

Ejercicio de Laboratorio: Patrón Builder en Java

Objetivo:

Implementar el patrón **Builder** para crear objetos complejos de manera flexible y mantenible.

Descripción:

Imagina que estás trabajando en un sistema de creación de vehículos. Necesitas implementar el patrón Builder para construir diferentes tipos de vehículos de manera eficiente.

Desarrollo:

1. Definir la Clase Vehículo:

Se crea una clase Vehicle con propiedades como modelo, color, año, precio, etc.

Implementa métodos getter y setter para las propiedades.

```
1 // Paso 1: Clase Vehicle
2 class Vehicle {
3     private String modelo;
4     private String color;
5     private int anio;
6     private double precio;
7
8     // Constructor, getters y setters
9 }
```

2. Crear la Interfaz Builder:

Define una interfaz VehicleBuilder con métodos para establecer las diferentes características del vehículo, como setModelo, setColor, setAnio, etc.

```
interface VehicleBuilder {
    void setModelo(String modelo);
    void setColor(String color);
    void setAnio(int anio);
    void setPrecio(double precio);
    VehicleBuilder build();
}
```

3. Implementar el Builder:

Crea una clase ConcreteVehicleBuilder que implemente la interfaz VehicleBuilder.

Dentro de esta clase, define métodos para establecer las características del vehículo según las necesidades.

```
2  class ConcreteVehicleBuilder implements VehicleBuilder
3      private Vehicle vehicle;
4
5      public ConcreteVehicleBuilder() {
6          this.vehicle = new Vehicle();
7      }
8
9      public void setModelo(String modelo) {
10         vehicle.setModelo(modelo);
11     }
12
13     public void setColor(String color) {
14         vehicle.setColor(color);
15     }
16
17     public void setAnio(int anio) {
18         vehicle.setAnio(anio);
19     }
20
21     public void setPrecio(double precio) {
22         vehicle.setPrecio(precio);
23     }
24
25     public Vehicle build() {
26         return vehicle;
27     }
28 }
```

Implementar el Director:

Crea una clase VehicleDirector que tenga un método construct que tome un VehicleBuilder como parámetro.

Dentro de construct, utiliza los métodos del VehicleBuilder para construir un objeto Vehicle complejo según un flujo predefinido.

```
2 class VehicleDirector {  
3     public void construct(VehicleBuilder builder) {  
4         builder.setModelo(modelo:"Sedan");  
5         builder.setColor(color:"Rojo");  
6         builder.setAnio(anio:2023);  
7         builder.setPrecio(precio:25000.0);  
8     }  
9 }
```

Probar el Patrón Builder:

Crear un objeto VehicleDirector y un objeto ConcreteVehicleBuilder.

Llama al método construct del director, pasando el constructor del vehículo como argumento.

Accede al objeto Vehicle construido y muestra sus características para verificar que se haya creado correctamente.

```
1 public class Main {      The declared package "" does not match the expecte  
2     Run | Debug  
3     public static void main(String[] args) {  
4         VehicleDirector director = new VehicleDirector();  
5         ConcreteVehicleBuilder builder = new ConcreteVehicleBuilder();  
6  
7         director.construct(builder);  
8         Vehicle vehicle = builder.build();  
9  
10        System.out.println("Vehículo construido:");  
11        System.out.println("Modelo: " + vehicle.getModelo());  
12        System.out.println("Color: " + vehicle.getColor());  
13        System.out.println("Año: " + vehicle.getAnio());  
14        System.out.println("Precio: $" + vehicle.getPrecio());  
15    }
```



UNIVERSIDAD TÉCNICA DE AMBATO
FACULTAD DE INGENIERÍA EN SISTEMAS, ELECTRÓNICA E INDUSTRIAL

CARRERA DE: SOFTWARE
Cdra. Universitaria (Predios Huachi) / Casilla 334/
Telefax: 03-2851894 – 2411537, Correo Electrónico: carrera.sistemas@uta.edu.ec
AMBATO-ECUADOR



Universidad Técnica de Ambato

Facultad de Ingeniería en Sistemas, Electrónica e Industrial

Título: Patrones de Diseño Creacionales.

Carrera: Software

Nivel y Paralelo: Quinto "A"

Alumnos participantes: Ayme David

Cortez Henry

Naranjo Edder

Tite Matías

Asignatura: Patrones de Software

Docente: Ing. Alberto Pérez

Instrucciones

Realizar 4 ejercicios prácticos de los patrones creacionales vistos en clases.

Desarrollo

1. Patrón Singleton

Objetivo:

Implementar el patrón Singleton para gestionar la configuración de una aplicación.

Descripción:

Imagina que estás desarrollando una aplicación que necesita cargar y acceder a la configuración en múltiples partes del sistema. Esta configuración puede incluir opciones como el idioma, la configuración de la interfaz de usuario, las preferencias de usuario, etc.



Utilizando el patrón Singleton, crearemos una clase ConfigManager que encapsulará la lógica de carga y acceso a la configuración. Esta clase tendrá un método estático getInstance() que devolverá la única instancia de la configuración. Si la instancia aún no existe, se creará la primera vez que se llame a getInstance(), y en sucesivas llamadas, simplemente se devolverá la instancia existente.

Desarrollo:

1. Definir la clase ConfigManager.

Se crea una clase llamada ConfigManager que actúa como el Singleton para gestionar la configuración de la aplicación.

```
● ● ●  
1 public class ConfigManager {  
2  
3 }
```

2. Declaración de Variables.

Se declara una variable estática instance de tipo ConfigManager que almacenará la única instancia de la clase, de igual manera se un objeto config de tipo Map<String, String> para almacenar la configuración de la aplicación.

```
● ● ●  
1 public class ConfigManager {  
2     private static ConfigManager instance = null;  
3     private HashMap<String, String> config;
```

3. Constructor Privado.

Se define un constructor privado para evitar la instanciación directa de la clase desde fuera de la misma, además dentro del constructor, se inicializa el mapa de configuración y se llama al método loadConfig() para cargar la configuración inicial.



```
● ● ●  
1 private ConfigManager() {  
2     config = new HashMap<>();  
3     loadConfig();  
4 }
```

4. Método getInstance.

Se define un método estático getInstance() que devuelve la única instancia de ConfigManager, si la instancia aún no existe (es decir, instance es null), se crea una nueva instancia de ConfigManager y se asigna a instance y si ya existe una instancia simplemente la devuelve.

```
● ● ●  
1 public static ConfigManager getInstance() {  
2     if (instance == null) {  
3         instance = new ConfigManager();  
4     }  
5     return instance;  
6 }
```

5. Método loadConfig.

Se define un método privado loadConfig() que carga la configuración inicial de la aplicación, en este ejemplo ya se establecen ciertos valores por defecto para simular la carga inicial de una configuración predeterminada.



```
 1  private void loadConfig() {  
 2      config.put("lenguaje", "es");  
 3      config.put("tema", "oscuro");  
 4      config.put("fuente", "Arial");  
 5      config.put("zonaHoraria", "UTC");  
 6  
 7      System.out.println("Configuración cargada");  
 8  }
```

6. Método getConfig(String key).

Se define un método público getConfig(String key) que devuelve el valor de configuración correspondiente a la clave especificada.

```
 1  public String getConfig(String key) {  
 2      return config.get(key);  
 3  }
```

7. Método main().

En el método main se ejemplifica el uso del singleton, se llama a getInstance() dos veces para obtener dos instancias de ConfigManager, pero ambas referencian la misma instancia debido a que es un Singleton. Se imprime la comparación de las dos instancias, lo que debería resultar en true y, por último, se accede a la configuración utilizando getConfig(String key) para obtener las configuraciones precargadas en este ejemplo.



```
1 public class App {  
2     public static void main(String[] args) {  
3         ConfigManager confInstanceOne = ConfigManager.getInstance();  
4         ConfigManager confInstanceTwo = ConfigManager.getInstance();  
5  
6         System.out.println(confInstanceOne == confInstanceTwo);  
7  
8         String language = confInstanceOne.getConfig("lenguaje");  
9         String theme = confInstanceTwo.getConfig("tema");  
10        String font = confInstanceOne.getConfig("fuente");  
11        String timeZone = confInstanceTwo.getConfig("zonaHoraria");  
12  
13        System.out.println("Lenguaje: " + language);  
14        System.out.println("Tema: " + theme);  
15        System.out.println("Fuente: " + font);  
16        System.out.println("Zona horaria: " + timeZone);  
17    }  
18 }
```

Configuración cargada
true
Lenguaje: es
Tema: oscuro
Fuente: Arial
Zona horaria: UTC

2. Patrón Factory Method

Objetivo:

Implementar el patrón Factory para crear instancias de objetos sin exponer la lógica de creación de objetos directamente en el código cliente.

Descripción:

Imagina que estás trabajando en un sistema de creación de vuelos. Necesitas implementar el patrón Factory para asignar distintos tipos de vuelos de manera eficiente, ya que un vuelo puede ser comercial, internacional, nacional o uno contratado.

En este caso, un vuelo tiene: un **origen**, un **destino**, una **fecha** de salida y una **hora** de partida; Un vuelo de carga adicionalmente tiene un **peso máximo** que puede llevar; Un vuelo contratado (charter) solo puede ser solicitado si existe algún **cliente** (cedula o nombre); Un vuelo internacional tiene que pagar un **impuesto** por salir al exterior; y finalmente, un vuelo nacional tiene que pagar una **tarifa** de abordaje. Luego de haber establecido las características de cada vuelo, se desea que se imprima todos los datos de dicho vuelo junto a su tipo perteneciente.



Desarrollo:

1. Definir la clase Flight.

Se crea la clase Flight con las propiedades: origen, destino, fecha y hora. Adicionalmente, se crea un constructor vacío y un método para asignar valores a cada atributo, este método recibirá un array de argumentos. Adicionalmente, se crean getters y setters.

```
import java.time.LocalDate;
import java.time.LocalTime;
public class Flight {
    private String origin;
    private String destination;
    private LocalDate date;
    private LocalTime time;

    public void setAttributes(Object ... attributes) {
        this.origin = attributes[0].toString();
        this.destination = attributes[1].toString();
        this.date = (LocalDate) attributes[2];
        this.time = (LocalTime) attributes[3];
    }

    public Flight() {
    }
```

2. Sobrescribir método `toString`.

Se sobrescribirá el método `toString` para hacer que imprima los datos del objeto en lugar de su asignación de memoria. Esto se realiza aplicando polimorfismo.

```
@Override
public String toString() {
    return "destino=" + destination + ", fecha=" + date + ", hora=" + time + ", origen=" + origin ;
}
```

3. Definir la clase `CargoFlight`.

Se crea una clase `CargoFlight` con las propiedades: `maxWeight` y la extenderemos de la clase `Flight` aplicando herencia. Adicionalmente, se crea un constructor vacío y se sobrescribe el método para asignar valores a cada atributo. Por último, se crearán getters y setters de los nuevos atributos.



```
public class CargoFlight extends Flight{  
  
    private double maxWeight;  
  
    @Override  
    public void setAttributes(Object... atributes) {  
  
        super.setAttributes(atributes);  
        this.maxWeight = Double.parseDouble(atributes[4].toString());  
    }  
  
    public CargoFlight(){  
    }  
}
```

4. Sobrescribir método **toString**.

Se sobrescribirá el método **toString** para hacer que imprima los datos del objeto junto al tipo de **Flight**.

```
@Override  
public String toString() {  
    return "VueloCarga ["+super.toString()+", pesoMaximo=" + maxWeight + "]";  
}
```

5. Definir la clase **CharterFlight**.

Se crea una clase **CharterFlight** con las propiedades: **customer** y la extenderemos de la clase **Flight** aplicando herencia. Adicionalmente, se crea un constructor vacío y se sobrescribe el método para asignar valores a cada nuevo atributo. Por último, se crearán getters y setters de los nuevos atributos.

```
public class CharterFlight extends Flight{  
    private String customer;  
  
    @Override  
    public void setAttributes(Object... atributes) {  
  
        super.setAttributes(atributes);  
        this.customer = (String)atributes[4];  
    }  
  
    public CharterFlight(){  
    }  
}
```



6. Sobrescribir método `toString`.

Se sobrescribirá el método `toString` para hacer que imprima los datos del objeto junto al tipo de `Flight`.

```
@Override  
public String toString() {  
    return "VueloCharter ["+super.toString()+"], cliente=" + customer + "]";  
}
```

7. Definir la clase `InternationalFlight`.

Se crea una clase `InternationalFlight` con las propiedades: `tax` y la extenderemos de la clase `Flight` aplicando herencia. Adicionalmente, se crea un constructor vacío y se sobrescribe el método para asignar valores a cada atributo. Por último, se crearán getters y setters de los nuevos atributos.

```
public class InternationalFlight extends Flight{  
    private double tax;  
  
    @Override  
    public void setAttributes(Object... attributes) {  
  
        super.setAttributes(attributes);  
        this.tax = (double)attributes[4];  
    }  
  
    public InternationalFlight(){  
    }  
}
```

8. Sobrescribir método `toString`.

Se sobrescribirá el método `toString` para hacer que imprima los datos del objeto junto al tipo de `Flight`.

```
@Override  
public String toString() {  
    return "VueloInternacional ["+super.toString()+"], impuesto=" + tax + "]";  
}
```

9. Definir la clase `NationalFlight`.

Se crea una clase `NationalFlight` con las propiedades: `fee` y la extenderemos de la clase `Flight` aplicando herencia. Adicionalmente, se crea un constructor vacío y se sobrescribe el



método para asignar valores a cada atributo. Por último, se crearán getters y setters de los nuevos atributos.

```
public class NationalFlight extends Flight{  
    private double fee;  
  
    public NationalFlight(){  
    }  
  
    @Override  
    public void setAttributes(Object... attributes) {  
  
        super.setAttributes(attributes);  
        this.fee = (double)attributes[4];  
    }  
}
```

10. Sobrescribir método `toString`.

Se sobrescribirá el método `toString` para hacer que imprima los datos del objeto junto al tipo de `Flight`.

```
@Override  
public String toString() {  
    return "VueloNacional ["+super.toString()+", tarifa=" + fee + "]";  
}
```

11. Definir la clase `FactoryFlight`.

Para implementar el patrón Factory, se usará un `TreeMap` que contendrá la clave y el valor para cada elemento. En este caso, la clave será el tipo de `Flight` y el valor será la instancia de la clase de ese tipo.

```
public class FactoryFlight {  
  
    private final static TreeMap<String, Flight> flights = new TreeMap<String, Flight>();  
    static{  
        flights.put(key:"carga", new CargoFlight());  
        flights.put(key:"contrato", new CharterFlight());  
        flights.put(key:"internacional", new InternationalFlight());  
        flights.put(key:"nacional", new NationalFlight());  
    }  
}
```

12. Crear método `getFlight`.

Para obtener el valor de alguna de las claves, es necesario acceder al mapa. Para ello se crea el método `getFlight ()`, que recibe como parámetro el tipo de vuelo que se desea



instanciar. Este devolverá la instancia de un objeto Flight. Este método será estático para no tener que instanciar la clase FactoryFlight.

```
public static Vuelo getVuelo(String tipo){  
    return vuelos.get(tipo.toLowerCase());  
}
```

13. Probar el patrón Factory.

Se crea un objeto de tipo Flight para cada tipo de vuelo, y se llama al método estático getFlight de la clase Factory. Entonces, se llama a la función setAttributes () para asignar atributos a cada uno de los objetos de tipo Flight. Por último, se imprime para comprobar que el toString sea el correcto para cada tipo de Flight.

```
public class App  
{  
    Run | Debug  
    public static void main( String[] args )  
    {  
        Flight carga = FactoryFlight.getFlight(tipo:"carga");  
        carga.setAttributes(...atributes:"Quito", "Guayaquil", LocalDate.parse(text:"2021-06-01"),  
        LocalTime.parse(text:"10:00"), 200);  
        System.out.println(carga);  
  
        Flight charter = FactoryFlight.getFlight(tipo:"charter");  
        charter.setAttributes(...atributes:"Ecuador", "USA", LocalDate.parse(text:"2021-06-01"),  
        LocalTime.parse(text:"10:00"), "Daniel Noboa");  
        System.out.println(charter);  
  
        Flight internacional = FactoryFlight.getFlight(tipo:"internacional");  
        internacional.setAttributes(...atributes:"Ecuador", "Francia", LocalDate.parse  
        (text:"2021-06-01"), LocalTime.parse(text:"10:00"), 150.40);  
        System.out.println(internacional);  
  
        Flight nacional = FactoryFlight.getFlight(tipo:"nacional");  
        nacional.setAttributes(...atributes:"Quito", "Guayaquil", LocalDate.parse(text:"2021-06-01"),  
        LocalTime.parse(text:"10:00"), 50.40);  
        System.out.println(nacional);  
    }  
}
```

14. Revisar resultados.

Se revisa que en la consola se imprima correctamente cada instancia de vuelo.

```
VueloCarga [destino=Guayaquil, fecha=2021-06-01, hora=10:00, origen=Quito, pesoMaximo=200.0]  
VueloCharter [destino=USA, fecha=2021-06-01, hora=10:00, origen=Ecuador, cliente=Daniel Noboa]  
VueloInternacional [destino=Francia, fecha=2021-06-01, hora=10:00, origen=Ecuador, impuesto=150.4]  
VueloNacional [destino=Guayaquil, fecha=2021-06-01, hora=10:00, origen=Quito, tarifa=50.4]  
PS C:\xampp\htdocs\Quinto\patrones de diseño\factory_ejercicio>
```



UNIVERSIDAD TÉCNICA DE AMBATO

FACULTAD DE INGENIERÍA EN SISTEMAS, ELECTRÓNICA E INDUSTRIAL

CARRERA DE: SOFTWARE

Cdla. Universitaria (Predios Huachi) / Casilla 334/
Telefax: 03-2851894 – 2411537, Correo Electrónico: carrera.sistemas@uta.edu.ec
AMBATO-ECUADOR



3. Patrón Builder

Objetivo:

Implementar el patrón Builder para crear objetos complejos de manera flexible y mantenible.

Descripción:

Vamos a desarrollar un sistema para gestionar reservas de hoteles. El sistema permitirá configurar reservas de habitaciones de hotel con múltiples opciones personalizables tales como tipo de habitación, número de huéspedes, necesidades especiales (como accesibilidad), opciones de comidas, y duración de la estancia.

El sistema debe permitir a los gerentes de hotel y a su personal crear nuevos prototipos de habitaciones y suites, incluyendo información como nombre de la habitación, descripción, precio por noche, y características específicas de cada tipo de habitación, como el tamaño, el número de camas, vistas disponibles, y amenidades especiales. Además, es esencial que este sistema ofrezca la capacidad de clonar configuraciones existentes de habitaciones, permitiendo modificaciones rápidas para crear nuevas variantes sin tener que empezar de cero cada vez.

Desarrollo:

1. Definir la Clase HotelReservation.

Se crea una clase HotelReservation con propiedades como: roomType, numberOfGuests, needsAccessibility, meals, nights. Implementando su constructor, y sus métodos getter y setter para cada propiedad.

```
public class HotelReservation {
    private String roomType;
    private int numberOfGuests;
    private boolean needsAccessibility;
    private String meals;
    private int nights;

    public HotelReservation(String roomType, int numberOfGuests,
                           boolean needsAccessibility, String meals, int nights) {
        this.roomType = roomType;
        this.numberOfGuests = numberOfGuests;
        this.needsAccessibility = needsAccessibility;
        this.meals = meals;
        this.nights = nights;
    }
}
```



2. Definir la Interfaz ReservationBuilder

Esta interfaz declara métodos para configurar cada aspecto de un objeto HotelReservation, como el tipo de habitación, número de huéspedes, accesibilidad, opciones de comida y duración de la estancia.

```
public interface ReservationBuilder {  
    ReservationBuilder setRoomType(String roomType);  
    ReservationBuilder setGuests(int numberOfGuests);  
    ReservationBuilder setAccessibilityNeeds(boolean needsAccessibility);  
    ReservationBuilder setMealOptions(String meals);  
    ReservationBuilder setDuration(int nights);  
    HotelReservation build();  
}
```

3. Definir la Clase HotelReservationBuilder

Esta clase concreta se encarga de implementar los métodos de la interfaz ReservationBuilder. Mantiene una instancia del producto que está siendo construido y ofrece métodos para configurar cada parte de ese producto. Una vez todas las partes están configuradas, el método build() devuelve el producto final.

```
public class HotelReservationBuilder implements ReservationBuilder{  
    private String roomType;  
    private int numberOfGuests;  
    private boolean needsAccessibility;  
    private String meals;  
    private int nights;  
  
    @Override  
    public ReservationBuilder setRoomType(String roomType) {  
        this.roomType = roomType;  
        return this;  
    }
```



4. Definir la Clase ReservationDirector

Esta clase utiliza al ReservationBuilder para crear diferentes "tipos" predefinidos de reservas, como "estándar", "lujo", u otras reservaciones que la empresa decida agregar a futuro. simplificando el proceso de creación para los clientes que no necesitan configuraciones personalizadas.

```
public class ReservationDirector {  
    private ReservationBuilder builder;  
  
    public ReservationDirector(ReservationBuilder builder) {  
        this.builder = builder;  
    }  
  
    public HotelReservation createStandardReservation() {  
        return builder.setRoomType(roomType:"Standard")  
            .setGuests(numberOfGuests:2)  
            .setAccessibilityNeeds(needsAccessibility:false)  
            .setMealOptions(meals:"No meals")  
            .setDuration(nights:3)  
            .build();  
    }  
  
    public HotelReservation createLuxuryReservation() {  
        return builder.setRoomType(roomType:"Suite")  
            .setGuests(numberOfGuests:2)  
            .setAccessibilityNeeds(needsAccessibility:true)  
            .setMealOptions(meals:"Full board")  
            .setDuration(nights:7)  
            .build();  
    }  
}
```

5. Probar el Patrón Builder

En App.java, se inicializa un HotelReservationBuilder y un ReservationDirector para manejar la creación de reservas de hotel. Utilizando el director, se generan una reserva estándar y una de lujo mediante métodos que configuran automáticamente los parámetros de cada reserva. Adicionalmente, se muestra la flexibilidad del Builder con la creación de una reserva personalizada, donde se ajustan manualmente detalles como el tipo de habitación y las opciones de comidas. Cada tipo de reserva es presentada en consola utilizando printReservationDetails(), que organiza y muestra los atributos de cada reserva, demostrando la utilidad del patrón Builder en situaciones de configuración estandarizada y personalizada.



```
public class App {  
    Run|Debug  
    public static void main(String[] args) {  
        HotelReservationBuilder builder = new HotelReservationBuilder();  
        ReservationDirector director = new ReservationDirector(builder);  
  
        HotelReservation standardReservation = director.createStandardReservation();  
        printReservationDetails(title:"Reserva Estándar", standardReservation);  
  
        HotelReservation luxuryReservation = director.createLuxuryReservation();  
        printReservationDetails(title:"Reserva de Lujo", luxuryReservation);  
  
        HotelReservation customReservation = builder.setRoomType(roomType:"Deluxe")  
            .setGuests(numberOfGuests:3)  
            .setAccessibilityNeeds(needsAccessibility:true)  
            .setMealOptions(meals:"Breakfast only")  
            .setDuration(nights:4)  
            .build();  
        printReservationDetails(title:"Reserva Personalizada", customReservation);  
    }  
  
    private static void printReservationDetails(String title, HotelReservation reservation) {  
        System.out.println(title + " creada:");  
        System.out.println("Tipo de habitación: " + reservation.getRoomType());  
        System.out.println("Número de huéspedes: " + reservation.getNumberOfGuests());  
        System.out.println("Accesibilidad: " + (reservation.isNeedsAccessibility() ? "Sí" : "No"));  
        System.out.println("Opción de comidas: " + reservation.getMeals());  
        System.out.println("Duración de la estancia: " + reservation.getNights() + " noches\n");  
    }  
}
```

6. Resultados del Patrón Builder

Una vez ejecuta dicho patrón, presentó los siguientes resultados en consola al ejecutarlo.

```
Reserva Estándar creada:  
Tipo de habitación: Standard  
Número de huéspedes: 2  
Accesibilidad: No  
Opción de comidas: No meals  
Duración de la estancia: 3 noches  
  
Reserva de Lujo creada:  
Tipo de habitación: Suite  
Número de huéspedes: 2  
Accesibilidad: Sí  
Opción de comidas: Full board  
Duración de la estancia: 7 noches
```



Reserva Personalizada creada:
Tipo de habitación: Deluxe
Número de huéspedes: 3
Accesibilidad: Sí
Opción de comidas: Breakfast only
Duración de la estancia: 4 noches

4. Patrón Prototype

Objetivo:

Implementar el patrón Prototype con la finalidad de optimizar el proceso de creación de objetos en una aplicación de gestión de inventario para una tienda de electrónicos.

Descripción:

En un herbolario, es común gestionar una variedad de plantas con diferentes propiedades y características. Para optimizar el proceso de registro de nuevas plantas y permitir la personalización de estas, se requiere implementar el patrón Prototype.

El sistema debe permitir a los empleados crear nuevos prototipos de plantas, incluyendo información como nombre, descripción, precio y características específicas de la planta, como tipo de suelo preferido y altura.

Desarrollo:

1. Se crea la interfaz ClonablePlantInterface en donde se implementa el método clone, el que retorna una instancia de ClonablePlantInterface. Este método será implementado por clases que deseen permitir la clonación de sus instancias.

```
3  public interface ClonablePlantInterface {  
4      ClonablePlantInterface clone();  
5  }
```

2. Luego se crea una clase Plant que implementa la interfaz ClonablePlantInterface. La clase Plant tiene tres propiedades privadas: name, description, y price. Hay un constructor que inicializa estos campos, y métodos get y set para acceder y modificar las propiedades.



```
3 class Plant implements ClonablePlantInterface {  
4     private String name;  
5     private String description;  
6     private double price;  
7  
8     public Plant(String name, String description, double price) {  
9         this.name = name;  
10        this.description = description;  
11        this.price = price;  
12    }  
13  
14    public String getName() {  
15        return this.name;  
16    }  
17  
18    public void setName(String name) {  
19        this.name = name;  
20    }  
21  
22    public String getDescription() {  
23        return this.description;  
24    }  
25  
26    public void setDescription(String description) {  
27        this.description = description;  
28    }  
29
```

3. Una vez que hayamos hecho los getters y setters implementamos el método clone() de la interfaz ClonablePlantInterface. Dentro del método, se crea una nueva instancia de Planta con los mismos valores de nombre, descripción y precio de la instancia actual, dicha instancia la retornamos.

También sobrescribimos el método `toString()` ya que nos proporcionara una representación en forma de String de la instancia de `Plant`, incluyendo su nombre, descripción, y precio. Tal como se puede ver a continuación:



UNIVERSIDAD TÉCNICA DE AMBATO

FACULTAD DE INGENIERÍA EN SISTEMAS, ELECTRÓNICA E INDUSTRIAL

CARRERA DE: SOFTWARE
Cdra. Universitaria (Predios Huachi) / Casilla 334/
Telefax: 03-2851894 – 2411537, Correo Electrónico: carrera.sistemas@uta.edu.ec
AMBATO-ECUADOR



```
class Plant implements ClonablePlantInterface {  
    @Override  
    public ClonablePlantInterface clone() {  
        return new Plant(this.name, this.description, this.price);  
    }  
  
    @Override  
    public String toString() {  
        return "Plant [name=" + name + ", description=" + description + ", price=" + price + "]  
    };  
}
```

4. Luego creamos la clase InventoryManager para manejar el almacenamiento y la clonación de prototipos de plantas, permitiendo duplicar objetos sin necesidad de conocer los detalles de su construcción.
 - La clase tiene un atributo privado prototipos que es un HashMap, donde la clave es un String y el valor es una ClonablePlantInterface.
 - El método addPrototype permite añadir nuevos prototipos de planta al HashMap.
 - El método createPlant intenta obtener un prototipo del HashMap utilizando la clave proporcionada. Si el prototipo existe (no es null), se clona utilizando el método clonar() y se devuelve la copia. Si no existe, retorna null.
 - Hay un método getPrototypes() que devuelve el HashMap completo de prototipos.

```
import java.util.HashMap;  
  
public class InventoryManager {  
    private HashMap<String, ClonablePlantInterface> prototypes = new HashMap<>();  
  
    public void addPrototype(String name, ClonablePlantInterface prototype) {  
        prototypes.put(name, prototype);  
    }  
  
    public ClonablePlantInterface createPlant(String key) {  
        ClonablePlantInterface prototype = prototypes.get(key);  
        if (prototype != null) {  
            return prototype.clone();  
        }  
        return null;  
    }  
  
    public HashMap<String, ClonablePlantInterface> getPrototypes() {  
        return this.prototypes;  
    }  
}
```



Probar el patrón Prototype

- Se crea una instancia de InventoryManager, que se supone que maneja la creación de prototipos de plantas.
- Se crea una nueva Plant llamada prototipoLavanda con nombre "Lavanda", una descripción "Planta aromática", y un precio de 5.0.
- Se añade el prototipoLavanda al gestor de inventario con la clave "Lavanda".
- Se crea lavanda1 y lavanda2 clonando el prototipo "Lavanda" desde el gestor de inventario, que devuelve objetos de tipo Plant.
- Se cambia el precio de lavanda1 a 10.0 y de lavanda2 a 15.0, mostrando que las instancias clonadas son independientes entre sí.
- Finalmente, se imprime la información de lavanda1 y lavanda2 a la consola, que debería mostrar sus propiedades incluyendo los nuevos precios establecidos.

```
public class Main {  
    Run | Debug  
    public static void main(String[] args) {  
        InventoryManager manager = new InventoryManager();  
  
        Plant prototipoLavanda = new Plant(name:"Lavanda", description:"Planta aromática", price:5.0);  
  
        manager.addPrototype(name:"lavanda", prototipoLavanda);  
  
        //Clonacion de la planta Lavanda  
        Plant lavanda1 = (Plant) manager.createPlant(key:"lavanda");  
        Plant lavanda2 = (Plant) manager.createPlant(key:"lavanda");  
  
        //Cambio de precio de la planta Lavanda  
  
        lavanda1.setPrice(price:10.0);  
        lavanda2.setPrice(price:15.0);  
  
        //Impresion de las plantas Lavanda  
        System.out.println("\nPlanta 1: " + lavanda1 + "\nPlanta 2: " + lavanda2);  
    }  
}
```

Impresión por consola:

```
Planta 1: Plant [name=Lavanda, description=Planta aromática, price=10.0]  
Planta 2: Plant [name=Lavanda, description=Planta aromática, price=15.0]
```



UNIVERSIDAD TÉCNICA DE AMBATO
FACULTAD DE INGENIERÍA EN SISTEMAS, ELECTRÓNICA E INDUSTRIAL

CARRERA DE: SOFTWARE

Cdla. Universitaria (Predios Huachi) / Casilla 334/
Telefax: 03-2851894 – 2411537, Correo Electrónico: carrera.sistemas@uta.edu.ec
AMBATO-ECUADOR



Link de los ejercicios:

https://drive.google.com/file/d/106sp3u8m6pcNFVB_bd182RAEJKaVCi94/view?usp=sharing



INTEGRANTES: Camino Jose, Jimenez Patricio, Jurado Adrian, Moreira Josue

CARRERA: Software

PARALELO: A

ASIGNATURA: Patrones de diseño de software

NIVEL: 5

FECHA: 05/30/2024

TEMA: Patrones creacionales

PATRÓN SINGLETON

Objetivo:

Implementar el patrón Singleton para asegurarse que solo exista una sola instancia del objeto que vamos a utilizar.

Descripción:

Supongamos que estamos trabajando en un sistema de manejo de aeropuertos, para asegurarse que solo exista una instancia de la base de datos donde tenemos almacenado la información de nuestros aeropuertos podemos usar el patrón singleton.

Desarrollo:

1. Definir los atributos que simularan ser tablas de nuestra base de datos de Aeropuertos.

```
public class DbAirport {  
    private Set<String> airports = Set.of(e1:"MAD", e2:"BCN", e3:"LHR", e4:"CDG", e5:"FRA",  
    e6:"IST", e7:"AMS", e8:"FCO", e9:"CPH", e10:"ZRH");  
    private Set<String> airplanes = Set.of(e1:"A320", e2:"A330", e3:"A350", e4:"B737",  
    e5:"B747", e6:"B777", e7:"B787");  
    private Set<String> flights = Set.of(e1:"IB051", e2:"IB052", e3:"IB053", e4:"IB054",  
    e5:"IB055", e6:"IB056", e7:"IB057", e8:"IB058", e9:"IB059", e10:"IB060");  
}
```

2. Implementamos un atributo estático en nuestra clase para llamar la instancia de nuestra base de datos sin declaración de un constructor.

```
private static DbAirport instance;
```

3. Implementamos el patrón singleton asegurándose de que solo exista una instancia de esa clase (nuestra base de datos).

```
public static DbAirport getInstance() {  
    if (instance == null) {  
        instance = new DbAirport();  
    }  
    return instance;  
}
```

4. Implementamos métodos para acceder a la información de nuestra base de datos que simulan tablas.



```
public Set<String> getAirports() {  
    return this.airports;  
}  
  
public Set<String> getAirplanes() {  
    return this.airplanes;  
}  
  
public Set<String> getFlights() {  
    return this.flights;  
}
```

5. Probar el patrón Singleton. Conseguimos la referencia de nuestra base de datos y usamos nuestros métodos de clase para acceder a nuestros aeropuertos, vuelos y aviones que tenemos en nuestros aeropuertos. Realizamos una comprobación de que ambas instancias son iguales.

```
public class App {  
    Run | Debug  
    public static void main(String[] args) {  
        var bAirport = DbAirport.getInstance();  
        var bAirport2 = DbAirport.getInstance();  
        var airports = bAirport.getAirports();  
        var airplanes = bAirport.getAirplanes();  
        var flights = bAirport.getFlights();  
        for(String airport : airports) {  
            System.out.println("Aiports:\n" + airport);  
        }  
  
        System.out.println();  
  
        for(String airplane : airplanes) {  
            System.out.println("Airplanes\n" + airplane);  
        }  
  
        System.out.println();  
  
        for(String flight : flights) {  
            System.out.println("Flights\n" + flight);  
        }  
  
        System.out.println();  
  
        System.out.println("Check If the two instances are the same\n" + (bAirport == bAirport2));  
    }  
}
```

Resultados:



```
Aiports:  
ZRH  
  
Airplanes  
A320  
  
Flights  
IB054  
  
Check If the two instances are the same  
true
```

PATRÓN BUILDER

Objetivo:

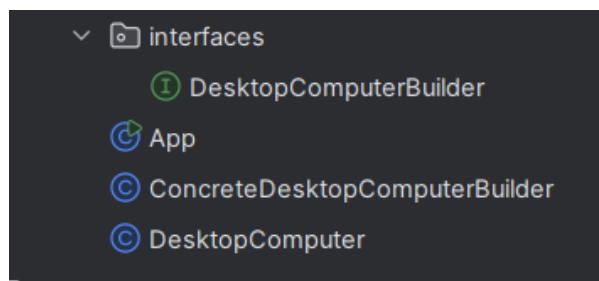
Implementar el patrón **Builder** para la creación de objetos complejos de forma que sea mantenible y a su vez flexible

Descripción:

Dado el caso de trabajar en un sistema de creación de computadoras de escritorio. Se necesita implementar el patrón Builder para construir diferentes tipos de computadoras de escritorio, y que estas creaciones se logren de forma eficiente y efectiva.

Desarrollo:

Estructura del ejercicio:



1. Definir la interface DesktopComputerBuilder

Se crea la interface DesktopComputerBuilder donde albergara los métodos de setCpu, set Gpu, setRamGB, setStorageGB, setOs y el método build para la creación de los objetos.



```
1 package com.patrones.interfaces;
2
3 import com.patrones.DesktopComputer;
4
5 public interface DesktopComputerBuilder {
6     DesktopComputerBuilder setCpu(String cpu);
7
8     DesktopComputerBuilder setGpu(String gpu);
9
10    DesktopComputerBuilder setRamGB(int ramGB);
11
12    DesktopComputerBuilder setStorageGB(int storageGB);
13
14    DesktopComputerBuilder setOs(String os);
15
16    DesktopComputer build();
17}
```

2. Definir la Clase DesktopComputer

Se crea la clase DesktopComputer la cual se encargará de tener las características y métodos que, en base a la logística, tendría una computadora de escritorio.

```
1 package com.patrones;
2
3 public class DesktopComputer {
4     private String cpu;
5     private String gpu;
6     private int ramGB;
7     private int storageGB;
8     private String os;
9
10    public DesktopComputer(String cpu, String gpu, int ramGB, int storageGB, String os) {
11        this.cpu = cpu;
12        this.gpu = gpu;
13        this.ramGB = ramGB;
14        this.storageGB = storageGB;
15        this.os = os;
16    }
17    > public String getCpu() { return cpu; }
18    > public String getGpu() { return gpu; }
19    > public int getRamGB() { return ramGB; }
20    > public int getStorageGB() { return storageGB; }
21    > public String getOs() { return os; }
22
23    @Override
24    public String toString() {
25        return "DesktopComputer{" +
26            "cpu='" + cpu + '\'' +
27            ", gpu='" + gpu + '\'' +
28            ", ramGB=" + ramGB +
29            ", storageGB=" + storageGB +
30            ", os='" + os + '\'' +
31            '}';
32    }
33}
34
```

3. Definir la Clase ConcreteDesktopComputerBuilder

Se crea la clase y se implementa la interface, clase cuyo contenido será la sobre escritura de los atributos y métodos encontrados en la interface, pero en base a las



características propias que tendría una computadora de escritorio además en el método build de esta clase, se encontrara la creación de los objetos.

```
1 package com.patrones;
2
3 import com.patrones.interfaces.DesktopComputerBuilder;
4
5 public class ConcreteDesktopComputerBuilder implements DesktopComputerBuilder { 6 usages
6     private String cpu; 2 usages
7     private String gpu; 2 usages
8     private int ramGB; 2 usages
9     private int storageGB; 2 usages
10    private String os; 2 usages
11
12    @Override 1 usage
13    public ConcreteDesktopComputerBuilder setCpu(String cpu) {...}
14    @Override 1 usage
15    public ConcreteDesktopComputerBuilder setGpu(String gpu) {...}
16    @Override 1 usage
17    public ConcreteDesktopComputerBuilder setRamGB(int ramGB) {...}
18    @Override 1 usage
19    public ConcreteDesktopComputerBuilder setStorageGB(int storageGB) {...}
20    @Override 1 usage
21    public ConcreteDesktopComputerBuilder setOs(String os) {...}
22    @Override 1 usage
23    public DesktopComputer build() {
24        return new DesktopComputer(cpu, gpu, ramGB, storageGB, os);
25    }
26}
27
28}
```

PROBAR EL PATRÓN BUILDER

Creacion de un objeto DesktopComputer

Primero, se crea una instancia del builder concreto `ConcreteDesktopComputerBuilder`. Luego, se utilizan los métodos encadenados del builder para establecer los diferentes atributos del objeto `DesktopComputer`, como la CPU, la GPU, la cantidad de RAM, el almacenamiento y el sistema operativo. Esto permite configurar las propiedades del objeto de manera flexible y en cualquier orden. Finalmente, se llama al método `build()` para obtener el objeto `DesktopComputer` completamente configurado. Una vez construido, se imprime el objeto utilizando `System.out.println()`. Este enfoque proporciona una manera clara y concisa de construir objetos complejos con múltiples atributos en Java.



```
1 package com.patrones;
2 import com.patrones.interfaces.DesktopComputerBuilder;
3
4
5 public class App
6 {
7     public static void main( String[] args )
8     {
9
10         DesktopComputerBuilder builder = new ConcreteDesktopComputerBuilder();
11         DesktopComputer computer = builder
12             .setCpu("Intel Core i7")
13             .setGpu("NVIDIA GeForce RTX 3080")
14             .setRamGB(16)
15             .setStorageGB(512)
16             .setOs("Windows 10")
17             .build();
18
19         System.out.println(computer);
20     }
21
22 }
```

```
Run App x
G : 
"C:\Program Files\Java\jdk-19\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2024.1\lib\idea_rt.jar=53151:C:\Program Files\JetBrains\IntelliJ IDEA 2024.1\bin" -Dfile.encoding=UTF-8 DesktopComputer{cpu='Intel Core i7', gpu='NVIDIA GeForce RTX 3080', ramGB=16, storageGB=512, os='Windows 10'}
Process finished with exit code 0
```

FACTORY METHOD

Objetivo

Implementar en un ejercicio sobre una pizzería el patrón de diseño Factory method para la creación de objetos de tipo Pizza.

Descripción

Imagina que estás desarrollando un sistema de pedidos en línea para un restaurante de pizzas que ofrece una amplia variedad de pizzas, como pizza de pepperoni, pizza vegetariana, pizza de carne, etc. Cada tipo de pizza tiene su propia receta y proceso de preparación específico.

Desarrollo

1. Creación de la interfaz “Pizza”.

La interfaz pizza contendrá los métodos encargados del proceso de creación de una pizza.



```
package com.factory.method;
💡
public interface Pizza { 9 usages 2 implementations
    void prepare(); 1 usage 2 implementations
    void bake(); 1 usage 2 implementations
    void cut(); 1 usage 2 implementations
    void packUp(); 1 usage 2 implementations
}
```

2. Creación de la clase “BeefPizza”.

BeefPizza al ser un tipo de pizza implementa y sobre escribe los métodos de la interfaz pizza para su uso,

```
package com.factory.method;
💡
public class BeefPizza implements Pizza{ 1 usage
    @Override 1 usage
    public void prepare() {
        System.out.println("La pizza de carne en camino");
    }

    @Override 1 usage
    public void bake() {
        System.out.println("Horneando la pizza de carne");
    }

    @Override 1 usage
    public void cut() {
        System.out.println("Cortando la pizza de carne");
    }

    @Override 1 usage
    public void packUp() {
        System.out.println("Empacando la pizza de carne" );
    }
}
```

3. Creación de la clase “NeapolitanPizza”

Al igual que beefPizza, NeapolitanPizza implementa los metodos de pizza para su uso.



```
package com.factory.method;

public class NeapolitanPizza implements Pizza{ 1 usage
    @Override 1 usage
    public void prepare() {
        System.out.println("Pizza napolitana en camino");
    }

    @Override 1 usage
    public void bake() {
        System.out.println("Horneando la pizza napolitana");
    }

    @Override 1 usage
    public void cut() {
        System.out.println("Cortando la pizza napolitana");
    }

    @Override 1 usage
    public void packUp() {
        System.out.println("Empacando la pizza napolitana");
    }
}
```

4. Creación de la clase abstracta “Pizzeria”

Pizzeria es la encargada de la implementación de los métodos, en general, se puede definir como la clase “Factory” en el ejercicio.

```
public abstract class Pizzeria { 4 usages 2 inheritors

    public void preparePizza() { 2 usages
        System.out.println("Preparando pizza, por favor espera un momento");
        Pizza pizza = createPizza();
        pizza.prepare();
    }

    public void bakePizza() { 2 usages
        System.out.println("Horneando pizza...");
        Pizza pizza = createPizza();
        pizza.bake();
    }

    public void cutPizza() { 2 usages
        System.out.println("Cortando pizza...");
        Pizza pizza = createPizza();
        pizza.cut();
    }

    public void packUpPizza() { 2 usages
        System.out.println("Empacando pizza...");
        Pizza pizza = createPizza();
        pizza.packUp();
    }

    public abstract Pizza createPizza(); 4 usages 2 implementations
}
```



5. Creación de las clases relacionadas al factory Pizzeria

Implementa la clase Pizzeria para encargarse de la creación de los objetos relacionados, en este caso, la creación NeapolitanPizza.

```
public class NeapolitanPizzaPizzeria extends Pizzeria { 1 usage

    @Override 4 usages
    public Pizza createPizza() {
        System.out.println("Haciendo Pizza Napolitana...");
        return new NeapolitanPizza();
    }
}
```

Implementa la clase Pizzeria para encargarse de la creación de los objetos relacionados, en este caso, la creación BeefPizza.

```
public class BeefPizzaPizzeria extends Pizzeria { 1 usage
    @Override 4 usages
    public Pizza createPizza() {
        System.out.println("Haciendo pizza de carne...");
        return new BeefPizza();
    }
}
```

6. Implementación en la clase “App” para probar el código.

Creamos los objetos beefPizzaRes y neapolitanPizzaRes, siendo de tipo BeefPizzaPizzeria y NeapolitanPizzaPizzeria respectivamente, ejecutando los métodos relacionados a cada uno, los cuales eran, preparePizza, bakePizza, cutPizza y packUpPizza.

```
public class App {
    public static void main(String[] args) {
        Pizzeria beefPizzaRes = new BeefPizzaPizzeria();
        beefPizzaRes.preparePizza();
        beefPizzaRes.bakePizza();
        beefPizzaRes.cutPizza();
        beefPizzaRes.packUpPizza();

        System.out.println("=====");

        Pizzeria neapolitanPizzaRes = new NeapolitanPizzaPizzeria();
        neapolitanPizzaRes.preparePizza();
        neapolitanPizzaRes.bakePizza();
        neapolitanPizzaRes.cutPizza();
        neapolitanPizzaRes.packUpPizza();
    }
}
```



7. Ejecución del código

```
Preparando pizza, por favor espera un momento
La pizza de carne en camino
Horneando pizza...
Horneando la pizza de carne
Cortando pizza...
Cortando la pizza de carne
Empacando pizza...
Empacando la pizza de carne
=====
Preparando pizza, por favor espera un momento
Pizza napolitana en camino
Horneando pizza...
Horneando la pizza napolitana
Cortando pizza...
Cortando la pizza napolitana
Empacando pizza...
Empacando la pizza napolitana
```

PATRÓN PROTOTYPE

Objetivo:

Implementar el patrón Prototype para crear copias del objeto original sin tener que crear una nueva instancia de este.

Descripción:

Supongamos que vamos a crear una API de Pokemones para consumirla en algún servicio, la mayoría de los objetos entre pokemones tendrán las mismas propiedades, en lugar de crear una nueva instancia del pokemon podemos clonar el mismo y cambiarle los atributos específicos para cada tipo de pokemon.

Desarrollo:

1. Definimos una clase abstracta que implementará el método que nos permitirá crear la clonación de los objetos

```
public abstract class Pokemon {
    public abstract Pokemon clone();
}
```



2. Creamos la clase de pokemon tipo “Aqua” que heredara de la clase Pokemon, implementamos sus atributos, su constructor, sus getters y setters.

```
public class Aqua extends Pokemon {  
  
    private int power;  
    private int health;  
    private String type;  
    private String color = "undefined";  
  
    public Aqua() {}  
  
    public Aqua(int power, int health, String type) {  
        this.power = power;  
        this.health = health;  
        this.type = type;  
    }  
  
    public int getPower() {  
        return power;  
    }  
  
    public void setPower(int power) {  
        this.power = power;  
    }  
}
```

Implementamos el método clone que nos obliga nuestra clase abstracta y así poder clonar nuestro objeto, sobre escribimos el método toString() para poder leer mejor el objeto al momento de las demostraciones de este patrón.

```
@Override  
public Pokemon clone() {  
    var aqua = new Aqua(this.getPower(), this.getHealth(), this.getType());  
    aqua.setColor(this.getColor());  
    return aqua;  
}  
  
@Override  
public String toString() {  
    return "power " + getPower() + ", Health " + getHealth() + ", Type " +  
           getType() + ", color " + getColor();  
}
```

3. Creamos la clase de pokemon tipo “Terra” que heredara de la clase Pokemon, implementamos sus atributos, su constructor, sus getters y setters.



```
public class Terra extends Pokemon {  
    private int power;  
    private int health;  
    private String type;  
    private int height;  
  
    public Terra() {}  
  
    public Terra(int power, int health, String type) {  
        this.power = power;  
        this.health = health;  
        this.type = type;  
    }  
  
    public int getPower() {  
        return power;  
    }  
  
    public void setPower(int power) {  
        this.power = power;  
    }  
  
    public int getHealth() {  
        return health;  
    }  
}
```

Implementamos el método clone que nos obliga nuestra clase abstracta y así poder clonar nuestro objeto, sobre escribimos el método toString() para poder leer mejor el objeto al momento de las demostraciones de este patrón.

```
@Override  
public Pokemon clone() {  
    var terra = new Terra(this.getPower(), this.getHealth(), this.getType());  
    terra.setHeight(this.getHeight());  
    return terra;  
}  
  
@Override  
public String toString() {  
    return "Power " + getPower() + ", Health " + getHealth() + ", Type " +  
        getType() + ", Height " + getHeight();  
}
```

Probar el patrón prototype

Declaramos un objeto original y otra la copia del objeto y modificamos los atributos necesarios para ese tipo de pokémon, así ya estaremos probando el método clone y consigo el patrón prototype, a los objetos originales les añadió en una lista cada uno para observar como es el objeto original y como es la copia con los atributos modificados.



```
public class App {
    Run | Debug
    public static void main(String[] args) {
        List<Pokemon> pokemons = new ArrayList<>();
        List<Pokemon> pokemonsClone = new ArrayList<>();
        var squirtle = new Aqua();
        squirtle.setPower(power:500);
        squirtle.setHealth(health:100);
        squirtle.setType(type:"Water");
        pokemons.add(squirtle);
        var gyarados = (Aqua) squirtle.clone();
        gyarados.setColor(color:"Blue");
        pokemonsClone.add(gyarados);

        var tirtuga = new Terra();
        tirtuga.setPower(power:350);
        tirtuga.setHealth(health:120);
        tirtuga.setType(type:"Ground");
        pokemons.add(tirtuga);
        var silicobra = (Terra) tirtuga.clone();
        silicobra.setHeight(height:2);
        pokemonsClone.add(silicobra);

        for(Pokemon pokemon: pokemons) {
            System.out.println("Pokemons original: " + pokemon);
        }

        for(Pokemon pokemon: pokemonsClone) {
            System.out.println("Pokemons clone: " + pokemon);
        }
    }
}
```

Resultados:

```
Pokemons original: power 500, Healt 100, Type Water, color undefined
Pokemons original: Power 350, Healt 120, Type Ground, Height 0
Pokemons clone: power 500, Healt 100, Type Water, color Blue
Pokemons clone: Power 350, Healt 120, Type Ground, Height 2
```



ESTUDIANTES:

- Emilia Galarza
- Jair Mera
- Pablo Villacrés
- Daniel Zhu

CARRERA: Software

PARALELO: A

ASIGNATURA: Patrones de Software

FECHA: 30/04/2024

NIVEL: 5

TEMA: Patrones de Diseño Creacionales

1. Singleton

Problemática: En un almacén grande que maneja múltiples productos, es crucial tener un sistema centralizado y consistente para gestionar el inventario. Dado que múltiples empleados pueden necesitar acceder y actualizar el inventario simultáneamente desde diferentes terminales o dispositivos, es esencial que todos vean y modifiquen la misma instancia del inventario para evitar discrepancias, duplicaciones o errores de datos.

Solución:

Utilizando el patrón Singleton, podemos asegurarnos de que el sistema de gestión de inventario mantenga una única instancia del inventario a través de la cual todos los accesos y actualizaciones se sincronizan correctamente. Esto garantiza que el estado del inventario sea coherente, actualizado y accesible de manera uniforme en todo momento.

Implementación en Java:

1. Clase Singleton InventoryManager:

Esta clase gestiona el inventario y asegura que sólo exista una instancia del inventario en todo el sistema.

Proporciona métodos para añadir, eliminar y consultar productos.



```
public class InventoryManager {  
    private static InventoryManager instance;  
    private Map<String, Integer> inventory;  
  
    private InventoryManager() {  
        inventory = new HashMap<>();  
    }  
  
    public static synchronized InventoryManager getInstance() {  
        if (instance == null) {  
            instance = new InventoryManager();  
        }  
        return instance;  
    }  
  
    public void addProduct(String product, int quantity) {  
        inventory.put(product, inventory.getOrDefault(product, 0) + quantity);  
    }  
  
    public void removeProduct(String product, int quantity) {  
        if (inventory.containsKey(product) && inventory.get(product) >= quantity) {  
            inventory.put(product, inventory.get(product) - quantity);  
        } else {  
            System.out.println("Insufficient stock or product does not exist.");  
        }  
    }  
  
    public int getProductQuantity(String product) {  
        return inventory.getOrDefault(product, 0);  
    }  
}
```

2. Demostración de uso del InventoryManager:

Simulamos el acceso y la modificación del inventario desde diferentes partes del sistema.

```
public class Main {  
    public static void main(String[] args) {  
        InventoryManager manager = InventoryManager.getInstance();  
  
        manager.addProduct("Laptop", 50);  
        manager.addProduct("Smartphone", 30);  
  
        manager.removeProduct("Laptop", 5);  
  
        System.out.println("Cantidad de Laptops: " + manager.getProductQuantity("Laptop"));  
        System.out.println("Cantidad de Smartphones: " + manager.getProductQuantity("Smartphone"));  
    }  
}
```



2. Factory

Se crea unidades militares (Infantry y Artillery) utilizando el patrón Factory. La fábrica (MilitaryUnitFactory) recibe el tipo de unidad y devuelve una instancia correspondiente de la unidad militar solicitada. Luego, en la clase principal (Main), se crea unidades de infantería y artillería y se muestra su funcionamiento.

Crear la interfaz que contiene los métodos comunes de todas las clases que van a existir en la fábrica

```
public interface MilitaryUnitInterface
    void attack();
    void move();
}
```

Crear las clases que van a crearse en la fábrica

```
public class Infantry implements MilitaryUnitInterface {
    @Override
    public void attack() {
        System.out.println("Infanteria atacando de cerca");
    }

    @Override
    public void move() {
        System.out.println("Infanteria moviendose rápidamente");
    }
}
```



```
public class Artillery implements MilitaryUnitInterface {
    @Override
    public void attack() {
        System.out.println("Artilleria atacando desde lejos");
    }

    @Override
    public void move() {
        System.out.println("Artilleria moviendose lentamente");
    }
}
```

Crear la clase fábrica con el método estático que crea las clases según el tipo que se pase

```
public class MilitaryUnitFactory {
    private static final Map<String, MilitaryUnitInterface> militaryUnits = new TreeMap<>();

    static {
        militaryUnits.put("infantry", new Infantry());
        militaryUnits.put("artillery", new Artillery());
    }

    public static MilitaryUnitInterface createMilitaryUnit(String type) {
        return militaryUnits.get(type);
    }
}
```



Se prueba la funcionalidad de la fábrica

```
public static void main(String[] args) {  
    MilitaryUnitInterface infantry = MilitaryUnitFactory.createMilitaryUnit("infantry");  
    infantry.attack();  
    infantry.move();  
  
    MilitaryUnitInterface artillery = MilitaryUnitFactory.createMilitaryUnit("artillery");  
    artillery.attack();  
    artillery.move();  
}
```

Infanteria atacando de cerca
Infanteria moviendose rápidamente
Artilleria atacando desde lejos
Artilleria moviendose lentamente



3. Builder

Se ha desarrollado un ejercicio implementando el patrón de diseño Builder en Java para ilustrar una forma eficiente y segura de construir consultas SQL dinámicamente. El patrón Builder es particularmente útil en este contexto ya que las consultas SQL pueden tener numerosos componentes opcionales y variantes que dependen de los requisitos específicos en tiempo de ejecución.

Crear la interfaz Builder que define los pasos para construir el objeto complejo:

```
public interface IQueryBuilder {  
    IQueryBuilder select(String columns);  
    IQueryBuilder from(String table);  
    IQueryBuilder where(String condition);  
    IQueryBuilder orderBy(String orderBy);  
    IQueryBuilder limit(int limit);  
    IQueryBuilder offset(int offset);  
    String build();  
}
```

Crear la clase QueryBuilder implementando la intefaz creada IQueryBuilder:

```
public class QueryBuilder implements IQueryBuilder {  
  
    private StringBuilder query = new StringBuilder();  
  
    public QueryBuilder select(String columns) {  
        query.append("SELECT ").append(columns).append(" ");  
        return this;  
    }  
  
    public QueryBuilder from(String table) {  
        query.append("FROM ").append(table).append(" ");  
        return this;  
    }  
  
    public QueryBuilder where(String condition) {  
        query.append("WHERE ").append(condition).append(" ");  
        return this;  
    }  
  
    public QueryBuilder orderBy(String orderBy) {  
        query.append("ORDER BY ").append(orderBy).append(" ");  
        return this;  
    }  
  
    public QueryBuilder limit(int limit) {  
        query.append("LIMIT ").append(limit).append(" ");  
        return this;  
    }  
  
    public QueryBuilder offset(int offset) {  
        query.append("OFFSET ").append(offset).append(" ");  
        return this;  
    }  
}
```



Agregar un método que nos permita “construir” el elemento que hemos ido cargando por partes

```
public class QueryBuilder implements IQueryBuilder {  
  
    public String build() {  
        return query.toString();  
    }  
  
}
```

Utilizar lo creado en el archivo main:

```
public static void main( String[] args )  
{  
    String queryBuilder = new QueryBuilder()  
        .select("name, age")  
        .from("users")  
        .where("age > 18")  
        .orderBy("name")  
        .limit(10)  
        .offset(0)  
        .build();  
    System.out.println(queryBuilder);  
}
```

Resultado obtenido:

```
SELECT name, age FROM users WHERE age > 18 ORDER BY name LIMIT 10 OFFSET 0  
SrPabliss ➔ builder-pattern ➔
```



4. Prototype

Problemática:

Supongamos que tenemos una aplicación que permite a los usuarios crear y personalizar formularios para diferentes eventos o necesidades. Cada formulario puede ser bastante complejo, incluyendo múltiples campos, configuraciones de validación, y estilos. Los usuarios a menudo necesitan crear formularios que son muy similares entre sí, con solo pequeñas modificaciones entre versiones.

Solución:

Utilizando el patrón Prototype, podemos permitir a los usuarios copiar un formulario existente y luego modificar esta copia, lo que ahorra tiempo y reduce errores en comparación con recrear un formulario similar desde cero. Este enfoque es eficiente porque se clonian los objetos complejos en lugar de reconstruirlos, preservando el estado base que no requiere modificación.

Implementación en JAVA:

1. Interfaz Prototype:

Se define un método para clonar el objeto.

```
public interface Prototype<T> {  
    T clone();  
}
```

2. Clase Form:

Se implementa la interfaz Prototype.

Se gestiona varios campos y configuraciones de un formulario.



```
import java.util.ArrayList;
import java.util.List;

public class Form implements Prototype<Form> {
    private String title;
    private List<String> fields;
    private boolean isPublic;

    public Form(String title, boolean isPublic) {
        this.title = title;
        this.fields = new ArrayList<>();
        this.isPublic = isPublic;
    }

    public void addField(String field) {
        fields.add(field);
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public void setPublic(boolean isPublic) {
        this.isPublic = isPublic;
    }

    @Override
    public Form clone() {
        Form copy = new Form(this.title, this.isPublic);
        for (String field : this.fields) {
            copy.addField(field);
        }
        return copy;
    }

    @Override
    public String toString() {
        return "Form{" +
            "title='" + title + '\'' +
            ", fields=" + fields +
            ", isPublic=" + isPublic +
            '}';
    }
}
```

3. Demostración de uso del Form:

Se crea un formulario base y luego clona para crear una versión modificada.



UNIVERSIDAD TÉCNICA DE AMBATO
FACULTAD DE INGENIERÍA EN SISTEMAS, ELECTRÓNICA E INDUSTRIAL

PERÍODO ACADÉMICO: MARZO 2024 – JULIO 2024



```
public class Main {  
    public static void main(String[] args) {  
        Form baseForm = new Form("Event Registration", true);  
        baseForm.addField("Name");  
        baseForm.addField("Email");  
        baseForm.addField("Date of Event");  
  
        Form companyEventForm = baseForm.clone();  
        companyEventForm.addField("Company Name");  
        companyEventForm.setTitle("Company Event Registration");  
  
        System.out.println(baseForm);  
        System.out.println(companyEventForm);  
    }  
}
```



UNIVERSIDAD TÉCNICA DE AMBATO



FACULTAD DE INGENIERÍA EN SISTEMAS, ELECTRÓNICA E
INDUSTRIAL

PATRONES DE SOFTWARE

PERÍODO ACADÉMICO: MARZO – AGOSTO 2024

PATRONES DE DISEÑO CREACIONALES

Integrantes: - Paul Villacis

- Jefferson Poaquiza
- Christian Sanchez
- Fernando Sánchez

Curso: 5to SW

Fecha: 30/04/2024

- 1) Realizar 4 ejercicios prácticos de los patrones revisados en clase:
Creacionales: Singleton, Factory, Builder, Prototype

PATRON BUILDER

Objetivo:

Implementar el patrón Builder para crear objetos de tipo "Tarea" en un sistema de gestión de tareas.

Descripción:

Imagina que estás trabajando en un sistema de gestión de tareas y necesitas implementar el patrón Builder para construir diferentes tipos de tareas de manera eficiente.

Desarrollo:

1.1 Definir la clase Task(Tarea) con sus atributos, constructores, getters y setters

```
1 package com.Villacis.Paul.ejercicioBuilder;
2
3 public class Task {
4     private String title;
5     private String description;
6     private String assignee;
7     private boolean completed;
8
9     public Task() {
10         super();
11     }
12
13     public Task(String title, String description, String assignee, boolean completed) {
14         super();
15         this.title = title;
16         this.description = description;
17         this.assignee = assignee;
18         this.completed = completed;
19     }
20
21     public String getTitle() {
22         return title;
23     }
24
25     public void setTitle(String title) {
26
27     public String getDescription() {
28         return description;
29     }
30
31     public void setDescription(String description) {
32         this.description = description;
33     }
34
35     public String getAssignee() {
36         return assignee;
37     }
38
39     public void setAssignee(String assignee) {
40         this.assignee = assignee;
41     }
42
43     public boolean isCompleted() {
44         return completed;
45     }
46
47     public void setCompleted(boolean completed) {
48         this.completed = completed;
49     }
50
51
52     @Override
53     public String toString() {
54         return "Task{" + "\n\ttitle='" + title + '\'' + ",\n\tdescription='" + description + '\'' + ",\n\tassignee=" +
55             + assignee + '\'' + ",\n\tcompleted=" + completed + "\n}";
56     }
57
58 }
59 }
```

1.2 Crear la Interfaz TaskBuilder:

Esta interfaz define los métodos que deben implementarse para construir una tarea. Cada método establece una propiedad de la tarea, como el título, la descripción, el responsable y si está completada. El método build() devuelve la tarea construida.

```
1 package com.Villacis.Paul.ejercicioBuilder;
2
3 public interface TaskBuilder {
4     void setTitle(String title);
5     void setDescription(String description);
6     void setAssignee(String assignee);
7     void setCompleted(boolean completed);
8     Task build();
9 }
10
```

1.3 Implementar el Builder:

Esta clase ConcreteTaskBuilder implementa la interfaz TaskBuilder. Se encarga de construir una tarea concreta, inicializando un objeto Task al inicio. Cada método setXxx() establece una propiedad de la tarea con los valores proporcionados. El método build() devuelve la tarea construida.

```
package com.Villacis.Paul.ejercicioBuilder;

public class ConcreteTaskBuilder implements TaskBuilder {

    private Task task;

    public ConcreteTaskBuilder() {
        this.task = new Task();
    }

    // Metodos para establecer las propiedades de la tarea

    @Override
    public void setTitle(String title) {
        task.setTitle(title);
    }

    @Override
    public void setDescription(String description) {
        task.setDescription(description);
    }

    @Override
    public void setAssignee(String assignee) {
        task.setAssignee(assignee);
    }

    @Override
    public void setCompleted(boolean completed) {
        task.setCompleted(completed);
    }

    // Metodo para construir y devolver la tarea
    @Override
    public Task build() {
        return task;
    }
}
```

1.4 Implementar el Director:

Esta clase TaskManager actúa como el director en el patrón Builder.

Tiene un método createTask() que toma un objeto TaskBuilder como parámetro.

Dentro de este método, se utiliza el objeto TaskBuilder para construir una tarea compleja siguiendo un flujo predefinido. Se establecen las diferentes propiedades de la tarea, como el título, la descripción, el responsable y si está completada.

```
1 package com.Villacis.Paul.ejercicioBuilder;
2
3 public class TaskManager {
4     public void createTask(TaskBuilder builder) {
5         builder.setTitle("Implementar la función de búsqueda");
6         builder.setDescription("Desarrollar la función de búsqueda en el sistema");
7         builder.setAssignee("Carlos Mera");
8         builder.setCompleted(false);
9     }
10 }
11 |
```

1.5 Probar el Patrón Builder:

En esta clase Main, se realiza la prueba del patrón Builder. Se crea un objeto TaskManager y un objeto ConcreteTaskBuilder. Luego, se llama al método createTask del manager, pasando el builder de tarea como argumento.

Este método utiliza el builder para construir una tarea siguiendo un flujo predefinido. Después, se accede al objeto Task construido y se muestran sus características para verificar que se haya creado correctamente.

```
package com.Villacis.Paul.ejercicioBuilder;

public class App
{
    public static void main(String[] args) {
        // Crear un objeto TaskManager y un objeto ConcreteTaskBuilder
        TaskManager manager = new TaskManager();
        ConcreteTaskBuilder builder = new ConcreteTaskBuilder();

        // Llamar al método createTask del manager, pasando el builder de tarea como argumento
        manager.createTask(builder);

        // Acceder al objeto Task construido y mostrar sus características para verificar que se haya creado correctamente
        Task task = builder.build();
        System.out.println(task.toString());
    }
}
```

1.6 Conclusiones

El patrón Builder separa la construcción de un objeto complejo de su representación, lo que permite que el director (TareaManager) y el constructor (ConcreteTareaBuilder) trabajen de manera independiente. Esto promueve un diseño más modular y flexible, donde cada componente se enfoca en una tarea específica. Al utilizar el patrón Builder, el proceso de construcción de objetos complejos se vuelve más flexible y mantenible. Si en el futuro se necesitan realizar cambios en la estructura de la tarea o en el flujo de construcción, es posible hacerlo fácilmente sin afectar otras partes del código. Esto facilita la adaptación del sistema a nuevos requisitos y promueve la reutilización de código.

PATRÓN PROTOTYPE

Objetivos

- Optimizar el proceso de creación de objetos, reduciendo el tiempo y los recursos al clonar prototipos existentes en lugar de crear nuevos objetos desde cero.
- Incrementar la flexibilidad y facilidad de mantenimiento del sistema, adaptando y modificando fácilmente los objetos clonados según las necesidades, sin afectar a los objetos originales.

Descripción del Problema

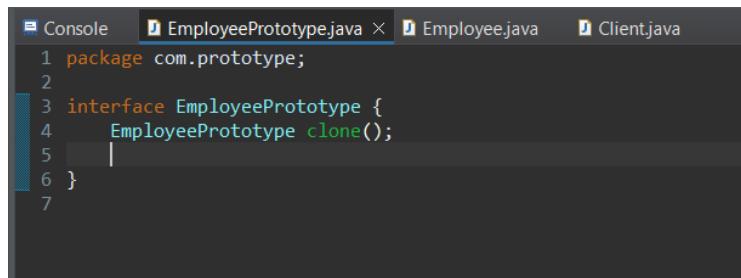
En el contexto de un sistema de gestión de empleados, crear nuevos objetos de empleado de manera eficiente puede ser un desafío, especialmente si los empleados comparten muchas características similares. El objetivo es evitar la creación redundante de objetos de empleado y mejorar la eficiencia del sistema.

Solución Propuesta

Para abordar este problema, se aplicó el patrón Prototype. Se creó una interfaz llamada `EmpleadoPrototype`, que define un método `clone()` para clonar un objeto de empleado. Luego, se implementó esta interfaz en la clase '`Empleado`', que representa un empleado en el sistema. Cada vez que se necesita un nuevo objeto de empleado, se clona un prototipo existente en lugar de crear uno desde cero.

Implementación

1. **Interfaz EmpleadoPrototype:** Esta interfaz define el método `clone()`, que debe ser implementado por las clases que deseen soportar la clonación de objetos de empleado.



```
1 package com.prototype;
2
3 interface EmployeePrototype {
4     EmployeePrototype clone();
5 }
6
7
```

2. **Clase Empleado:** Esta clase concreta implementa la interfaz `EmpleadoPrototype`. Almacena los detalles de un empleado, como su nombre y cargo, salario, y proporciona una implementación del método `clone()` que devuelve una copia superficial del objeto empleado.

```
Console × EmployeePrototype.java × Employee.java × Client.java
1 package com.prototype;
2
3 class Employee implements EmployeePrototype {
4
5     private String name;
6     private String position;
7     private double salary;
8 }
```

Implementación del constructor de la clase.

```
10
11 public Employee(String name, String position, double salary) {
12     super();
13     this.name = name;
14     this.position = position;
15     this.salary = salary;
16 }
17
```

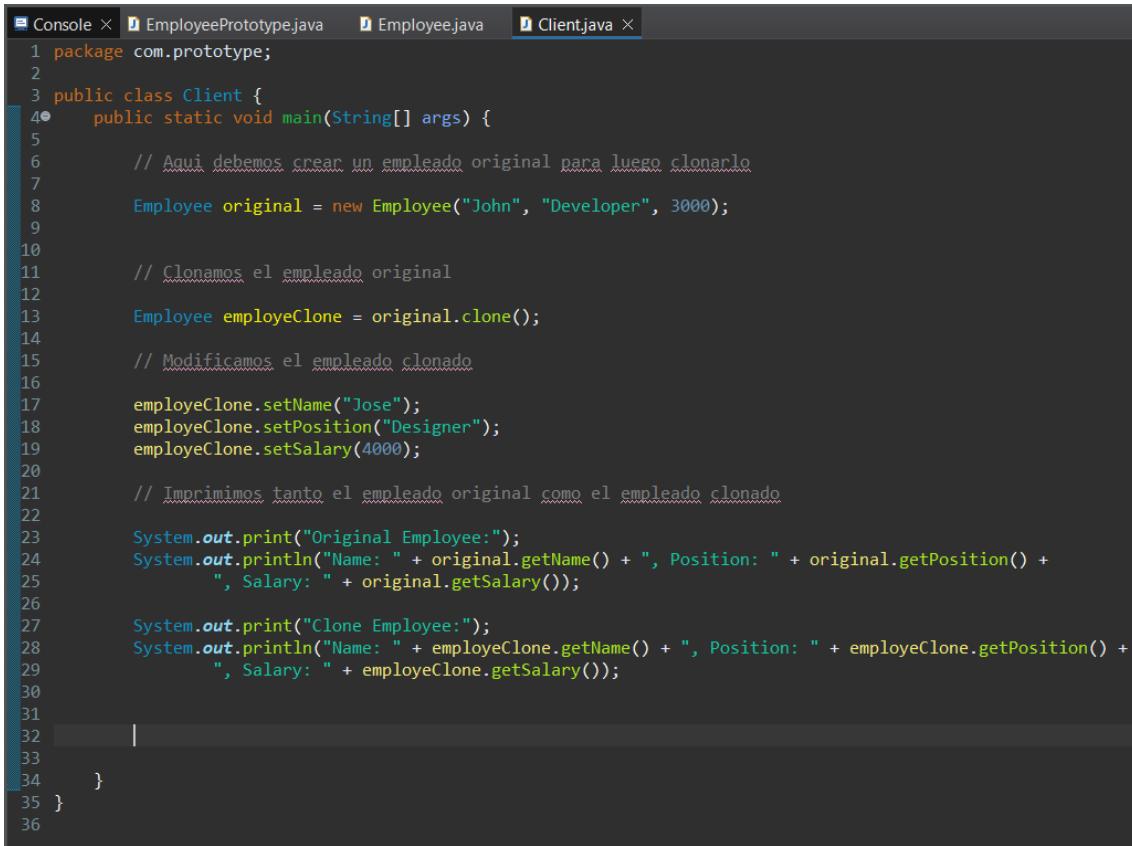
Implementación de los métodos Setters y Getters.

```
21
22 public String getName() {
23     return name;
24 }
25
26 public void setName(String name) {
27     this.name = name;
28 }
29
30 public String getPosition() {
31     return position;
32 }
33
34 public void setPosition(String position) {
35     this.position = position;
36 }
37
38 public double getSalary() {
39     return salary;
40 }
41
42 public void setSalary(double salary) {
43     this.salary = salary;
44 }
```

Implementación del método clone que nos devuelve un empleado con las características asignadas.

```
15
16 @Override
17 public Employee clone() {
18     // TODO Auto-generated method stub
19     return new Employee(this.name, this.position, this.salary);
20 }
21
```

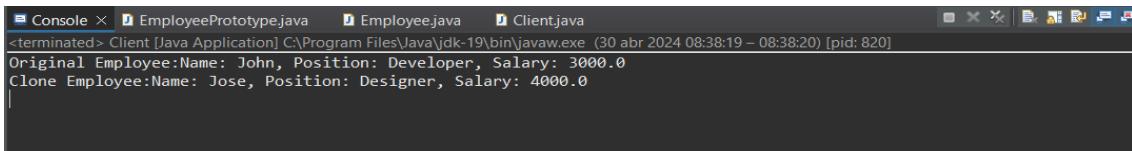
3. Cliente: En la clase Cliente, que actúa como el punto de entrada de la aplicación, se crea un objeto original de empleado y luego se clona según sea necesario. Se pueden modificar los atributos del objeto clon sin afectar al original.



```
1 package com.prototype;
2
3 public class Client {
4     public static void main(String[] args) {
5         // Aquí debemos crear un empleado original para luego clonarlo
6         Employee original = new Employee("John", "Developer", 3000);
7
8         // Clonamos el empleado original
9         Employee employeClone = original.clone();
10
11        // Modificamos el empleado clonado
12
13        employeClone.setName("Jose");
14        employeClone.setPosition("Designer");
15        employeClone.setSalary(4000);
16
17        // Imprimimos tanto el empleado original como el empleado clonado
18
19        System.out.print("Original Employee:");
20        System.out.println("Name: " + original.getName() + ", Position: " + original.getPosition() +
21                           ", Salary: " + original.getSalary());
22
23        System.out.print("Clone Employee:");
24        System.out.println("Name: " + employeClone.getName() + ", Position: " + employeClone.getPosition() +
25                           ", Salary: " + employeClone.getSalary());
26
27    }
28
29 }
30
31
32
33
34 }
35
36 }
```

Resultado por consola.

Aquí observamos que tenemos un empleado original con sus características y el empleado clonado igual con sus características propias.



```
<terminated> Client [Java Application] C:\Program Files\Java\jdk-19\bin\javaw.exe (30 abr 2024 08:38:19 – 08:38:20) [pid: 820]
Original Employee:Name: John, Position: Developer, Salary: 3000.0
Clone Employee:Name: Jose, Position: Designer, Salary: 4000.0
```

Conclusiones

- La aplicación del patrón Prototype en el sistema de gestión de empleados demostró ser altamente eficiente y flexible. Al permitir la clonación de objetos existentes, se evita la redundancia en la creación de objetos similares, lo que mejora la eficiencia del sistema.
- La su eficiencia y flexibilidad en la implementación del patrón Prototype también promueve una mayor coherencia en la estructura y el comportamiento de los objetos clonados. El uso de prototipos existentes como base para la creación de nuevos objetos garantiza que todas las instancias compartan características comunes y estén sujetas a un flujo de creación coherente.

PATRÓN FACTORY

Objetivo: Se desea implementar en el sistema de gestión de pedidos de una empresa de repartos a domicilio el patrón Factory para la creación de los diferentes tipos de órdenes de entrega que ofrece la empresa en su catálogo.

Descripción: La empresa de repartos a domicilio tiene diferentes tipos de órdenes de entrega, como las de comida, las de paquetes grandes, las de documentos importantes, etc. Cada tipo tiene su propio proceso específico de gestión y logística. Los cuales son complejos y muy extensos por lo cual se deben tratar de manera prosaica tal y como se hace en el patrón Factory.

Desarrollo:

1. Primero creamos la interfaz orden de entrega con métodos que se implementaran en las clases según las necesidades específicas de la orden que se quiera crear.

```
5 package interfaces;
6
7 /**
8 *
9 * @author Alejo
10 */
11 public interface OrdenDeEntregaInterfaz {
12     void procesarOrden();
13     void asignarRepartidor();
14     void enviarNotificacion();
15 }
```

2. Se crea la superclase Orden que implementara la interfaz OrdenDeEntregaInterfaz, pero estos serán abstractos pues la implementación real será por parte de sus clases hijas. Además esta clase tendrá los atributos que deben tener todos los pedidos

```
abstract class Orden {
    private int id;
    private String direccionEntrega;
    private boolean entregada;
    private Date fechaCreacion;
    private Date fechaEntrega;
    private String observaciones;
    public Orden() {
    }
    public Orden(int id, String direccionEntrega, String observaciones) {
        this.id = id;
        this.direccionEntrega = direccionEntrega;
        this.entregada = false;
        this.fechaCreacion = new Date();
        this.observaciones = observaciones;
    }
    public abstract void procesarOrden();
    public abstract void asignarRepartidor();
    public abstract void enviarNotificacion();

    public int getId() {
```

```

public int getId() {
    return id;
}
public void setId(int id) {
    this.id = id;
}
public String getDireccionEntrega() {
    return direccionEntrega;
}
public void setDireccionEntrega(String direccionEntrega) {
    this.direccionEntrega = direccionEntrega;
}
public boolean isEntregada() {
    return entregada;
}
public void setEntregada(boolean entregada) {
    this.entregada = entregada;
}
public Date getFechaCreacion() {
    return fechaCreacion;
}
public void setFechaCreacion(Date fechaCreacion) {
    this.fechaCreacion = fechaCreacion;
}
public Date getFechaEntrega() {
    return fechaEntrega;
}
public void setFechaEntrega(Date fechaEntrega) {
    this.fechaEntrega = fechaEntrega;
}

```

1. Creamos diferentes subclases de la superclase Orden que implementaran los métodos de la interfaz de acuerdo con sus necesidades, además le añadimos un constructor vacío que será el que la factory usará.
Los constructores tendrán una visibilidad de paquete pues fuera del paquete la forma en la que se crea objetos será a través del factory.

La clase OrdenDeComida

```

public class OrdenDeComida extends Orden {
    OrdenDeComida(int id, String direccionEntrega, String observaciones) {
        super(id, direccionEntrega, observaciones);
    }

    OrdenDeComida() {

    }
    public void procesarOrden() {
        System.out.println("Procesando orden de comida...");
        // Lógica específica para órdenes de comida
    }

    public void asignarRepartidor() {
        System.out.println("Asignando repartidor para la orden de comida...");
        // Lógica para asignar un repartidor a la orden de comida
    }

    public void enviarNotificacion() {
        System.out.println("Enviando notificación al cliente de la orden de comida...");
        // Lógica para enviar una notificación al cliente sobre la orden de comida
    }
}

```

La clase OrdenDePaquete

```
public class OrdenDePaquete extends Orden {  
    OrdenDePaquete(int id, String direccionEntrega, String observaciones) {  
        super(id, direccionEntrega, observaciones);  
    }  
  
    OrdenDePaquete() {  
    }  
  
    public void procesarOrden() {  
        System.out.println("Procesando orden de paquete grande...");  
        // Lógica específica para órdenes de paquetes grandes  
    }  
  
    public void asignarRepartidor() {  
        System.out.println("Asignando repartidor para la orden de paquete grande...");  
        // Lógica para asignar un repartidor a la orden de paquete grande  
    }  
  
    public void enviarNotificacion() {  
        System.out.println("Enviendo notificación al cliente de la orden de paquete grande...");  
        // Lógica para enviar una notificación al cliente sobre la orden de paquete grande  
    }  
}
```

A continuación

La clase OrdenDeDocumentos

```
public class OrdenDeDocumento extends Orden {  
    OrdenDeDocumento(int id, String direccionEntrega, String observaciones) {  
        super(id, direccionEntrega, observaciones);  
    }  
    OrdenDeDocumento() {  
    }  
    public void procesarOrden() {  
        System.out.println("Procesando orden de documento importante...");  
        // Lógica específica para órdenes de documentos importantes  
    }  
    public void asignarRepartidor() {  
        System.out.println("Asignando repartidor para la orden de documento importante...");  
        // Lógica para asignar un repartidor a la orden de documento importante  
    }  
    public void enviarNotificacion() {  
        System.out.println("Enviendo notificación al cliente de la orden de documento importante...");  
        // Lógica para enviar una notificación al cliente sobre la orden de documento importante  
    }  
}
```

Activar Windows

- Finalmente creamos el Factory este tendrá un método crearOrden que retornará un objeto del tipo Orden y recibirá como parámetro un String que representa el tipo de orden que vamos a crear.

```
public class OrdenFactory {
    public static Orden crearOrden(String tipo) {
        switch (tipo.toLowerCase()) {
            case "comida":
                return new OrdenDeComida();
            case "paquete":
                return new OrdenDePaquete();
            case "documento":
                return new OrdenDeDocumento();
            default:
                throw new IllegalArgumentException("Tipo de orden no válido: " + tipo);
        }
    }
}
```

Un ejemplo de la aplicación de este patrón puede ser

```
/*
public class FactoryEmpresaDelivery {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        Orden ordenDeComida = OrdenFactory.crearOrden("comida");
        ordenDeComida.procesarOrden();
        ordenDeComida.asignarRepartidor();
        ordenDeComida.enviarNotificacion();

        Orden ordenDePaquete = OrdenFactory.crearOrden("paquete");
        ordenDePaquete.procesarOrden();
        ordenDePaquete.asignarRepartidor();
        ordenDePaquete.enviarNotificacion();

        Orden ordenDeDocumento = OrdenFactory.crearOrden("documento");
        ordenDeDocumento.procesarOrden();
        ordenDeDocumento.asignarRepartidor();
        ordenDeDocumento.enviarNotificacion();
    }
}
```

Conclusiones:

- La aplicación del patrón Factory permite crear diferentes tipos de órdenes de entrega (como órdenes de comida, paquetes grandes o documentos importantes) sin modificar el código cliente. Esto facilita la gestión de diversos tipos de pedidos que pueda tener la empresa.
- En caso de que un sistema de software tenga varios objetos de la misma naturaleza, es conveniente crear una interfaz general para la creación de los mismo, pues con eso no se expone la lógica de creación.
- La lógica de creación de órdenes se centraliza en la fábrica, lo que facilita la gestión y mantenimiento del código. Cualquier cambio en la forma en que se crean las órdenes (por ejemplo, la adición de nuevos tipos de órdenes) solo necesita realizarse en la fábrica, lo que simplifica el proceso de desarrollo.

PATRÓN SINGLETON

Objetivo:

Implementar el patrón **Singleton** para asegurar que una clase contenga una única instancia y proporcione un punto de acceso global de manera efectiva.

Descripción:

Imagina que se está trabajando en una empresa de software en donde los empleados cumplen un rol diferente y se requiere implementar el patrón Singleton para obtener la información de los empleados y suministrar un punto de acceso global a dicha instancia.

Desarrollo:

1. Definir la Clase Employee:

Se crea una clase Employee con propiedades privadas como name, role, assignment, etc.

Implementa el constructor con sus respectivos parámetros y se emplea para crear instancias de la clase Employee y se asigna valores a los atributos.

```
1 package com.grupo1.PatronSingleton;
2
3 public class Employee {
4
5     private String name;
6     private String role;
7     private String assignment;
8
9     public Employee(String name, String role, String assignment) {
10         this.name = name;
11         this.role = role;
12         this.assignment = assignment;
13     }
14 }
```

2. Crear la Clase Printer:

Define una clase Printer con una instancia estática y privado de tipo Printer, es decir que se emplea para almacenar la única instancia de la clase, también un atributo de nrOfPages para el conteo de las páginas que se van a imprimir y se crea un constructor privado.

También se crea un método getInstance para comparar si existe o no existe una única instancia de la clase. Por último, se define un atributo público print que me permite imprimir el texto y el número de páginas.

```

1 package com.grupo1.PatronSingleton;
2
3 public class Printer {
4
5     private static Printer instance;
6     private int nrOfPages;
7
8     private Printer() {
9
10    }
11
12     public static Printer getInstance() {
13         return instance == null ? instance = new Printer() : instance;
14     }
15
16     public void print(String text) {
17         System.out.println(text + "\n" + "Paginas impresas hoy " + ++nrOfPages + "\n" + "*****");
18     }
19
20 }
21

```

3. Implementar el Singleton:

En la clase Employee creo un método llamado printCurrentAssignment que me permite imprimir la información del empleado.

```

15     public void printCurrentAssignment() {
16         Printer printer = Printer.getInstance();
17         printer.print("Employee:" + name + "\n" + "Role:" + role + "\n" + "Assignment:" + assignment + "\n");
18     }
19 }
20
21

```

Probar el patrón Singleton:

Crear una instancia de la clase Employee con los atributos asignados en dicha clase y luego imprimir la información asignada para cada empleado.

```

*App.java ×
1 package com.grupo1.PatronSingleton;
2
3 public class App {
4
5     public static void main(String[] args) {
6         Employee fernando = new Employee("Fernando", "coordinador", "Organiza las ideas");
7         Employee maria = new Employee("Maria", "evaluadora", "evalua las ventajas e inconvenientes");
8         Employee jose = new Employee("Jose", "especialista", "Aporta conocimiento específico");
9         Employee juan = new Employee("Juan", "desarrollador", "Desarrolla el producto");
10
11         fernando.printCurrentAssignment();
12         maria.printCurrentAssignment();
13         jose.printCurrentAssignment();
14         juan.printCurrentAssignment();
15     }
16 }
17

```

Conclusiones

- Es un patrón que abarca la compresión general del concepto de instancia y de la programación orientada a objetos, en la implementación se puede utilizar una clase como si fuera una instancia misma.
- La propia clase es encargada de crear la única instancia y autorizar el acceso global a dicha instancia a través de un método de clase.
- Se declara el constructor de la clase como privado para que no se pueda instanciar directamente.

UNIVERSIDAD TECNICA DE AMBATO

INTEGRANTES: Joshua Herrera, Kenneth Paredes, Josué López, Gerson Zunta

FECHA: 30-04-2024

ASIGNATURA: Patrones de diseño de Software

DIRECCION ONEDRIVE A LA CARPETA CON LOS PROYECTOS

[PatronesSoftWare](#)

PATRON SINGLETON

El patrón singleton nos permite realizar una única instancia de un objeto con el objetivo de evitar el uso de múltiples instancias de objetos innecesarias que vuelven a nuestro código repetitivo y poco legible. En este caso se procederá a explicar la implementación del mismo patrón detallando cada una de las clases que interfieren en el mismo.

PROBLEMA

En aplicaciones que necesitan llevar un registro o conteo de acciones de cada usuario , es indispensable establecer un sistema de registro de datos de inicio de sesión, acciones, actividades, modificaciones o cierres de sesión. Por lo que es necesario crear un archivo de logging que me permita llevar todas esas acciones en un mismo archivo centralizado.

SOLUCIÓN

Si se implementa a nuestro sistema el patrón singleton, se garantiza que todos los componentes usen la misma instancia del archivo de log, lo que facilita la gestión de la documentación y nos asegura que el proceso de loggin sea coherente y centralizado.

Crear una clase que para crear un registro log utilizando el patrón singleton

1. Para poder resolver el problema es necesario crear la clase Log, la cual debe tener un constructor privado para evitar que se instancien múltiples veces y un método para instanciarlo que me devuelva la misma instancia en caso de que previamente haya sido llamado.

```

1 package main.SingletonImplementation;
2
3 import java.io.FileWriter;
4
5 public class Logger {
6     private static Logger instance;
7     private static PrintWriter printWriter;
8     private static DateTimeFormatter dtf = DateTimeFormatter.ofPattern("yyyy/MM/dd HH:mm:ss");
9
10    private Logger() {
11        try {
12            FileWriter fileWriter = new FileWriter("C:/logs/log.txt", true);
13            printWriter = new PrintWriter(fileWriter, true);
14        } catch (IOException e) {
15            System.out.println("error " + e);
16        }
17    }
18
19    public static Logger getInstance() {
20        if (instance == null) {
21            instance = new Logger();
22            System.out.println("Instancia creada");
23        } else {
24            System.out.println("Ya existe una instancia de Log");
25        }
26        return instance;
27    }
28
29    public void log(String message) {
30        LocalDateTime now = LocalDateTime.now();
31        printWriter.println("[" + dtf.format(now) + "] " + message);
32    }
33
34    public void close() {
35        printWriter.close();
36    }
37}
38
39
40
41
42
43

```

Ilustración 1. Clase Log

2. Se realiza una clase Main para comprobar que se cree únicamente una sola instancia. Además, con el método log se crearán algunos registros para verificar que funcione correctamente.

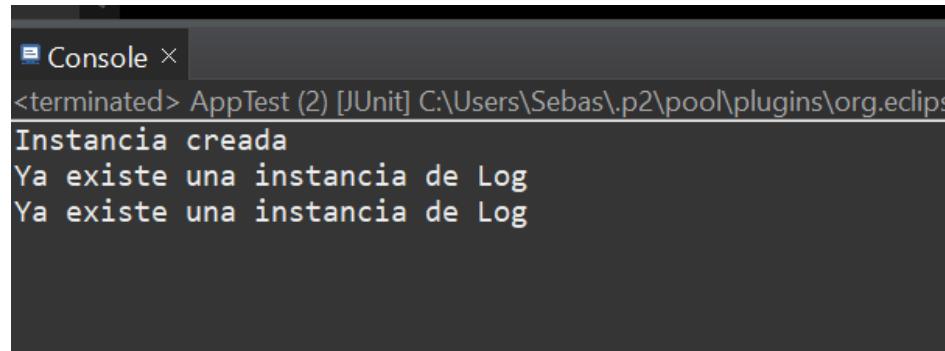
```

1 package main.SingletonImplementation;
2
3 /**
4  * Hello world!
5  */
6
7 public class App
8 {
9     public static void main( String[] args )
10    {
11        Logger logger = Logger.getInstance();
12
13        logger.log("Aplicacion iniciada");
14        logger.log("Inicializando la aplicacion");
15        logger.log("Apagando la aplicacion");
16        logger.close();
17
18        Logger.getInstance();
19        |
20    }
21 }
22

```

Ilustración 2. Claase Main

En la consola debería aparecer un mensaje de la primera instancia creada, y como en el código creamos una segunda instancia podremos visualizar que no se puede crear otra instancia.



```
<terminated> AppTest (2) [JUnit] C:\Users\Sebas\.p2\pool\plugins\org.eclipse.jdt.core_3.11.0.v20150611-1410\jdt.core\src\org\eclipse\jdt\core\internal\JavaModelManager.java:113: error: cannot find symbol
symbol: class IPath
location: class org.eclipse.jdt.core.IJavaElement
1 error
```

Ilustración 3. Consola que muestra la ejecución del programa

TESTING

Para revisar que nuestro código siga el patron singleton es necesario realizar testing , el primero verificará dos instancias creadas y verificará si la segunda es la misma instancia que la primera.

```
public void testSingletonInstance() {  
  
    Logger logger1 = Logger.getInstance();  
    Logger logger2 = Logger.getInstance();  
  
    assertEquals("Las instancias deben ser la misma", logger1, logger2);  
}
```

Ilustración 4. Método para verificar que las instancias sean las mismas

También un test para verificar que la clase ha sido instanciada nos ayudaremos del método assertNotNull

```
public void testSingletonInitialization() {  
  
    Logger logger = Logger.getInstance();  
  
    assertNotNull("La instancia ahora debe estar inicializada", logger);  
}
```

Ilustración 5. Método verificar que se cree una instancia con el método creado en Log

Finalmente se realiza la ejecución del test para verificar que no haya problemas

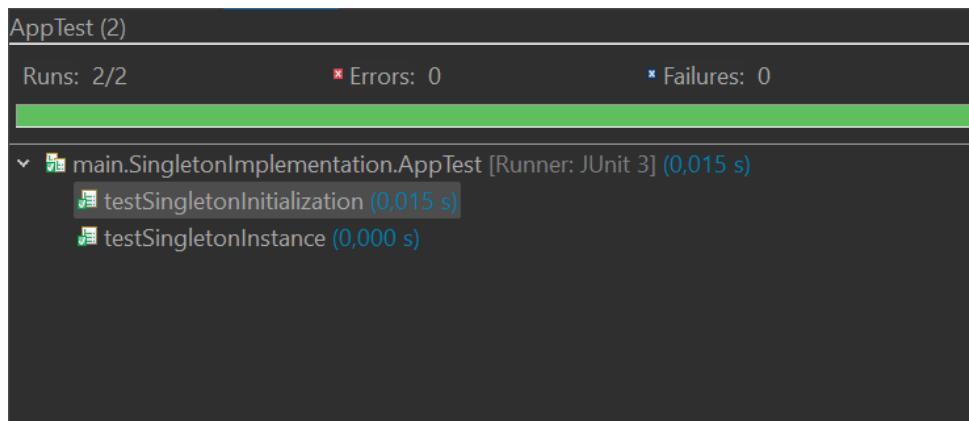


Ilustración 6. Test realizado correctamente

PATRON PROTOTYPE

El patrón Prototype nos permite crear nuevos objetos clonando un prototipo existente, en lugar de crearlos desde cero. Esto es útil cuando la creación de un objeto es costosa o compleja, ya que clonar un objeto existente es más eficiente. A continuación, se describe la implementación del patrón Prototype utilizando una clase Usuario como prototipo.

PROBLEMA

En aplicaciones donde se requiere crear múltiples instancias de un objeto similar, como usuarios con datos predefinidos, puede resultar ineficiente crear cada objeto desde cero. Se necesita una forma de crear nuevas instancias basadas en un objeto prototipo.

SOLUCIÓN

La solución propuesta utiliza el patrón Prototype para clonar un objeto Usuario prototipo y crear nuevas instancias a partir de él. La clase Usuario implementa la interfaz Cloneable y proporciona un método `clone()` que utiliza `super.clone()` para realizar la clonación.

IMPLEMENTACIÓN

La clase **Usuario** contiene los datos básicos de un usuario, como nombre, apellido, edad y registro civil. Implementa la interfaz Cloneable y proporciona un método **clone()** que utiliza **super.clone()** para clonar el objeto.

```
1 package com.gerson.zunta.PatronPrototype;
2
3 import org.jetbrains.annotations.MustBeInvokedByOverride;
4
5 public class Usuario implements Cloneable {
6     String Nombre;
7     String Apellido;
8     int edad;
9     String Registro_Civil;
10
11     public Usuario clone() {
12         try {
13             return (Usuario) super.clone();
14         } catch (CloneNotSupportedException e) {
15             return null;
16         }
17     }
18
19     public String getNombre() {
20         return Nombre;
21     }
22
23     public void setNombre(String nombre) {
24         Nombre = nombre;
25     }
26
27     public String getApellido() {
28         return Apellido;
29     }
30
31     public void setApellido(String apellido) {
32         Apellido = apellido;
33     }
34
35     public int getEdad() {
36         return edad;
37
38     public void setEdad(int edad) {
39         this.edad = edad;
40     }
41
42
43     public String getRegistro_Civil() {
44         return Registro_Civil;
45     }
46
47     public void setRegistro_Civil(String registro_Civil) {
48         Registro_Civil = registro_Civil;
49     }
50
51     @Override
52     public boolean equals(Object o) {
53         if (this == o) return true;
54         if (o == null || getClass() != o.getClass()) return false;
55         Usuario usuario = (Usuario) o;
56         if (edad != usuario.edad) return false;
57         if (Nombre != null ? !Nombre.equals(usuario.Nombre) : usuario.Nombre == null) return false;
58         if (Apellido != null ? !Apellido.equals(usuario.Apellido) : usuario.Apellido == null) return false;
59         return Registro_Civil != null ? Registro_Civil.equals(usuario.Registro_Civil) : usuario.Registro_Civil == null;
60     }
61
62     @Override
63     public int hashCode() {
64         int result = Nombre != null ? Nombre.hashCode() : 0;
65         result = 31 * result + (Apellido != null ? Apellido.hashCode() : 0);
66         result = 31 * result + edad;
67         result = 31 * result + (Registro_Civil != null ? Registro_Civil.hashCode() : 0);
68         return result;
69     }
70 }
```

```
.....return Registro_Civil != null ? Registro_Civil.equals(usuario.Registro_Civil) : usuario.Registro_Civil == null; }  
.....}  
• .. @Override  
.. public int hashCode() {  
.....int result = Nombre != null ? Nombre.hashCode() : 0;  
.....result = 31 * result + (Apellido != null ? Apellido.hashCode() : 0);  
.....result = 31 * result + (Registro_Civil != null ? Registro_Civil.hashCode() : 0);  
.....return result; }  
.....}  
• .. @Override  
.. public String toString() {  
.....return "Usuario{" +  
....."Nombre='" + Nombre + '\'' +;  
.....", Apellido='" + Apellido + '\'' +;  
.....", edad=" + edad +;  
.....", Registro_Civil='" + Registro_Civil + '\'' +;  
....."}"; }  
.....}  
}
```

El método **main** de la clase **App** crea un prototipo de usuario con datos predefinidos y luego clona este prototipo para **crear dos nuevos usuarios**. Los datos de los nuevos usuarios se modifican para demostrar que son instancias independientes.

```

1 package com.gerson.zunta.PatronPrototype;
2
3 import org.slf4j.Logger;
4
5 public class App {
6     private static final Logger Log = LoggerFactory.getLogger(App.class);
7
8     public static void main(final String[] args) {
9
10        Usuario prototipoUsuario = new Usuario();
11        prototipoUsuario.setNombre("Nombre");
12        prototipoUsuario.setApellido("Apellido");
13        prototipoUsuario.setEdad(30);
14        prototipoUsuario.setRegistro_Civil("Registro Civil");
15
16        Usuario nuevoUsuario1 = prototipoUsuario.clone();
17        Log.info("Nuevo usuario 1: " + nuevoUsuario1);
18
19        nuevoUsuario1.setNombre("Marco");
20        nuevoUsuario1.setEdad(25);
21        Log.info("Nuevo usuario 1 modificado: " + nuevoUsuario1);
22
23        Usuario nuevoUsuario2 = prototipoUsuario.clone();
24        Log.info("Nuevo usuario 2: " + nuevoUsuario2);
25    }
26
27 }
28
29 }
```

Consola

```

terminated> App (4) [Java Application] C:\Program Files\Java\jdk-21\bin\javaw.exe [30 abr 2024, 13:51:11 - 13:51:13] [pid: 21176]
2024-04-30 13:51:13,377 [main] INFO c.g.z.P.App - Nuevo usuario 1: Usuario{Nombre='Nombre', Apellido='Apellido', edad=30, Registro_Civil='Registro Civil'}
2024-04-30 13:51:13,379 [main] INFO c.g.z.P.App - Nuevo usuario 1 modificado: Usuario{Nombre='Marco', Apellido='Apellido', edad=25, Registro_Civil='Registro Civil'}
2024-04-30 13:51:13,379 [main] INFO c.g.z.P.App - Nuevo usuario 2: Usuario{Nombre='Nombre', Apellido='Apellido', edad=30, Registro_Civil='Registro Civil'}
```

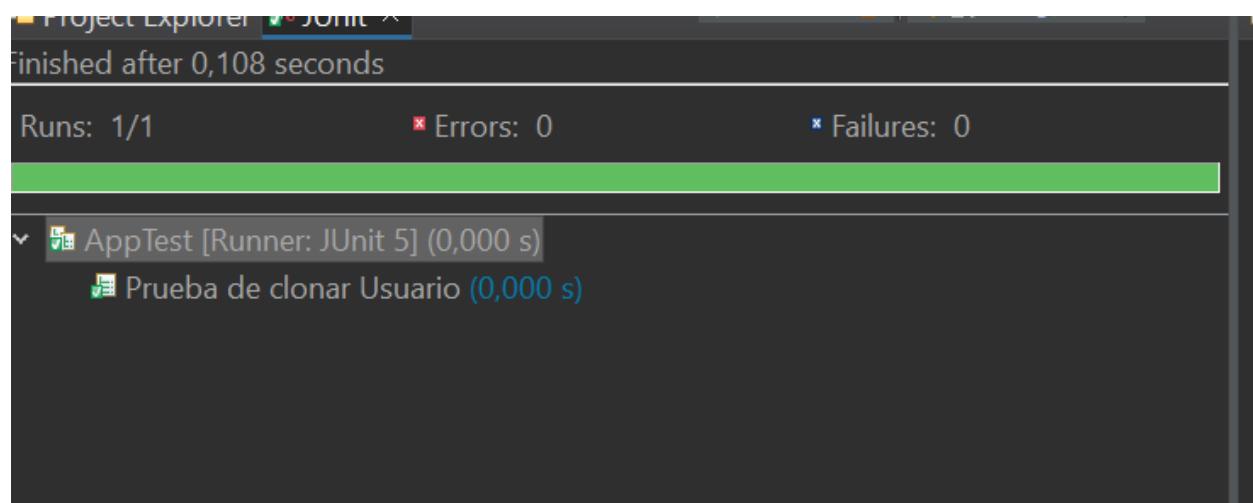
La clase **AppTest** contiene pruebas unitarias para verificar que la clonación de usuarios funcione correctamente, asegurando que los nuevos usuarios sean instancias independientes y que los datos clonados coincidan con los del prototipo.

```

1 package com.gerson.zunta.PatronPrototype;
2
3 import org.junit.jupiter.api.BeforeEach;
4
5 public class AppTest {
6     private Usuario prototipoUsuario;
7
8     @BeforeEach
9     public void setUp() {
10        prototipoUsuario = new Usuario();
11        prototipoUsuario.setNombre("Nombre");
12        prototipoUsuario.setApellido("Apellido");
13        prototipoUsuario.setEdad(30);
14        prototipoUsuario.setRegistro_Civil("Registro Civil");
15    }
16
17    @Test
18    @DisplayName("Prueba de clonar Usuario")
19    public void testClone() {
20        Usuario nuevoUsuario = prototipoUsuario.clone();
21
22        assertNotSame(prototipoUsuario, nuevoUsuario);
23
24        assertEquals(prototipoUsuario.getNombre(), nuevoUsuario.getNombre());
25        assertEquals(prototipoUsuario.getApellido(), nuevoUsuario.getApellido());
26        assertEquals(prototipoUsuario.getEdad(), nuevoUsuario.getEdad());
27        assertEquals(prototipoUsuario.getRegistro_Civil(), nuevoUsuario.getRegistro_Civil());
28    }
29
30
31 }
32
33
34
35
36
37
38
39
40

```

Ejecución Test



PATRON BUILDER

Objetivo:

Implementar el patrón Builder para facilitar la configuración y ensamblaje personalizado de computadoras, mejorando la eficiencia y precisión del proceso.

Descripción:

En una tienda de computadoras, es habitual manejar una amplia variedad de componentes con diferentes especificaciones y características para satisfacer las diversas necesidades de los clientes. Para optimizar el proceso de ensamblaje de computadoras personalizadas y permitir la personalización detallada de estas, se requiere implementar el patrón Builder.

El sistema debe permitir a los técnicos ensamblar nuevas configuraciones de computadoras, incluyendo información como el tipo de CPU, GPU, cantidad de RAM, y capacidad de almacenamiento. Cada componente puede ser seleccionado específicamente para ajustarse a las necesidades de rendimiento, gráficos, o almacenamiento del usuario, facilitando la creación de sistemas altamente personalizados y eficientes.

Desarrollo:

1. Interfaz ComputerBuilder

Define los métodos necesarios para construir paso a paso las partes de un objeto Computer.

Métodos Definidos:

- **setCPU(String cpu):** Establece el procesador de la computadora.
- **setGPU(String gpu):** Establece la tarjeta gráfica de la computadora.
- **setRAM(int ram):** Establece la cantidad de RAM en gigabytes.
- **setStorage (int storage):** Establece la capacidad de almacenamiento en gigabytes.
- **build():** Devuelve el producto final, es decir, el objeto Computer completamente ensamblado.



```
1  public interface ComputerBuilder {  
2      ComputerBuilder setCPU(String cpu);  
3      ComputerBuilder setGPU(String gpu);  
4      ComputerBuilder setRAM(int ram);  
5      ComputerBuilder setStorage(int storage);  
6      Computer build();  
7  }  
8
```

2. Clase Concreta GamingComputerBuilder

Implementa la interfaz ComputerBuilder y se encarga de construir una computadora específica, en este caso, una computadora de juegos.

Implementación:

En esta clase se implementa cada método de ComputerBuilder para elegir componentes de alta especificación que son típicos en una computadora de juegos, como un CPU de alto rendimiento, una GPU potente, amplia memoria RAM y grandes opciones de almacenamiento.

```
● ● ●

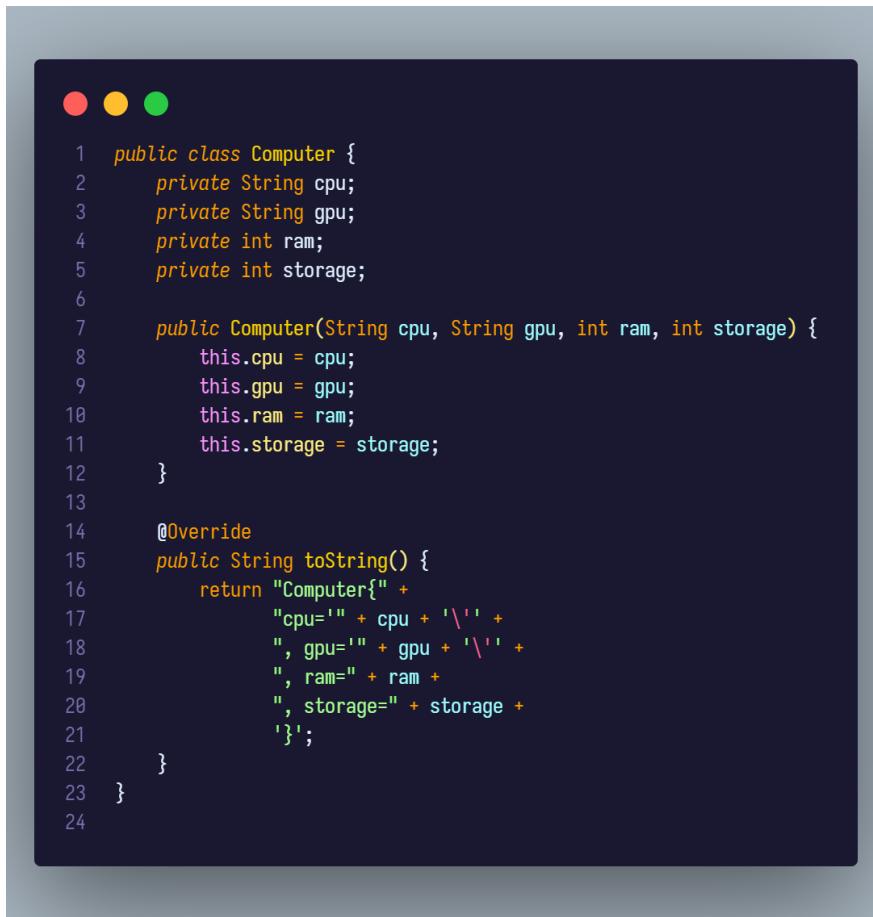
1  public class GamingComputerBuilder implements ComputerBuilder {
2      private String cpu;
3      private String gpu;
4      private int ram;
5      private int storage;
6
7      @Override
8      public ComputerBuilder setCPU(String cpu) {
9          this.cpu = cpu;
10         return this;
11     }
12
13     @Override
14     public ComputerBuilder setGPU(String gpu) {
15         this.gpu = gpu;
16         return this;
17     }
18
19     @Override
20     public ComputerBuilder setRAM(int ram) {
21         this.ram = ram;
22         return this;
23     }
24
25     @Override
26     public ComputerBuilder setStorage(int storage) {
27         this.storage = storage;
28         return this;
29     }
30
31     @Override
32     public Computer build() {
33         return new Computer(cpu, gpu, ram, storage);
34     }
35 }
36
```

3. Clase Producto Computer

Representa el producto final que es la computadora ensamblada.

Propiedades:

- **String cpu:** Tipo de procesador.
- **String gpu:** Tipo de tarjeta gráfica.
- **int ram:** Cantidad de RAM en gigabytes.
- **int storage:** Capacidad de almacenamiento en gigabytes.



The screenshot shows a code editor window with a dark theme. At the top, there are three colored circular icons (red, yellow, green). Below them, the Java code for the Computer class is displayed. The code defines a constructor that takes four parameters (cpu, gpu, ram, storage) and initializes the corresponding private fields. It also overrides the toString() method to return a string representation of the computer's specifications.

```
1 public class Computer {
2     private String cpu;
3     private String gpu;
4     private int ram;
5     private int storage;
6
7     public Computer(String cpu, String gpu, int ram, int storage) {
8         this.cpu = cpu;
9         this.gpu = gpu;
10        this.ram = ram;
11        this.storage = storage;
12    }
13
14    @Override
15    public String toString() {
16        return "Computer{" +
17            "cpu=\"" + cpu + '\"' +
18            ", gpu=\"" + gpu + '\"' +
19            ", ram=" + ram +
20            ", storage=" + storage +
21            '}';
22    }
23 }
24
```

4. Clase ComputerDirector

Utiliza una instancia de ComputerBuilder para crear computadoras predefinidas, como modelos de juegos o de oficina, simplificando el proceso de creación.

Métodos:

- **buildGamingPC():** Construye una computadora con especificaciones adecuadas para juegos.
- **buildOfficePC():** Construye una computadora con especificaciones adecuadas para tareas de oficina, optimizando costos y eficiencia energética.

```
1  public class ComputerDirector {
2      private ComputerBuilder builder;
3
4      public ComputerDirector(ComputerBuilder builder) {
5          this.builder = builder;
6      }
7
8      public Computer buildGamingPC() {
9          return builder.setCPU("AMD Ryzen 9")
10             .setGPU("NVIDIA RTX 3080")
11             .setRAM(32)
12             .setStorage(2000)
13             .build();
14      }
15
16     public Computer buildOfficePC() {
17         return builder.setCPU("Intel Core i5")
18             .setGPU("Integrated Graphics")
19             .setRAM(16)
20             .setStorage(500)
21             .build();
22     }
23 }
24 }
```

5. Implementación en el Main

```
1  public class Main {
2      public static void main(String[] args) {
3          ComputerBuilder builder = new GamingComputerBuilder();
4          ComputerDirector director = new ComputerDirector(builder);
5
6          // Construir una computadora de juegos
7          Computer gamingPC = director.buildGamingPC();
8          System.out.println(gamingPC);
9
10         // Construir una computadora personalizada
11         Computer customPC = builder.setCPU("Intel Core i9")
12             .setGPU("NVIDIA RTX 3080")
13             .setRAM(32)
14             .setStorage(2048)
15             .build();
16         System.out.println(customPC);
17     }
18 }
19 }
```

Ejecución en Consola:

```
Roaming\Code\User\workspaceStorage\8affcb68
Computer{
    cpu='AMD Ryzen 9'
    , gpu='NVIDIA RTX 3080'
    , ram=32
    , storage=2000
}

Computer{
    cpu='Intel Core i9'
    , gpu='NVIDIA RTX 3080'
    , ram=32
    , storage=2048
}

PS C:\Users\raulp\OneDrive\Desktop\QUINTO S
```

PATRON FACTORY

El patrón de diseño Factory, también conocido como fábrica, es un patrón creacional que proporciona una interfaz para crear objetos de un tipo determinado sin especificar su clase concreta. En lugar de instanciar objetos directamente utilizando su constructor, el patrón Factory utiliza un método común en una clase separada, conocida como fábrica, para crear objetos según el tipo solicitado. Esto promueve la encapsulación, ya que oculta la lógica de creación de objetos detrás de una interfaz única, lo que permite una fácil extensión y modificación del código. Además, el patrón Factory facilita la adopción del principio de "programar para una interfaz, no para una implementación", lo que conduce a un código más flexible, mantenable y escalable.

PROBLEMA

Se enfrenta al desafío de desarrollar un sistema de gestión para una fábrica de vehículos. Se requiere una funcionalidad que permita fabricar varios tipos de vehículos, como coches y motocicletas, de manera eficiente y flexible. Sin embargo, se busca evitar la dependencia directa entre la lógica de creación de vehículos y las clases concretas de coches y motocicletas, con el fin de facilitar la extensión del sistema en el futuro.

SOLUCIÓN

Para abordar esta problemática, se implementará el patrón Factory. Se creará una clase denominada VehicleFactory, la cual actuará como una fábrica para producir objetos Vehicle basados en el tipo especificado. Esta fábrica encapsulará la lógica de creación de vehículos y ofrecerá un método estático llamado createVehicle(), el cual tomará como parámetro el tipo de vehículo que se desea fabricar ("car" para coches y "motorcycle" para motocicletas).

Al emplear la fábrica VehicleFactory, se podrá crear vehículos sin la necesidad de preocuparse por los detalles de implementación de cada tipo de vehículo. Esto permitirá mantener un código modular y fácilmente extensible, ya que se podrán añadir nuevos tipos de vehículos en el futuro simplemente creando nuevas clases concretas que implementen la interfaz Vehicle, sin necesidad de modificar la lógica de la fábrica.

1. Implementar la interfaz vehicle

```
1 package factory;
2
3 public interface Vehicle {
4     void manufacture();
5 }
6
```

2. Crear la clase carro

```
1 package factory;
2
3 public class Car implements Vehicle{
4     @Override
5     public void manufacture() {
6         System.out.println("Car manufactured");
7     }
8 }
9
```

3. Crear la clase motocicleta

```
1 package factory;
2
3 public class Motorcycle implements Vehicle{
4     @Override
5     public void manufacture() {
6         System.out.println("Motorcycle manufactured");
7     }
8 }
```

4. Crear la clase factory

```
1 package factory;
2
3 public class VehicleFactory {
4
5     public static Vehicle createVehicle(string type) {
6         if (type.equals("Car")) {
7             return new Car();
8         } else if (type.equals("Motorcycle")) {
9             return new Motorcycle();
10        } else {
11            System.out.println("Invalid vehicle type");
12            return null;
13        }
14    }
15}
16
```

5. Testear si son válidos

```
src > test > java > factory > AppTest.java > AppTest > testCreateInvalidVehicle()
1 package factory;
2
3 import static org.junit.Assert.assertTrue;
4 import static org.junit.Assert.fail;
5
6 import org.junit.Test;
7
8 /**
9  * Unit test for simple App.
10 */
11 public class AppTest
12 {
13     @Test
14     public void testCreateCar() {
15         Vehicle car = VehicleFactory.createVehicle(type:"Car");
16         assertTrue(car instanceof Car);
17     }
18
19     @Test
20     public void testCreateMotorcycle() {
21         Vehicle motorcycle = VehicleFactory.createVehicle(type:"Motorcycle");
22         assertTrue(motorcycle instanceof Motorcycle);
23     }
24
25     //test para cuando falla
26     @Test
27     public void testCreateInvalidVehicle() {
28         Vehicle vehicle = VehicleFactory.createVehicle(type:"Invalid");
29         if (vehicle != null) {
30             fail();
31         }
32     }
33 }
```



UNIVERSIDAD TÉCNICA DE AMBATO
FACULTAD DE INGENIERÍA EN SISTEMAS, ELECTRÓNICA E INDUSTRIAL
CARRERA DE SOFTWARE

Cdra. Universitaria (Predios Huachi) / Casilla 334 / Telefax: 03-2851894 – 2411537, Correo Electrónico: carrera.sistemas@uta.edu.ec
AMBATO-ECUADOR



Estudiantes: Karen Guatumillo, Hamilton Jumbo, Edwin López, Oscar Ramírez

Carrera: Software

Fecha: 27/Abril/2024

Asignatura: Patrones De Diseño

Nivel: Quinto

Paralelo: "A"

Tema: Patrones de Diseño Creacionales

Realizar 4 ejercicios prácticos de los patrones revisados en clase:

➤ PATRÓN SINGLETON

Objetivo: Implementar el Patrón Singleton para garantizar que solo exista una única instancia de la clase principal en Java Maven.

Descripción: Imagina que estas con un gestor de internet y lo quieres conectar a un dispositivo final como tu celular, debes de crear una aplicación que te permita garantizar que dicha conexión se vaya realizando una única vez en toda tu aplicación.

Desarrollo:

1. Se inicializa un proyecto con el nombre de PatronSGLT en eclipse con Maven.
2. Creamos la clase en la que se va a implementar el Patrón Singleton, en esta clase se debe de crear las variables con las que se podrá validar si el dispositivo está conectado o no.

```
>  App.java
>  InternerManager.java
```

3. Luego se añade un constructor el cual debe de ser privado para garantizar que no se esté creando una instancia de fuera de esta clase.
4. Es recomendable añadir una clase la cual verifique que si la instancia no existe pues cree esa misma instancia caso contrario no y devuelva la que ya está en ejecución, esto permite ayudar a la memoria a solo invocar al método cuando este el dispositivo desconectado.
5. Los demás métodos como el conect() y el disconnect() solo son una demostración en la que se imprime un mensaje que indique en caso de cumplirse cualquier situación.



UNIVERSIDAD TÉCNICA DE AMBATO
FACULTAD DE INGENIERÍA EN SISTEMAS, ELECTRÓNICA E INDUSTRIAL
CARRERA DE SOFTWARE

Cdra. Universitaria (Predios Huachi) / Casilla 334 / Telefax: 03-2851894 – 2411537, Correo Electrónico: carrera.sistemas@uta.edu.ec
AMBATO-ECUADOR



```
package PatronSingletonGrupo2.PatronSGLT;

public class InternerManager
{
    private static InternerManager instance;
    private boolean isConnected;

    public InternerManager()
    {
        this.isConnected = false;
    }

    public static InternerManager getInstance()
    {
        if( instance == null)
        {
            instance = new InternerManager();
        }
        return instance;
    }

    public boolean isConnected()
    {
        return isConnected;
    }

    public void connect()
    {
        isConnected = true;
        System.out.println("Se logro establecer conexion en tu dispositivo");
    }

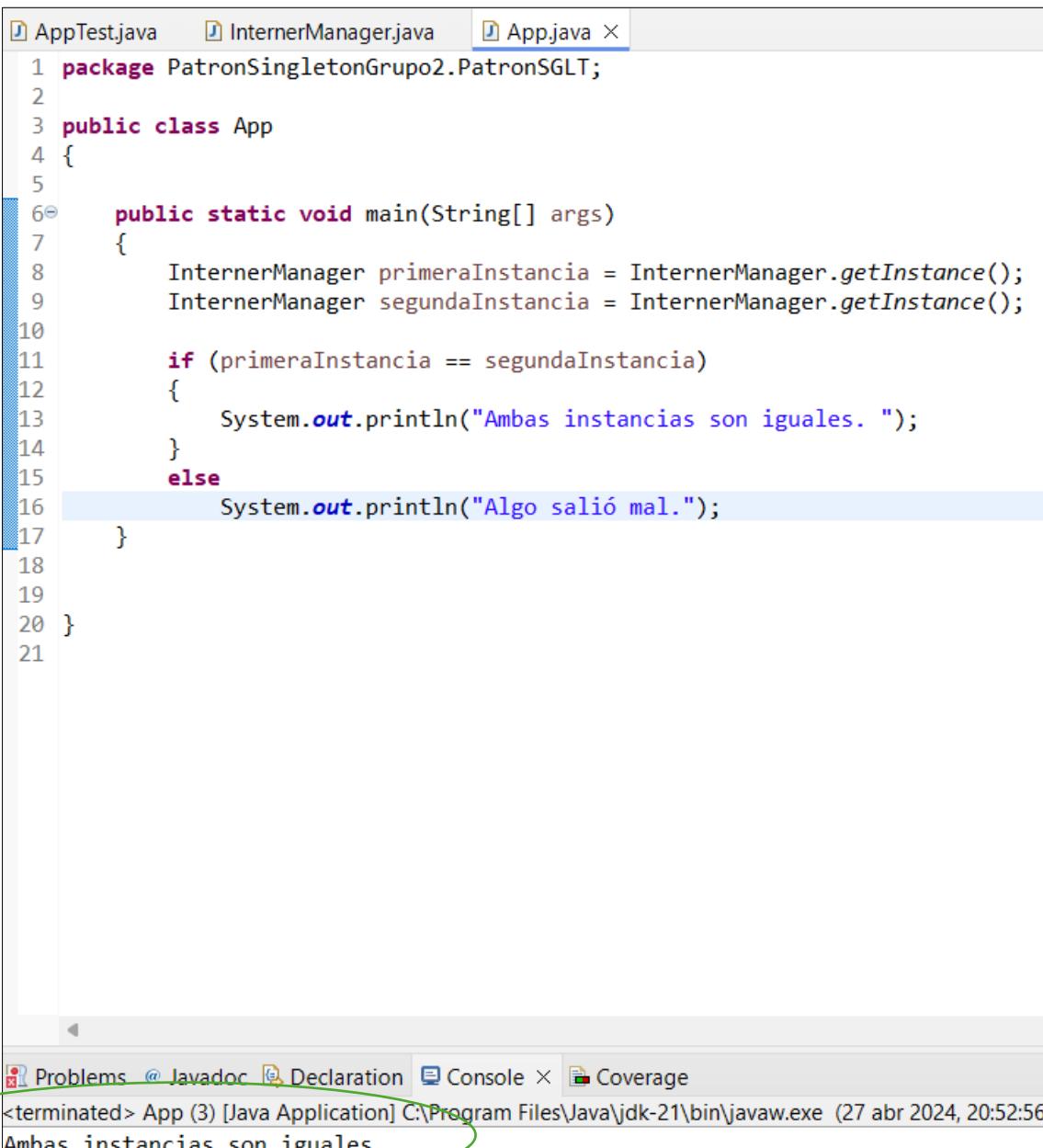
    public void disconnect()
    {
        isConnected = false;
        System.out.println("Su dispositivo esta desconectado");
    }
}
```

6. El resultado que se espera es que, a la hora de llamar dos instancias de la clase, deba aparecer que esa misma instancia es igual y no son dos instancias diferentes de la misma clase, entonces una forma fácil de probar es con un if como se demuestra:



UNIVERSIDAD TÉCNICA DE AMBATO
FACULTAD DE INGENIERÍA EN SISTEMAS, ELECTRÓNICA E INDUSTRIAL
CARRERA DE SOFTWARE

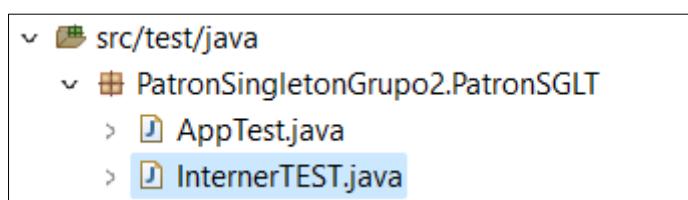
Cdra. Universitaria (Predios Huachi) / Casilla 334 / Telefax: 03-2851894 – 2411537, Correo Electrónico: carrera.sistemas@uta.edu.ec
AMBATO-ECUADOR



```
AppTest.java InternerManager.java App.java X
1 package PatronSingletonGrupo2.PatronSGLT;
2
3 public class App
4 {
5
6     public static void main(String[] args)
7     {
8         InternerManager primeraInstancia = InternerManager.getInstance();
9         InternerManager segundaInstancia = InternerManager.getInstance();
10
11         if (primeraInstancia == segundaInstancia)
12         {
13             System.out.println("Ambas instancias son iguales. ");
14         }
15         else
16             System.out.println("Algo salió mal.");
17     }
18
19
20 }
```

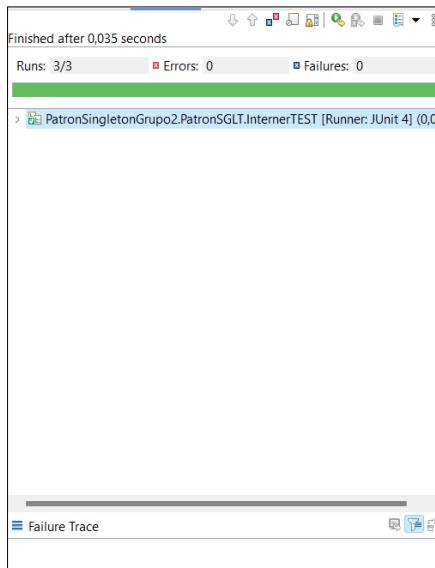
Problems @ Javadoc Declaration Console X Coverage
<terminated> App (3) [Java Application] C:\Program Files\Java\jdk-21\bin\javaw.exe (27 abr 2024, 20:52:56)
Ambas instancias son iguales.

7. Por último, se va a añadir una clase la cual va a implementar las pruebas unitarias, se debe de tomar en cuenta que en el patrón Singleton no se evidencia claramente en pruebas unitarias si no mas bien en los pasos que se detallaron anteriormente.





8. Corremos el TDD y instanciamos tres pruebas para verificar el patrón implementado.



```
import static org.junit.Assert.assertTrue;
import org.junit.Test;
public class InternerTEST {
    @Test
    public void testSingletonInstance() {
        InternerManager primeraInstancia = InternerManager.getInstance();
        InternerManager segundaInstancia = InternerManager.getInstance();
        assertEquals(primeraInstancia, segundaInstancia);
    }
    @Test
    public void testConnect() {
        InternerManager manager = InternerManager.getInstance();
        assertFalse(manager.isConnected());
        manager.connect();
        assertTrue(manager.isConnected());
    }
    @Test
    public void testDisconnect() {
        InternerManager manager = InternerManager.getInstance();
        manager.connect();
        assertTrue(manager.isConnected());
        manager.disconnect();
        assertFalse(manager.isConnected());
    }
}
```

➤ PATRÓN FACTORY

Objetivo:

Implementar el patrón **Factory** para crear objetos en una superclase que permita a las subclases alterar el tipo de objetos que serán creados.

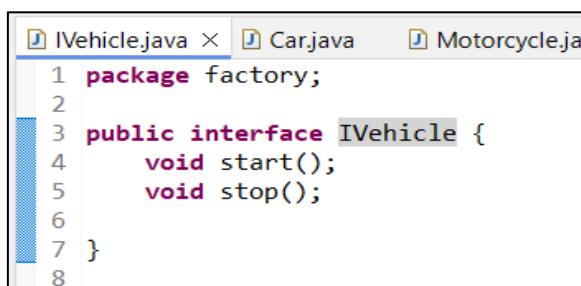
Descripción:

Piensa que estamos diseñando un sistema de creación de medios de transporte que requiere una implementación flexible y escalable de distintos vehículos. Para lograr esto, vamos a aplicar el patrón Factory.

Desarrollo:

1. Definir la Interfaz IVehicle:

Se crea una interfaz denominada IVehicle que contenga métodos comunes en el contexto de la creación de métodos de transporte, en este caso start y stop.



```
package factory;
public interface IVehicle {
    void start();
    void stop();
}
```



2. Crear la clase Car, Motorcycle, Truck y Plane:

Se construye cada una de las clases que van a hacer referencia a un tipo de transporte distinto y se implementa la interfaz IVehicle con sus métodos respectivos, modificándolos y adaptándolos a cada clase de transporte. Además, se pueden incluir métodos propios como en la clase Plane.

```
IVehicle.java Car.java × Motorcycle.java Truck.java Plane.java
1 package factory;
2
3 public class Car implements IVehicle{
4
5     @Override
6     public void start() {
7         System.out.println("---> Run car");
8     }
9
10    @Override
11    public void stop() {
12        System.out.println("---> Stop car");
13    }
14
15 }
16
17 }
18 }
```

```
IVehicle.java Car.java Motorcycle.java × Truck.java Plane.java
1 package factory;
2
3 public class Motorcycle implements IVehicle{
4
5     @Override
6     public void start() {
7         System.out.println("---> Run motorcycle");
8     }
9
10    @Override
11    public void stop() {
12        System.out.println("---> Stop motorcycle");
13    }
14
15 }
16
17 }
```

```
IVehicle.java Car.java Motorcycle.java Truck.java × Plane.java
1 package factory;
2
3 public class Truck implements IVehicle {
4
5     @Override
6     public void start() {
7         System.out.println("---> Run truck");
8     }
9
10    @Override
11    public void stop() {
12        System.out.println("---> Stop truck");
13    }
14
15 }
16
17 }
```



UNIVERSIDAD TÉCNICA DE AMBATO
FACULTAD DE INGENIERÍA EN SISTEMAS, ELECTRÓNICA E INDUSTRIAL
CARRERA DE SOFTWARE

Cdra. Universitaria (Predios Huachi) / Casilla 334 / Telefax: 03-2851894 – 2411537, Correo Electrónico: carrera.sistemas@uta.edu.ec
AMBATO-ECUADOR



```
IVehicle.java Car.java Motorcycle.java Truck.java Plane.java X
1 package factory;
2
3 public class Plane implements IVehicle {
4
5     @Override
6     public void start() {
7         System.out.println("---> Run plane");
8     }
9
10    @Override
11    public void stop() {
12        System.out.println("---> Stop plane");
13    }
14
15    public void fly() {
16        System.out.println("---> Take off the plane");
17    }
18
19 }
20
21 }
```

3. Definir la clase VehicleFactory:

En esta clase se implementa el patrón Factory, pues se crean instancias de diferentes tipos de vehículos según el vehículo especificado.

El método `createVehicle(String typeOfVehicle)` toma como argumento una cadena `typeOfVehicle` que especifica el tipo de vehículo que se desea crear.

El método verifica el valor de `typeOfVehicle` utilizando comparaciones de cadenas insensibles a mayúsculas y minúsculas (`equalsIgnoreCase()`) y en caso de existir crea y devuelve una instancia de esa clase.

```
IVehicle.java Car.java Motorcycle.java Truck.java Plane.java Main.java
1 package factory;
2
3 public class VehicleFactory {
4
5     public IVehicle createVehicle(String typeOfVehicle) {
6         if(typeOfVehicle.equalsIgnoreCase("Car")) {
7             return new Car();
8         }
9         if(typeOfVehicle.equalsIgnoreCase("Motorcycle")) {
10            return new Motorcycle();
11        }
12        if(typeOfVehicle.equalsIgnoreCase("Truck")) {
13            return new Truck();
14        }
15        if(typeOfVehicle.equalsIgnoreCase("Plane")) {
16            return new Plane();
17        }
18        System.out.println("Unknown vehicle");
19        return null;
20    }
21
22 }
```

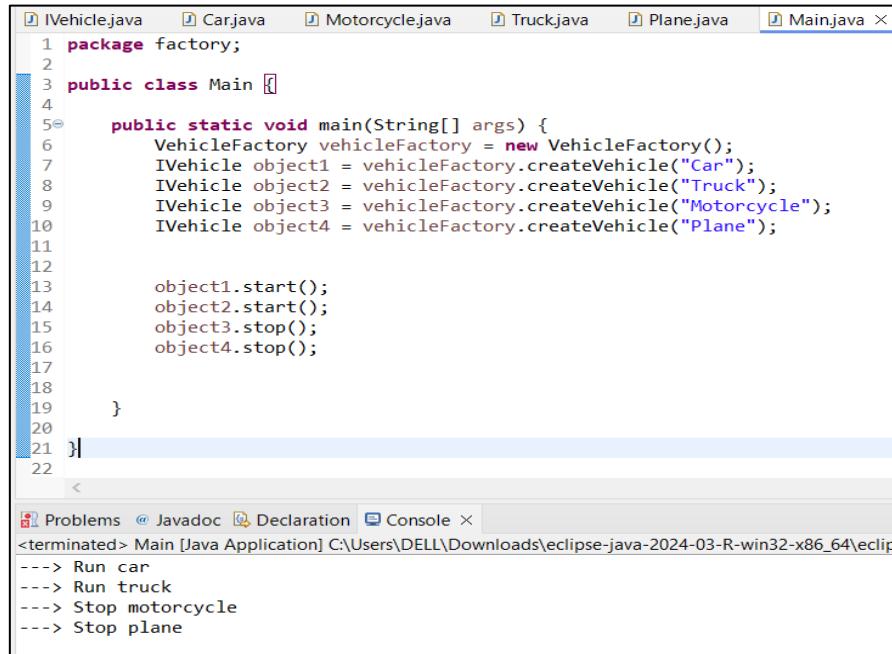
4. Implementar la clase Main:

La clase `Main` crea una instancia de `VehicleFactory`, que es la fábrica responsable de crear diferentes tipos de vehículos. Luego, utiliza esta fábrica para crear instancias de cuatro tipos diferentes de vehículos: carro "Car", camión "Truck", motocicleta "Motorcycle" y avión "Plane" que permite invocar a sus métodos.



UNIVERSIDAD TÉCNICA DE AMBATO
FACULTAD DE INGENIERÍA EN SISTEMAS, ELECTRÓNICA E INDUSTRIAL
CARRERA DE SOFTWARE

Cdla. Universitaria (Predios Huachi) / Casilla 334 / Telefax: 03-2851894 – 241537, Correo Electrónico: carrera.sistemas@uta.edu.ec
AMBATO-ECUADOR



```
IVehicle.java Car.java Motorcycle.java Truck.java Plane.java Main.java
1 package factory;
2
3 public class Main {
4
5     public static void main(String[] args) {
6         VehicleFactory vehicleFactory = new VehicleFactory();
7         IVehicle object1 = vehicleFactory.createVehicle("Car");
8         IVehicle object2 = vehicleFactory.createVehicle("Truck");
9         IVehicle object3 = vehicleFactory.createVehicle("Motorcycle");
10        IVehicle object4 = vehicleFactory.createVehicle("Plane");
11
12        object1.start();
13        object2.start();
14        object3.stop();
15        object4.stop();
16
17    }
18
19 }
20
21 }
```

Problems Declaration Console <terminated> Main [Java Application] C:\Users\DELL\Downloads\eclipse-java-2024-03-R-win32-x86_64\clip

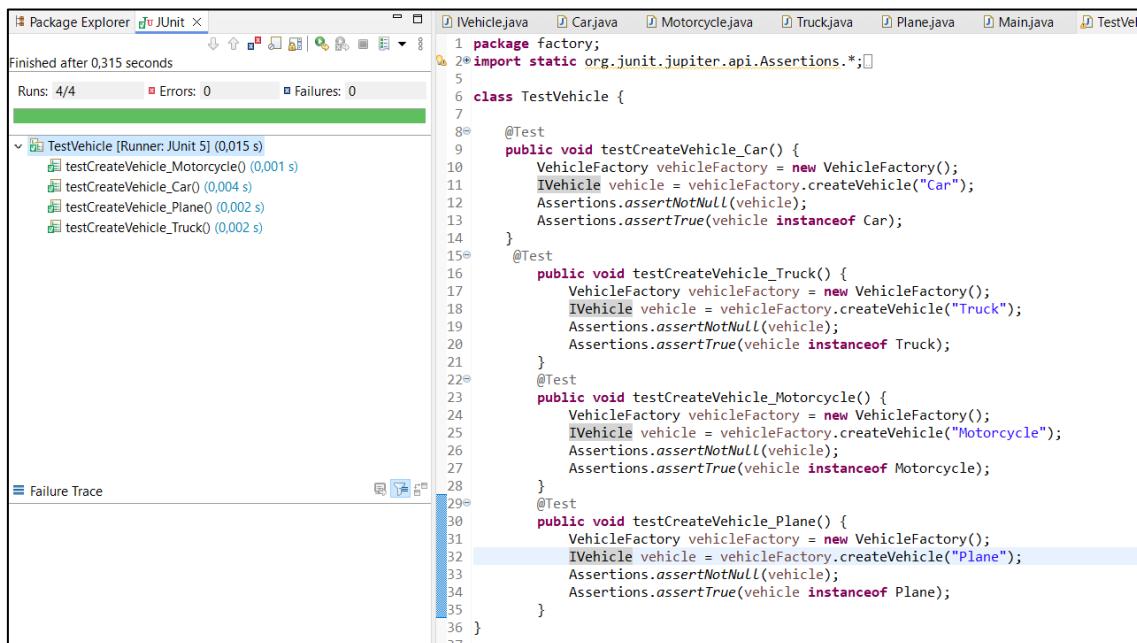
```
>>> Run car
>>> Run truck
>>> Stop motorcycle
>>> Stop plane
```

5. Crear la clase de pruebas TestVehicle:

Estas pruebas unitarias verifican si la clase VehicleFactory puede crear correctamente instancias de diferentes clases de vehículos según el tipo especificado.

Dentro de cada método de prueba, se instancia VehicleFactory y se llama al método createVehicle(String typeOfVehicle) con un tipo de vehículo específico.

Luego, se realizan las aserciones para verificar si el vehículo creado no es nulo y si es una instancia del tipo esperado de vehículo.



Package Explorer JUnit >

Finished after 0,315 seconds

Runs: 4/4 Errors: 0 Failures: 0

TestVehicle [Runner: JUnit 5] (0,015 s)

- testCreateVehicle_Motorcycle() (0,001 s)
- testCreateVehicle_Car() (0,004 s)
- testCreateVehicle_Plane() (0,002 s)
- testCreateVehicle_Truck() (0,002 s)

```
IVehicle.java Car.java Motorcycle.java Truck.java Plane.java Main.java TestVeh
1 package factory;
2
3 import static org.junit.jupiter.api.Assertions.*;
4
5 class TestVehicle {
6
7     @Test
8     public void testCreateVehicle_Car() {
9         VehicleFactory vehicleFactory = new VehicleFactory();
10        IVehicle vehicle = vehicleFactory.createVehicle("Car");
11        Assertions.assertNotNull(vehicle);
12        Assertions.assertTrue(vehicle instanceof Car);
13    }
14
15    @Test
16    public void testCreateVehicle_Truck() {
17        VehicleFactory vehicleFactory = new VehicleFactory();
18        IVehicle vehicle = vehicleFactory.createVehicle("Truck");
19        Assertions.assertNotNull(vehicle);
20        Assertions.assertTrue(vehicle instanceof Truck);
21    }
22
23    @Test
24    public void testCreateVehicle_Motorcycle() {
25        VehicleFactory vehicleFactory = new VehicleFactory();
26        IVehicle vehicle = vehicleFactory.createVehicle("Motorcycle");
27        Assertions.assertNotNull(vehicle);
28        Assertions.assertTrue(vehicle instanceof Motorcycle);
29    }
30
31    @Test
32    public void testCreateVehicle_Plane() {
33        VehicleFactory vehicleFactory = new VehicleFactory();
34        IVehicle vehicle = vehicleFactory.createVehicle("Plane");
35        Assertions.assertNotNull(vehicle);
36        Assertions.assertTrue(vehicle instanceof Plane);
37    }
}
```



➤ PATRÓN BUILDER

Objetivo:

Implementar el patrón creacional **Builder** para permitir la construcción de objetos complejos paso a paso, proporcionando una mayor flexibilidad y mantenibilidad al proceso de creación.

Descripción:

Se plantea un problema referente a la creación de diferentes tipos de pizzas dentro un local, por lo cual se quiere establecer el patrón Builder para mejorar la eficiencia en la producción.

Desarrollo:

1. Definir la clase Pizza

Creamos la clase Pizza con los atributos que necesitamos (Masa, Salsa, Relleno) y solo utilizaremos los métodos para obtener la información (getters).

```
class Pizza {  
  
    private final String dough;  
    private final String sauce;  
    private final String topping;  
  
    public Pizza(String dough, String sauce, String topping) {  
        this.dough = dough;  
        this.sauce = sauce;  
        this.topping = topping;  
    }  
  
    public String getDough() {  
        return dough;  
    }  
  
    public String getSauce() {  
        return sauce;  
    }  
  
    public String getTopping() {  
        return topping;  
    }  
}
```

2. Definir la interfaz IPizzaBuilder

Creamos una interfaz que tendrá los métodos para ingresar información (setters) y adicionalmente uno para la creación de un objeto de la clase Pizza.

```
public interface IPizzaBuilder {  
    IPizzaBuilder setDough(String dough);  
    IPizzaBuilder setSauce(String sauce);  
    IPizzaBuilder setTopping(String topping);  
    Pizza build();  
}
```



UNIVERSIDAD TÉCNICA DE AMBATO
FACULTAD DE INGENIERÍA EN SISTEMAS, ELECTRÓNICA E INDUSTRIAL
CARRERA DE SOFTWARE

Cdra. Universitaria (Predios Huachi) / Casilla 334 / Telefax: 03-2851894 – 2411537, Correo Electrónico: carrera.sistemas@uta.edu.ec
AMBATO-ECUADOR



3. Creación de la clase PizzaBuilder

A esta clase se le implementa la interfaz creada anteriormente para obtener los métodos obligatorios, también deberemos añadir los atributos de la clase Pizza para mejorar la eficiencia en la lógica de builder.

Cada uno de los métodos dará el valor recibido a sus atributos y devolverá el mismo, para posteriormente retornar todo el objeto creado en el método build().

```
public class PizzaBuilder implements IPizzaBuilder {

    private String dough;
    private String sauce;
    private String topping;

    @Override
    public PizzaBuilder setDough(String dough) {
        this.dough = dough;
        return this;
    }

    @Override
    public PizzaBuilder setSauce(String sauce) {
        this.sauce = sauce;
        return this;
    }

    @Override
    public PizzaBuilder setTopping(String topping) {
        this.topping = topping;
        return this;
    }

    @Override
    public Pizza build() {
        return new Pizza(dough, sauce, topping);
    }
}
```

4. Creación de la clase PizzaDirector

La clase director se utiliza principalmente para crear objetos con valores preestablecidos, por ello utilizaremos un objeto de la clase PizzaBuilder para ingresar información a los atributos y posteriormente retornar el objeto creado.



UNIVERSIDAD TÉCNICA DE AMBATO
FACULTAD DE INGENIERÍA EN SISTEMAS, ELECTRÓNICA E INDUSTRIAL
CARRERA DE SOFTWARE

Cdra. Universitaria (Predios Huachi) / Casilla 334 / Telefax: 03-2851894 – 2411537, Correo Electrónico: carrera.sistemas@uta.edu.ec
AMBATO-ECUADOR



```
class PizzaDirector {  
    private final PizzaBuilder builder;  
    public PizzaDirector(PizzaBuilder builder) {  
        this.builder = builder;  
    }  
    public Pizza makeMargherita() {  
        return builder  
            .setDough("Thin Crust")  
            .setSauce("Tomato")  
            .setTopping("Mozzarella")  
            .build();  
    }  
    public Pizza makePepperoni() {  
        return builder  
            .setDough("Thick Crust")  
            .setSauce("Tomato")  
            .setTopping("Pepperoni")  
            .build();  
    }  
}
```

5. Pruebas realizadas al patrón Builder

Instanciamos tanto al constructor como al director para realizar pruebas unitarias, empezando por el ingreso de cada uno de los atributos manualmente, para continuar con la creación de un objeto completo gracias a los valores predefinidos en el director (Peperoni y Margherita).

```
public class PizzaTest {  
  
    private PizzaBuilder builder = new PizzaBuilder();  
    private PizzaDirector director = new PizzaDirector(builder);  
  
    @Test  
    public void testDough() {  
        Pizza pizza = builder.setDough("suave").build();  
        assertEquals("suave", pizza.getDough());  
    }  
  
    @Test  
    public void testSauce() {  
        Pizza pizza = builder.setSauce("picante").build();  
        assertEquals("picante", pizza.getSauce());  
    }  
  
    @Test  
    public void testTopping() {  
        Pizza pizza = builder.setTopping("jamón").build();  
        assertEquals("jamón", pizza.getTopping());  
    }  
  
    @Test  
    public void testDirectorPeperoni() {  
        Pizza pizza = director.makePepperoni();  
  
        assertEquals("Thick Crust", pizza.getDough());  
        assertEquals("Tomato", pizza.getSauce());  
        assertEquals("Pepperoni", pizza.getTopping());  
    }  
  
    @Test  
    public void testDirectorMargherita() {  
        Pizza pizza = director.makeMargherita();  
  
        assertEquals("Thin Crust", pizza.getDough());  
        assertEquals("Tomato", pizza.getSauce());  
        assertEquals("Mozzarella", pizza.getTopping());  
    }  
}
```



➤ PATRÓN PROTOTYPE

Objetivo:

Implementar el patrón creacional **Prototype** para la clonación de objetos existentes sin que el código dependa de sus clases.

Descripción:

En el contexto de una tienda que ofrece diversos productos, surge la necesidad de optimizar la creación y gestión de objetos similares. Para abordar este desafío, se propone aplicar el patrón Prototype.

Desarrollo:

1. Definir la clase Producto

Creamos la clase Producto con los atributos que necesitamos (nombre, precio, stock) y generamos tanto el constructor como los getters y setters.

```
3  public class Producto implements IProducto {  
4      private String name;  
5      private double price;  
6      private int stock;  
7  
8      // contructors  
9  
10     public Producto() {  
11    }  
12  
13     public Producto(String name, double price, int stock) {  
14         this.name = name;  
15         this.price = price;  
16         this.stock = stock;  
17     }  
18     tabnine: test | explain | document | ask  
19     // getters and setters
```

2. Definir la interfaz IProducto

Creamos una interfaz que extienda la clase Clonable que tendrá el método clonar.

```
3  public interface IProducto extends Cloneable{  
4      IProducto clone();  
5  }
```



3. Implementación de la interfaz IProducto en la clase Producto

A esta clase se le implementa la interfaz IProducto para obtener el método obligatorio clone(). El cual se sobrescribe el método con la lógica para retornar el objeto clonado. Además, se añade el método toString para visualizar de mejor manera la respuesta.

```
3  public class Producto implements IProducto {  
4      @Override  
5      public String toString() {  
6          return "Producto [name=" + name + ", price=" + price + ", stock=" + stock + "]";  
7      }  
8  
9      tabnine: test | explain | document | ask  
10     @Override  
11     public IProducto clone() {  
12         Producto clone = new Producto();  
13         clone.setName(this.getName());  
14         clone.setPrice(this.getPrice());  
15         clone.setStock(this.getStock());  
16         return clone;  
17     }  
18  
19 } ?
```

4. Probar el patrón prototype en la clase main.

Instanciamos un objeto de tipo producto (product1) y le asignamos de nombre “Lavadora”, precio “1000” y Stock de “10”. Para lo cual deseamos crear un clon de dicho objeto. Donde se cambiaria el nombre a “Secadora” y el precio a “1100”.

```
3  public class App  
4  {  
5      Run | Debug | tabnine: test | explain | document | ask  
6      public static void main( String[] args )  
7      {  
8          Producto product1 = new Producto();  
9          product1.setName(name:"Lavadora");  
10         product1.setPrice(price:1000);  
11         product1.setStock(stock:10);  
12  
13         Producto product2 = (Producto) product1.clone();  
14  
15         System.out.println(product1);  
16         System.out.println(product2);  
17         product2.setName(name:"Secadora");  
18         product2.setPrice(price:1100);  
19         System.out.println(product1);  
20         System.out.println(product2);  
21  
22     }  
23 } ?
```



UNIVERSIDAD TÉCNICA DE AMBATO
FACULTAD DE INGENIERÍA EN SISTEMAS, ELECTRÓNICA E INDUSTRIAL
CARRERA DE SOFTWARE

Cdra. Universitaria (Predios Huachi) / Casilla 334 / Telefax: 03-2851894 – 2411537, Correo Electrónico: carrera.sistemas@uta.edu.ec
AMBATO-ECUADOR



Se comprueba imprimiendo por consola.

```
PS D:\Capeta de Oscar\5to Semestre\Patrones de diseño\ProyectosJava\EjemploPrototype>
e diseño\ProyectosJava\EjemploPrototype\ejprototype\target\classes' 'com.ramirez.oscar
Producto [name=Lavadora, price=1000.0, stock=10]
Producto [name=Lavadora, price=1000.0, stock=10]
-----
Producto [name=Lavadora, price=1000.0, stock=10]
Producto [name=Secadora, price=1100.0, stock=10]
```

5. Pruebas realizadas al patrón prototype

Instanciamos la clase producto para realizar las pruebas unitarias, empezamos por el test que comprueba que se ha clonado exitosamente.

```
1 package com.ramirez.oscar;
2
3 import static org.junit.Assert.*;
4
5 import org.junit.Test;
6
7 public class AppTest {
8
9     tabnine: test | explain | document | ask
10    @Test
11    public void testClonacionExitosa() {
12        Producto product1 = new Producto(name:"Lavadora", price:1000, stock:10);
13        Producto product2 = (Producto) product1.clone();
14
15        // Verificar que los objetos son diferentes
16        assertNotSame(product1, product2);
17
18        // Verificar que los atributos son iguales
19        assertEquals(product1.getName(), product2.getName());
20        assertEquals(product1.getPrice(), product2.getPrice(), delta:0);
21        assertEquals(product1.getStock(), product2.getStock());
22    }
23 }
```

Después verificamos la independencia entre la originar y el objeto clonado.

```
23 @Test
24 public void testIndependenciaEntreOriginalYClon() {
25     Producto product1 = new Producto(name:"Lavadora", price:1000, stock:10);
26     Producto product2 = (Producto) product1.clone();
27
28     // Modificar el clon
29     product2.setName(name:"Secadora");
30     product2.setPrice(price:1100);
31     product2.setStock(stock:8);
32
33     // Verificar que el original no cambió
34     assertEquals(expected:"Lavadora", product1.getName());
35     assertEquals(expected:1000, product1.getPrice(), delta:0);
36     assertEquals(expected:10, product1.getStock());
37 }
```



UNIVERSIDAD TÉCNICA DE AMBATO
FACULTAD DE INGENIERÍA EN SISTEMAS, ELECTRÓNICA E INDUSTRIAL
CARRERA DE SOFTWARE

Cdra. Universitaria (Predios Huachi) / Casilla 334 / Telefax: 03-2851894 – 2411537, Correo Electrónico: carrera.sistemas@uta.edu.ec
AMBATO-ECUADOR



Ahora modificamos los atributos del clone y comprobamos que se ha realizado el cambio de valores.

```
tabnine: test | explain | document | ask
39  @Test
40  public void testModificacionAtributosClon() {
41      Producto product1 = new Producto(name:"Lavadora", price:1000, stock:10);
42      Producto product2 = (Producto) product1.clone();
43
44      // Modificar el clon
45      product2.setName(name:"Secadora");
46      product2.setPrice(price:1100);
47      product2.setStock(stock:8);
48
49      // Verificar que el clon cambió
50      assertEquals(expected:"Secadora", product2.getName());
51      assertEquals(expected:1100, product2.getPrice(), delta:0);
52      assertEquals(expected:8, product2.getStock());
53  }
```

Después, comprobamos la creación de múltiples clones de los cuales cambiamos sus parámetros y visualizamos si no son iguales a el objeto original.

```
55  @Test
56  public void testCreacionMultiplesClones() {
57      Producto product1 = new Producto(name:"Lavadora", price:1000, stock:10);
58      Producto product2 = (Producto) product1.clone();
59      Producto product3 = (Producto) product1.clone();
60
61      // Modificar los clones
62      product2.setName(name:"Secadora");
63      product2.setPrice(price:1100);
64      product3.setStock(stock:8);
65
66      // Verificar que los clones son independientes
67      assertNotEquals(product1.getName(), product2.getName());
68      assertNotEquals(product1.getPrice(), product2.getPrice(), delta:0);
69      assertNotEquals(product1.getStock(), product3.getStock());
70  }
```

Por último, se testeó que la referencia sea diferente entre el objeto clonado y el clon.

```
tabnine: test | explain | document | ask
72  @Test
73  public void testReferenciaDiferente() {
74      Producto product1 = new Producto(name:"Lavadora", price:1000, stock:10);
75      Producto product2 = (Producto) product1.clone();
76
77      // Verificar que los objetos tienen diferente referencia de memoria
78      assertNotSame(product1, product2);
79  }
```

- ✔ Test run at 4/28/2024, 12:18:23 PM
- ✔ ⚡ testClonacionExitosa()
- ✔ ⚡ testCreacionMultiplesClones()
- ✔ ⚡ testIndependenciaEntreOriginalYClon()
- ✔ ⚡ testModificacionAtributosClon()
- ✔ ⚡ testReferenciaDiferente()



I. PORTADA

UNIVERSIDAD TÉCNICA DE AMBATO

Facultad de Ingeniería en Sistemas, Electrónica e Industrial

Título: Patrones de Diseño Creacionales

Carrera: Software

Nivel y Paralelo: 5 A

Alumnos participantes:
Espinoza Lata Jordy Joel
Flores Tacuaman Elvis Sebastian
Ramírez Moreta Jade Yadira
Rivera Claudio Alejandro Sebastian

Módulo y Docente: Aplicaciones Distribuidas Ing. Mg. Pablo I. Morales P



Contenido

1.1	Tema.....	3
1.2	Objetivos	3
1.3	Marco Teórico	3
1.4	Singleton.....	4
1.5	Factory.....	6
1.6	Builder	9
1.7	Prototype	12
1.8	Referencias	15



1.1 Tema

Patrones de Diseño Creacionales

1.2 Objetivos

Objetivo General

Implementar cuatro ejercicios con el fin de entender de mejor manera como se implementan los diferentes patrones de diseño

Objetivos específicos

Investigar los siguientes patrones de diseño

Desarrollar ejercicios de cada patrón investigado

1.3 Marco Teórico

Singleton:

El patrón Singleton garantiza que una clase tenga una única instancia y proporciona un punto de acceso global a esa instancia. Es útil cuando se necesita controlar estrictamente cómo se accede a una clase y garantizar que solo exista una instancia de ella en todo el programa. Al usar el Singleton, se evita la creación repetida de la misma instancia y se facilita el acceso a los recursos compartidos. Sin embargo, su implementación debe ser cuidadosa para garantizar que la instancia única se maneje correctamente en entornos multi-hilo y para evitar problemas de concurrencia.

Factory:

El patrón Factory proporciona una interfaz para crear objetos de una familia de clases relacionadas sin especificar las clases concretas. Permite encapsular la creación de objetos y abstraer la lógica de creación de su uso. Con Factory, se puede crear un objeto sin conocer los detalles de su implementación, lo que facilita la adición de nuevos tipos de objetos sin modificar el código existente. Además, al centralizar la lógica de creación, se mejora el mantenimiento y la escalabilidad del código al evitar la dispersión de la lógica de creación en múltiples lugares.

Builder:

El patrón Builder separa la construcción de un objeto complejo de su representación, permitiendo la creación de diferentes representaciones del mismo objeto utilizando el mismo proceso de construcción. Proporciona una forma más flexible y paso a paso de crear objetos, permitiendo configurar y personalizar el objeto antes de finalizar su construcción. El Builder abstracta el proceso de construcción del objeto, lo que hace que el código sea más legible y



mantenible, especialmente cuando se trabaja con objetos que tienen muchas configuraciones posibles. Además, facilita la creación de objetos inmutables y ayuda a evitar la necesidad de múltiples constructores con parámetros opcionales.

Prototype:

El patrón Prototype permite la creación de nuevos objetos mediante la copia de un objeto existente, evitando la necesidad de crear objetos desde cero. Es útil cuando la creación directa de objetos es costosa o complicada, ya sea debido a la complejidad del objeto o a la necesidad de inicializarlo con datos específicos. Al utilizar el Prototype, se pueden crear nuevos objetos mediante la clonación de un prototipo existente, lo que facilita la creación de objetos similares con diferentes valores de estado. Esto promueve la reutilización de código y mejora el rendimiento al evitar la redundancia en la creación de objetos.

1.4 Ejercicio Singleton

Creación de la Clase EventLogger:

- Implementación del Singleton: La clase EventLogger implementa el patrón Singleton para garantizar que solo exista una única instancia en toda la aplicación.
- Instancia única: Se declara una instancia privada y estática de la clase (private static EventLogger instance).
- Constructor privado: Se declara un constructor privado para evitar la creación de instancias fuera de la clase (private EventLogger()).
- Método estático getInstance(): Este método estático proporciona acceso a la única instancia de la clase. Si la instancia aún no ha sido creada, se crea una nueva instancia y se devuelve.
- Método logEvent(String event): Este método agrega un evento al registro de eventos. El evento se añade a una lista interna y se muestra un mensaje en la consola indicando que se ha registrado un evento.
- Método getEventLog(): Este método devuelve el registro completo de eventos.



```
J EventLogger.java > EventLogger
1 import java.util.ArrayList;
2 import java.util.List;
3
4 public class EventLogger {
5     private static EventLogger instance;
6
7     private List<String> eventLog;
8
9     private EventLogger() {
10         eventLog = new ArrayList<>();
11     }
12
13     public static synchronized EventLogger getInstance() {
14         if (instance == null) {
15             instance = new EventLogger();
16         }
17         return instance;
18     }
19
20     public void logEvent(String event) {
21         eventLog.add(event);
22         System.out.println("Evento registrado: " + event);
23     }
24
25     public List<String> getEventLog() {
26         return eventLog;
27     }
28 }
```

Creación de la clase Main:

- Se obtiene la instancia del Singleton EventLogger utilizando el método estático getInstance().
- Se agregan eventos al registro utilizando el método logEvent(String event).
- Se obtiene el registro completo de eventos utilizando el método getEventLog().



```
J Mainjava X
J Mainjava > ...
1 public class Main {
2     Run | Debug
3     public static void main(String[] args) {
4         EventLogger logger = EventLogger.getInstance();
5
6         logger.logEvent(event:"\nInicio de sesión de usuario");
7         logger.logEvent(event:"Creación de un nuevo archivo");
8         logger.logEvent(event:"Cierre de sesión de usuario");
9
10        System.out.println(x:"\nRegistro de eventos:");
11        for (String event : logger.getEventLog()) {
12            System.out.println(event);
13        }
14    }
}
```

1.5 Ejercicio Factory

1. Clase TipoDePago:

La clase TipoDePago es una enumeración que define los diferentes métodos de pago disponibles en el sistema. Cada tipo de pago se representa como una constante en la enumeración, lo que facilita su identificación y uso en el programa.

```
J TipoDePago.java X
J TipoDePago.java > ↗TipoDePago
1 public enum TipoDePago {
2     PAYPAL, TARJETA_CREDITO, TRANSFERENCIA_BANCARIA, EFECTIVO
3 }
```

2. Interfaz Pago:

La interfaz Pago define un contrato para las clases que representan métodos de pago específicos. Contiene un único método abstracto crearPago(), que debe ser implementado por las clases que la implementen. Este método representa el proceso de realizar un pago utilizando un método de pago particular.

```
J Pago.java X
J Pago.java > ...
1 public interface Pago {
2     void crearPago();
3 }
```

3. Clase PagoPayPal:

La clase PagoPayPal implementa la interfaz Pago. Su método crearPago() imprime la forma de pago seleccionada, este método realizaría el proceso de pago utilizando PayPal.



```
J PagoPayPal.java X
J PagoPayPal.java > ...
1 public class PagoPayPal implements Pago {
2     @Override
3     public void crearPago(){
4         System.out.println("Procesando pago a través de PayPal...");
5     }
6 }
7
```

4. Clase PagoTransferencia:

Similar a PagoPayPal, la clase PagoTransferencia también implementa la interfaz Pago. Su método crearPago() imprime la forma de pago, este método realizaría el proceso de pago mediante una transferencia bancaria.

```
J PagoTransferencia.java X
J PagoTransferencia.java > ...
1 public class PagoTransferencia implements Pago{
2     @Override
3     public void crearPago(){
4         System.out.println("Procesando pago a través de una Transferencia...");
5     }
6 }
7
```

5. Clase PagoTarjeta:

Al igual que las anteriores, la clase PagoTarjeta implementa la interfaz Pago. Su método crearPago() imprime la forma de pago, este método sería responsable de realizar el proceso de pago utilizando una tarjeta de crédito.

```
J PagoTarjeta.java X
J PagoTarjeta.java > ...
1 public class PagoTarjeta implements Pago{
2     @Override
3     public void crearPago(){
4         System.out.println("Procesando pago a través de la Tarjeta...");
5     }
6 }
7
```

6. Clase PagoEfectivo:

La clase PagoEfectivo también implementa la interfaz Pago. Su método crearPago() imprime la forma de pago, este método representa el proceso de pago utilizando efectivo.

```
J PagoEfectivo.java X
J PagoEfectivo.java > ...
1 public class PagoEfectivo implements Pago {
2     @Override
3     public void crearPago(){
4         System.out.println("El pago se realiza de manera física (Efectivo)");
5     }
6 }
7
```

7. Clase PagoFactory:

La clase PagoFactory es una fábrica que proporciona una instancia concreta de la interfaz Pago según el tipo de pago especificado. Contiene un método obtenerPago(TipoDePago tipoDePago) que toma un TipoDePago como argumento y devuelve una instancia



correspondiente de la clase de pago apropiada. Utiliza un switch para determinar qué clase de pago instanciar según el tipo de pago proporcionado.

```
J PagoFactory.java X
J PagoFactory.java > ...
1  public class PagoFactory {
2
3      public Pago obtenerPago(TipoDePago tipoDePago) throws NoSuchFieldException{
4
5          return switch(tipoDePago){
6              case PAYPAL-> new PagoPayPal();
7              case EFECTIVO-> new PagoEfectivo();
8              case TARJETA_CREDITO-> new PagoTarjeta();
9              case TRANSFERENCIA_BANCARIA-> new PagoTransferencia();
10         };
11     }
12 }
```

8. Clase Main:

La clase Main contiene el método main, que es el punto de entrada del programa. En el método main, se crea una instancia de PagoFactory y se utiliza para obtener una instancia de Pago según el tipo de pago seleccionado por el usuario. Esto se logra mediante la interacción con el usuario a través de la entrada estándar (en este caso, mediante un menú simple). Una vez obtenida la instancia de Pago, se llama al método crearPago() en esa instancia, lo que inicia el proceso de pago correspondiente.

```
J PagoFactory.java J Main.java 1 X
J Main.java > ...
1  import java.util.Scanner;
2
3  public class Main {
4      Run | Debug
5      public static void main(String[] args) throws NoSuchFieldException {
6          Scanner scanner = new Scanner(System.in);
7          PagoFactory pagoFactory = new PagoFactory();
8
8          System.out.println("Seleccione el método de pago:");
9          System.out.println("1. PayPal");
10         System.out.println("2. Tarjeta de crédito");
11         System.out.println("3. Transferencia bancaria");
12         System.out.println("4. Efectivo");
13
14         int opcion = scanner.nextInt();
15         TipoDePago tipoDePago;
16
17         switch (opcion) {
18             case 1:
19                 tipoDePago = TipoDePago.PAYPAL;
20                 break;
21             case 2:
22                 tipoDePago = TipoDePago.TARJETA_CREDITO;
23                 break;
24             case 3:
25                 tipoDePago = TipoDePago.TRANSFERENCIA_BANCARIA;
26                 break;
27             case 4:
28                 tipoDePago = TipoDePago.EFECTIVO;
29                 break;
30             default:
31                 System.out.println("Opción no válida. Seleccionando PayPal por defecto.");
32                 tipoDePago = TipoDePago.PAYPAL;
33                 break;
34         }
35
36         Pago pago = pagoFactory.obtenerPago(tipoDePago);
37         pago.crearPago();
38     }
39 }
```



1.6 Ejercicio Builder

1. Definir la Clase Robot:

Se crea una clase Robot con atributos como tipo, cabeza, brazo, piernas, etc. Implementando métodos setter para las propiedades y un método `toString` para poder sobrescribir cada uno de los robots.

```
package PatronBuilder;

public class Robot {

    private String tipo;
    private String cabeza;
    private String torso;
    private String brazos;
    private String piernas;
```

```
public Robot(String tipo) {
    this.tipo = tipo;
}

public void setCabeza(String cabeza) {
    this.cabeza = cabeza;
}

public void setTorso(String torso) {
    this.torso = torso;
}

public void setBrazos(String brazos) {
    this.brazos = brazos;
}

public void setPiernas(String piernas) {
    this.piernas = piernas;
}

@Override
public String toString() [
    return "Robot " + tipo + " - Cabeza: " + cabeza + ", Torso: " +
        torso + ", Brazos: " + brazos + ", Piernas: " + piernas;
]
```

2. Crear la interfaz Builder

Se define una interfaz `RobotBuilder` con métodos para construir las diferentes características del robot, como `buildCabeza`, `buildTorso`, `buildBrazos`, `buildPiernas` y `getRobot` que vendrá a devolver el robot completo.



```
1 package PatronBuilder;
2
3
4 public interface RobotBuilder {
5
6     void buildCabeza();
7     void buildTorso();
8     void buildBrazos();
9     void buildPiernas();
10    Robot getRobot();
11 }
12
13 }
```

3. Implementar el Builder

Se crea una clase Robot que implemente la interfaz RobotBuilder. Dentro de esta clase se definen métodos para establecer las características que tendrá el robot explorador.

```
1 package PatronBuilder;
2
3 public class RobotExploradorBuilder implements RobotBuilder {
4     private Robot robot;
5
6     public RobotExploradorBuilder() {
7         this.robot = new Robot("Explorador");
8     }
9
10    public void buildCabeza() {
11        robot.setCabeza(cabeza:"Cámara de visión 360");
12    }
13
14    public void buildTorso() {
15        robot.setTorso(torso:"Sensores de ambiente");
16    }
17
18    public void buildBrazos() {
19        robot.setBrazos(brazos:"Brazos extensibles");
20    }
21
22    public void buildPiernas() {
23        robot.setPiernas(piernas:"Ruedas todo terreno");
24    }
25
26    public Robot getRobot() {
27
28        return robot;
29    }
30 }
```

4. Implementar el director



Se crea una clase RobotDirector que tenga un método construct que tome un RobotBuilder como parámetro. Dentro de construct, utiliza los métodos del RobotBuilder para construir un objeto de Robot

```
1 package PatronBuilder;
2
3 public class RobotDirector {
4
5
6     public void construct(RobotBuilder builder ){
7
8         builder.buildCabeza();
9         builder.buildBrazos();
10        builder.buildTorso();
11        builder.buildPiernas();
12
13    }
14 }
```

5. Probar el patrón Builder

Se crea un objeto RobotDirector y un objeto de RobotBuilder. Llama al método construct del director, pasando el constructor del robot como argumento. Accede al objeto Robot construido y se imprime sus características para verificar que se haya creado correctamente.

```
1 package PatronBuilder.Principal;
2 import PatronBuilder.Robot;
3 import PatronBuilder.RobotBuilder;
4 import PatronBuilder.RobotDirector;
5 import PatronBuilder.RobotExploradorBuilder;
6 import PatronBuilder.RobotMilitarBuilder;
7
8
9
10 public class Main {
11     Run | Debug
12     public static void main(String[] args) {
13
14         //se instancian las clases
15
16         RobotDirector director = new RobotDirector();
17
18         RobotBuilder builder = new RobotMilitarBuilder();
```

```
// Construir un robot militar
19
20         director.construct(builder);
21         Robot robotMilitar = builder.getRobot();
22         System.out.println(robotMilitar);
23
24         // Construir un robot explorador
25
26         builder = new RobotExploradorBuilder();
27         director.construct(builder);
28         Robot robotExplorador = builder.getRobot();
29         System.out.println(robotExplorador);
```

```
pi\AppData\Roaming\Code\User\workspaceStorage\d71ae1c286fba8b69fffb5e56af878f3\redhat.java\jdt_ws\Patrones_d7160254\bin' 'PatronBuilder.Principal.Main'
Robot Militar - Cabeza: Casco de combate, Torso: Chaleco antibalas, Brazos: Armas integradas, Piernas: Botas reforzadas
Robot Explorador - Cabeza: Cámara de visión 360, Torso: Sensores de ambiente, Brazos: Brazos extensibles, Piernas: Ruedas todo terreno
PS C:\Patrones>
```



1.7 Ejercicio Prototype

En este ejercicio, se te pide crear una clase CuentaAHImpl que represente una cuenta de ahorro bancaria utilizando el patrón Prototype. Deberás implementar métodos para establecer y obtener el monto de la cuenta, así como un método para clonar una instancia de CuentaAHImpl. Luego, en la clase App, se realizarán operaciones para crear una cuenta de ahorro, establecer su monto, clonarla y comparar las referencias.

1. Primer Paso

- Primero creamos la interfaz llamada ICuenta la cual extendemos de Clonable que es una clase propia de Java.
- En esta interfaz establecemos un método llamado clonar() que me va a devolver la misma interfaz por si existe una clase que implemente esa interfaz y cumpla con ese objetivo.

```
App.java  ICuenta.java ×
1 package com.alejo.inter;
2
3 public interface ICuenta extends Cloneable {
4
5     ICuenta clonar();
6
7 }
8 |
```

2. Segundo Paso

- Creamos otra Clase llamada CuentaAHImpl (Cuenta Ahorro Implementación) y implementamos la interfaz creada anteriormente ICuenta
- En esta clase creo dos atributos y un método de apoyo de tipo String y double, con un constructor en el cual inicializo la variable tipo como “AHORRO”.

```
App.java  ICuenta.java  CuentaAHImpl.java ×
1 package com.alejo.model;
2
3 import com.alejo.inter.ICuenta;
4
5 public class CuentaAHImpl implements ICuenta {
6
7     private String tipo;
8     private double monto;
9
10    public CuentaAHImpl() {
11        tipo = "AHORRO";
12    }
13}
```



3. Tercer Paso

- Cuando se invoca clonar() en una instancia de CuentaAHImpl, se intenta realizar una clonación utilizando el método clone().
- Se inicializa una variable cuenta como null para almacenar la copia clonada.
- Dentro de un bloque try-catch, se realiza la clonación mediante (CuentaAHImpl) clone(). Esto utiliza la implementación predeterminada de clone() en Java para crear una copia superficial de la instancia actual de CuentaAHImpl.
- Si la clonación es exitosa, la variable cuenta ahora contiene una copia independiente de la instancia original.
- Si la clonación falla debido a que CuentaAHImpl no implementa Cloneable o porque el método clone() no está accesible, se captura la excepción CloneNotSupportedException.
- Finalmente, el método clonar() devuelve la variable cuenta, que es la copia clonada de la instancia original.

```
14•      @Override
15      public ICuenta clonar() {
16          CuentaAHImpl cuenta = null;
17
18          try {
19              cuenta = (CuentaAHImpl) clone();
20          } catch (CloneNotSupportedException e) {
21              e.printStackTrace();
22          }
23
24          return cuenta;
25      }
26
```

4. Cuarto Paso

- Creamos el método toString() que devuelve una representación en forma de cadena (String) de la instancia de CuentaAHImpl, que incluye información sobre el tipo de cuenta y el monto actual.
- Este método es anotado con @Override, indicando que está sobrescribiendo el método toString() de la clase padre (Object).
- La implementación devuelve una cadena que contiene el tipo de cuenta (tipo) y el monto (monto) de la cuenta actual.
- Y creamos sus respectivo setter y getter de monto.

```
27•      @Override
28      public String toString() {
29          return "CuentaAHImpl [tipo=" + tipo + ", monto=" + monto + "]";
30      }
31
32•      public void setMonto(double monto) {
33          this.monto = monto;
34      }
35
36•      public double getMonto() {
37          return monto;
38      }
39
```



5. Quinto Paso

- La clase App contiene un método main() que demuestra cómo usar la clase CuentaAHImpl para crear una cuenta de ahorro, clonarla y realizar comparaciones entre las instancias.
- Se crea una nueva instancia de CuentaAHImpl llamada cuentaAhorro.
- Se utiliza el método setMonto(double monto) para establecer el monto de la cuenta de ahorro en 200.
- Se clona la instancia cuentaAhorro utilizando el método clonar().
- Se verifica si la cuenta clonada (cuentaClonada) no es null antes de imprimir su representación utilizando el método toString().
- Se compara la referencia de la cuenta clonada (cuentaClonada) con la referencia de la cuenta original (cuentaAhorro) utilizando el operador ==. Esto imprimirá true si ambas variables apuntan a la misma instancia de objeto (es decir, si no se ha creado una nueva instancia en la clonación).

```
*App.java × ICuentajava CuentaAHImpl.java
1 package com.alejo;
2
3 import com.alejo.model.CuentaAHImpl;
4
5 public class App {
6
7     public static void main(String[] args) {
8
9         CuentaAHImpl cuentaAhorro = new CuentaAHImpl();
10        cuentaAhorro.setMonto(200);
11        CuentaAHImpl cuentaClonada = (CuentaAHImpl) cuentaAhorro.clonar();
12
13
14        if (cuentaClonada != null) {
15            System.out.println(cuentaClonada);
16        }
17
18        System.out.println(cuentaClonada == cuentaAhorro);
19
20    }
21
22 }
23
```



- La salida del programa mostrará la representación de la cuenta clonada (si no es null) y el resultado de la comparación de referencias entre la cuenta clonada y la cuenta original.

The screenshot shows a Java application running in an IDE. The console tab is active, displaying the following output:

```
<terminated> app [Java Application] C:\Program Files\Java\jdk-11\bin\javaw.exe (29 abr 2020 10:45:43)  
CuentaAHImpl [tipo=AHORRO, monto=200.0]  
false
```

1.8 Referencias

- Rodríguez Portela, A. E. (2019). Patrón de diseño Singleton. *Arquitectura de Software. a Objetos, P. O. Patrones de diseño orientado a objetos.*
- Blanco, C. (2008). Patrones de diseño. *Ingeniería del Software I.*



UNIVERSIDAD TÉCNICA DE AMBATO
FACULTAD DE INGENIERÍA EN SISTEMAS, ELECTRÓNICA E INDUSTRIAL

CARRERA DE SOFTWARE

Cdra. Universitaria (Predios Huachi) / Casilla 334 / Telefax: 03-2851894 – 2411537

AMBATO-ECUADOR



Título: Patrones de Diseño Creacionales

Estudiantes participantes: Fiallos Edison

Lizano Alex

López David

Tiche Kiwar

Asignatura: Patrones de Software

Fecha: 30/04/2024

Carrera: Software

Nivel: Quinto

Introducción

En el ámbito del desarrollo de software, la aplicación efectiva de patrones de diseño juega un papel fundamental en la creación de sistemas robustos y mantenibles. En este contexto, el trabajo en equipo adquiere una importancia crucial, ya que permite aprovechar al máximo la experiencia y habilidades de cada miembro para alcanzar los objetivos del proyecto. En este informe, exploramos una experiencia de colaboración donde se llevó a cabo la implementación práctica de cuatro patrones de diseño fundamentales: Singleton, Factory, Builder y Prototype. A lo largo de este trabajo en equipo, se documentó meticulosamente cada etapa del proceso de creación de estos patrones, destacando tanto los desafíos enfrentados como las soluciones encontradas. Este estudio no solo revela la importancia de la colaboración en el desarrollo de software, sino también la eficacia de la aplicación de patrones de diseño reconocidos en la construcción de sistemas sólidos y flexibles.



Patrón Creacional Singleton

Objetivo

El objetivo del patrón Singleton es garantizar que una clase tenga solo una instancia y proporcionar un punto de acceso global a esa instancia. Esto se logra restringiendo la creación de nuevas instancias de la clase y proporcionando un método estático que devuelve la única instancia existente.

Descripción

Imaginemos un escenario en el que estamos desarrollando un sistema de gestión para un laboratorio de computadoras en una universidad. El laboratorio cuenta con un conjunto de computadoras que los estudiantes pueden reservar para realizar actividades académicas. Deseamos garantizar que solo exista un único laboratorio en todo el sistema, para evitar inconsistencias en la gestión de las reservas y para optimizar el uso de recursos.

Desarrollo

1. Clase Computadora

- Definimos la clase Computadora con atributos como marca, modelo y estado de uso (enUso).
- Proporcionamos métodos de acceso (getters) para los atributos marca y modelo.
- Definimos un método setEnUso(boolean) para modificar el estado de uso de la computadora.

```
src > main > java > singleton >  Computadora.java >  Computadora
1  package singleton;
2
3  public class Computadora {
4      private String marca;
5      private String modelo;
6      private boolean enUso;
7
8      public Computadora(String marca, String modelo) {
9          this.marca = marca;
10         this.modelo = modelo;
11         this.enUso = false;
12     }
13
14     public String getMarca() {
15         return marca;
16     }
17
18     public String getModelo() {
19         return modelo;
20     }
21
22     public boolean isEnUso() {
23         return enUso;
24     }
25
26     public void setEnUso(boolean enUso) {
27         this.enUso = enUso;
28     }
29
30     @Override
31     public String toString() {
32         return "Computadora{" +
33                 "marca='" + marca + '\'' +
34                 ", modelo='" + modelo + '\'' +
35                 ", enUso=" + enUso +
36                 '}';
37     }
38 }
```



2. Clase LaboratorioComputadoras

- Implementamos el patrón Singleton en esta clase.
- Creamos un constructor privado para evitar la creación de instancias externas.
- Definimos un método estático obtenerInstancia() que devuelve la única instancia del laboratorio. Si la instancia aún no existe, la crea.
- Creamos una lista para almacenar las computadoras disponibles en el laboratorio.
- Proporcionamos métodos para agregar computadoras al laboratorio, reservar una computadora y obtener la lista de computadoras disponibles.

```
src > main > java > singleton > LaboratorioComputadoras.java > LaboratorioComputadoras
1  package singleton;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  public class LaboratorioComputadoras {
7      private static LaboratorioComputadoras instancia;
8
9      private List<Computadora> computadorasDisponibles;
10
11     private LaboratorioComputadoras() {
12         computadorasDisponibles = new ArrayList<>();
13     }
14
15     public static LaboratorioComputadoras obtenerInstancia() {
16         if (instancia == null) {
17             instancia = new LaboratorioComputadoras();
18         }
19         return instancia;
20     }
21
22     public void agregarComputadora(Computadora computadora) {
23         computadorasDisponibles.add(computadora);
24     }
25
26     public Computadora reservarComputadora() {
27         if (!computadorasDisponibles.isEmpty()) {
28             return computadorasDisponibles.remove(index:0);
29         } else {
30             System.out.println("No hay computadoras disponibles en este momento.");
31             return null;
32         }
33     }
34
35     public List<Computadora> getComputadorasDisponibles() {
36         return computadorasDisponibles;
37     }
38
39 }
```

3. Clase Main

- En el método main, creamos una instancia del LaboratorioComputadoras.
- Creamos instancias de Computadora y las agregamos al laboratorio.



- Reservamos una computadora del laboratorio y la marcamos como "en uso".
- Intentamos reservar otra computadora y mostramos el estado del laboratorio.

```
src > main > java > singleton > Main.java > Main > main(String[])
1 package singleton;
2
3 public class Main {
4     Run | Debug
5     public static void main(String[] args) {
6         LaboratorioComputadoras laboratorio = LaboratorioComputadoras.obtenerInstancia();
7
8         Computadora comp1 = new Computadora(marca:"Dell", modelo:"Inspiron");
9         //Computadora comp2 = new Computadora("HP", "Pavilion");
10        //Computadora comp3 = new Computadora("Lenovo", "ThinkPad");
11
12        laboratorio.agregarComputadora(comp1);
13        //laboratorio.agregarComputadora(comp2);
14        //laboratorio.agregarComputadora(comp3);
15
16        Computadora compReservada = laboratorio.reservarComputadora();
17        if (compReservada != null) {
18            //compReservada.setEnUso(true);
19            System.out.println("Se reservó la computadora: " + compReservada);
20        }
21
22        /* Computadora otraCompReservada = laboratorio.reservarComputadora();
23        if (otraCompReservada != null) {
24            System.out.println("Se reservó otra computadora: " + otraCompReservada);
25        }*/
26
27        System.out.println("Estado de las computadoras en el laboratorio:");
28        for (Computadora comp : laboratorio.getComputadorasDisponibles()) {
29            System.out.println(comp);
30        }
31    }
32 }
```

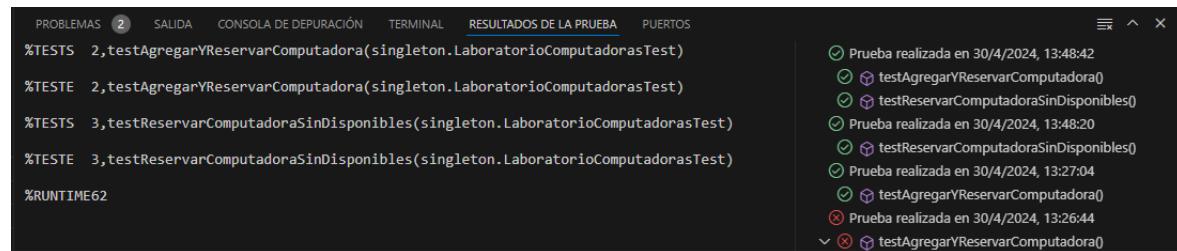
4. Clase LaboratorioComputadorasTest

- Creamos una instancia de LaboratorioComputadoras en el método setUp() antes de cada prueba.
- Definimos pruebas unitarias para verificar el comportamiento de LaboratorioComputadoras al agregar y reservar computadoras.
- En las pruebas, creamos instancias de Computadora, las agregamos al laboratorio, reservamos una o más computadoras y verificamos que el estado del laboratorio sea el esperado después de realizar estas acciones.



```
src > test > java > singleton > LaboratorioComputadorasTest.java > LaboratorioComputadorasTest > testAgregarYReservarComputadora()
1  package singleton;
2
3  import static org.junit.Assert.assertEquals;
4  import static org.junit.Assert.assertNotNull;
5  import static org.junit.Assert.assertNull;
6
7  import org.junit.Before;
8  import org.junit.Test;
9
10 public class LaboratorioComputadorasTest {
11     private LaboratorioComputadoras laboratorio;
12
13     @Before
14     public void setUp() {
15         laboratorio = LaboratorioComputadoras.obtenerInstancia();
16     }
17
18     @Test
19     public void testAgregarYReservarComputadora() {
20         Computadora comp1 = new Computadora(marca:"Dell", modelo:"Inspiron");
21         Computadora comp2 = new Computadora(marca:"HP", modelo:"Pavilion");
22
23         laboratorio.agregarComputadora(comp1);
24         laboratorio.agregarComputadora(comp2);
25
26         assertEquals(expected:2, laboratorio.getComputadorasDisponibles().size());
27
28         Computadora compReservada = laboratorio.reservarComputadora();
29         assertNotNull(compReservada);
30         assertEquals(expected:1, laboratorio.getComputadorasDisponibles().size());
31
32         Computadora otraCompReservada = laboratorio.reservarComputadora();
33         assertNotNull(otraCompReservada);
34         assertEquals(expected:0, laboratorio.getComputadorasDisponibles().size());
35
36         assertNull(laboratorio.reservarComputadora());
37     }
38
39     @Test
40     public void testReservarComputadoraSinDisponibles() {
41         assertNull(laboratorio.reservarComputadora());
42     }
43 }
44
45 }
```

5. Prueba exitosa

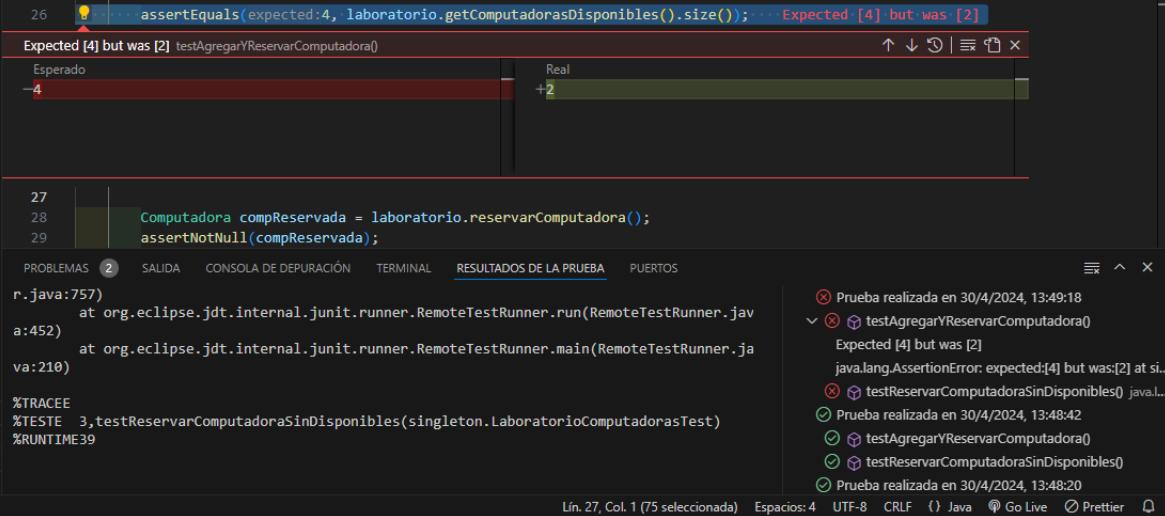


PROBLEMAS 2 SALIDA CONSOLA DE DEPURACIÓN TERMINAL RESULTADOS DE LA PRUEBA PUERTOS

```
%TESTS 2,testAgregarYReservarComputadora(singleton.LaboratorioComputadorasTest)
%TESTE 2,testAgregarYReservarComputadora(singleton.LaboratorioComputadorasTest)
%TESTS 3,testReservarComputadoraSinDisponibles(singleton.LaboratorioComputadorasTest)
%TESTE 3,testReservarComputadoraSinDisponibles(singleton.LaboratorioComputadorasTest)
%RUNTIME62
```

Prueba realizada en 30/4/2024, 13:48:42
 ⚡ ⓘ testAgregarYReservarComputadora()
 ⚡ ⓘ testReservarComputadoraSinDisponibles()
 Prueba realizada en 30/4/2024, 13:48:20
 ⚡ ⓘ testReservarComputadoraSinDisponibles()
 Prueba realizada en 30/4/2024, 13:27:04
 ⚡ ⓘ testAgregarYReservarComputadora()
 ✘ Prueba realizada en 30/4/2024, 13:26:44
 ✘ ⓘ testAgregarYReservarComputadora()

6. Prueba fallida



The screenshot shows the Eclipse IDE interface with the Java perspective. A JUnit test has failed, indicated by a red exclamation mark icon. The error message is "Expected [4] but was [2]" for the method "testAgregarYReservarComputadora". The "Real" result shows a value of "+2". The code being tested is:

```
26 assertEquals(expected:4, laboratorio.getComputadorasDisponibles().size()); ... Expected [4] but was [2]
27
28     Computadora compReservada = laboratorio.reservarComputadora();
29     assertNotNull(compReservada);
```

The "PROBLEMAS" view shows several errors and warnings, including:

- Prueba realizada en 30/4/2024, 13:49:18
- testAgregarYReservarComputadora() - Expected [4] but was [2]
- java.lang.AssertionError: expected:[4] but was:[2] at si...
- Prueba realizada en 30/4/2024, 13:48:42
- testReservarComputadoraSinDisponibles() java...
- Prueba realizada en 30/4/2024, 13:48:20
- testAgregarYReservarComputadora()
- testReservarComputadoraSinDisponibles()
- Prueba realizada en 30/4/2024, 13:48:20

Patrón Creacional Factory

Objetivo:

Implementar un sistema de gestión de la construcción de edificaciones y la producción de unidades o recursos utilizando el patrón Factory.

Descripción:

Supongamos que estás desarrollando un juego de estrategia en tiempo real (RTS) en el que los jugadores pueden construir diferentes tipos de edificaciones, como cuarteles, minas, fábricas, etc. Cada tipo de edificación produce un tipo específico de unidad o recurso.

Desarrollo:

1. Crear la interfaz de Edificios con valores que siempre debe tener un edificio.

Primero debemos crear una interfaz con los métodos a usar.

```
package com.Kiwar.Tiche.Interfaz.Factory;

public interface BuildingInterfaz {
    BuildingInterfaz build(String name, String owner, String location);
    String showBuilding();
}
```

2. Creamos una class enum que contenga los tipos de Edificios.

Creamos un enum con los tipos de Edificios que vamos a tener para facilitar la implementación.

```
package com.Kiwar.Tiche.Implement.Building;

public enum TypeBuilding {
    Barracks,Mine,Factory;
}
```

3. Creamos las clases e implementamos la interfaz.

Creamos las clases definidas en enum para poder implementar la interfaz y poder usar los métodos definidos en la interfaz posteriormente generaremos setters y getters para los atributos que son propios de la clase.



```
package com.Kiwar.Tiche.Implement.Building;

import java.util.HashMap;

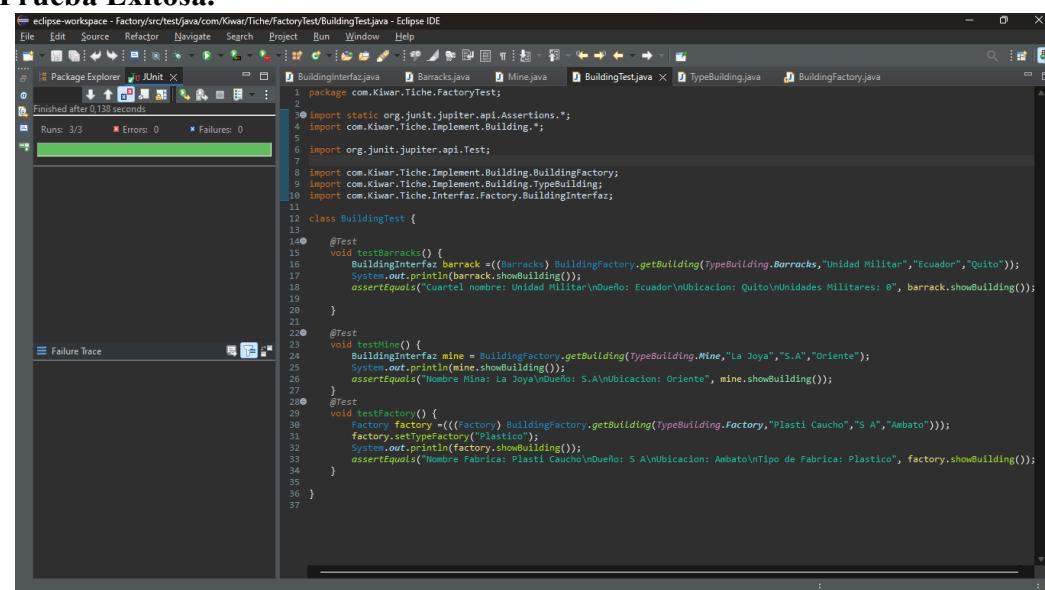
public class BuildingFactory {
    private static final Map<TypeBuilding, BuildingInterfaz> typeBuildings = new HashMap<TypeBuilding, BuildingInterfaz>();
    {
        put(TypeBuilding.Barracks, new Barracks());
        put(TypeBuilding.Mine, new Mine());
        put(TypeBuilding.Factory, new Factory());
    }
    public static BuildingInterfaz getBuilding(TypeBuilding typeBuilding, String name, String owner, String location) {
        return typeBuildings.get(typeBuilding).build(name, owner, location);
    }
}
```

5. Realizamos Pruebas con las salidas de showBuilding().

Realizamos las pruebas que creamos convenientes con el assertEquals para verificarlo con la salida esperada y lo que muestra el .showBuilding().

```
1 package com.Kiwar.Tiche.FactoryTest;
2
3 import static org.junit.jupiter.api.Assertions.*;
4 import com.Kiwar.Tiche.Implement.Building.*;
5
6 import org.junit.jupiter.api.Test;
7
8 import com.Kiwar.Tiche.Implement.Building.BuildingFactory;
9 import com.Kiwar.Tiche.Implement.Building.TypeBuilding;
10 import com.Kiwar.Tiche.Interfaz.Factory.BuildingInterfaz;
11
12 class BuildingTest {
13
14     @Test
15     void testBarracks() {
16         BuildingInterfaz barrack =((Barracks) BuildingFactory.getBuilding(TypeBuilding.Barracks,"Unidad Militar","Ecuador","Quito"));
17         System.out.println(barrack.showBuilding());
18         assertEquals("Cuartel nombre: Unidad Militar\nDueño: Ecuador\nUbicacion: Quito\nUnidades Militares: 0", barrack.showBuilding());
19     }
20
21     @Test
22     void testMine() {
23         BuildingInterfaz mine = BuildingFactory.getBuilding(TypeBuilding.Mine,"La Joya","S.A","Oriente");
24         System.out.println(mine.showBuilding());
25         assertEquals("Nombre Mina: La Joya\nDueño: S.A\nUbicacion: Oriente", mine.showBuilding());
26     }
27     @Test
28     void testFactor() {
29         Factory factory =((Factory) BuildingFactory.getBuilding(TypeBuilding.Factory,"Plasti Caucho","S A","Ambato"));
30         factory.setTypeFactory("Plastico");
31         System.out.println(factory.showBuilding());
32         assertEquals("Nombre Fabrica: Plasti Caucho\nDueño: S A\nUbicacion: Ambato\nTipo de Fabrica: Plastico", factory.showBuilding());
33     }
34
35
36 }
```

Prueba Exitosa.





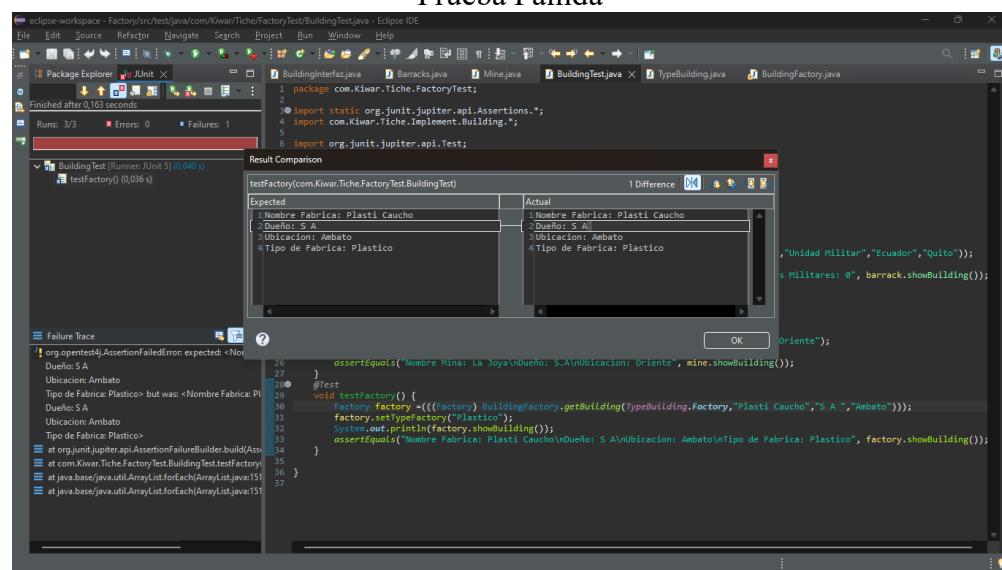
UNIVERSIDAD TÉCNICA DE AMBATO
FACULTAD DE INGENIERÍA EN SISTEMAS, ELECTRÓNICA E INDUSTRIAL
CARRERA DE SOFTWARE

Cdra. Universitaria (Predios Huachi) / Casilla 334 / Telefax: 03-2851894 – 2411537

AMBATO-ECUADOR



Prueba Fallida



The screenshot shows the Eclipse IDE interface with a JUnit test runner window titled "BuildingTest [Runner: JUnit 5] (0.040 s)". The status bar indicates "Finished after 0.163 seconds" with 3 runs, 0 errors, and 1 failure. The "Result Comparison" dialog is open, comparing the "Expected" and "Actual" outputs of a test method. The "Expected" output is:

```
Nombre Fabrica: Plasti Caucó
Dueño: S.A.
Ubicacion: Ambato
Tipo de Fabrica: Plastico
```

The "Actual" output is:

```
Nombre Fabrica: Plasti Caucó
Dueño: S.A.
Ubicacion: Ambato
Tipo de Fabrica: Plastico
,"Unidad Militar","Ecuador","Quito"));
s Militares: 0", barrack.showBuilding());
```

The "Failure Trace" panel shows the stack trace for the assertion failure:

```
org.opentest4j.AssertionFailedError expected: <None>
<None>
  at org.junit.jupiter.api.Assertions.assertequals(Assertions.java:100)
  at org.junit.jupiter.api.Assertions.assertEquals(Assertions.java:114)
  at com.Kiwar.Tiche.FactoryTest.BuildingTest.testFactory(BuildingTest.java:26)
  at org.junit.jupiter.api.TestFunctionBuilder.build(TestFunctionBuilder.java:151)
  at java.base/java.util.ArrayList.forEach(ArrayList.java:151)
  at java.base/java.util.ArrayList.forEach(ArrayList.java:151)
```



Patrón Creacional Builder

Objetivo

Implementar el patrón Builder en Java para la construcción de objetos complejos de una manera flexible, legible y mantenible.

Descripción

Suponiendo que trabajas en una compañía de tecnología que se encarga de fabricar y ensamblar portátiles, puedes deducir que tendrás que realizar dicho proceso para una gran cantidad de portátiles, mismos que serán distintos entre sí. En este caso es necesario implementar el patrón Builder para poder fabricar los portátiles de una manera eficiente.

Desarrollo

1. Definir la clase Computer:

Se crea la clase Computer, a partir de la cual de creará el objeto Computer, por lo mismo es necesario establecer en esta clase propiedades como procesador (processor), tarjeta gráfica (graphicCard), RAM (ram) y almacenamiento (storage).

También se implementa el constructor de la clase, y los getter y setter para las propiedades.

```
public class Computer {  
    private String processor;  
    private String graphicCard;  
    private int ram;  
    private int storage;  
  
    public Computer(){  
    }  
  
    public String getProcessor() {  
        return processor;  
    }  
    public void setProcessor(String processor) {  
        this.processor = processor;  
    }  
    public String getGraphicCard() {  
        return graphicCard;  
    }  
    public void setGraphicCard(String graphicCard) {  
        this.graphicCard = graphicCard;  
    }  
    public int getRam() {  
        return ram;  
    }  
    public void setRam(int ram) {  
        this.ram = ram;  
    }  
    public int getStorage() {  
        return storage;  
    }  
    public void setStorage(int storage) {  
        this.storage = storage;  
    }  
}
```



2. Crear la clase estática ComputerBuilder:

Es necesario crear una clase estática (ComputerBuilder) para implementar el builder dentro de la clase Computer. En la misma se establece los mismos atributos que la clase Computer

```
public static class ComputerBuilder{  
    private String processor;  
    private String graphicCard;  
    private int ram;  
    private int storage;
```

3. Crear la interfaz ComputerInterface:

Se define la interfaz ComputerInterface con los métodos necesarios para establecer los diferentes atributos de la clase Computer, adicionalmente se crea el método build que servirá para devolver un objeto Computer

```
public interface ComputerInterface {  
    public ComputerBuilder setProcessor(String processor);  
    public ComputerBuilder setGraphicCard(String graphicCard);  
    public ComputerBuilder setRam(int ram);  
    public ComputerBuilder setStorage(int storage);  
    public Computer build();  
}
```

4. Creamos un constructor:

En la clase Computer, es necesario implementar un constructor que reciba como parámetro un objeto ComputerBuild para asignar los atributos del objeto Computer.

```
public class Computer {  
    private String processor;  
    private String graphicCard;  
    private int ram;  
    private int storage;  
  
    public Computer(){  
    }  
  
    private Computer(ComputerBuilder builder) {  
        this.processor = builder.processor;  
        this.graphicCard = builder.graphicCard;  
        this.ram = builder.ram;  
        this.storage = builder.storage;  
    }  
}
```

5. Implementar la interfaz en la clase estática:

Para implementar el Builder, es necesario que la clase estática implemente la interfaz ComputerInterface. En la misma, se definirán los métodos para establecer las características de dicha clase. Finalmente, en el método build se construirá un nuevo objeto Computer que tome ComputerBuilder como parámetro, el mismo que será retornado por el mismo método build.



```
public static class ComputerBuilder implements ComputerInterface{
    private String processor;
    private String graphicCard;
    private int ram;
    private int storage;

    public ComputerBuilder setProcessor(String processor) {
        this.processor = processor;
        return this;
    }

    public ComputerBuilder setGraphicCard(String graphicCard) {
        this.graphicCard = graphicCard;
        return this;
    }

    public ComputerBuilder setRam(int ram) {
        this.ram = ram;
        return this;
    }

    public ComputerBuilder setStorage(int storage) {
        this.storage = storage;
        return this;
    }

    public Computer build() {
        return new Computer(this);
    }
}
```

6. Probar el Patrón Builder

Creamos un objeto Computer en el main y le asignamos los atributos a través de la clase ComputerBuilder llamando a cada uno de los métodos establecidos en el flujo, incluyendo al final el método build, que será el que permitirá crear el objeto Computer y lo retornará a la instancia. Finalmente, se accede al objeto computer construido y se muestran sus atributos para comprobar que se haya creado correctamente.

```
public class App
{
    Run | Debug
    public static void main( String[] args )
    {
        Computer computer = new Computer.ComputerBuilder()
            .setProcessor(processor:"AMD Ryzen")
            .setGraphicCard(graphicCard:"AMD Radeon Graphics")
            .setRam(ram:12)
            .setStorage(storage:500)
            .build();

        System.out.println("Procesador: " + computer.getProcessor());
        System.out.println("Tarjeta gráfica: " + computer.getGraphicCard());
        System.out.println("RAM: " + computer.getRam() + " GB");
        System.out.println("Almacenamiento: " + computer.getStorage() + " GB");
    }
}
```

Procesador: AMD Ryzen
Tarjeta gráfica: AMD Radeon Graphics
RAM: 12 GB
Almacenamiento: 500 GB



Patrón Creacional Prototype

Objetivo:

Implementar el patrón de diseño Prototype en un proyecto de juegos de roles para agilizar la creación de nuevos personajes, permitiendo la clonación dinámica de objetos con características predefinidas.

Descripción

Este proyecto se centra en el desarrollo de un juego de roles donde se ha implementado el patrón de diseño Prototype. Se han creado las clases base y derivadas necesarias para representar diferentes tipos de personajes, como guerreros y magos, que heredan de una clase principal llamada Personaje.

El patrón Prototype permite la creación de nuevos personajes de manera eficiente al clonar objetos existentes con características predefinidas. Esto significa que, en lugar de crear cada personaje desde cero, se pueden duplicar y modificar personajes existentes según las necesidades del juego.

Desarrollo

1. Creamos una interfaz llamada PrototipoPersonaje.

Esta interfaz define un método **clonar()** que devuelve un objeto del tipo PrototipoPersonaje. La interfaz extiende la interfaz **Cloneable**, lo que indica que las clases que la implementen podrán utilizar el método **clone()** para crear copias de sí mismas.

```
1 package com.fiallos.edison.Interfaz;
2
3 public interface PrototipoPersonaje extends Cloneable{
4
5     PrototipoPersonaje clonar();
6 }
7
```

2. Creación de la Clase Personaje:

Se crea una clase llamada **Personaje**, que implementa la interfaz **PrototipoPersonaje**.

La clase define atributos para el nombre del personaje y sus estadísticas como **fuerza**, **agilidad** e **inteligencia**.

Se proporcionan dos constructores: uno sin argumentos y otro que permite inicializar los atributos del personaje.

Se implementa el método **clonar()** de la interfaz **PrototipoPersonaje**, utilizando el método **clone()** de Java para realizar una clonación superficial del objeto.

Se proporcionan métodos **getter** y **setter** para acceder y modificar los atributos del personaje.

Se sobrescribe el método **toString()** para proporcionar una representación en forma de cadena del objeto Personaje, que incluye su nombre y estadísticas.



```
1 package com.fiallos.edison.modelo;
2
3 import com.fiallos.edison.Interfaz.PrototipoPersonaje;
4
5 public class Personaje implements PrototipoPersonaje{
6
7     protected String nombre;
8     protected int fuerza;
9     protected int agilidad;
10    protected int inteligencia;
11
12    public Personaje() {
13        super();
14    }
15
16    public Personaje(String nombre, int fuerza, int agilidad, int inteligencia) {
17        super();
18        this.nombre = nombre;
19        this.fuerza = fuerza;
20        this.agilidad = agilidad;
21        this.inteligencia = inteligencia;
22    }
23
24    @Override
25    public PrototipoPersonaje clonar() {
26        try {
27            return (PrototipoPersonaje) super.clone();
28        } catch (CloneNotSupportedException e) {
29
30            return null;
31        }
32    }
33
34
35    public String getNombre() {
36        return nombre;
37    }
38
39    public void setNombre(String nombre) {
40        this.nombre = nombre;
41    }
42
43    public int getFuerza() {
44        return fuerza;
45    }
46
47    public void setFuerza(int fuerza) {
48        this.fuerza = fuerza;
49    }
50
51    public int getAgilidad() {
52        return agilidad;
53    }
54
55    public void setAgilidad(int agilidad) {
56        this.agilidad = agilidad;
57    }
58
59    public int getInteligencia() {
60        return inteligencia;
61    }
62
63    public void setInteligencia(int inteligencia) {
64        this.inteligencia = inteligencia;
65    }
66
67    @Override
68    public String toString() {
69        return "Personaje [nombre=" + nombre + ", fuerza=" + fuerza + ", agilidad=" + agilidad + ", "
70                + inteligencia + "]";
71    }
72 }
```

3. Creación de la Clase Guerrero:



Se crea una clase llamada **Guerrero**, que extiende la clase **Personaje**.

La clase define un atributo adicional específico para los guerreros, que es el daño que pueden infiligrir en combate.

Se proporcionan dos constructores: uno sin argumentos y otro que permite inicializar tanto los atributos heredados de la clase Personaje como el atributo específico de los guerreros.

Se proporcionan métodos getter y setter para acceder y modificar el atributo de daño.

Se sobrescribe el método **toString()** para proporcionar una representación en forma de cadena del objeto **Guerrero**, que incluye su atributo de daño junto con las estadísticas heredadas de la clase **Personaje**.

```
1 package com.fiallos.edison.modelo;
2
3 public class Guerrero extends Personaje{
4
5     private int daño;
6
7     public Guerrero() {
8         super();
9         // TODO Auto-generated constructor stub
10    }
11
12     public Guerrero(String nombre, int fuerza, int agilidad, int inteligencia, int daño) {
13         super(nombre, fuerza, agilidad, inteligencia);
14         this.daño = daño;
15    }
16
17     public int getDaño() {
18         return daño;
19    }
20
21     public void setDaño(int daño) {
22         this.daño = daño;
23    }
24
25     @Override
26     public String toString() {
27         return "Guerrero [daño=" + daño + "] "+super.toString();
28     }
29
30
31
32 }
33
```

4. Creación de la Clase Mago:

Se crea una clase llamada **Mago**, que extiende la clase Personaje.

La clase define un atributo adicional específico para los magos, que es el maná que poseen para lanzar hechizos.

Se proporcionan dos constructores: uno sin argumentos y otro que permite inicializar tanto los atributos heredados de la clase **Personaje** como el atributo específico de los magos.

Se proporcionan métodos **getter** y **setter** para acceder y modificar el atributo de maná.

Se sobrescribe el método **toString()** para proporcionar una representación en forma de cadena del objeto **Mago**, que incluye su atributo de maná junto con las estadísticas heredadas de la clase **Personaje**.



```
1 package com.fiallos.edison.modelo;
2
3 public class Mago extends Personaje{
4
5     private int mana;
6
7     public Mago() {
8         super();
9         // TODO Auto-generated constructor stub
10    }
11
12    public Mago(String nombre, int fuerza, int agilidad, int inteligencia,int mana) {
13        super(nombre, fuerza, agilidad, inteligencia);
14        this.mana = mana;
15    }
16
17    public int getMana() {
18        return mana;
19    }
20
21    public void setMana(int mana) {
22        this.mana = mana;
23    }
24
25    @Override
26    public String toString() {
27        return "Mago [mana=" + mana + "] "+super.toString();
28    }
29
30 }
```

5. Creación de la Clase GeneradorPersonajes:

Se crea una clase llamada **GeneradorPersonajes**, que se encarga de generar y almacenar prototipos de personajes.

La clase utiliza un mapa (**Map**) para almacenar los prototipos de personajes, donde la clave (**String**) representa el tipo de personaje y el valor (**PrototipoPersonaje**) representa el prototipo correspondiente.

Se proporciona un método llamado **adicinarPrototipo()** para agregar un nuevo prototipo al mapa, tomando una clave y un objeto **PrototipoPersonaje** como argumentos.

Se proporciona un método estático llamado **obtenerPrototipo()** para obtener un prototipo de personaje del mapa utilizando una clave dada. Este método clona el prototipo utilizando el método **clonar()** de la interfaz **PrototipoPersonaje** antes de devolverlo.

```
1 package com.fiallos.edison.modelo;
2
3 import java.util.HashMap;
4
5 public class GeneradorPersonajes {
6
7     private static Map<String, PrototipoPersonaje> prototipos = new HashMap<>();
8
9     public void adicinarPrototipo(String llave, PrototipoPersonaje personaje) {
10        prototipos.put(llave, personaje);
11    }
12
13    public static PrototipoPersonaje obtenerPrototipo(String llave) {
14        return (PrototipoPersonaje) prototipos.get(llave).clonar();
15    }
16
17 }
```

La clase **GeneradorPersonajes** actúa como un repositorio centralizado de prototipos de personajes en el juego.



Permite registrar y almacenar diferentes tipos de prototipos de personajes, facilitando su acceso y uso en otras partes del sistema.

7. Puesta a prueba del Patrón Prototype

La clase App es la clase principal que contiene el método main, donde se lleva a cabo la ejecución del programa. En este método, se crean instancias de GeneradorPersonajes y diferentes tipos de personajes (guerreros y magos) para demostrar el uso del patrón Prototype.

```
package com.fiallos.edison.RolPrototype;
import com.fiallos.edison.modelo.GeneradorPersonajes;
public class App {
    public static void main(final String[] args) {
        GeneradorPersonajes gp = new GeneradorPersonajes();
        Guerrero guerreroOriginal = new Guerrero("Guerrero", 60, 87, 30, 175);
        gp.adicionarPrototipo("guerrero", guerreroOriginal);
        System.out.println("Guerrero inicial: "+guerreroOriginal);
        Guerrero guerreroClon = (Guerrero)gp.obtenerPrototipo("guerrero");
        guerreroClon.setDaño(90);
        System.out.println("Guerrero Clon: "+guerreroClon);
        System.out.println();
        Mago magoOriginal = new Mago("Mago", 45, 60, 88, 180);
        gp.adicionarPrototipo("mago", magoOriginal);
        System.out.println("Mago Inicial: "+magoOriginal);
        Mago magoClon = (Mago)gp.obtenerPrototipo("mago");
        magoClon.setInteligencia(94);
        System.out.println("Mago Clon: "+magoClon);
    }
}
```

8. Salida por consola

```
Problems @ Javadoc Declaration Console Progress
<terminated> App (4) [Java Application] E:\Programas\Eclipse\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.10.v20240120-1143\jre\b
Guerrero inicial: [daño=175] [nombre=Guerrero, fuerza=60, agilidad=87, inteligencia=30]
Guerrero Clon: [daño=90] [nombre=Guerrero, fuerza=60, agilidad=87, inteligencia=30]

Mago Inicial: [mana=180] [nombre=Mago, fuerza=45, agilidad=60, inteligencia=88]
Mago Clon:[mana=180] [nombre=Mago, fuerza=45, agilidad=60, inteligencia=94]
```

Como podemos apreciar La capacidad de clonar prototipos de guerreros y magos, registrados en un generador de personajes, permite la creación de nuevas instancias con atributos predefinidos, mientras que las modificaciones posteriores en los clones demuestran la independencia de estos respecto a los prototipos originales.



UNIVERSIDAD TÉCNICA DE AMBATO
FACULTAD DE INGENIERÍA EN SISTEMAS, ELECTRÓNICA E INDUSTRIAL

CARRERA DE SOFTWARE

Cdra. Universitaria (Predios Huachi) / Casilla 334 / Telefax: 03-2851894 – 2411537

AMBATO-ECUADOR



Conclusiones

La implementación práctica de los patrones de diseño Singleton, Factory, Builder y Prototype en un proyecto de trabajo en equipo, respaldada por una documentación exhaustiva de cada paso, ha resultado en un código modular, escalable y fácilmente mantenable. Estos patrones han permitido una gestión eficiente de recursos, una creación flexible de objetos y una reducción significativa de la duplicación de código. La documentación detallada no solo facilita el entendimiento y la colaboración dentro del equipo actual, sino que también establece una base sólida para futuras iteraciones del proyecto, promoviendo así el desarrollo continuo y exitoso del software.