PROGRAMACIÓN CON OBJETOS I

TRABAJO FINAL: “Mini-MiUNQ”

Alumnos: De Maio, Julian Daniel - Beltrame, Franco

Fecha: 08/12/2022

Alcance

Se implementaron todas las partes y funciones que el enunciado pedía que realizáramos. A continuación, dejamos algunas consideraciones acerca de la implementación de mensajes que fueron interpretados a nuestro criterio.

*Dimos por sentado y según lo hablado en consultas en clase, que el usuario común y el invitado nunca usarán el mensaje agregarUsuario:, así que no implementamos estos casos ya que damos por entendido que nunca lo usarán. Esto también sucede con darDeBaja: que solamente será utilizado por el usuario root y con modificarUsuario: que solamente será utilizado por el usuario root y el administrador.

*Inicializamos la instancia del sistema miniMiUNQ con el mensaje: (initializeConUsuarioRoot: unUsuario conUsuarioPublico: otroUsuario conUsuarioActual: otroUsuarioMas conDirectorioRaiz: unDirectorio). De esta manera cada vez que se haga una nueva instancia, se la puede inicializar con cualquier tipo de usuarioActual y también con cualquier directorio como raíz.

*Interpretamos que al momento de crear, modificar o eliminar un directorio o un archivo, cada usuario que sea el dueño de dicho directorio o archivo, es capaz de realizar las acciones mencionadas. Sin embargo, el usuarioRoot es el único con la capacidad de hacerlo con archivos o directorios que no le pertenezcan.

*Implementamos en la parte de Streams que solamente se pueden agregar masivamente usuarios comunes, ya que los usuarios con privilegios (usuario root y los administradores) serán una cantidad reducida y no es necesario cargarlos masivamente.

Modelo

En nuestra implementación los objetos más importantes son:

MiniMiUNQ: Es el responsable de la gran mayoría de las operaciones en nuestro modelo. Junto con Directorio y con ayuda de proveedores de estados el sistema MiniMiUNQ es el que nos permite iniciar sesión con diversos usuarios, agregar/modificar/dar de baja usuarios, movernos entre directorios, crear archivos en el disco real, administrar los permisos, entre otras funcionalidades.

Directorio: Es el nexo entre el sistema MiniMiUNQ y los subdirectorios/archivos ya que muchas veces el sistema le delega responsabilidades al mismo. Sus principales responsabilidades son crear/borrar archivos y subdirectorios cuando se lo ordena el sistema MiniMiUNQ.

Usuario: Es una SuperClase, de este objeto depende el inicio de sesión de los diferentes usuarios en el sistema MiniMiUNQ.

Dependiendo del tipo de usuario que esté iniciado sesión en el sistema, el mismo podrá realizar todas sus funcionalidades (agregar, dar de baja, modificar, etc).

Progreso

Como primer instancia, decidimos que para trabajar de manera remota y ordenada, una gran opción era utilizar TeamViewer, ya que nos permitía a ambos codear en la misma imagen de Cuis, sin necesidad de tener que compartir archivos ".st" para no perder el progreso.

Una vez leído el dominio, decidimos crear un bloc de notas en el cual llevamos dicho dominio a un formato más "concreto". De esta manera, cada vez que nos surgía una duda acerca del enunciado, nos era más fácil evacuarla. Una vez hecho esto, nos dimos cuenta que había diferentes usuarios, con diferentes capacidades, y que de estos dependían las acciones que se pueden realizar dentro del sistema MiniMiUNQ. A partir de eso, decidimos empezar a realizar los tests necesarios para modelar la superclase "Usuario" y sus subclases, las cuales se diferencian por su comportamiento.

En los primeros tests, fuimos probando los casos en los que un usuario ya registrado en la plataforma, registra a otro usuario que no lo está, y también contemplamos el caso en el que se agrega un usuario ya registrado. Con este comportamiento, decidimos implementar un double dispatch dinámico, que delegue la acción de registrar al usuario y que este último, accione según su estado de registro.

Luego, implementamos las reglas de negocio sobre los casos de inicio de sesión de los distintos usuarios al sistema MiniMiUNQ. Para esto decidimos inicializar a los usuarios con un nombre y una contraseña. A continuación, comenzamos a testear los casos en que un usuario agrega, modifica o da de baja a otro usuario y reutilizamos el proveedor de estados que creamos para el registro de usuarios, ya que no cualquier usuario puede realizar las acciones mencionadas anteriormente.

Cuando logramos implementar todos los comportamientos de los usuarios, creímos necesario seguir con la implementación de los elementos del sistema, es decir los directorios y los archivos. En primera instancia, definimos que una plataforma MiniMiUNQ está inicializada con un directorio, el cual es llamado “raíz”. En base a esto comenzamos a testear casos en los cuales se agregan nuevos directorios (llamados subdirectorios) a la raíz, siempre y cuando no haya un directorio con el mismo nombre, en ese caso, el sistema lanza error.

Para poder recordar el directorio anterior para realizar la implementación del mensaje ‘volverADirectorioPadre’ (que nos devuelve al directorio padre del actual), se nos ocurrió la idea de tener un colaborador interno en la terminal que simula ser la ruta como la conocemos en por ejemplo Windows. Después de pensar posibles implementaciones decidimos hacerla una OrderedCollection a la que cuando nos movemos a un directorio le añadimos el mismo, y cuando realizamos el comando ‘volverADirectorioPadre’ , removemos su último elemento. Esto nos permitió también tener una referencia dinámica del directorio actual puesto que este va a ser siempre el último elemento de la lista y se actualiza solo puesto que si añadimos un nuevo elemento, este pasa a ser el último y por ende el directorio actual, y si quitamos el último elemento (con el comando ya mencionado) este pasa a ser el directorio visitado anteriormente.

Una vez culminado el sistema de Directorios nos dedicamos a testear lo relacionado a la creación de archivos. No hay mucho para comentar sobre esto puesto que fue muy parecido a los directorios en cuanto a su creación, las excepciones y tal. Lo que sí varió con respecto a Directorio es que teníamos la habilidad de leer/escribir en los mismos; para realizar esto optamos que el archivo tenga un colaborador interno que sea un string al cual le concatenamos el contenido a escribir.

A partir de este momento ya nos empezamos a plantear la idea que Directorio y Archivo tenían ciertas características en común que podrían llevarnos a ubicarlos en una jerarquía que luego llamaríamos Elemento.

En cuanto a los permisos de lectura, escritura y borrado, decidimos que un elemento por default es un elemento sin permisos, es decir, privado. A partir de esto, creamos un mensaje de la plataforma llamado 'cambiarPermisoDe:alElemento:', el cual realiza una validación sobre el usuario actual para definir si el mismo cuenta con las capacidades para realizar dicha acción, antes de ejecutar el cambio de permiso.

Para la implementación de filtros, comenzamos creando un mensaje de la plataforma para cada filtro, y llegamos al punto en que dichos filtros debían ser combinables y ahí es donde todo el progreso que habíamos hecho, no funcionaba. Luego de pensarlo un rato, se nos ocurrió crear un mensaje de la plataforma llamado 'flitrarPor:' que recibe una colección de filtros, los cuales implementamos como subclases para cada tipo de filtro. Dependiendo del filtrado que se desea aplicar, se le agrega a la colección esperada cada clase de filtro con un mensaje que recibe como parámetro el valor por el que se va a filtrar.

Por último, encaramos la resolución de streams, que luego de mucha prueba y error, logramos hacer funcionar el test que extrae la información de los elementos del directorio actual (luego de correr el mismo, se genera el archivo de texto en la carpeta donde se ejecuta CuisUniversity) y también el de la carga masiva de usuarios comunes.

Dificultades

Las dificultades que se nos presentaron a la hora del desarrollo fue primeramente el desconocimiento del dominio ya que si bien es algo que utilizamos cotidianamente, no le prestamos la suficiente atención a su funcionamiento y su lógica. Una vez entendido el dominio comenzamos a hacer TDD y este nos fue guiando y pudimos avanzar en el trabajo.

Algo que nos costó fue aplicar el double dispatch dinámico, ya que no contábamos con tanta práctica en este tipo de implementaciones, pero una vez que logramos implementar uno, los siguientes nos resultaron mucho más amenos, ya que podíamos guiarnos del primero.

Además, al momento de llevar a cabo la implementación de una determinada parte del dominio, nos resultó difícil decidir qué camino tomar o de qué manera resolver dicha problemática, ya que uno intenta siempre elegir el camino que sea más fácil y que menos complicaciones le traiga.

Un claro ejemplo de esto, fue lo relatado anteriormente acerca de la implementación de filtros, ya que en un principio abordamos la problemática de una manera y llegamos a un punto en que nuestra resolución se volvía obsoleta, en consecuencia, tuvimos que empezar de cero y plantear una resolución diferente.

Nos encantaría poder decir que alguna parte del enunciado nos resultó fácil, pero ninguna lo fue. Cada vez que avanzábamos en el trabajo, se nos presentaba un nuevo desafío. Sin embargo, pudimos resolver la integridad del trabajo.

Conclusiones

Como conclusión, pensamos que este trabajo abarcó en su totalidad los temas vistos en la materia.

Uno de los conceptos que se aplicó en mayor medida fue el de proveedor de estados, utilizado para la creación de archivos y directorios (por ejemplo para saber si había uno creado anteriormente con el mismo nombre), en el registro de usuarios y la administración de los mismos.

También se utilizó bastante el concepto de double dispatch tanto en los proveedores como en otros mensajes, utilizamos el concepto de delegación, las validaciones para levantar excepciones, utilizamos jerarquías y además manejamos streams.

Más allá de todo, el trabajo nos ayudó a reforzar los conceptos vistos en la materia, ya que a pesar de tener conocimiento de ellos, no era nuestro fuerte llevarlos a la práctica.