TRABAJO PRÁCTICO GRUPAL:

SISTEMA DE GESTION DE BÚSQUEDAS LABORALES

INTEGRANTES:

Acuña Destrée Joaquín

Jeronimo Jimenez

FECHA DE ENTREGA:

Domingo 8/5/2022

Descripción de patrones de diseño

Para la realización del trabajo implementamos los siguientes patrones vistos en clase:

- ***Patron de diseño Singleton:***

Este patrón consiste en la restricción de la creación de objetos de una clase, permitiendo que sólo se pueda crear una única instancia de la clase, dándole un acceso global en el programa.

En el trabajo, usamos este patrón para instanciar un único Sistema, que es quien lleva a cabo la mayor parte de los necesarios para el funcionamiento del programa.

```
public class Sistema  {

    private static Sistema sistema=null;

    private Sistema() {

    }

    public static Sistema getSistema() {
        if(sistema==null)
            sistema= new Sistema();
        return sistema;
    }
```

- ***Patrón de diseño Factory:***

Este patrón nos permite, en un método static crear un objeto, y devolverlo sin saber de qué tipo es el objeto tratado.

En el trabajo lo utilizamos para la creación de empleados, empleadores y usuarios de la agencia ue se extienden de Usuario como vemos en el diagrama UML.

```
public class UsuarioFactory {

    public static Usuario getUsuario(String username, String password,String tipo) {
        Usuario respuesta=null;
        if(tipo.equals("Agencia"))
            respuesta=new Agencia(username,password);
        else if(tipo.equals("Empleador"))
            respuesta=new Empleador(username,password);
        else if(tipo.equals("Empleado"))
            respuesta=new Empleado(username,password);
        return respuesta;
    }

}
```

- ***Patrón de diseño Decorator***

Este patrón sirve para cuando se quiere “decorar” un objeto agregándole atributos o cambiándole el comportamiento de los métodos.

Usamos este método en la creación de los elementos del formulario de búsqueda, decorando cada uno de estos elementos agregándole un atributo peso.

//foto

- ***Patrón de diseño Double Dispatch***

Este es un patrón de doble envío. Se usa cuando se tiene un método que recibe como parámetro un objeto y depende de este objeto es lo que debería hacer.

Usamos este patrón para el cálculo de las coincidencias laborales. Por ejemplo, queriendo ver la compatibilidad de una locación. Si invocamos el método versus de la clase HomeOffice pasando una locación Presencial, el objeto home office desconoce qué objeto se le paso, pero sabe su clase, entonces el método versus invoca el método VersusHomeOffice de la locación presencial y esta proporciona el resultado.

```java
public class HomeOffice extends Locacion {

    public HomeOffice() {
        super();
    }

    @Override
    public double versus(Locacion loc) {
        return loc.versusHomeOfice();
    }
    @Override
    public double versusHomeOfice() {
        return 1;
    }
    @Override
    public double versusPresencial() {
        return -1;
    }
    @Override
    public double versusIndistinto() {
        return 1;
    }
```

```java
public class Presencial extends Locacion {

    public Presencial() {
        super();
    }

    @Override
    public double versus(Locacion loc) {
        return loc.versusPresencial();
    }
    @Override
    public double versusHomeOfice() {
        return -1;
    }
    @Override
    public double versusPresencial() {
        return 1;
    }
    @Override
    public double versusIndistinto() {
        return 1;
    }
```

Descripción de métodos mas significativos

```java
public void iniciarRondaEncuentros() {
    for (Empleador empleador : empleadores) {
        for (TicketEmpleador ticketEmpleador : empleador.getTicketsEmitidos()) {
                for (Empleado empleado : empleados) {
                if (ticketEmpleador.getEstado() == Ticket.ESTADO_ACTIVO
                        && empleado.getTicket().getEstado() == Ticket.ESTADO_ACTIVO) {
                    Entrevista entrevista = new Entrevista(empleador, empleado, ticketEmpleador, empleado.getTicket());
                    entrevista.setCompatibilidad(compatibilidad(empleado.getTicket(), ticketEmpleador));
                    entrevistas.add(entrevista);
                    entrevistas.sort((e1, e2) -> e1.getCompatibilidad() > e2.getCompatibilidad() ? -1 : 1);
                }
            }
        }
    }
}
```

IniciarRondaEncuentros() metodo de la clase Agencia lo ue hace es tomar cada empleador de la lista de empleadores de la agencia. Para cada empleador se toma cada uno de sus tickets. Despues si verifican ue los estados de este ticket y el ticket del empleado esten activos para generar una nueva entrevista, se llama a la funcion compatibilidad y se agrega esta entrevista a la lista entrevistas.

```java
private double compatibilidad(TicketEmpleado empleado, TicketEmpleador ticketEmpleador) {
    return
        ticketEmpleador.getFormulario().getLocacion().versus(empleado.getFormulario().getLocacion())
        + ticketEmpleador.getFormulario().getCargaHoraria().versus(empleado.getFormulario().getCargaHoraria())
        + ticketEmpleador.getFormulario().getEleccionEdad().versus(empleado.getFormulario().getEleccionEdad())
        + ticketEmpleador.getFormulario().getEstudios().versus(empleado.getFormulario().getEstudios())
        + ticketEmpleador.getFormulario().getExperiencia().versus(empleado.getFormulario().getExperiencia())
        + ticketEmpleador.getFormulario().getRemuneracion().versus(empleado.getFormulario().getRemuneracion())
        + ticketEmpleador.getFormulario().getTipoPuesto().versus(empleado.getFormulario().getTipoPuesto());
}
```

Compatibilidad(TicketEmpleado empleado, TicketEmpleador ticketEmpleador) metodo de la clase agencia ue devuelve un double ue es la suma de cada llamando de cada aspecto del ticketEmpleador al metodo versus(con parametro el mismo aspecto pero del ticket empleado) esto se resuelven con el patron double dispatch mostrado mas arriba.