Programación III

Trabajo Práctico Grupal

INFORME

"Sistema de búsquedas laborales"

-2° Parte-

Grupo n°: 7

Integrantes:

-Avalos, Wenceslao

-Lapiana, Santiago Nicolás

-Luna, Lautaro

-Sosa, Santiago

Fecha de entrega: 26/06/22

Introducción

La aplicación que se pidió desarrollar consiste en un sistema de búsquedas laborales que permite a dos tipos de usuarios: empleados pretensos y empleadores, buscar empleo/empleados. Para ello, se organizan distintas rondas: una de encuentros, una de elecciones y por último una contratación, de las cuales el administrador tiene control.

Tickets de Encuentro Laboral

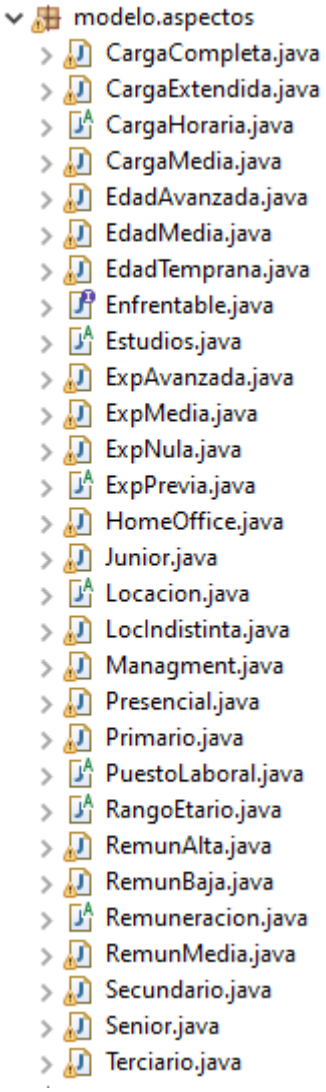
Cada empleado completará un formulario especificando su preferencia frente a distintos aspectos tales como Carga Horaria, Estudios Cursados, Experiencia Previa, Locación, Rango Etario, Remuneración, Tipo de Puesto Laboral. En el caso de los empleadores, completarán un formulario por cada tipo de puesto laboral que ofrecen, especificando la cantidad de vacantes que hay para cada tipo (de esta forma se creará un formulario para cada vacante). Luego, los formularios son pasados al sistema (la agencia), y éste les devuelve, al empleado un ticket de empleo, y al empleador un ticket de empleado con la cantidad de vacantes solicitadas.

Modificaciones respecto a la parte I:

Ahora, para la creación de tickets (tanto de empleado como empleador) llama únicamente al método creaFormulario, el cual llama a emiteFormulario dentro de él. El creaFormulario del empleador tiene los siguientes parámetros (el del Empleador es similar, pero se incluye la cantidad de puestos).

- public void creaFormulario (String locacion, String remuneracion, String cargaHoraria, String puestoLaboral, String expPrevia, String estudios, Peso peso)

Los aspectos y sus opciones tienen ahora una clase específica cada una, en lugar de utilizar matrices como se hacía anteriormente. Estas clases se encuentran en el paquete modelo.aspectos.



Para la creación del formulario se utiliza un FormularioFactory (se llama dentro de creaFormulario). De acuerdo a los parámetros de tipo String que se reciban se creará el Formulario.

Los atributos del Formulario serán objetos de las clases de los Aspectos. Tendrá un atributo para cada aspecto, los cuales podrán recibir cualquiera de las tres opciones del aspecto.

Estos cambios fueron de utilidad para el cálculo del puntaje de los aspectos, que se explica posteriormente (en la sección Ronda de Encuentros).

Ronda de Encuentros

Consiste en enfrentar a cada empleado con cada empleador (específicamente sus tickets) para determinar el nivel de coincidencia entre ambos. Ésto quedará reflejado en una Lista de Asignación que se creará para cada usuario, ordenada de mayor a menor según un puntaje de coincidencia.

Modificaciones respecto a la parte I:

El cálculo de puntaje de coincidencias se realiza en la Agencia en:

- public double calculaPuntajeEncuentro(TicketEmpleado ti, TicketEmpleo tj, Peso peso)

Cada aspecto implementa Enfrentable, de esta forma, se compara para cada aspecto, el atributo del formulario del empleado y empleador. Para esto, se utiliza el patrón Double Dispatch. Se tiene una clase padre para los aspectos, y de cada una de ellas heredan otras tres clases: las tres posibles posiblidades (por ej. cargaHoraria se divide en Media, Completa y Extendida). Cada una de estas clases hijas implementan los métodos enfrentarPri, enfrentarSeg y enfrentarTer, además del método enfrentar. Este último llama a alguno de los otros tres métodos pero del aspecto con el cual se quiere enfrentar. Además, se tiene cuidado para saber desde qué perspectiva utilizar la matriz y los pesos.

Ronda de Elecciones

Después de la Ronda de Encuentros, se dará un plazo a los usuarios para que hagan una elección en base a su Lista de Asignación. Cuando termina ese plazo, se realiza la Ronda de Elecciones, que organiza las elecciones en estructuras que luego serán procesadas por la Ronda de Contrataciones. Además, en esta fase se hace un cálculo parcial de los puntajes de la aplicación de cada usuario.

Ronda de Contrataciones

Por último, en la Ronda de Contrataciones, el sistema verifica que empleados y empleadores se eligieron mutuamente, considerando que se hayan elegido para el mismo empleo (ticket). Si hay coincidencia, entonces se instancia un nuevo contrato con ambas partes. Además se hace una modificación final de los puntajes de la aplicación de cada usuario.

Excepciones

Consideramos las siguientes excepciones:

- **PesoInvalidoException:** Se ingresó un valor que no está entre 0 y 1 en alguno de los atributos de Peso.
- **UsuarioRepetidoException:** Se quiso registrar un empleado con el mismo username que otro empleado, o un empleador con el mismo username que otro empleador.
- **UsuariosInsuficientesException:** No hay al menos un empleado y un empleador al realizar la ronda de encuentros laborales.

Desarrollo de la aplicación:

Nuevas funcionalidades.

- Aplicación de patrones vistos en la segunda etapa de la materia.
- Simulación con hilos.
- Persistencia.
- Interfaz gráfica

Patrón state:

El patrón State resultó conveniente para la manipulación de los estados de los tickets emitidos tanto por empleados o empleadores. Recordemos que los tickets poseen varios estados posibles entre ellos el estar: Activo, Finalizado, Suspendido o Cancelado.

La modificación de estos estados se logra desde la ventana de interfaz, y mediante un controlador se le avisa al modelo que debe modificar el/los tickets especificados.

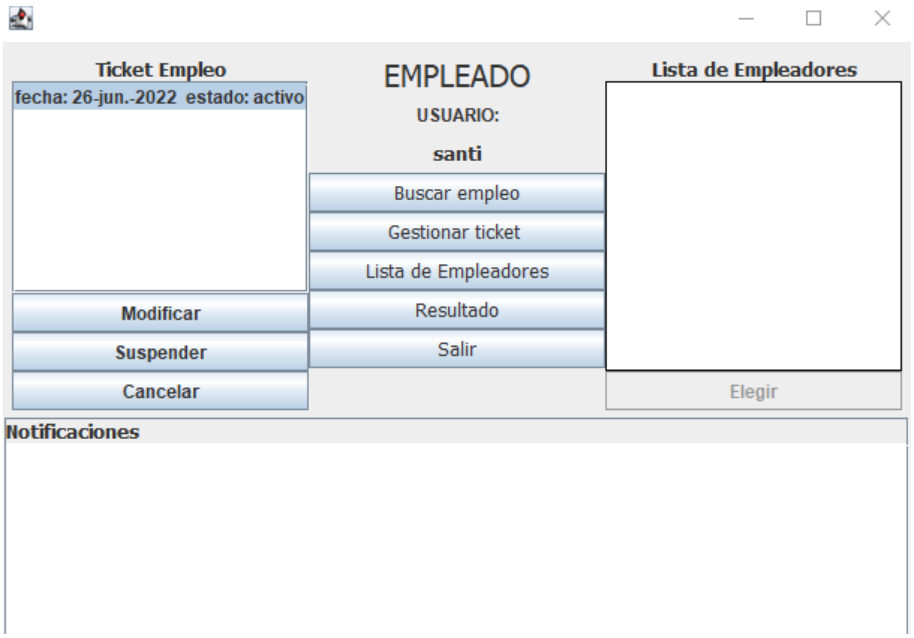
La clase contexto en este caso es el Ticket el cual posee de atributo una variable de tipo interfaz "*IEstadoTicket*". Esta interfaz posee los métodos que implementarán los estados mencionados antes, para modificarse si es necesario.

Se han agregado 4 clases para los 4 estados dichos, estas clases implementan "*IEstadoTicket*" y poseen un atributo de la clase contexto. Por ende genera una conocida doble referencia entre los estados y el ticket

```java
package modelo.estados;

public interface IEstadoTicket {
    void activarse();
    void cancelarse();
    void suspenderse();
    void finalizarse();
    boolean isActivo();
}
```

Interfaz IEstadoTicket



Perspectiva de usuario empleado para manipular, gestionar, emitir y modificar los estados de sus tickets.

Persistencia de la información y serialización

Como requisito de implementación se tiene que la persistencia de la información se hará mediante serialización y archivos. Cada vez que se inicia la aplicación se deben leer todos los archivos para levantar en memoria todos los datos.

Cada vez que la aplicación finaliza (y en cada demanda del usuario) se deben persistir los datos para actualizar los archivos. Para lograr la persistencia utilizamos:

Persistencia Binaria

- La clase a persistir y todas sus variables de instancia deben ser serializables.
- Deben implementar la interface Serializable

Utilización del Patrón DAO y DTO:

Se ha utilizado el ***Patrón DAO*** para independizar a la aplicación sobre la forma de acceder a los datos. Para cumplir con el prometido se procedió a utilizar una interfaz llamada *"IPersistencia*" la cual provee los métodos necesarios para insertar, actualizar, borrar y consultar información.

```
package persistencia;

import java.io.IOException;

public interface IPersistencia<Serializable> {
    void abrirInput(String nombre) throws IOException;

    void abrirOutput(String nombre) throws IOException;

    void cerrarOutput() throws IOException;

    void cerrarInput() throws IOException;

    void escribir(Serializable objecto) throws IOException;

    Serializable leer() throws IOException, ClassNotFoundException;
}
```

Aparte se aplicó el ***Patrón DTO*** debido a que la clase Agencia utiliza el Patron Singleton, y con este patrón mencionado, podemos transformar la Agencia en una clase serializable.

Se procedió a crear una clase idéntica a la agencia dentro del paquete "persistencia" la cual es llamada "AgenciaDTO", para realizar las transformaciones para la serialización y la manipulación de los datos se utilizaron los siguientes métodos de transformación dentro de una clase "UtilAgencia"

```java
package persistencia;

import modelo.Agencia;

public class UtilAgencia {

    public static AgenciaDTO AgenciaToAgenciaDTO(Agencia agencia) {
        AgenciaDTO respuesta = new AgenciaDTO();
        respuesta.setFondos(agencia.getFondos());
        respuesta.setContratos(agencia.getContratos());
        respuesta.setEleccionesEmpleadores(agencia.getEleccionesEmpleadores());
        respuesta.setEleccionesEmpleados(agencia.getEleccionesEmpleados());
        respuesta.setEmpleadores(agencia.getEmpleadores());
        respuesta.setEmpleados(agencia.getEmpleados());
        respuesta.setAdministradores(agencia.getAdministradores());
        return respuesta;
    }

    public static void AgenciaDTOToAgencia(AgenciaDTO agenciaDTO, Agencia agencia) {
        agencia.setFondos(agenciaDTO.getFondos());
        agencia.setContratos(agenciaDTO.getContratos());
        agencia.setEleccionesEmpleadores(agenciaDTO.getEleccionesEmpleadores());
        agencia.setEleccionesEmpleados(agenciaDTO.getEleccionesEmpleados());
        agencia.setEmpleadores(agenciaDTO.getEmpleadores());
        agencia.setEmpleados(agenciaDTO.getEmpleados());
        agencia.setAdministradores(agenciaDTO.getAdministradores());
    }
}
```

Información a persistir

La información que se desea persistir es toda la relacionada con los usuarios registrados en la Agencia y todos sus tickets realizados. Entre los datos persistidos se encuentran:

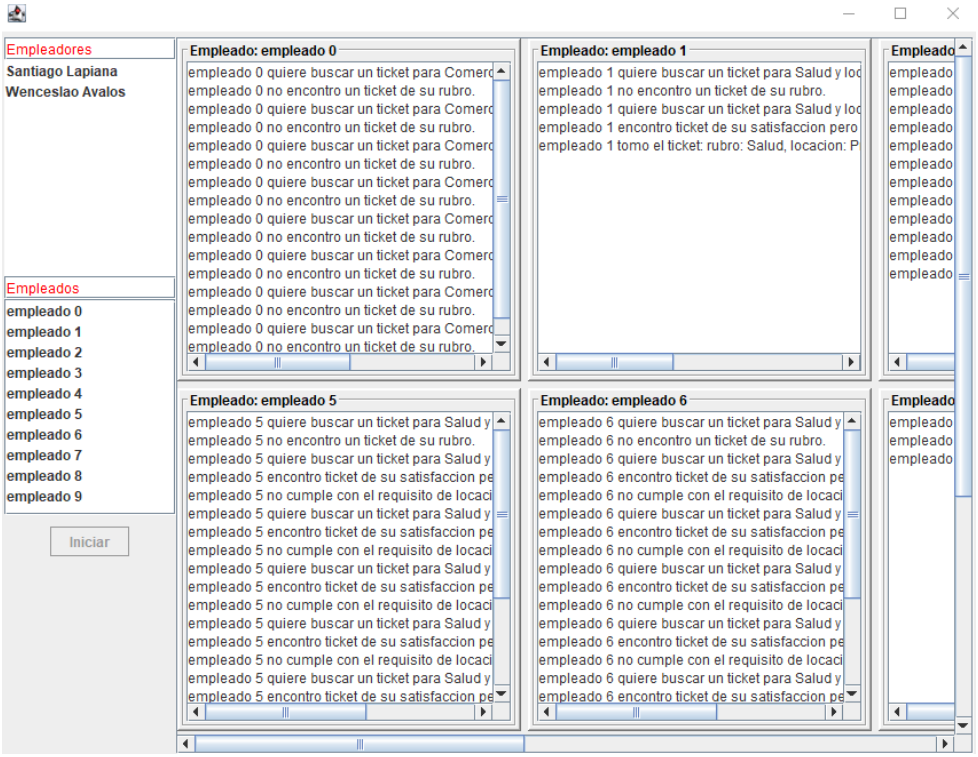
-La Agencia con todos sus datos, entre ellos:
-Lista de Empleadores
-Lista de Empleados Pretensos
-Los contratos generados.
-Tickets activos.

Simulación con Hilos:

Para una correcta visualización de la simulación mostrada con los hilos de los empleados y empleadores se contó con una ventana “VSimulación” específica y modelada especialmente para esta demostración. Dentro de esta ventana se generará 1 panel por cada hilo que esté interactuando con la bolsa de trabajo, informando constantemente que es lo que está haciendo. Estos paneles son observadores de los hilos, y mediante un método “Notificador” se le informa que es lo que debe mostrar por pantalla. Un modelado similar a los vistos en las clases teórico-prácticas.

Debido a que los empleados y empleadores se extienden de una clase “NoAdmin” la cual surge de “Usuarios”, y cómo deben ser concurrentes, se procedió a implementar en “NoAdmin” la interfaz “Runnable”.

Se cuenta con un controlador específico llamado “ControladorThread” el cual actualizará las listas de las vistas de hilos que están compitiendo por un ticket en la bolsa (o bien queriendo emitir uno) y el que se encarga de iniciar los métodos run de cada uno accediendo al modelo.



Simulación de hilos iniciada con 2 Empleadores y 10 Empleados.

Patrón MVC y Observer-Observable.

El patrón MVC permite separar la lógica de negocio de la interfaz de usuario y así repartir responsabilidades. Se divide en 3 componentes cómo indican sus siglas (modelo, ventana y controlador).

La ventana es la encargada de comunicarle al controlador que es lo que desea el usuario. El controlador en base a lo comunicado modificará el modelo si es necesario y se comunicará con la ventana nuevamente.

Para la implementación de los susodicho, se cuenta con tres paquetes diferenciados los cuales son “Vista”, “Modelo” y “Controlador”.

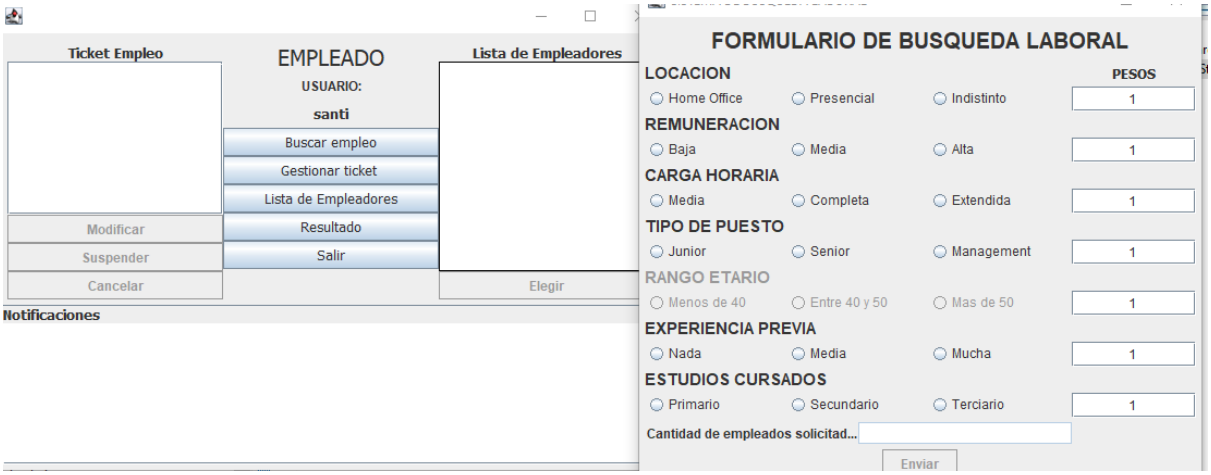
En la *Vista* se implementan dos interfaces para modelar los comportamientos de las ventanas las cuales son, IVistaLogin e IVistaUsuario.

Aparte están desarrolladas todas las clases necesarias para mostrar una interfaz gráfica que permite la interacción entre los usuarios (Administradores, Empleadores o Empleados) con la aplicación.

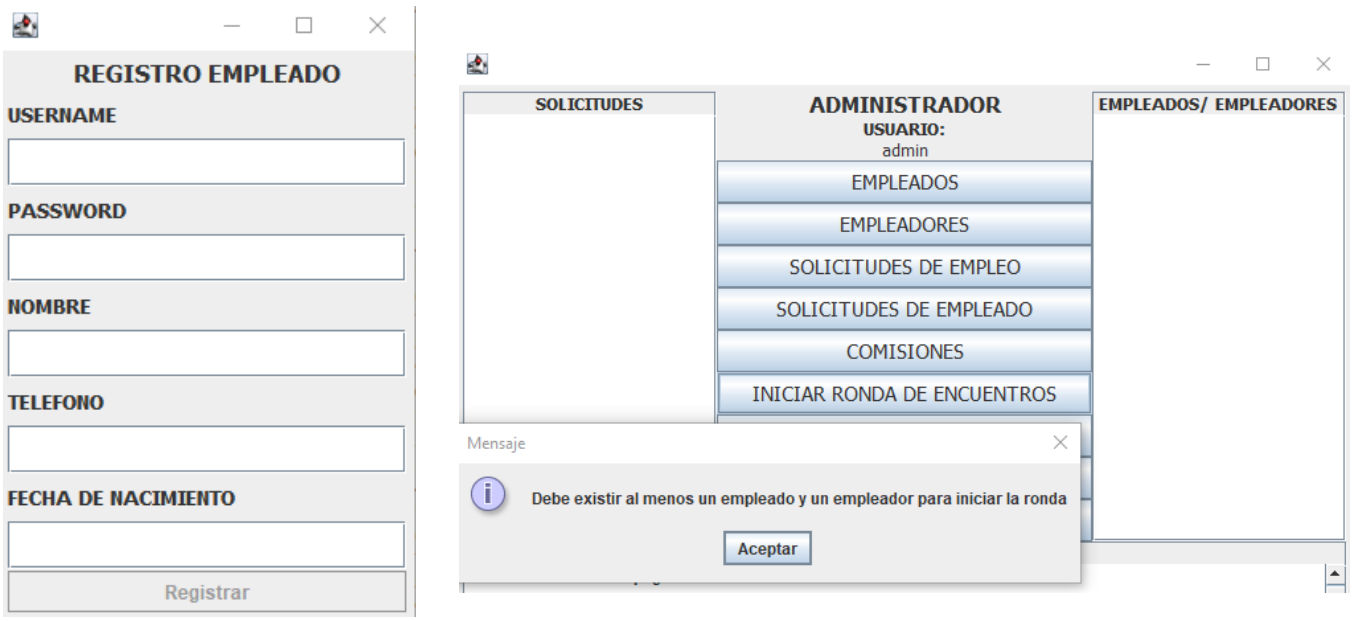
Se han utilizado 4 *controladores* que implementan ActionListener para actuar de acuerdo a lo que indique la Ventana y para comunicarse con el modelo con el fin de modificarlo o bien sea de cerrar o abrir ventanas.

El patrón Observer-Observable se ha utilizado para ayudar a la correcta implementación de estas interacciones mencionadas, lo cual se puede ver de forma clara en el ControladorLogin. El controlador observa a la agencia (observable) y al mismo tiempo hace un action listener de la ventana. Por ende toda acción que dispare un evento desde la vista será atendido en el controlador el cual dirige el flujo de controles.

Se llama a los métodos update() de cada observador desde el modelo, indicando que hay un cambio en el estado del objeto observado (Agencia).



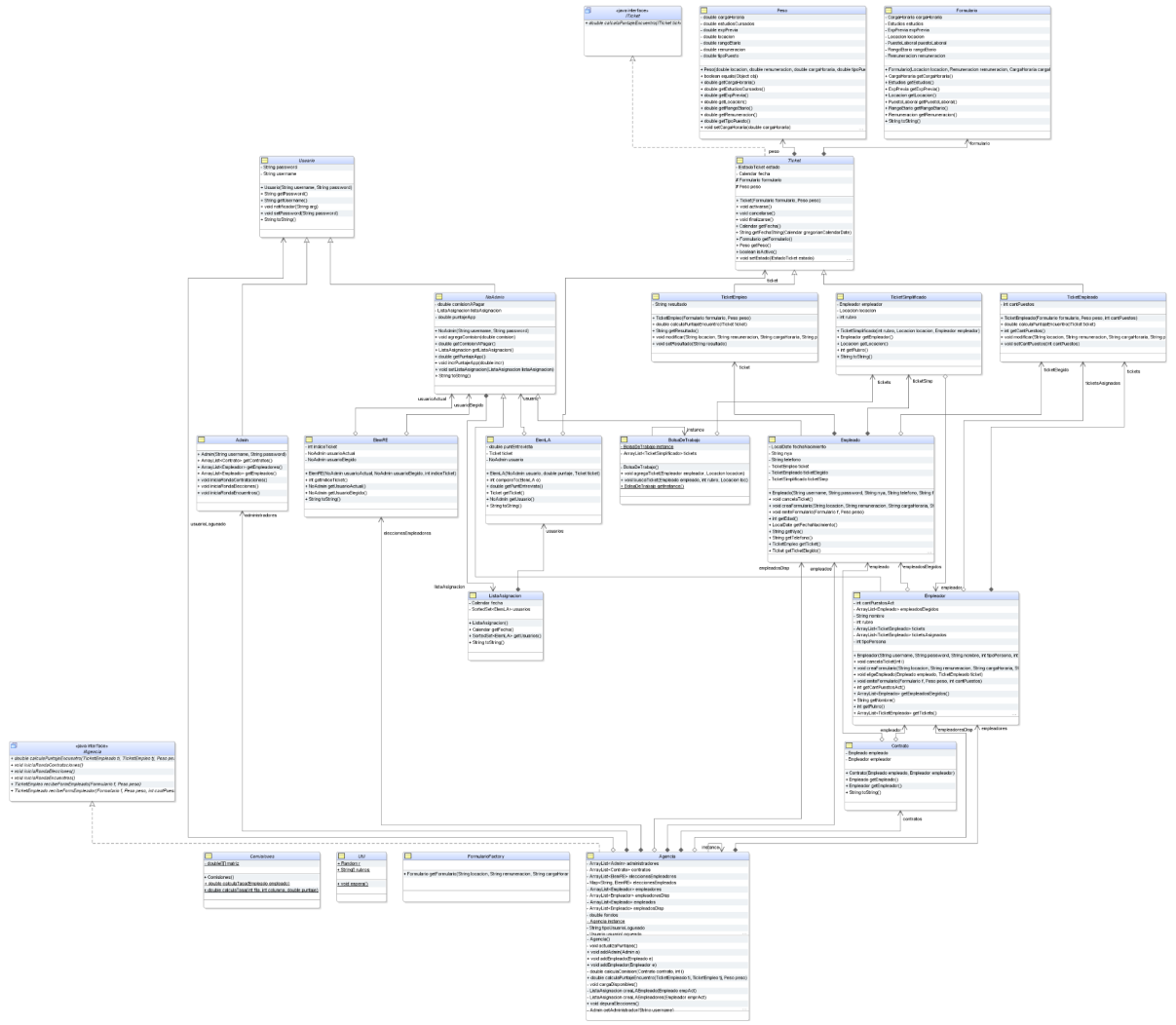
.



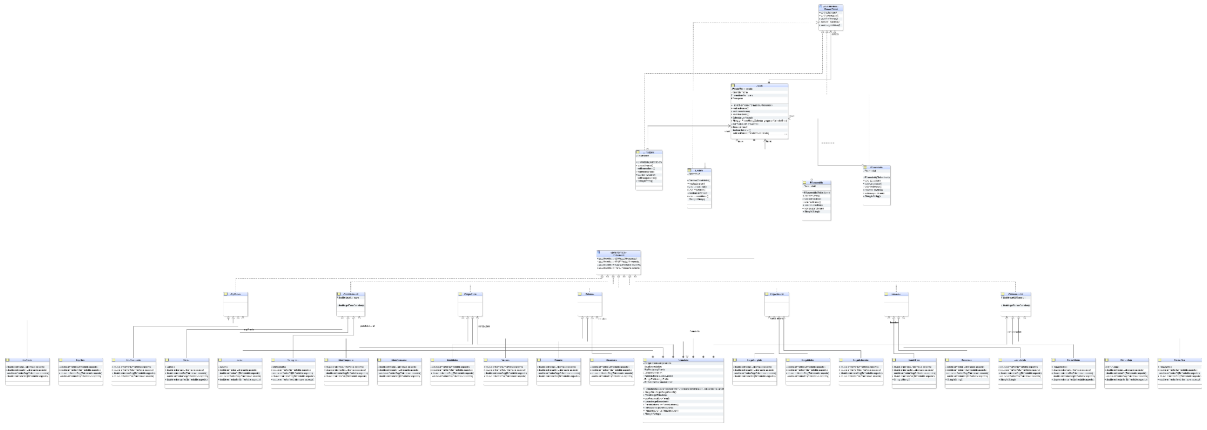
Algunas de las ventanas utilizadas en la aplicación

Diagramas UML

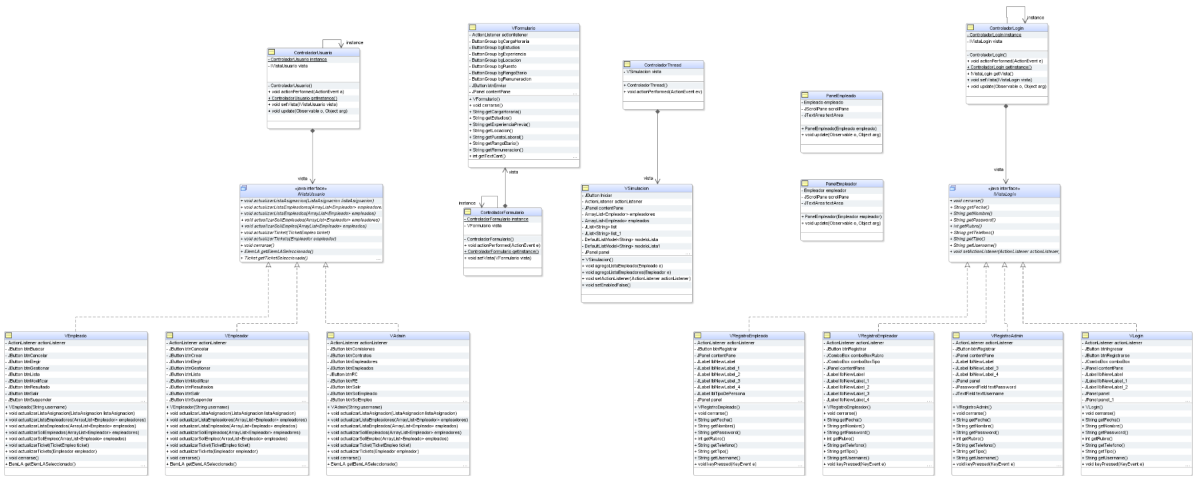
A continuación insertamos los diagramas UML correspondientes a las relaciones entre las clases de nuestro trabajo. Para verlo con más detalle recomendamos ver los archivos .png adjuntados.



“Diagrama UML del paquete modelo “



“Diagrama UML del paquete modelo.aspectos y modelo.estados”



“Diagrama UML del paquete vista y el paquete controlador”

Conclusiones finales:

Durante el desarrollo de la segunda parte de este trabajo se han explorado nuevos patrones de diseño tales como el Observer Observable, Patrón State, MVC, DAO/DTO y complementarlos con los implementados con anterioridad.

Se intelectualizó la importancia de generar código con posibilidad de ser extendido, sin la necesidad de hacer un refactor completo del mismo. Gracias a esto fue relativamente sencillo agregar las nuevas funcionalidades solicitadas.

En esta segunda entrega dos de las cosas más desafiantes e interesantes fueron, por un lado poder generar una interfaz gráfica funcional a todo lo desarrollado y por el otro lado poder persistir los datos cargados en el sistema.

Por último queremos mencionar que ha sido interesante y a la vez gratificante aprender a distribuir las tareas, siendo de vital importancia la puesta en común de criterios a la hora de generar el código.