Programación 3

TP GRUPAL

Sistema De Gestión de Búsquedas Laborales

Integrantes:

- **Boolls, Nicolás**
- **Vega Imbalde, Aureliano**
- **Vicente, Paola**

Índice

Breve Descripción …… 2
Aclaraciones …… 2
Patrones de diseño utilizados …… 2-3
Arquitectura de 3 capas …… 3-4
Implementación de la concurrencia …… 4-8
Implementación del patrón Decorator …… 8

Breve Descripción:
El presente trabajo describe las cuestiones más importantes del Sistema de Gestión de Búsquedas Laborales. También se incluye el diagrama de clases, y se explican los patrones de diseño utilizados. Además, se agregó un apartado de “aclaraciones” donde mencionamos las particularidades que establecimos en nuestro Sistema.

Aclaraciones:

- Corregimos las incoherencias en los valores resultantes de las matrices del cálculo de coincidencias laborales.

- La remuneración puede ser de $40.000, $80.000 ó $120.000.

Patrones de diseño utilizados:

- *Singleton*: patrón de diseño creacional que restringe la creación de objetos de una clase a una sola instancia, pudiendo ser accedida de forma global. Es muy útil de utilizar en casos en donde una clase solo puede tener un objeto. Por ejemplo, solo puede haber un planeta Tierra.

  En este proyecto se utilizó para la clase ‘Sistema’ y en BolsaDeTrabajo.

- *Factory*: patrón de diseño creacional que permite crear un objeto de una clase sin que el cliente sepa la lógica de creación. Este, además, permite poder crear un objeto al momento de la ejecución, sin necesidad de hacerlo antes.

  En este proyecto se utilizó para el registro del Empleador y para crear el formulario de búsqueda (se especificó el tipo de cada uno de sus atributos).

- *Double Dispatch*: resuelve el conflicto que sucede cuando un comportamiento no solo depende del tipo del objeto que recibe el mensaje sino también del del que se pasa como parámetro. Es decir, se utiliza cuando el objeto se comporta de una forma distinta dependiendo de sus subclases pero también dependiendo de la subclase del objeto con el que interactúa.

  En este proyectó se utilizó principalmente para el cálculo de coincidencias laborales.

- *Decorator*: patrón de diseño estructural que permite añadir funcionalidades a un objeto de forma dinámica. De esta manera se evita crear un exceso de clases hijas de una clase padre.

En este proyecto se utilizó para el atributo ‘’persona’’ de la clase Empleador. Permitiendo así ‘’decorar’’ a persona (ya sea física o jurídica) para que pueda ser en salud, en comercio local o en comercio internacional. De esta manera, lo que en un principio podrían ser seis clases, con este patrón son tres.

- *Observer-Observable*: patrón de diseño de comportamiento utilizado para observar los cambios de comportamiento en un objeto y, a su vez, notificarlos. Esto se genera a través de dos objetos: el observado y el observador. Como sus nombres lo indican, el primero es al que se analizan sus cambios, y el segundo es el que observa estos cambios y actúa en cuanto a ellos.

  En este proyecto se utilizó para observar los eventos ocurridos en el estado del Ticket Simplificado cuando se efectúa la concurrencia.

- *State:* patrón de diseño de comportamiento utilizado cuando el comportamiento de un objeto cambia dependiendo de su estado. Por ejemplo, en un estado puede que tenga la capacidad de realizar una acción que, en otro estado, no.

  En este proyecto se utilizó para indicar los cambios de estado de los tickets (solo en TicketBuscaEmpleo y TicketBuscaEmpleador, ya que ambos son subclases de Ticket). Estos pueden tener cuatro estados: activo, suspendido, finalizado o cancelado; y dependiendo de estos es cómo se comportarán los tickets.

- *DAO-DTO:* sirve para transferir información entre un cliente y un servidor. Esto lo hace a través de la creación de objetos planos (DTO) con una serie de atributos del objeto que se quiere persistir (DAO) que puedan ser enviados o recuperados del servidor, sin modificar la clase del objeto a persistir.

  En este proyecto se implementó en el objeto 'Objeto' con el fin de persistir los datos necesarios de la clase sistema debido al Singleton utilizado en la misma.

El patrón Factory y el patrón Double Dispatch fueron utilizados de forma conjunta para el cálculo de coincidencias laborales.

Aspectos a resaltar:

- *Arquitectura en tres capas*:

  En este proyecto, aplicamos el modelo de arquitectura en tres capas. Aplicamos este modelo para poder separar cada aspecto del programa en un grupo y así hacer más fácil su implementación. Además permite poder trabajar de forma simultánea en cada capa sin verse modificadas por el cambio de las otras. Asimismo, para poder aplicar el MVC, es crucial la separación en capas.
  Las capas son las siguientes:

- Presentación: es el contacto con el usuario; este solo tiene acceso a esta capa. Acá solo se piden y se muestran datos y se envían mensajes al usuario.

- Negocio: acá se realizan las operaciones más complejas. Es el punto de encuentro entre los datos de las diferentes entidades de la capa de modelo. Básicamente, tomando los datos necesarios, se realizan los cálculos para darle información al usuario en la capa de presentación.

- Modelo: es donde se encuentran todas las entidades. Acá están las clases con sus atributos. Pueden tener métodos propios pero no muy complejos y deben ser internos a cada clase. Es decir, no interactúan con otras.

La comunicación entre capas es lineal. La capa de presentación solo le puede pedir datos a la capa de negocio. La capa de negocio solo le puede pedir datos a la capa de modelo. Es decir, una capa solo puede pedir datos a la capa de abajo y solo puede enviar datos a la capa de arriba.

En nuestro proyecto pusimos, dentro de la capa de presentación, a las ventanas y los controladores; dentro de la capa de negocio, al Sistema y al main de la concurrencia; y dentro de la capa de modelo, a todas entidades del programa (empleados, empleadores, administradores, tickets, formularios, listas, etc).

- *Implementación de la concurrencia*:

El ticket simplificado tiene 4 atributos. 3 de tipo String (tipoDeTrabajo, locacion y estado) y un empleador de tipo Empleador. Estos tickets siempre se inicializan en "disponible".

Además tiene el método cambiaEstado que llama a los métodos setChanged y notifyObservers de la clase Observable.

```
public void cambiaEstado(String estado)
{
    this.setChanged();
    this.notifyObservers(estado);
}
```

El run del Empleado es de la siguiente manera:

```java
public void run()
{
    int i = 0;
    while (i < 10 && this.ticketSimplificado == null)
    {
        TicketSimplificado ticket;
        String tipoDeTrabajo = null;
        Random r = new Random();
        int num = r.nextInt(3);
        switch (num)
        {
        case 0:
            tipoDeTrabajo = "RubroComercioInternacional";
            break;
        case 1:
            tipoDeTrabajo = "RubroComercioLocal";
            break;
        case 2:
            tipoDeTrabajo = "RubroSalud";
            break;
        }
        ticket = BolsaDeTrabajo.getInstancia().buscaEmpleo(this.getNombre(), tipoDeTrabajo);
        if (ticket != null)
        {
            Util.espera(3000);
            String locEmpl = this.getTicket().getFormulario().getLocacion().diceTipo();
            if (locEmpl.equals(ticket.getLocacion()) || locEmpl.equals("Indistinto")
                    || ticket.getLocacion().equals("Indistinto"))
            {
                this.ticketSimplificado = ticket;
                BolsaDeTrabajo.getInstancia().noDevuelveTicket(this.getNombre(), ticket);
            } else
                BolsaDeTrabajo.getInstancia().devuelveTicket(this.getNombre(), ticket);

        }
        i++;
    }
}
```

Primero selecciona el tipo de trabajo de forma aleatoria, luego busca un ticket que sea compatible con ese tipo de trabajo (explicado más abajo). Si encuentra ticket, es decir, si no es null, hace lo siguiente: espera unos segundos (para simular la concurrencia) y luego verifica que la locación del ticket y la de este empleado sean compatibles. Si lo son, toma el ticket de forma definitiva e indica que no lo devuelve (explicado más abajo). Si no lo son, indica que lo devuelve (explicado más abajo). Este proceso lo hace 10 veces o hasta que se quede de forma definitiva con un ticket.

El run del Empleador es de la siguiente manera:

```
public void run()
{
    Observador obs = new Observador();
    TicketSimplificado ticket=null;
    for(int i = 0; i < 3; i++)
    {
        String locacion=null;
        Random r = new Random();
        int num = r.nextInt(3);
        switch (num)
        {
        case 0:
            locacion = "HomeOffice";
            break;
        case 1:
            locacion = "Presencial";
            break;
        case 2:
            locacion = "Indistinto";
            break;
        }
        ticket = new TicketSimplificado(this.persona.diceRubro(),locacion,this);
        obs.agregarObservable(ticket);
        BolsaDeTrabajo.getInstancia().agregaEmpleo(ticket);
        Util.espera(3000);
    }

}
```

Primero se crea un observador, después se elige un tipo de locación de forma aleatoria y luego agrega un ticket con esa locación y con el rubro del mismo Empleador (explicado más abajo). Además se agrega el ticket a la lista de obervados del obervador. Por último se espera unos milisegundos para simular la concurrencia.

Creamos una clase Singleton llamada BolsaDeTrabajo donde están todos los métodos synchronized. Esta tiene un array (llamado "tickets") con todos los tickets que los empleadores van creando.

El método para agregar un ticket es el siguiente:

```
public synchronized void agregaEmpleo(TicketSimplificado ticket)
{
    this.tickets.add(ticket);
    this.notifyAll();
}
```

El método buscaEmpleo que utiliza el Empleado es el siguiente:
(Si no se ve la imagen, se puede descargar y ampliar, o directamente ver el código)

```
public synchronized TicketSimplificado buscaEmpleo(String nombre, String tipoDeTrabajo)
{
    TicketSimplificado ticket = null;
    int i = 0;
    while(tickets.size() == 0) //si no hay tickets, debe esperar a que los empladores generen
        try
        {
            this.ventana.getTextArea().append("Todavia no hay empleos disponibles.");
            this.wait();
        } catch (InterruptedException e1)
        {
            e1.printStackTrace();
        }
    while (i < tickets.size() && ticket == null)
    {
        if (tickets.get(i).getTipoDeTrabajo().equals(tipoDeTrabajo))
        {
            ticket = tickets.get(i);
            while (ticket.getEstado().equals("en consulta"))
                try
                {
                    this.ventana.getTextArea().append(nombre + " no consulta el ticket de " + ticket.getEmpleador().getNombre()+ " porque ya esta en consulta.\n");
                    this.wait();
                } catch (InterruptedException e)
                {
                    e.printStackTrace();
                }

            if (!ticket.getEstado().equals("no disponible"))
            {
                ticket.cambiaEstado("en consulta"); //esto es para llamar a los metodos de observable
                this.ventana.getTextArea().append(nombre + " esta en proceso de consulta con '" + ticket.getEmpleador().getNombre()+"'.\n");
            } else
                ticket = null;
        }
        i++;
    }
    return ticket;
}
```

Primero entra en un ciclo while que solo se corta si ya se recorrieron todos los tickets y no se consiguió ninguno, o si se consiguió un ticket compatible. Si el ticket es compatible en cuanto al tipo de trabajo, se verifica su estado. Si está en estado de consulta (es decir que otro thread también tuvo coincidencia en cuanto al tipo de trabajo y está verficando si también es compatible en cuanto a la locación), se espera a que deje de estarlo. Una vez que deja de estarlo, se verifica si el ticket sigue disponible. Si lo está, lo toma y lo deja en estado de consulta porque ahora le toca a él verificar la compatibilidad de la locación, y llama a cambiaEstado del ticket (explicado previamente). En cambio, si no está más disponible (es decir, el thread anterior lo tomó de forma definitiva), el ticket queda en null. Si es null, pasa al siguiente ticket y hace todo de nuevo. Si no lo es, se corta el ciclo. El ciclo también se corta si recorre todos los tickets y no compatibiliza con ninguno.

El método devulveTicket es el siguiente:

```
public synchronized void devuelveTicket(String nombre, TicketSimplificado ticket)
{
    System.out.println(nombre + " no consigue trabajo porque su locacion no es compatible con la de '"
            + ticket.getEmpleador().getNombre()+"'.\n");
    ticket.cambiaEstado("disponible"); //esto es para llamar a los metodos de observable
    this.notifyAll();
}
```

Este método se aplica cuando se verifica que las locaciones del ticket y del Empleado NO son compatibles. El estado del ticket vuelva a ser disponible y se hace un notifyAll para indicar este cambio de estado.
El metodo noDevuelveTicket:

```
public synchronized void noDevuelveTicket(String nombre, TicketSimplificado ticket)
{
    System.out.println(nombre + " consigue empleo con '" + ticket.getEmpleador().getNombre() + "'.\n");
    this.tickets.remove(ticket);
    ticket.cambiaEstado("no disponible"); //esto es para llamar a los metodos de observable
    this.notifyAll();
}
```

Este método se aplica cuando se verifica que las locaciones del ticket y del Empleado SI son compatibles. El ticket se elimina del array de tickets de la BolsaDeTrabajo, el estado del ticket pasa a ser NoDisponible y se hace un notifyAll para indicar este cambio de estado.

- *Implementación del patrón Decorator*:

  Dado que la tabla de comisiones a los empleadores no tenía la forma que acostumbramos para aplicar el decorator, la adaptamos de manera tal que tenga sentido usar este patrón.

  Para aplicar el decorator las clases bases llaman a un método y luego las clases que ‘’decoran’’ también llaman a ese método y modifican el resultado obtenido por la clase base. Pero esta modificación es siempre la misma, es decir, es indistinta a la clase base.

  El problema que nos surgió con la tabla indicada en el trabajo, es que el decorator (cualquiera de los tres: salud, comercio local y comercio internacional) no es el mismo para ambos tipos de personas ¿A qué nos referimos con esto? Veamos un ejemplo de un ejercicio resuelto en teoría: el método getArmadura de la clase Mago es 500 y de la clase Elfo es 1000. El método getArmadura del decorator Tierra es +25%. Es decir que, indistintamente de la clase base, cualquier objeto que sea de Tierra, al realizar el método getArmadura, se le sumará un 25%. Entonces, el Mago de Tierra tendrá un 625 (500 + 500*25%) de armadura y el Elfo de Tierra un 1250 (1000 + 1000*25%).

  En cambio, en lo planteado en el trabajo final, el porcentaje de Salud, Comercio Local y Comercio Internacional es DISTINTO para Persona Física que para Persona Jurídica. Por ejemplo, la salud para Persona Física es 60%, pero para Jurídica es 80%. Entonces pareciera que se debería aplicar dos tipos distintos de decorator salud, uno para cada tipo de persona.

  Pero nos dimos cuenta de lo siguiente, la tabla siempre seguía la misma lógica: en persona jurídica, sin importar el rubro, la comisión siempre era un 20% mayor que la de persona física. Entonces, lo que se nos ocurrió fue que el método para calcular la comisión en persona física dé como resultado 0% y para persona jurídica 20%. Luego, los métodos que ‘’decoran’’ tienen cada uno su respectivo resultado a la hora de calcular la comisión: Salud +60%, ComercioLocal +70% y ComercioInternacional +80%. Estos últimos porcentajes son indistintos al tipo de persona. Por lo tanto, se pueden hacer todas las combinaciones indicadas en la tabla. Por ejemplo, una persona jurídica en salud sería el 20% de PersonaJuridica más el 60% de Salud; dando así un 80%.