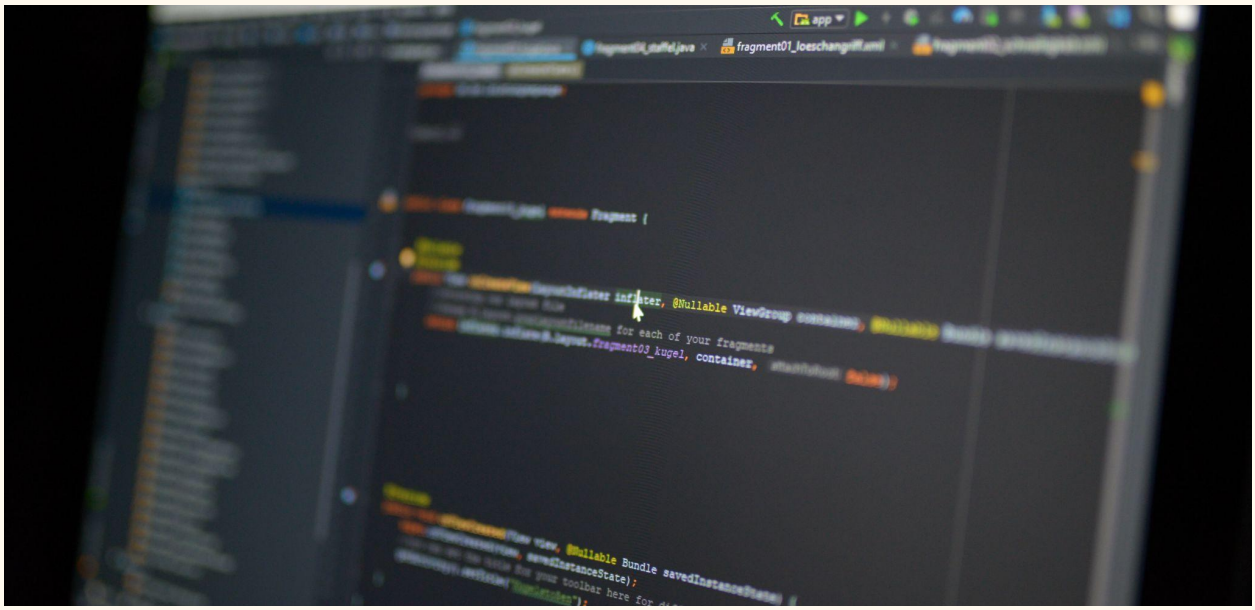
PROGRAMACIÓN III

TRABAJO PRÁCTICO GRUPAL PARTE 2

Facal, Simón
Presa, Martiniano
Reale, Valentina
Sánchez, Milton



BOLSA DE TRABAJO

URL DEL VIDEO:

https://www.youtube.com/watch?v=Q8d3SNwT2LY&ab_channel=Sim%C3%B3nFacal

Modificaciones de la parte 1

Patrón State

Estado del ticket

El gran cambio de la primera parte fue aplicar el Patrón State en el estado de los tickets. El estado de los tickets era un String y, cada vez que teníamos que ejecutar una acción que dependía de este utilizábamos un if con la condición de preguntar el estado para determinar si debía o no ejecutarse el código.

Con la implementación de este patrón evitamos esos if´s y dejamos que el ticket sea responsable de saber determinar qué acción puede y no puede ejecutar según su estado.

¿Cómo lo implementamos?

1. Creamos la interfaz ***IState*** con todos los métodos correspondientes al cambio de estado y a todas las acciones que puede hacer un ticket.
2. Creamos las clases que representan los estados, todas implementan la interfaz ***IState*** y,en el constructor, pasamos como parámetro la referencia a la clase de contexto, en este caso ***Ticket***, y se la asignamos al atributo privado con el mismo nombre.
3. En la clase ***Ticket***, nuestra clase de contexto, cambiamos el atributo estado de ser ***String*** a ser IState y en el constructor inicializamos el estado en activo. También se implementaron todos los métodos de la interfaz ***IState*** pero SIN implementar esta interfaz y dentro de cada método se delegó la responsabilidad al estado.

Comentarios

- En las clases que representan los estados, si bien se implementa la interfaz, no todos los métodos tienen código dentro, ya que esto depende de si el estado que representa la clase puede o no puede ejecutar ese método. Por ejemplo, un ticket en estado **cancelado** no puede ser activado, o sea que no puede pasar a estado **activo**, en este caso el método activar dentro de la clase ***CanceladoState*** lanza una excepción con un mensaje para que luego este se muestre en la ventana.
- Las clases que representan los estados tienen la referencia del ***Ticket*** así pueden utilizar los métodos públicos del ticket y sobre todo para poder hacer la transición de un estado a otro.

- En la clase Ticket no se implementa la interfaz IState ya que, si bien tienen los mismos métodos que un estado, este no comparte herencia de tipo: **un ticket no es un estado**.

Concurrencia

Simulación

En la simulación existen cuatro entidades que forman parte:

- El ticket simplificado, que guarda el rubro y la locación del trabajo, como así también la referencia del empleador.
- La bolsa de trabajo, es el recurso compartido en donde se almacenan los tickets simplificados.
- El empleador, que se encarga de generar los tickets simplificados y los envía a la bolsa de trabajo.
- El empleado pretenso, que es el que intenta obtener un ticket que se adecúe a sus preferencias.

Al ser tanto los empleadores como los empleados pretensos usuarios que debían actuar a la vez y en tiempo real, aplicamos el concepto de concurrencia, que separa el procesamiento en distintos hilos de ejecución. En este caso, un hilo para cada empleador y empleado pretenso.

¿Cómo lo implementamos?

- Creamos las clases ***Simulacion_Empleador*** y ***Simulacion_EmpleadoPretenso***, que se extienden de ***Persona_Empleador*** y ***Persona_EmpleadoPretenso*** respectivamente. Esto fue principalmente para no invadir el resto del sistema con la implementación de la interfaz ***Runnable*** y atributos que no se iban a utilizar por fuera de la simulación. Junto con esas clases, creamos ***BolsaDeTrabajo*** y ***TicketSimplificado***.
- Implementamos el método run en el empleador y el empleado pretenso.
- Implementamos los métodos ***poneTicketSimplificado***, ***sacaTicketSimplificado*** y ***analizaTicketSimplificado***, haciéndolos synchronized para que los recorra un solo hilo a la vez.

¿Cómo funciona?

- Empleador: genera tickets simplificados (puede crear hasta tres) y decide aleatoriamente qué tipo de locación y de rubro tendrán, y luego pone el ticket en la bolsa de trabajo. En

ese momento, notifica a todos los empleados en espera para que vuelvan a buscar tickets en la bolsa.

- Empleado pretenso: tiene diez intentos para obtener un ticket que se adecúe a sus preferencias. Primero, intenta sacar un ticket de la bolsa, pero si no encuentra ningún ticket que coincida con el rubro que él eligió, se queda en espera (el hilo se "duerme"), hasta que le notifiquen algún cambio. Si encuentra un ticket que cumple con la condición, lo saca de la bolsa y lo analiza. Luego de esto, si coincide la locación con su preferencia, se queda con el ticket y termina su búsqueda. Si no cumple con la condición, lo devuelve a la bolsa y notifica a los empleados en espera para que vuelvan a buscar tickets en la bolsa e intenta de nuevo sacar un ticket de la bolsa (hasta que se le acaben los intentos).

Comentarios

- Al principio, por una malinterpretación del trabajo práctico, creímos que los empleados pretensos debían observar la bolsa de trabajo. Implementamos el patrón observer para que, cuando un empleador pusiera un ticket en la bolsa, notificaran a los observers para que vayan a buscar un ticket. Esto causaba un conflicto entre los métodos run y update, que se ejecutarían independientemente de lo que haga el otro, y unirlos haciendo, por ejemplo, que el método update notifique al método run hacía la idea cada vez más confusa. La descartamos y aplicamos sólo la concurrencia, y al patrón observer le dimos otro uso.

Patrón Observer / Observable

Este patrón lo implementamos junto con la simulación. Consta de un observador, el cuál es el 'ControladorVistaSimulación' cuyos observables son todos los hilos de empleados y empleadores. Cada vez que alguno de ellos actúa, ya sea para poner un ticket en la bolsa, en el caso de los empleadores, o sacar uno, en el caso de los empleados pretensos, se informa al observador este 'cambio de estado'. El controlador de la 'VistaSimulacion', cuando recibe esta información, se encarga de escribir en la consola el nuevo estado de la bolsa, esto se detalla en el método 'update' perteneciente a su clase. Los empleadores únicamente pueden poner hasta tres tickets, mientras que los empleados pretensos pueden hacer hasta 10 intentos; una vez que sacan un ticket (porque los rubros coinciden), éste puede coincidir o no con sus intereses de locación. Si ésta no es compatible, vuelve a ingresar el ticket en la bolsa.

En el paquete 'simulacion' se encuentra una clase con métodos para crear la simulación (instanciar los hilos, comenzar su ejecuciones, agregar los objetos observables al observer), otro

para iniciar la simulación (el cual únicamente asigna false al atributo booleano ‘simulacionFinalizada’ perteneciente a la clase ‘BolsaDeTrabajo’) y uno para detenerla. Estos métodos son invocados por el controlador en el instante en que se toca el botón ‘Iniciar’ o ‘Detener’ pertenecientes a la vista.

Persistencia

Para la persistencia del sistema, aplicamos archivos XML, acompañado de los patrones DTO y DAO. Usamos el patrón DAO, creando una interfaz para la persistencia y una clase de persistencia XML que la implementa. Todas las clases que intervienen en la persistencia son independientes del modelo, y se encuentran en el paquete ‘persistencia’. Luego en cada clase que nos resultó relevante pusimos un constructor sin parámetros, getters y setters para sus atributos, de forma que la persistencia se realice correctamente. Usamos el patrón DTO para persistir la clase Administrador ya que implementa patrón Singleton, por lo tanto no persistimos la clase en sí, sino sus atributos. Creamos una clase ‘AdministradorDTO’ la cual contiene los atributos de la clase ‘Administrador’ junto con sus getters y setters. Luego, al leer el archivo con los objetos persistidos, se debe hacer una conversión de ‘AdministradorDTO’ a ‘Administrador’, los métodos para realizar esta acción se encuentran en la clase ‘UtilAdministrador’.

Realizamos la persistencia de todos los usuarios, junto con sus tickets, listas de asignación, elecciones y contratos. De esta manera, si se cierra la aplicación, al volver a ingresar queda toda esta información guardada. Cada vez que algún usuario se registra, borra su cuenta, crea un ticket o realiza alguna modificación, actualizamos la información en nuestro archivo XML.

Patrón MVC

Diseñando las vistas

Para generar la interfaz de comunicación con el usuario utilizamos este patrón. Creamos una ventana de tipo ***JFrame*** que tiene un panel (llamado ***contentPane***) que contiene diez paneles de tipo ***JPanel***. A estos paneles los llamamos “vistas”, los cuales, según interaccione el usuario, se irán mostrando y escondiendo de la ventana.

Tanto la ventana como las vistas tienen una interfaz propia la cual posee todos los métodos que necesita el controlador para obtener información de la vista. Creamos un controlador por cada vista, estos en el constructor reciben como parámetro una referencia al ***IVentana***. El

IVentana posee getters de todas las ***IVista*** y del ***contentPane*** principal, así la ventana principal puede cambiar de vista desde todos los controladores. Para lograr esto, utilizamos una herramienta llamada ***CardLayout***, que nos permitía tratar a las vistas como cartas de un mazo, mostrando solo la que está al frente.

Lo siguiente que hicimos fue deshabilitar botones en caso de que no se cumplan ciertas condiciones. Así es el caso de los registros de usuarios, el inicio de sesión o la creación de tickets. Para esto, tuvimos que implementar los eventos, que ante alguna interacción con el usuario, los métodos correspondientes habilitan o deshabilitan botones según corresponda. Para los ***textField*** usamos el evento ***keyRelease***, que se activa en cuento el usuario deja de presionar una tecla, y el evento ***itemStateChange*** para los ***comboBox*** y los ***radioButton***, que se activa cuando hubo un cambio en el estado del componente.

Añadiendo los controladores

Todos los controladores implementan la interfaz ActionListener y están compuestos por:

- Una referencia a la ***Ventana principal*** (para obtener referencia a la vista y al contentPane)
- Una referencia a la ***vista*** que le corresponde (para delegar funciones visuales)
- Una referencia al ***contentPane*** principal(para poder cambiar de vista)
- El método sobrescrito,de la interfaz ActionListener, ***actionPerformed***(para recibir comandos de la vista y ejecutar acciones)

Fuimos añadiendo controladores a medida que los íbamos necesitando. Lo primero que quisimos implementar fue la transición de una vista a otra, y junto con eso, se setea el tamaño de la ventana cada vez que una vista es mostrada para que se adecúe a esta.

Luego, implementamos las acciones de los botones según el comando enviado por cada uno. Comenzamos con ***VistaInicial*** para que controle el inicio de sesión, seguimos con los registros de los usuarios, luego las funcionalidades, luego la gestión de tickets y finalmente con la simulación.

Todos cumplen la función primordial de relacionar su vista con el modelo. El controlador obtiene información de la vista y hace los cambios pertinentes en el modelo como también catchea excepciones del modelo y le delega a la vista la función de crear ventanas emergentes para que el usuario sea notificado de ciertos errores que provocaron sus acciones.

Otros cambios

- Adición de las siguientes excepciones (todas tienen como parámetro un String que será el mensaje de una ventana emergente):
    - ***CambioDeEstadoFallidoException***: se lanza cuando un ticket intenta cambiar de estado y no puede. Por ejemplo, un ticket cancelado intentando pasar al estado activo.
    - ***ErrorCodigoException***: se lanza cuando se intenta registrar al administrador y se ingresó un código erróneo.
    - ***ErrorUsuarioException***: se lanza cuando se ingresa un usuario que no existe en el sistema.
    - ***UsuarioYaRegistradoException***: se lanza cuando se intenta registrar un usuario ya existente.
- Eliminación de la clase ***RondaDeElecciones*** del modelo: Esta clase tenía como objetivo crear las listas de elecciones de los usuarios de forma aleatoria. Al comenzar a crear la parte visual y volvernos conscientes de que podíamos obtener esta lista directamente desde la vista y entregársela al controlador para que se la settee al usuario, la clase quedó obsoleta y por eso decidimos descartarla.
- Cambio de algunos métodos de visualización en el modelo: métodos como, por ejemplo, ***visualizarEmpleadores*** o ***visualizarEmpleadosPretensos*** de la ***funcionalidadAdministrador*** hacían un ***System.out*** a la consola, cuando lo correcto era que devuelvan un String y que los System.out estén en el main.