;

public class PersonalAdministrativo extends Persona {
        private String sector;

    public PersonalAdministrativo(String codigo, String nombres, String apellidos, String DNI, String sector) {
        super(codigo, nombres, apellidos, DNI);
        this.sector = sector;
    }

}

```

Clase Policia

```java
package clases;

import interfaces.ISaldos;

public class Policia extends Persona implements ISaldos {
    private int edad;
    private int rango;
    private double saldo;

    public Policia(String nombres, String apellidos, String codigo, String DNI, int edad, int rango) {
        super(codigo, nombres, apellidos, DNI);
        this.edad=edad;
        this.rango=rango;
    }

    public int getEdad() {
        return edad;
    }

    @Override
    public double getSaldo() {
        return saldo;
    }

    @Override
    public void calcularS() {
        saldo = 120 * rango;
    }

    @Override
    public String toString() {
        return super.toString()
                +"\nEdad: "+edad
                +"\nRango: "+rango
                +"\nSaldo: "+saldo;
    }
}
```

Interface ISaldos

```java
package interfaces;

public interface ISaldos {
    public void calcularS();
    public double getSaldo();
}
```

Clase Lector

```java
package clases;

import java.util.Scanner;

public class Lector { // Wrapper.

    private static Scanner S = new Scanner(System.in);

    public static String leer(String pregunta) {
        System.out.println(pregunta);
        return S.nextLine();
    }

}
```

Clase Principal

```java
package principal;

public class Principal {

    public static void main(String[] args) {
        SistemaRegistros.getInstancia().ejecutar();
//          new SistemaRegistros().ejecutar();
//          SistemaRegistros S = new SistemaRegistros();
    }

}
```

Clase SistemaRegistros

```java
package principal;

import clases.Lector;
import clases.PersonalAdministrativo;
import clases.Policia;
import java.util.ArrayList;

public class SistemaRegistros {

    private static SistemaRegistros unicaInstancia;

    public static SistemaRegistros getInstancia() {
        if(unicaInstancia == null)
            unicaInstancia = new SistemaRegistros();
        return unicaInstancia;
    }

    private SistemaRegistros() {}

    private ArrayList<Policia> ListaPolicia = new ArrayList();
    private ArrayList<PersonalAdministrativo> ListaAdministrativo = new ArrayList();
    private final String nombreComisaria = "AGUILAS NEGRAS";

    public void ejecutar() {
        int opt;
        do{
            opt = menu();
            switch (opt){
                case 1:
                    registrarPolicia();
                    break;
                case 2:
                    registrarPersonal();
                    break;
                case 3:
                    calcularSaldo();
                    break;
                case 4:
                    listarPoliciaEdad();
                    break;
                case 5:
                    listarPoliciaSueldo();
                    break;
                case 6:
                    listarPersonal();
                    break;
```

```
47                  case 7:
48                      System.out.println("Saliendo del programa");
49                      break;
50                  default:
51                      System.out.println("La opcion no es valida");
52              }
53
54          }while (opt!=7);
55      }
56
57      public int  menu(){
58              String opciones=
59                  "\n MENU PINCIPAL DE LA COMISARÍA "+nombreComisaria+
60                  "\n1.Registrar Policia "+
61                  "\n2.Resgistrar PersonalAdministrativo"+
62                  "\n3.Calcular el saldo del policia"+
63                  "\n4.Listar ascendentemente los Policias por edad"+
64                  "\n5.Listar Segun el sueldo de los policias"+
65                  "\n6.Listar alfabeticamente al PersonalAdministrativo por apellidos"+
66                  "\n7.Salir"+
67                  "\nSeleccione Opcion: ";
68              return Integer.parseInt(Lector.leer(opciones));
69      }
70
71
72      public void registrarPolicia(){
73          String nombre=Lector.leer("Ingrese nombre: ");
74          String apellido=Lector.leer("Ingrese apellidos: ");
75          String codigo=Lector.leer("Ingrese codigo: ");
76          String DNI=Lector.leer("Ingrese DNI: ");
77          int edad=Integer.parseInt(Lector.leer("Ingrese su edad: "));
78          int rango = -1;
79          while(rango<1||rango>5) // while( !(rango>=1&&rango<=5) )
80              rango = Integer.parseInt(Lector.leer("Ingrese su rango: "));
81  //          do{
82  //              rango = Integer.parseInt(Lector.leer("Ingrese su rango: "));
83  //          }while(rango<1||rango>5);
84          ListaPolicia.add( new Policia(nombre,apellido,codigo,DNI,edad,rango) );
85      }
86
```

```
87      private void registrarPersonal() {
88          String codigo = Lector.leer("Ingrese código: ");
89          String nombres = Lector.leer("Ingrese nombres: ");
90          String apellidos = Lector.leer("Ingrese apellidos: ");
91          String DNI = Lector.leer("Ingrese DNI: ");
92          String sector = Lector.leer("Ingrese sector: ");
93          ListaAdministrativo.add( new PersonalAdministrativo(codigo, nombres, apellidos, DNI, sector) );
94      }
95
96      private void calcularSaldo() {
97  //          for(int i=0; i<ListaPolicia.size(); ++i)
98  //              ListaPolicia.get(i).calcularS();
99
100 //          FOR EACH.
101         for(Policia p : ListaPolicia)
102             p.calcularS();
103     }
104
105     private void listarPoliciaEdad() {
106 //          MÉTODO DE LA BURBUJA.
107         for(int i=0; i<ListaPolicia.size()-1; ++i)
108             for(int j=i+1; j<ListaPolicia.size(); ++j)
109                 if(ListaPolicia.get(i).getEdad() > ListaPolicia.get(j).getEdad()) {
110                     Policia temporal = ListaPolicia.get(i);
111                     ListaPolicia.set(i, ListaPolicia.get(j));
112                     ListaPolicia.set(j, temporal);
113                 }
114
115         for(Policia p : ListaPolicia)
116             System.out.println( p.toString() );
117     }
118
119     private void listarPoliciaSueldo() {
120 //          MÉTODO DE LA BURBUJA.
121         for(int i=0; i<ListaPolicia.size()-1; ++i)
122             for(int j=i+1; j<ListaPolicia.size(); ++j)
123                 if(ListaPolicia.get(i).getSaldo() < ListaPolicia.get(j).getSaldo()) {
124                     Policia temporal = ListaPolicia.get(i);
125                     ListaPolicia.set(i, ListaPolicia.get(j));
126                     ListaPolicia.set(j, temporal);
127                 }
128
129         for(Policia p : ListaPolicia)
130             System.out.println( p.toString() );
131     }
132
133     private void listarPersonal() {
134 //          MÉTODO DE LA BURBUJA.
135         for(int i=0; i<ListaAdministrativo.size()-1; ++i)
136             for(int j=i+1; j<ListaAdministrativo.size(); ++j)
137                 if(ListaAdministrativo.get(i).getApellidos().compareTo(ListaAdministrativo.get(j).getApellidos())>0) {
138                     PersonalAdministrativo temporal = ListaAdministrativo.get(i);
139                     ListaAdministrativo.set(i, ListaAdministrativo.get(j));
140                     ListaAdministrativo.set(j, temporal);
141                 }
142
143         for(PersonalAdministrativo adm : ListaAdministrativo)
144             System.out.println( adm.toString() );
145     }
146
147
148
149
150 }
```

Luego de la correcta implementación del patrón no podemos realizar lo siguiente:

Clase Principal (main)

```
1   package principal;
2
3   public class Principal {
4
5       public static void main(String[] args) {
6           SistemaRegistros.getInstancia().ejecutar();
7           new SistemaRegistros().ejecutar();
8   //          SistemaRegistros S = new SistemaRegistros();
9       }
10
11  }
12
```

No podemos instanciar la clase ya que tenemos la restricción con el constructor de la clase al ponerlo privado y también al haber establecido el método de la clase de tipo privado y estático como se mostrará a continuación:

Clase SistemaRegistros

```
8   public class SistemaRegistros {
9
10      private static SistemaRegistros unicaInstancia;
11
12      public static SistemaRegistros getInstancia() {
13          if(unicaInstancia == null)
14              unicaInstancia = new SistemaRegistros();
15          return unicaInstancia;
16      }
17
18      private SistemaRegistros() {}
19
```

Como se muestra en la siguiente representación tampoco es posible:

Clase Principal (main)

```
1   package principal;
2
3   public class Principal {
4
5       public static void main(String[] args) {
6           SistemaRegistros.getInstancia().ejecutar();
7   //          new SistemaRegistros().ejecutar();
8           SistemaRegistros S = new SistemaRegistros();
9       }
10
11  }
12
```

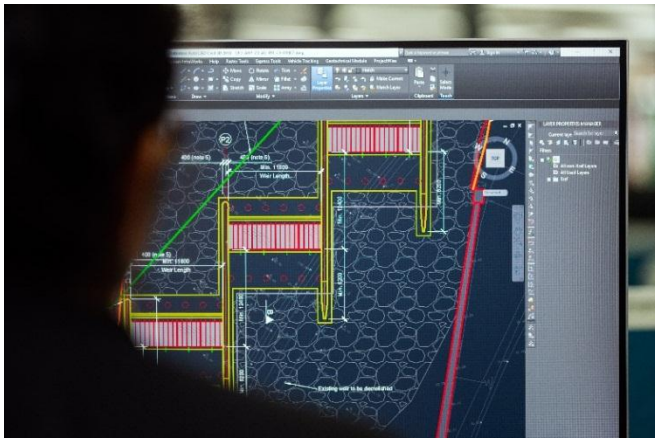
ALGUNAS PRUEBAS….

```
 MENU PINCIPAL DE LA COMISARÍA AGUILAS NEGRAS
1.Registrar Policia
2.Resgistrar PersonalAdministrativo
3.Calcular el saldo del policia
4.Listar ascendentemente los Policias por edad
5.Listar Segun el sueldo de los policias
6.Listar alfabeticamente al PersonalAdministrativo por apellidos
7.Salir
Seleccione Opcion:
```

Podemos registrar información del personal del sistema, calcular los sueldos según su rango, listar los policías o personal administrativo, para motivos de ejemplo listaremos el personal policial por orden ascendente según la edad.

```
 MENU PINCIPAL DE LA COMISARÍA AGUILAS NEGRAS
1.Registrar Policia
2.Resgistrar PersonalAdministrativo
3.Calcular el saldo del policia
4.Listar ascendentemente los Policias por edad
5.Listar Segun el sueldo de los policias
6.Listar alfabeticamente al PersonalAdministrativo por apellidos
7.Salir
Seleccione Opcion:
4

Código: 789456
Nombres: Johan
Apellidos: Meza
DNI: 159852
Edad: 22
Rango: 2
Saldo: 0.0

Código: 1098785205
Nombres: Jhon Mario
Apellidos: Meza Rios
DNI: 159753
Edad: 24
Rango: 1
Saldo: 120.0
```

STRUCTURAL PATTERNS



Structural Patterns

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

Son patrones que nos facilitan la modelización de nuestro software especificando la forma en la que unas clases se relacionan con otras.
Los patrones estructurales describen como las clases y objetos pueden ser combinados para formar grandes estructuras y proporcionar nuevas funcionalidades.
Estos objetos adicionados pueden ser incluso objetos simples u objetos compuestos.

ADAPTER

El objetivo del patrón Adapter es convertir la interfaz de una clase existente en la interfaz esperada por los clientes también existentes de modo que puedan trabajar de manera conjunta. Se trata de conferir a una clase existente una nueva interfaz para responder a las necesidades de los clientes. Es ADAPTAR un objeto existente.

DIAGRAMA DE CLASES

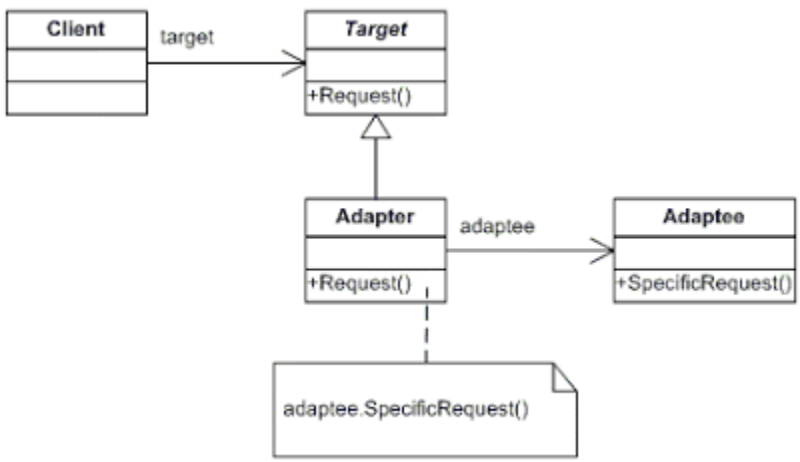
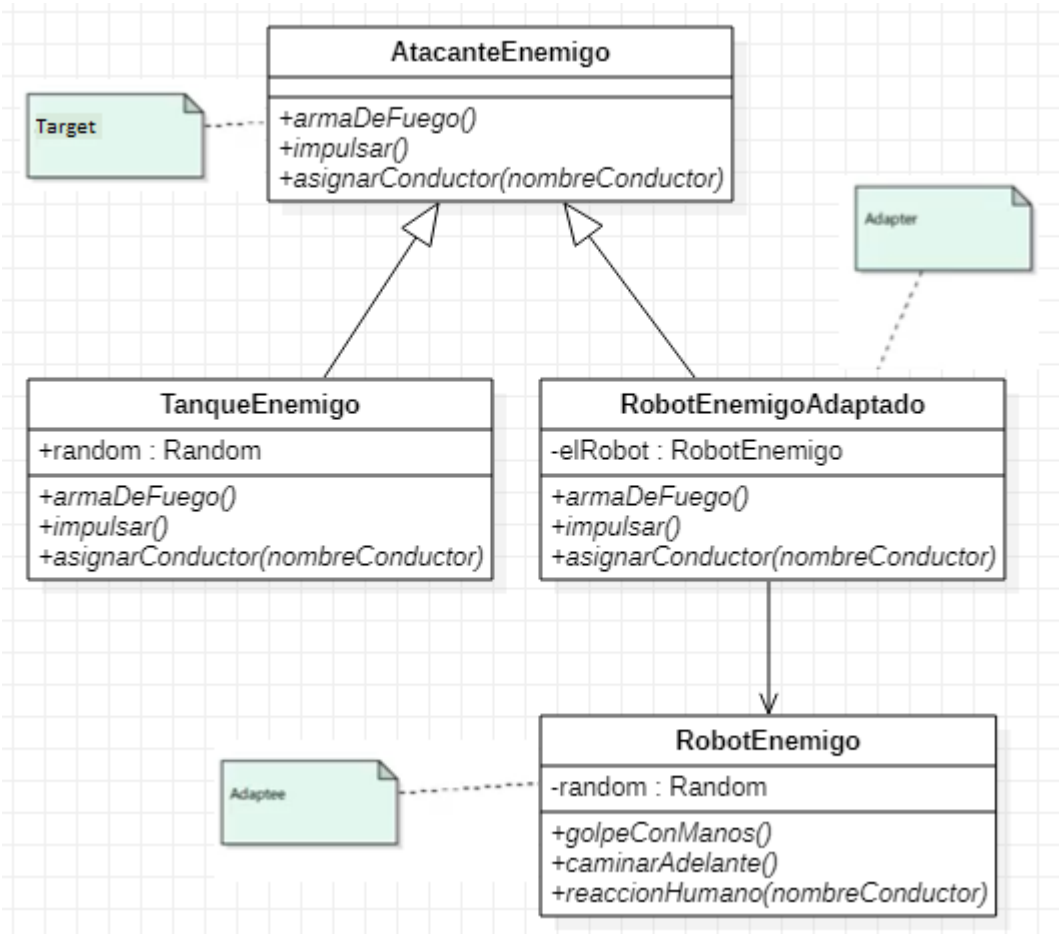
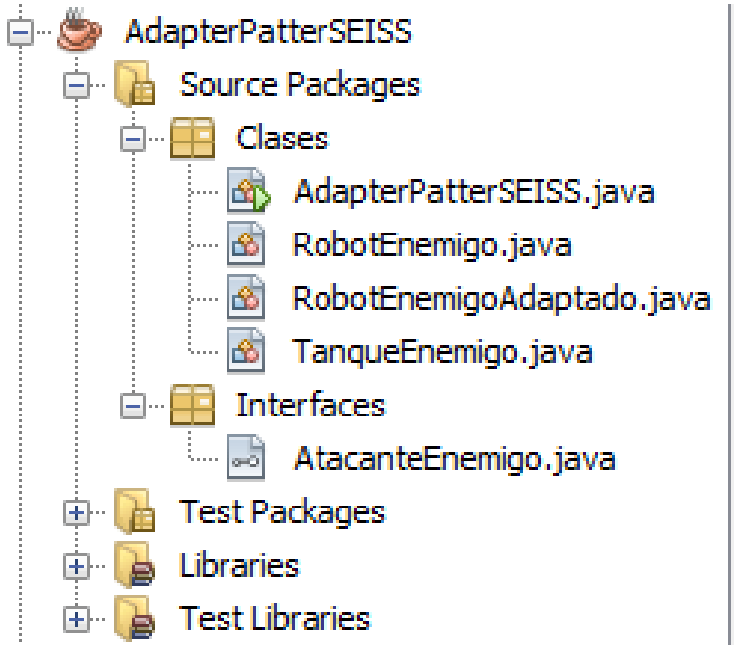


Imagen tomada: https://www.hojjatk.com/2012/11/adapter-design-pattern.html

APLICACIÓN

Se requiere desarrollar un juego el cual permita añadir diferentes tipos de enemigos, se debe tener la posibilidad de portar un arma, andar y alguien que lo controle. Se desea agregar un tipo de enemigo (Robot) con funcionalidades diferentes a las demás, se debe adaptar el enemigo (robot) sin afectar la lógica inicial del juego. Con este patrón se puede adaptar al enemigo (robot) haciendo un tipo de puente para adaptar la clase enemiga a las funcionalidades del resto de enemigos.





Estructura de Carpetas

Interface AtacanteEnemigo

```
package Interfaces;

public interface AtacanteEnemigo {
    public void armaDeFuego();
    public void impulsar();
    public void asignarConductor(String nombreConductor);
}
```

Clase TanqueEnemigo

```java
package Clases;

import Interfaces.AtacanteEnemigo;
import java.util.Random;

public class TanqueEnemigo implements AtacanteEnemigo{

    Random generador = new Random();

    @Override
    public void armaDeFuego() {
        int danoAtaque = generador.nextInt(10) + 1;
        System.out.println("Tanque Enemigo hizo " + danoAtaque + " de daño");
    }

    @Override
    public void impulsar() {
        int movimiento = generador.nextInt(5) + 1;
        System.out.println("Tanque enemigo se movio " + movimiento + " espacios");
    }

    @Override
    public void asignarConductor(String nombreConductor) {
        System.out.println(nombreConductor + " esta conducion el tanque");
    }
}
```

Clase RobotEnemigo

```java
package Clases;

import java.util.Random;

public class RobotEnemigo {
    Random generador = new Random();

    public void golpeConManos()
    {
        int danoAtaque = generador.nextInt(10) + 1;
        System.out.println("Robot Enemigo Causo " + danoAtaque + " de daño con las manos");
    }

    public void caminarAdelante() {
        int movimiento = generador.nextInt(5) + 1;
        System.out.println("Robot enemigo camino hacia adelante " + movimiento + " espacios");
    }

    public void reaccionHumano(String nombreConductor) {
        System.out.println("Robot enemigo golpea a " + nombreConductor);
    }
}
```

Clase RobotEnemigoAdaptado

```java
package Clases;

import Interfaces.AtacanteEnemigo;

public class RobotEnemigoAdaptado implements AtacanteEnemigo{

    RobotEnemigo elRobot;

    public RobotEnemigoAdaptado(RobotEnemigo nuevoRobot)
    {
        elRobot = nuevoRobot;

    }

    @Override
    public void armaDeFuego() {
        elRobot.golpeConManos();
    }

    @Override
    public void impulsar() {
        elRobot.caminarAdelante();
    }

    @Override
    public void asignarConductor(String nombreConductor) {
        elRobot.reaccionHumano(nombreConductor);
    }
}
```

Clase AdapterPatterSEISS (main)

```
package Clases;

public class AdapterPatterSEISS {

    public static void main(String[] args) {
        TanqueEnemigo tanque = new TanqueEnemigo();

        RobotEnemigo robot = new RobotEnemigo();

        RobotEnemigoAdaptado robotAdaptado = new RobotEnemigoAdaptado(robot);

        System.out.println("El robot");
        robot.reaccionHumano("Paul");
        robot.caminarAdelante();
        robot.golpeConManos();
        System.out.println("--------------------");

        System.out.println("El tanque Enemigo");
        tanque.asignarConductor("Frank");
        tanque.impulsar();
        tanque.armaDeFuego();
        System.out.println("--------------------");

        System.out.println("El robot adaptado");
        robotAdaptado.asignarConductor("Mark");
        robotAdaptado.impulsar();
        robotAdaptado.armaDeFuego();
    }
}
```

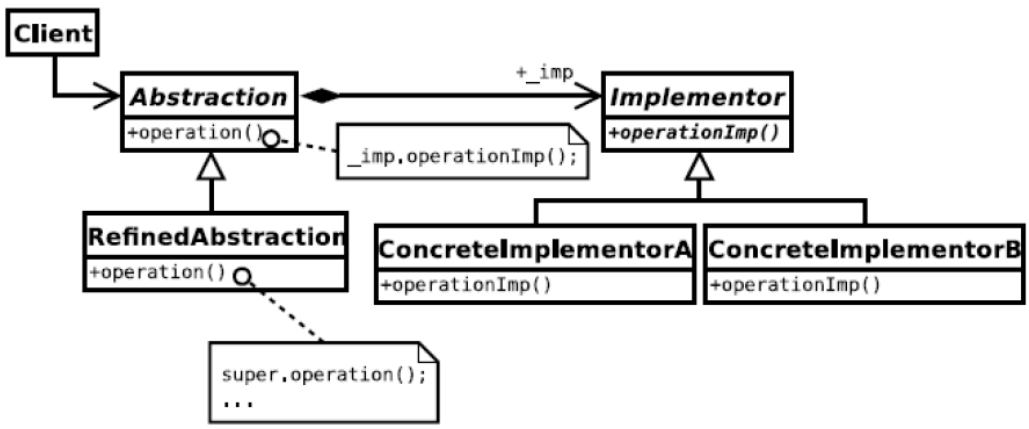
ALGUNAS PRUEBAS….

```
El robot
Robot enemigo golpea a Paul
Robot enemigo camino hacia adelante 5 espacios
Robot Enemigo Causo 4 de daño con las manos
--------------------
El tanque Enemigo
Frank esta conducion el tanque
Tanque enemigo se movio 1 espacios
Tanque Enemigo hizo 10 de daño
--------------------
El robot adaptado
Robot enemigo golpea a Mark
Robot enemigo camino hacia adelante 5 espacios
Robot Enemigo Causo 5 de daño con las manos
```

BRIDGE

El objetivo del patrón Bridge es separar el aspecto de implementación de un objeto de su aspecto de representación y de interfaz

DIAGRAMA DE CLASES



APLICACIÓN

Se requiere una aplicación que ensamble los automóviles a gusto del cliente y muestre su desarrollo desde su producción hasta la terminación.

Clase Assemble

```
public class Assemble implements workShop{
    public void work(){
        System.out.println(" and Assemble");
    }
}
```

Clase Produce

```
public class Produce implements workShop{
    public void work(){
        System.out.print("Produce");
    }
}
```

Clase Automobile Abstract

```
public abstract class Automobile {
    protected  workShop worShop1;
    protected workShop worShop2;
    protected Automobile( workShop ws1, workShop ws2){
        this.worShop1=ws1;
        this.worShop2=ws2;
    }
    abstract public void manufacture();
    }
```

Clase bus

```
public class bus  extends Automobile{
 public  bus (workShop ws1, workShop ws2){
  super (ws1,ws2);
 }
    @Override
    public void manufacture() {
        System.out.println("Bus is:");
        worShop1.work();
        worShop2.work();
    }
```

Clase Taxi

```
public class Taxi  extends Automobile{
 public  Taxi (workShop ws1, workShop ws2){
  super (ws1,ws2);
 }
     @Override
    public void manufacture() {
        System.out.println("\nTaxi is:");
        worShop1.work();
        worShop2.work();
    }
```

Clase User (main)

```
public class User {
    public static void main(String[] args) {
       Automobile Bus=new bus (new Produce(),new Assemble());
   Bus.manufacture();
    Automobile taxi=new Taxi(new Produce(),new Assemble());
   taxi.manufacture();
    }
}
```

ALGUNAS PRUEBAS….

DECORATOR

El objetivo del patrón Decorator es agregar dinámicamente funcionalidades suplementarias a un objeto

DIAGRAMA DE CLASES

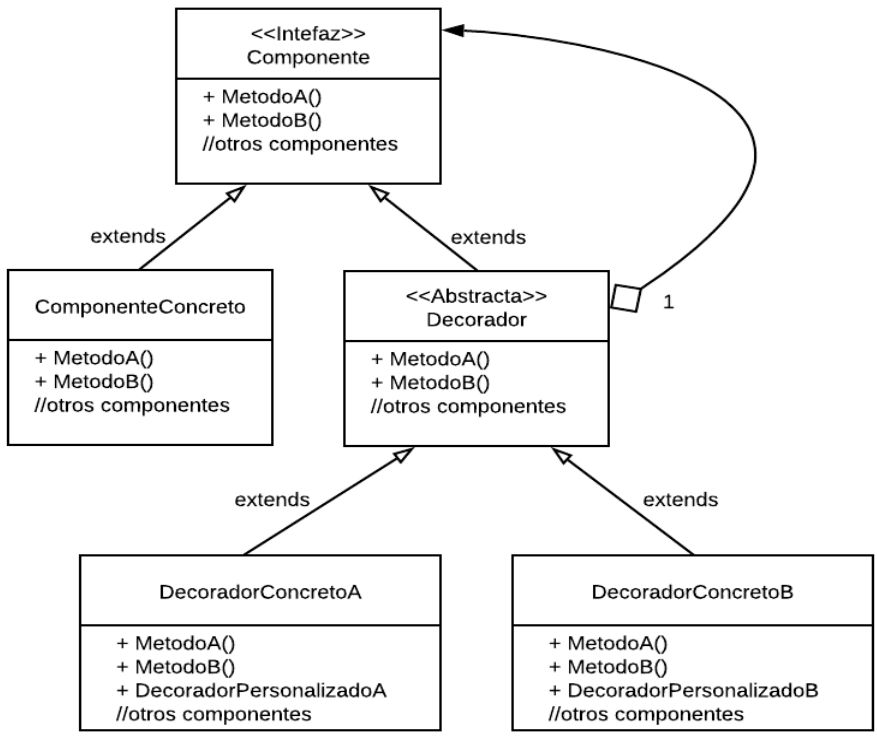
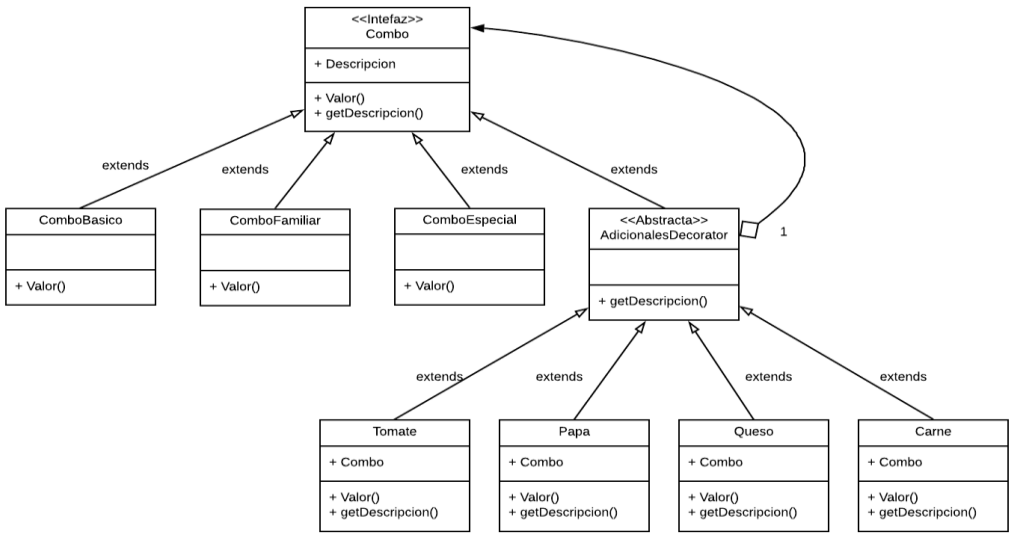


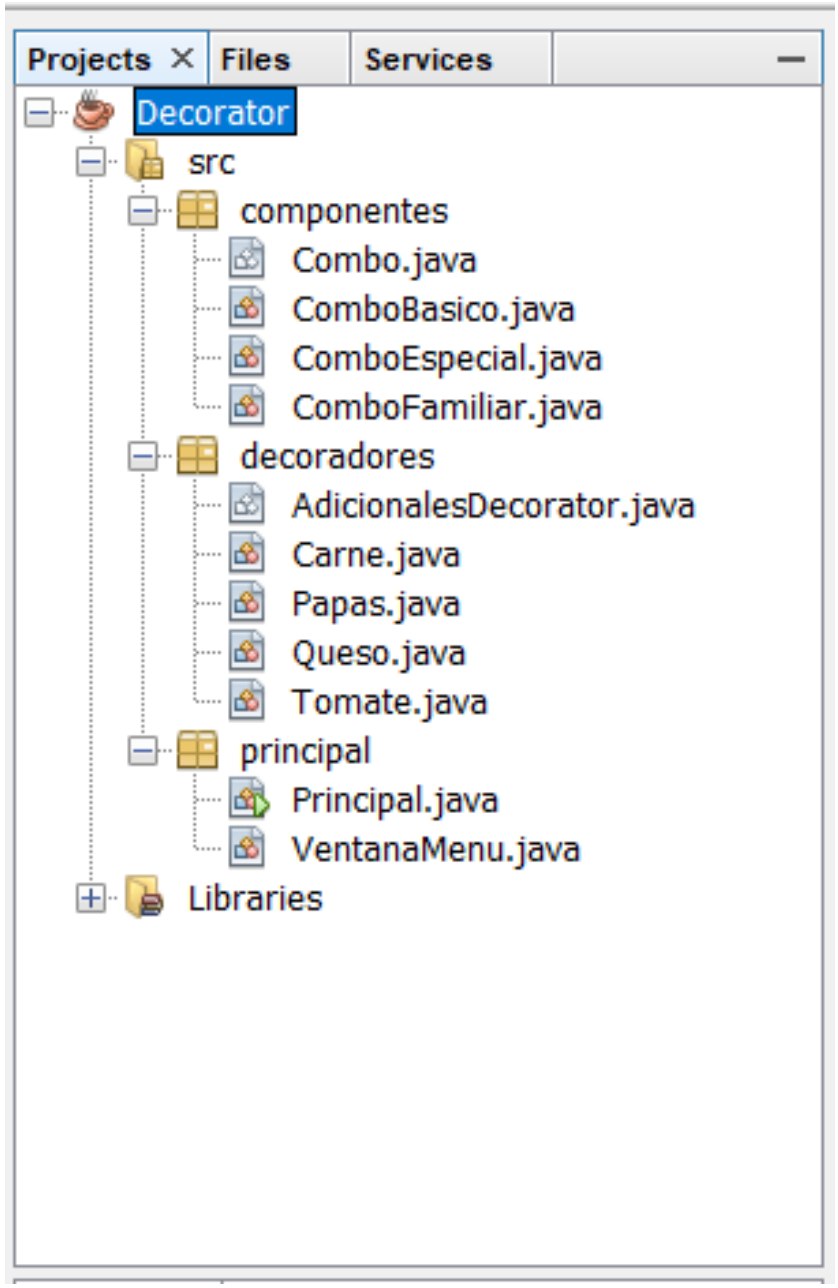
Diagrama de Clases Genérico

APLICACIÓN

Un restaurante de comidas rápidas ofrece 3 tipos de combos (Combo Básico, Combo Familiar, Combo Especial) cada combo tiene características diferentes en cuanto a cantidad, porciones, salsas entre otros, el restaurante también ofrece la posibilidad de aumentar el pedido mediante diferentes porciones adicionales (Tomate, Papas, Carne, Queso), se desea crear un sistema de pedidos que permita al usuario seleccionar el combo deseado, así como armar su propio pedido con las porciones adicionales, el sistema mostrara el pedido.



La solución que nos da el patrón Decorator es solo utilizar la herencia para que las clases Decorator tengan el mismo tipo de los objetos a decorar y utilizaremos la composición para determinar el comportamiento de forma dinámica y en tiempo de ejecución, relacionando los decoradores con los componentes concretos, así no modificaríamos la lógica de las clases existentes cada vez que queramos agregar una nueva funcionalidad al sistema.



Estructura de Carpetas

Clase Combo Abstracta

```java
package componentes;

/**
 * Clase combo, esta clase indica la clase
 * padre del tipo de combo disponible, cuenta con una
 * descripcion y un precio
 * @author jhonm
 *
 */
public abstract class Combo {

        String descripcion = "";

        public String getDescripcion()
        {
            return descripcion;
        }

        public abstract int valor();

}
```

Clase ComboBasico

```java
package componentes;

/**
 * Indica un tipo de combo basico,
 * heredando de la clase padre Combo
 * @author jhonm
 *
 */
public class ComboBasico extends Combo{

    public ComboBasico() {
        descripcion="Porcion de Papas Fritas, " +
                "Salsa, Queso, Hamburgueza Sencilla, Gaseosa";
    }

    @Override
    public int valor() {
        return 6200;
    }
}
```

Clase ComboEspecial

```java
package componentes;
/**
 * Indica un tipo de combo Especial,
 * heredando de la clase padre Combo
 * @author jhonm
 *
 */
public class ComboEspecial extends Combo{

        public ComboEspecial()
        {
                descripcion="Doble Porcion de Papas Fritas,3 tipos " +
                            "de Salsa, Doble Queso, Hamburgueza Especial " +
                            "Doble Carne, Doble Tomate, Gaseosa";
        }

        @Override
        public int valor() {
                return 10400;
        }


}

```

Clase ComboFamiliar

```java
package componentes;
/**
 * Indica un tipo de combo Familiar,
 * heredando de la clase padre Combo
 * @author jhonm
 *
 */
public class ComboFamiliar extends Combo
{

        public ComboFamiliar(){
                descripcion="Doble Porcion de Papas Fritas, " +
                            "salsa,doble queso, hamburgueza " +
                            "Familiar,doble tomate, gaseosa";
        }

        @Override
        public int valor() {
                return 7500;
        }

}

```

Clase AdicionalesDecorator Abstracta

```java
package decoradores;
import componentes.Combo;


/**
 * (Decorator)
 * Clase abstracta de los productos adicionales,
 * cuenta con una descripcion del producto
 * @author jhonm
 *
 */
public abstract class AdicionalesDecorator extends Combo{

        public abstract String getDescripcion();
}
```

Clase Carne

```java
package decoradores;
import componentes.Combo;


/**
 * (Decorator Concreto)
 * clase Carne, siendo esta un adicional,
 * cuenta con una descripcion y un precio
 * @author jhonm
 *
 */
public class Carne extends AdicionalesDecorator{

        Combo combo;

        public Carne(Combo combo)
        {
                this.combo=combo;
        }

        @Override
        public String getDescripcion() {
                return combo.getDescripcion()+" , Porcion de Carne";
        }

        @Override
        public int valor() {
                return 2500+combo.valor();
        }
}
```

Clase Papas

```java
package decoradores;
import componentes.Combo;

/**
 * (Decorator Concreto)
 * clase papas, siendo esta un adicional, cuenta
 * con una descripcion y un precio
 * @author jhonm
 *
 */
public class Papas extends AdicionalesDecorator{

	Combo combo;

	public Papas(Combo combo)
	{
		this.combo=combo;
	}

	@Override
	public String getDescripcion() {
		return combo.getDescripcion()+" , Porcion de Papas";
	}

	@Override
	public int valor() {
		return 1500+combo.valor();
	}

}
```

Clase Queso

```java
package decoradores;
import componentes.Combo;

/**
 * (Decorator Concreto)
 * clase queso, siendo esta un adicional,
 * cuenta con una descripcion y un precio
 * @author jhonm
 *
 */
public class Queso extends AdicionalesDecorator{

	Combo combo;

	public Queso(Combo combo)
	{
		this.combo=combo;
	}

	@Override
	public String getDescripcion() {
		return combo.getDescripcion()+" , Porcion de Queso";
	}

	@Override
	public int valor() {
		return 1000+combo.valor();
	}

}
```

Clase Tomate

```java
package decoradores;
import componentes.Combo;

/**
 * (Decorator Concreto)
 * clase tomate, siendo esta un adicional,
 * cuenta con una descripcion y un precio
 * @author jhonm
 *
 */
public class Tomate extends AdicionalesDecorator{

        Combo combo;

        public Tomate(Combo combo)
        {
            this.combo=combo;
        }


        @Override
        public String getDescripcion() {
            return combo.getDescripcion()+" , Porcion de Tomate";
        }

        @Override
        public int valor() {
            return 100+combo.valor();
        }

}

```

Clase Principal (main)

```java
package principal;

/**
 * Clase principal del aplicativo
 * @author jhonm
 *
 */
public class Principal {

    public static void main(String[] args) {
            VentanaMenu ventana=new VentanaMenu();
            ventana.setVisible(true);
        }

}

```

Finalmente en el paquete principal tenemos la clase donde se ejecuta el programa y la ventana que representa el Menú de Selección desde el cual el usuario puede seleccionar el Combo y las porciones a pedir, al enviar el pedido el sistema de forma dinámica por medio del patrón

Decorator valida las combinaciones solicitadas y calcula el precio Total del pedido

Clase VentanaMenu

```
1   package principal;
2
3   import java.awt.event.ActionEvent;
4   import java.awt.event.ActionListener;
5
6   import javax.swing.JButton;
7   import javax.swing.JCheckBox;
8   import javax.swing.JComboBox;
9   import javax.swing.JFrame;
10  import javax.swing.JLabel;
11  import javax.swing.JOptionPane;
12  import javax.swing.JScrollPane;
13  import javax.swing.JTextArea;
14
15  import componentes.Combo;
16  import componentes.ComboBasico;
17  import componentes.ComboEspecial;
18  import componentes.ComboFamiliar;
19  import decoradores.Carne;
20  import decoradores.Papas;
21  import decoradores.Queso;
22  import decoradores.Tomate;
23
24
25  /**
26   * Clase GUI, es la ventana interfaz de
27   * usuario, en la que se permite la seleccion de
28   * un combo y el calculo de su valor con o sin adicionales
29   * @author jhonm
30   *
31   */
32  public class VentanaMenu extends JFrame implements ActionListener{
33
34          JLabel titulo,adicionales;
35          JComboBox combos;
36          String arregloCombos[] = { "Seleccione","Combo Basico",
37                                     "Combo Familiar", "Combo Especial" };
38          JTextArea area;
39          private JScrollPane scroll;
40          JTextArea area2;
41          private JScrollPane scroll2;
42          JButton aceptar,cancelar;
43          JCheckBox tomate,papas,carne,queso;
44
```

```
45          public VentanaMenu()
46          {
47            tomate= new JCheckBox();
48            tomate.setText("Tomate");
49            tomate.setBounds(190, 30, 70, 25);
50            tomate.setEnabled(false);
51
52            papas= new JCheckBox();
53            papas.setText("Papas");
54            papas.setBounds(280, 30, 70, 25);
55            papas.setEnabled(false);
56
57            carne= new JCheckBox();
58            carne.setText("Carne");
59            carne.setBounds(190, 50, 70, 25);
60            carne.setEnabled(false);
61
62            queso= new JCheckBox();
63            queso.setText("Queso");
64            queso.setBounds(280, 50, 70, 25);
65            queso.setEnabled(false);
66
67            aceptar=new JButton();
68            aceptar.setText("Enviar Pedido");
69            aceptar.setBounds(70, 320, 150, 25);
70
71            cancelar=new JButton();
72            cancelar.setText("Salir");
73            cancelar.setBounds(230, 320, 90, 25);
74
75            titulo= new JLabel();
76            titulo.setText("MENU COMBOS");
77            titulo.setBounds(20, 10, 150, 25);
78
79            adicionales= new JLabel();
80            adicionales.setText("Seleccione los Adicionales");
81            adicionales.setBounds(195, 10, 180, 25);
82
83            combos= new JComboBox();
84            combos.setBounds(20, 40, 150, 25);
85            combos.setModel(new javax.swing.DefaultComboBoxModel(arregloCombos));
              combos.addActionListener(this);
87
```

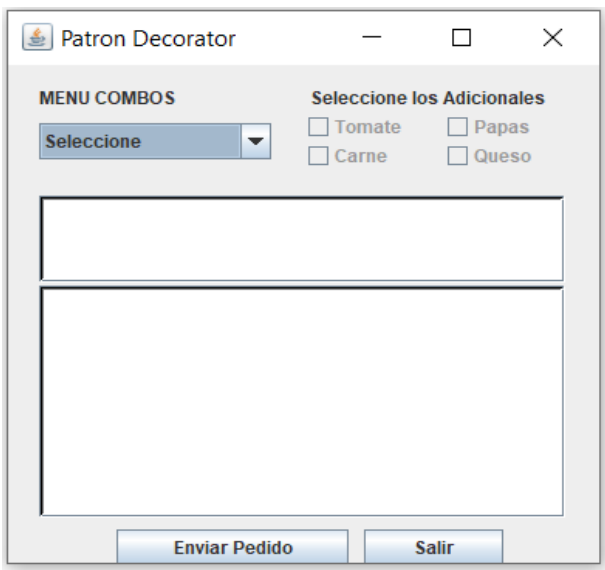
```
88              scroll = new JScrollPane();
89              area = new JTextArea();
90                    area.setEditable(false);
91                    area.setFont(new java.awt.Font("Verdana", 0, 12));
92                    area.setLineWrap(true);
93                    area.setWrapStyleWord(true);
94                    area.setBorder(javax.swing.BorderFactory.createBevelBorder(
95                                  javax.swing.border.BevelBorder.LOWERED, null, null, null,
96                                  new java.awt.Color(0, 0, 0)));
97                    scroll.setViewportView(area);
98                    scroll.setBounds(20, 90, 340, 60);
99
100                   scroll2 = new JScrollPane();
101                     area2 = new JTextArea();
102                           area2.setEditable(false);
103                           area2.setFont(new java.awt.Font("Verdana", 0, 12));
104                           area2.setLineWrap(true);
105                           area2.setWrapStyleWord(true);
106                           area2.setBorder(javax.swing.BorderFactory.createBevelBorder(
107                                       javax.swing.border.BevelBorder.LOWERED, null, null, null,
108                                       new java.awt.Color(0, 0, 0)));
109                           scroll2.setViewportView(area2);
110                           scroll2.setBounds(20, 152, 340, 160);
111
112
              combos.addActionListener(this);
              aceptar.addActionListener(this);
              cancelar.addActionListener(this);
              tomate.addActionListener(this);
              queso.addActionListener(this);
              carne.addActionListener(this);
              papas.addActionListener(this);
120
              add(queso);
              add(carne);
              add(papas);
              add(tomate);
              add(cancelar);
              add(aceptar);
              add(scroll);
              add(scroll2);
              add(adicionales);
              add(titulo);
              add(combos);
132
```

```
132
            setSize(400,390);
            setTitle("Patron Decorator");
            setLocationRelativeTo(null);
            setLayout(null);
137
138         }
139
140
141
142         @Override
            public void actionPerformed(ActionEvent e)
144         {
145             if (e.getSource()==aceptar)
146             {
                    if (combos.getSelectedIndex()==1)
148                 {
149                     Combo comboBasico=new ComboBasico();
150                     area.setText(comboBasico.getDescripcion());
151                     enviarPedido(comboBasico);
152                 }
153                 else if (combos.getSelectedIndex()==2)
154                 {
155                     Combo comboFamiliar=new ComboFamiliar();
156                     area.setText(comboFamiliar.getDescripcion());
157                     enviarPedido(comboFamiliar);
158                 }
159                 else if (combos.getSelectedIndex()==3)
160                 {
161                     Combo comboEspecial=new ComboEspecial();
162                     area.setText(comboEspecial.getDescripcion());
163                     enviarPedido(comboEspecial);
164                 }
165                 else{
166                     JOptionPane.showMessageDialog(null, "No ha realizado " +
167                             "ningun pedido","Advertencia!!!",JOptionPane.WARNING_MESSAGE);
168                 }
169             }
170             if (e.getSource()==cancelar)
171             {
172                 System.exit(0);
173             }
174             if (e.getSource()==combos)
175             {
176                 verificaSeleccion();
177
178             }
```

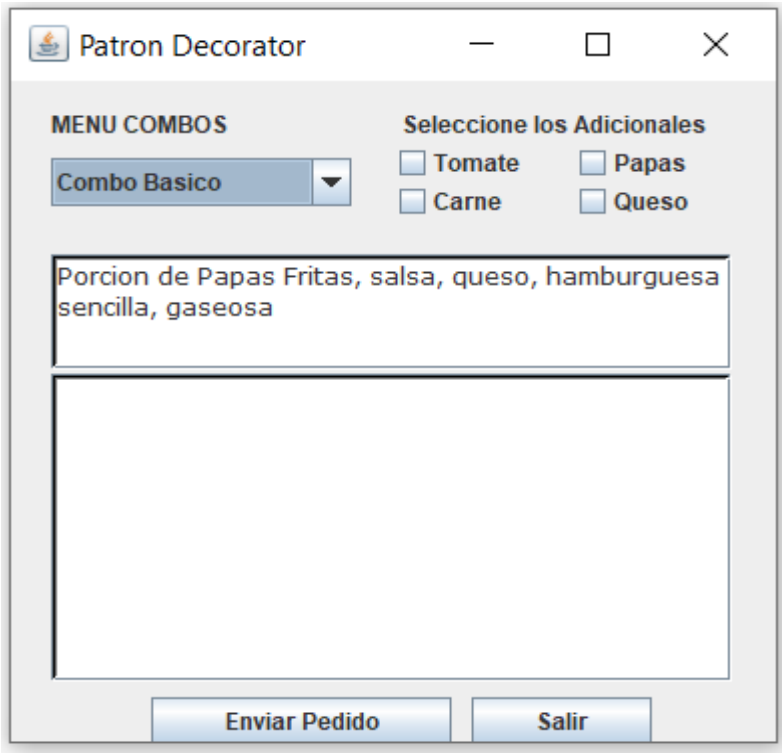
```
179         }
180
181         /**
182          * Aplicamos conceptos de polimorfismo para definir
183          * la porcion del combo seleccinado y luego
184          * se envia al area2 correspondiente a la del envio
185          * definido por el usuario
186          * @param combo
187          */
188         private void enviarPedido(Combo combo)
189         {
190                         if (queso.isSelected())
191                         {
192                                 combo=new Queso(combo);
193                         }
194                         if (carne.isSelected())
195                         {
196                                 combo=new Carne(combo);
197                         }
198                         if (papas.isSelected())
199                         {
200                                 combo=new Papas(combo);
201                         }
202                         if (tomate.isSelected())
203                         {
204                                 combo=new Tomate(combo);
205                         }
206                 area2.setText("Su Pedido: \n"+combo.getDescripcion()+"\n\n" +
207                                 "Valor:\n $"+combo.valor());
208         }
209
210         /**
211          * Valida la seleccion realizada y envia la correspondiente
212          * descripcion al area de texto
213          */
214         private void verificaSeleccion() {
215                 if (combos.getSelectedIndex()!=0)
216                 {
217                         habilitar(true);
                        if (combos.getSelectedIndex()==1)
219                         {
220                                 area.setText("Porcion de Papas Fritas, salsa, queso," +
221                                                 " hamburguesa sencilla, gaseosa");
222                                 area2.setText("");
223                         }
224                         else if (combos.getSelectedIndex()==2)
225                         {
```

```
226                             area.setText("Doble Porcion de Papas Fritas,salsa,doble queso," +
227                                     " hamburguesa Familiar,doble tomate, gaseosa");
228                             area2.setText("");
229                     }
230                     else if (combos.getSelectedIndex()==3)
231                     {
232                             area.setText("Doble Porcion de Papas Fritas,3 tipos de salsa," +
233                                     " doble queso, hamburguesa Especial Doble Carne," +
234                                     " Doble tomate, gaseosa");
235                             area2.setText("");
236                     }
237
238             }
239             else{
240                     area.setText("");
241                     area2.setText("");
242                     habilitar(false);
243             }
244
245         }
246
247
248
249         private void habilitar(boolean b)
250         {
251                 tomate.setEnabled(b);
252                 queso.setEnabled(b);
253                 carne.setEnabled(b);
254                 papas.setEnabled(b);
255                 tomate.setSelected(false);
256                 queso.setSelected(false);
257                 carne.setSelected(false);
258                 papas.setSelected(false);
259         }
260
261
262     }
263
```

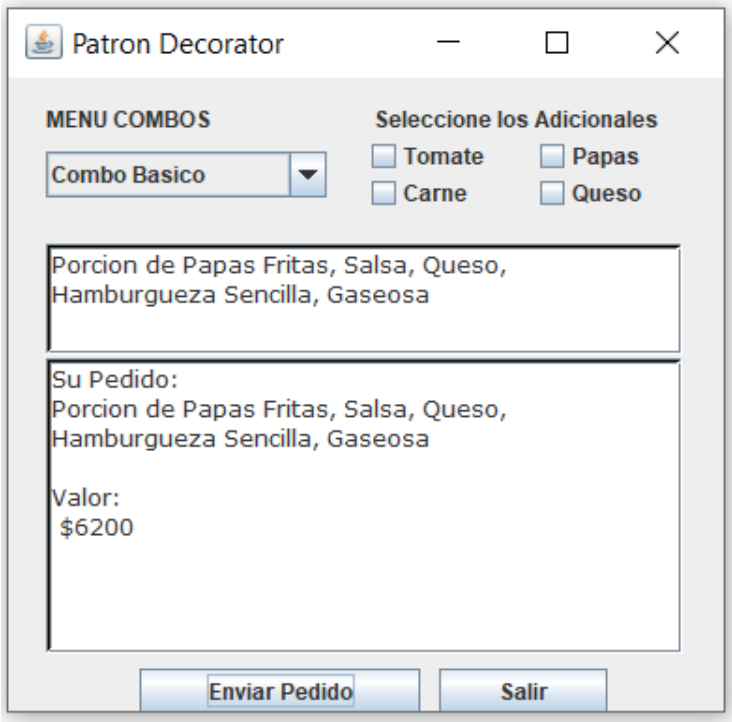
ALGUNAS PRUEBAS….



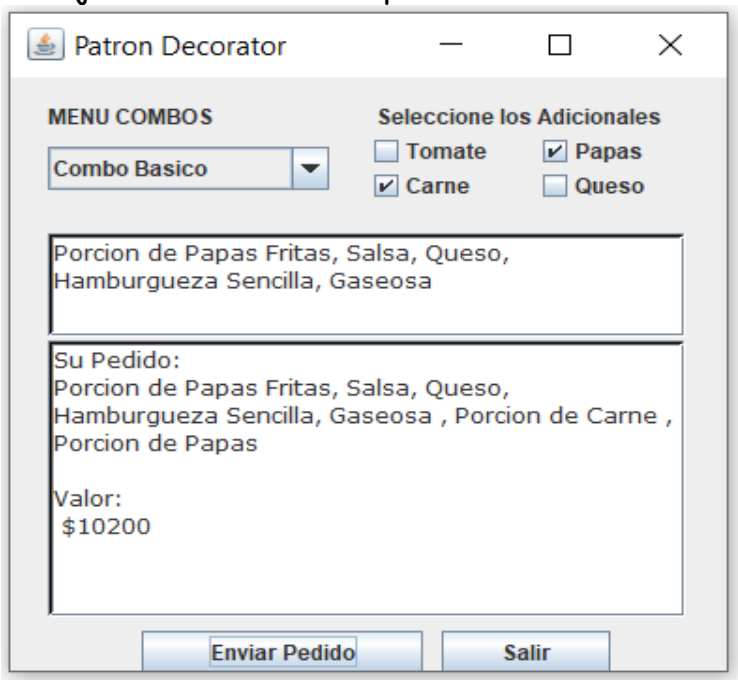
Al seleccionar un combo se reflejará los ingredientes del producto combo seleccionado



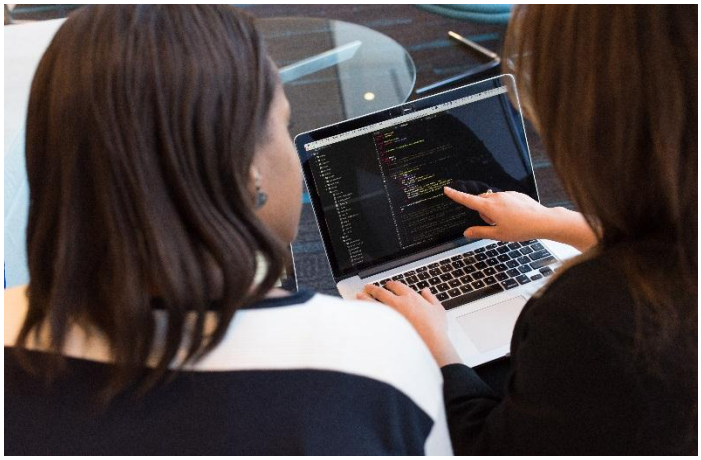
Se habilitarán los adicionales, si decidimos seleccionar alguno, el precio variara de acuerdo a la cantidad y tipos de adicionales seleccionados:



Al presionar enviar pedido nos muestra un valor del combo básico por 6.200, si eligiéramos adicionales, y volviéramos a presionar el botón enviar pedido, el precio del pedido aumentara y se dichos adicionales se reflejaran en la orden de pedido:



BEHAVIORAL PATTERNS



Behavioral Patterns

- Command
- Chain of Responsability
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

Los patrones de comportamiento nos ayudan a definir la comunicación e iteración entre los objetos de un sistema.
El propósito de este patrón es reducir el acoplamiento entre los objetos. Se usan para gestionar algoritmos, relaciones y responsabilidades entre objetos.

Se centran en la interacción entre asociaciones de clases y objetos definiendo cómo se comunican entre sí.

OBSERVER

El objetivo de este patrón es definir una dependencia de uno a muchos entre objetos; de forma que cuando un objeto cambie de estado se notifique y se actualicen automáticamente todos los objetos que dependan de él.

DIAGRAMA DE CLASES

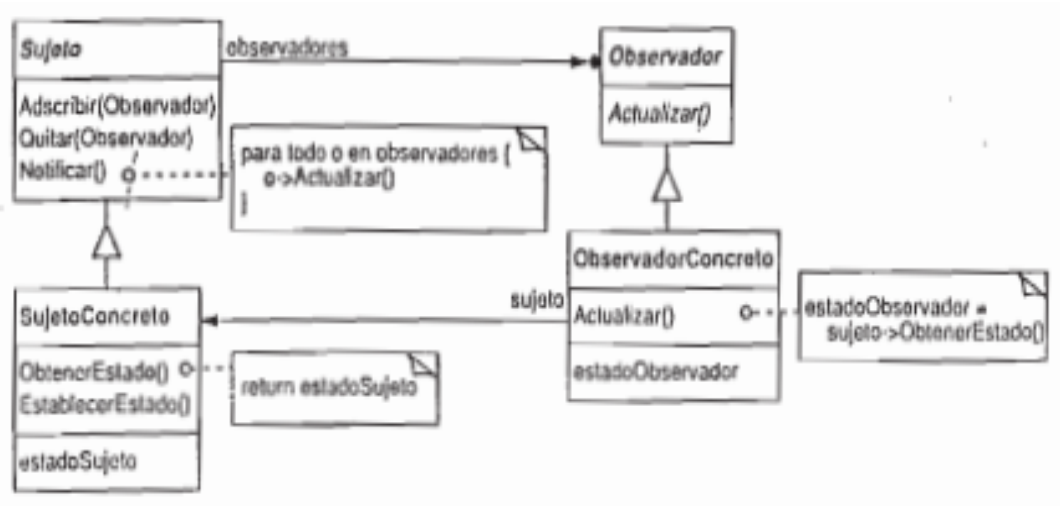
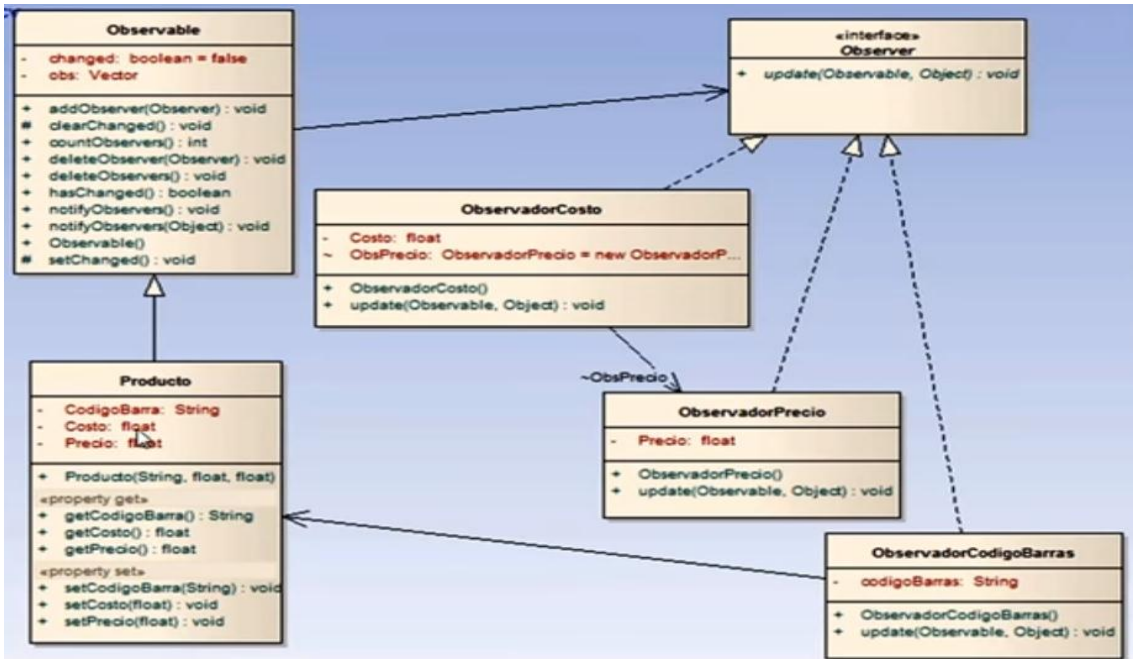
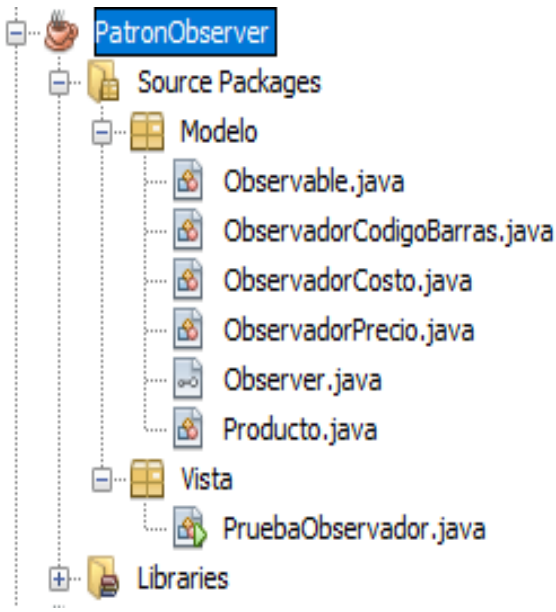


Diagrama de Clases Genérico

APLICACIÓN

En un sistema para la facturación y venta de un supermercado, se requiere que si producto tiene algún cambio en su precio, costo o código de barras, sus datos sean agregados o se actualicen inmediatamente.





Estructura de Carpetas

Clase Producto

```
package Modelo;
import java.util.Observable;

public class Producto extends Observable {

    private String CodigoBarra;
    private float Precio;
    private float Costo;

    public String getCodigoBarra() {
        return CodigoBarra;
    }

    public float getPrecio() {
        return Precio;
    }

    public float getCosto() {
        return Costo;
    }

    public void setPrecio(float Precio) {
        this.Precio = Precio;
        setChanged();
        notifyObservers(new Float(Precio));
    }

    public void setCodigoBarra(String CodigoBarra) {
        this.CodigoBarra = CodigoBarra;
        setChanged();
        notifyObservers(CodigoBarra);
    }

    public void setCosto(float Costo) {
        this.Costo = Costo;
        setChanged();
        notifyObservers(new Float(Costo));
    }

    public Producto(String codigoBarra, float costo, float precio) {
        this.CodigoBarra = codigoBarra;
        this.Precio = precio;
        System.out.println("Producto Creado Codigo: " + codigoBarra
                + " Costo: " + costo + " Precio: " + precio);
    }
}
```

Clase Observable

```
package Modelo;

import java.util.Vector;

public class Observable {

    private boolean changed = false;
    private Vector obs;

    public Observable() {
        obs = new Vector();
    }

    public synchronized void addObserver(Observer o) {
        if (o == null) {
            throw new NullPointerException();
        }
        if (!obs.contains(o)) {
            obs.addElement(o);
        }
    }

    public synchronized void deleteObserver(Observer o) {
        obs.removeElement(o);
    }

    public void notifyObservers() {
        notifyObservers(null);
    }
  public void notifyObservers(Object arg) {
      Object[] arrLocal;

      synchronized (this) {
          if (!changed) {
              return;
          }
          arrLocal = obs.toArray();
          clearChanged();
      }

      for (int i = arrLocal.length - 1; i >= 0; i--) {
          ((Observer) arrLocal[i]).update(this, arg);
      }
  }

  public synchronized void deleteObservers() {
      obs.removeAllElements();
  }

  protected synchronized void setChanged() {
      changed = true;
  }

  protected synchronized void clearChanged() {
      changed = false;
  }
```

```
    public synchronized boolean hasChanged() {
        return changed;
    }

    public synchronized int countObservers() {
        return obs.size();
    }
}
```

Clase ObservadorCodigoBarras

```
package Modelo;
import java.util.Observable;
import java.util.Observer;

public class ObservadorCodigoBarras implements Observer {

    private String codigoBarras;

    public ObservadorCodigoBarras() {
        codigoBarras = null;
        System.out.println("ObservadorCodigoBarras creado es " + codigoBarras);
    }

    @Override
    public void update(Observable o, Object arg) {
        if (arg instanceof String) {
            codigoBarras = (String) arg;
            System.out.println("ObservadorCodigoBarras: Cambiado a " + codigoBarras);
        } else {
            System.out.println("ObservadorCodigoBarras Reportandose Todo Igual!");
        }
    }
}
```

Clase ObservadorCosto

```
package Modelo;
import java.util.Observable;
import java.util.Observer;

public class ObservadorCosto implements Observer {

    private float Costo;
    ObservadorPrecio ObsPrecio=new ObservadorPrecio();
    public ObservadorCosto() {
        Costo = 0;
        System.out.println("ObservadorCosto creado costo es " + Costo);
    }

    @Override
    public void update(Observable o, Object arg) {
        if (arg instanceof Float) {
            Costo = ((Float) arg).floatValue();
            float Precio=(Costo*1.1f);
            Precio = ((Float)Precio).floatValue();
            System.out.println("Observador de Costo: ! Cambiado Costo a " + Costo);
            ObsPrecio.update(o, Precio);
        } else {
            System.out.println("Observador de Costo: !Todo Igual Nada Cambiado en Costo!");
        }
    }
}
```

Clase ObservadorPrecio

```
package Modelo;
import java.util.Observable;
import java.util.Observer;

public class ObservadorPrecio implements Observer {

    private float Precio;

    public ObservadorPrecio() {
        Precio = 0;
        System.out.println("ObservadorPrecio creado Precio es " + Precio);
    }

    @Override
    public void update(Observable o, Object arg) {
        if (arg instanceof Float) {
            Precio = ((Float) arg).floatValue();
            System.out.println("Observador de Precio: ! Cambiado Precio a " + Precio);
        } else {
            System.out.println("Observador de Precio: !Todo Igual Nada Cambiado en Costo!");
        }
    }
}
```

Interface Observer

```
package Modelo;

public interface Observer {

    void update(Observable o, Object arg);
}
```

ALGUNAS PRUEBAS….

Para la comprobar este ejemplo se creara la clase PruebaObservador, en primer lugar añadiremos el producto numero "123456", con un costo de: 1987.29, y un precio de: 2123.10 (sin ningún código de barras).
Luego se crearan los observadores de código y precio(que depende del costo).
Luego podemos manipular los elementos del producto, añadiremos el código de barras numero "654321" y también cambiaremos el costo por: 18879.6.

Clase PruebaObservador

```
package Vista;

import Modelo.ObservadorCodigoBarras;
import Modelo.ObservadorCosto;
import Modelo.Producto;

public class PruebaObservador {

    public static void main(String args[]) {

        Producto p = new Producto("123456", 1987.29f, 2123.10f);
        ObservadorCodigoBarras ObsCodigo = new ObservadorCodigoBarras();
        ObservadorCosto ObsCosto = new ObservadorCosto();

        p.addObserver(ObsCodigo);
        p.addObserver(ObsCosto);

        System.out.println("----Cambio de Codigo Barras---");
        p.setCodigoBarra("654321");
        System.out.println("----Cambio de Costo---");
        p.setCosto(18879.6f);
    }
}
```

```
run:
Producto Creado Codigo: 123456 Costo: 1987.29 Precio: 2123.1
ObservadorCodigoBarras creado es null
ObservadorPrecio creado Precio es 0.0
ObservadorCosto creado costo es 0.0
----Cambio de Codigo Barras---
Observador de Costo: !Todo Igual Nada Cambiado en Costo!
ObservadorCodigoBarras: Cambiado a 654321
----Cambio de Costo---
Observador de Costo: ! Cambiado Costo a 18879.6
Observador de Precio: ! Cambiado Precio a 20767.56
ObservadorCodigoBarras Reportandose Todo Igual!
BUILD SUCCESSFUL (total time: 0 seconds)
```

Se pueden evidenciar:
La creación del objeto con el código 123456 y costo 1987.29 y precio 2123.1
Se agregan los respectivos observadores que inicialmente se muestran nulos o con valores de 0.
Después cambiamos el código de barras y se muestra que el observador de costo sigue sin cambios, pero el observador de código de barras si observa que ha cambiado.
Luego cambiamos el costo y de la misma manera el precio cambia ya que depende del nuevo costo, pero el código de barras sigue igual.
Con esto podemos evidenciar como los observadores son notificados del estado actual del producto y se actualizan si así lo requieren.

BIBLIOGRAFIA

Gamma, E., Vlissides, J., Johnson, R., & Helm, R. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software.* Miami: Addison-Wesley.

G. Booch, I. Jacobson, J. Rumbaugh. *El Lenguaje Unificado de Modelado*. Guía del
usuario. Addison-Wesley,1999.

Herbert Schilt. *Java 2. Manual de Referencia.* McGraw-Hill, 2001. ISBN: 8448131738

J. Rumbaugh, I. Jacobson, G. Booch, *El Lenguaje Unificado de Modelado. Manual de
referencia*. Addison-Wesley, 2000.

K. Arnold, J. Gosling. *The Java Programming Language*. The Java Series, Addison-Wesley, 1998

Larman, Craig. *UML y patrones*. Segunda Edición. Prentice Hall. 2002.
Sitio Web de Víctor Gulías http://www.fi.udc.es/~gulias [Último acceso 20 de enero de 2009]

M. Fowler, K. Scott. *UML Distilled.* Addison-Wesley Longman, 1997.

VIDEOS

https://www.youtube.com/watch?v=9XnsOpjclUg

https://informaticapc.com/patrones-de-diseno/builder.php

https://www.youtube.com/watch?v=Vag254mmj0M&t=212s

https://www.youtube.com/watch?v=R6Ef64hDwGo&t=302s

https://www.edicioneseni.com/open/mediabook.aspx?idR=f779ded5638192f8a72486770aaa1b3b

http://arantxa.ii.uam.es/~eguerra/docencia/0708/04%20Creacion.pdf

https://www.youtube.com/watch?v=fUKwjOq5Dn4

https://www.youtube.com/watch?v=f_KJScnY7TA

https://www.youtube.com/watch?v=qG286LQM6BU

https://www.youtube.com/watch?v=MQaFPvZ4u3g&list=PLd0lZIptCEwOplc9fZ8rv5yj-Vo7C6q7k

http://codejavu.blogspot.com/2013/07/ejemplo-patron-de-diseno-decorator.html

https://experto.dev/patron-de-diseno-decorator-en-java

https://profeuttec.yolasite.com/resources/Patrones%20de%20dise%C3%B1o%20-%20Erich%20Gamma.pdf

https://www.youtube.com/watch?v=OeHBKrYeX08

https://www.youtube.com/watch?v=QiKrKNTdGGs

ANEXO No 1

PARTICIPANTES DEL SEMILLERO SEIIS

NOMBRES		APELLIDOS		Celular	e-mail	
Brayan	Nicolas	Aceros	Guerrero	3166359726	bnaceros@uts.edu.co	ESTUDIANTE UTS
Lenin	Fabian	Bolivar	Rojas	3177850492	lfbolivar@uts.edu.co	ESTUDIANTE UTS
Sergio	Ivan	Diaz	Vega	3219474401	sergioidiaz@uts.edu.co	ESTUDIANTE UTS
Diego	Alexander	Garcia	Echeverria	3142703593	dalexandergarcia@uts.edu.co	ESTUDIANTE UTS
Albert	Stephen	Gerena	Castellanos	3134355371	agerena@uts.edu.co	ESTUDIANTE UTS
Jhon	Mario	Meza	Rios	3133922896	jmariomeza@uts.edu.co	ESTUDIANTE UTS
Erwin		Meza	Vega	3154070506	emezav@unicauca.edu.co	PROFESOR ASOCIADO UNICAUCA
Eliecer		Montero	Ojeda	3163351067	emontero@correo.uts.edu.co	PROFESOR TITULAR UTS
Jose	Antonio	Otalora	Porras	3002588211	jaotalora@uts.edu.co	ESTUDIANTE UTS
Sergio	Ivan	Villamizar	Luna	3506837410	sivillamizar@uts.edu.co	ESTUDIANTE UTS