



Universidad
Zaragoza



Escuela Universitaria
Politécnica - Teruel
UniversidadZaragoza

Práctica 3

Encapsulación de una aplicación legada

Radu Constantin Robu
Alberto Pérez Blasco

Sistemas Legados

Escuela Universitaria Politécnica de Teruel
Departamento de Informática e Ingeniería de Sistemas

27 de enero de 2022

Resumen ejecutivo

Este documento recoge la realización de la práctica 3 de la asignatura de Sistemas Legados. Para la realización de la práctica se ha estudiado primero el entorno de trabajo, instalando las aplicaciones necesarias para su correcto funcionamiento.

Posteriormente, se ha realizado un estudio de la aplicación legada con la que se ha tenido que trabajar. Estudiando su funcionamiento, operaciones permitidas y los datos necesarios.

Una vez acabado este paso, se ha realizado el diseño de la aplicación moderna, y del wrapper necesario para conectar la aplicación legada con la aplicación moderna.

Por último, se ha implementado la aplicación moderna, con su interfaz y sus métodos, así como los métodos del wrapper para la comunicación entre la aplicación moderna y la aplicación legada.

Índice

1. Introducción	3
2. Entorno de trabajo	3
3. Funcionamiento de la aplicación legada	4
4. Diseño	5
4.1. Diagrama de clases	5
4.2. Diagramas de secuencia	6
5. Comunicación entre el wrapper y x3270	12
5.1. Redirección de los canales estándar (salida y entrada)	12
5.2. Envío de comandos	12
5.3. Comprobación de comando ejecutado correctamente	14
5.4. Lectura de la pantalla	15
5.5. Wait(output) propio	15
6. Creación de la vista	16
7. Funcionamiento de la aplicación	17
7.1. Nuevo fichero de tareas	17
7.2. Añadir tarea	18
7.3. Eliminar tarea	22
7.4. Buscar tareas	24
7.5. Listar tareas	25
7.6. Salir	26
8. Conclusiones	28
Referencias	28

1. Introducción

La práctica 3 de la asignatura de Sistemas Legados está enfocada a la aplicación de los conocimientos teóricos de la asignatura para la implementación de un encapsulador o *Wrapper*. Este encapsulador se debe emplear para implementar una aplicación gráfica que permita, a través del encapsulador, comunicarse y emplear una aplicación legada ejecutada sobre un supuesto IBM ESA/390 con el sistema operativo MUSIC/SP.

El objetivo principal de la práctica es definir un API que pueda posteriormente ser utilizado por el software del sistema moderno donde se quiere integrar la aplicación legada. Además, se debe desarrollar un programa con interfaz gráfica que ejecute la aplicación legada mediante el wrapper con el fin de probar el correcto funcionamiento de todo el sistema.

2. Entorno de trabajo

Para la ejecución de la aplicación legada se ha empleado MUSIC/SP a través del emulador s3270. Para comunicarse con el emulador, ha sido necesario redireccionar la entrada y salida. De esta forma, el emulador, ejecutado como proceso, puede recibir y enviar datos al programa implementado.

Al principio del proceso de realización de la práctica, se pensó en implementar la interfaz de usuario en Python, realizando la comunicación con una aplicación servidor implementada en Java.

Sin embargo, debido a la falta de tiempo y de complicaciones a la hora de emplear Python junto con la librería s3270, se decidió realizar la práctica en Java empleando Java JDK 11 y ejecutando s3270 como proceso.

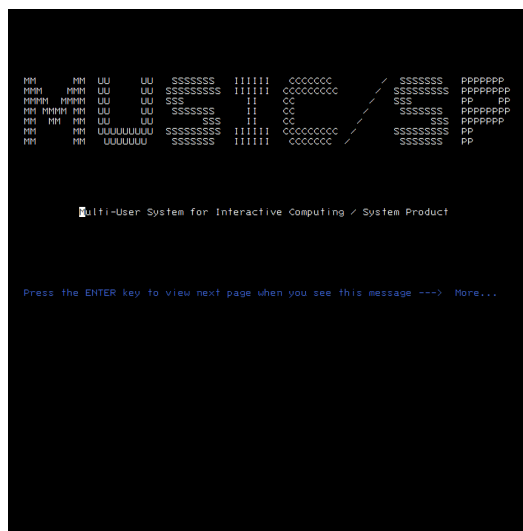


Figura 1: Emulador de sistema operativo MUSIC/SP con interfaz gráfica.

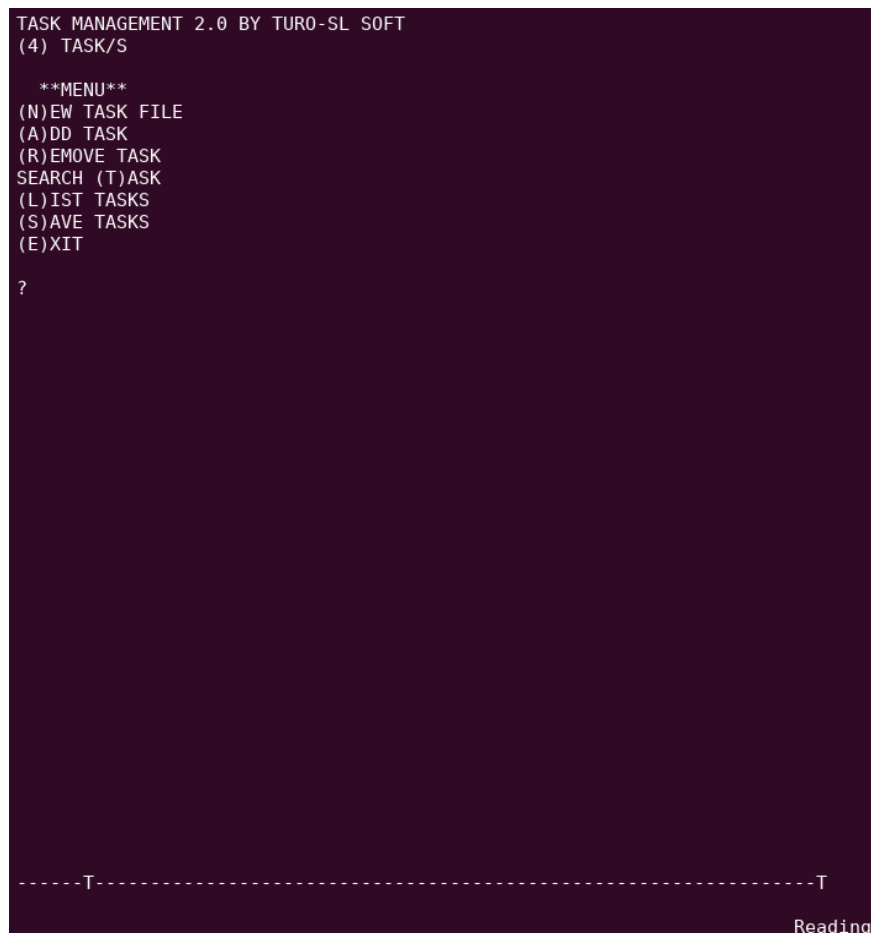


Figura 2: Versión terminal sin pantalla del emulador de MUSIC/SP.

3. Funcionamiento de la aplicación legada

La aplicación legada (Task Management 2.0), está formada por un menú en el que se muestran las distintas opciones que se pueden realizar (Figura figura 3). Siendo estas:

- Crear un nuevo fichero de tareas (lugar en el que se almacenan las tareas que se van añadiendo, eliminando).
- Añadir una nueva tarea al fichero (necesario id, nombre, descripción y fecha).
- Eliminar una tarea existente (necesario proporcionar un id).
- Buscar tareas (como patrón de búsqueda se utilizará la fecha).
- Listar las tareas almacenadas en el fichero.
- Guardar los cambios realizados en la sesión.
- Salir (en caso de no haber guardado los cambios se preguntará al usuario si desea guardarlos).

The image shows a terminal window with a dark background and light-colored text. At the top, it says 'TASK MANAGEMENT 2.0 BY TURO-SL SOFT' and '(4) TASK/S'. Below that is a menu titled '**MENU**' with the following options: '(N)EW TASK FILE', '(A)DD TASK', '(R)EMOVE TASK', 'SEARCH (T)ASK', '(L)IST TASKS', '(S)AVE TASKS', and '(E)XIT'. A question mark '?' is on the line following the menu. At the bottom of the terminal, there is a dashed line with 'T' at each end, and the word 'Reading' is visible in the bottom right corner of the terminal window.

```
TASK MANAGEMENT 2.0 BY TURO-SL SOFT
(4) TASK/S

  **MENU**
(N)EW TASK FILE
(A)DD TASK
(R)EMOVE TASK
SEARCH (T)ASK
(L)IST TASKS
(S)AVE TASKS
(E)XIT
?

-----T-----T
Reading
```

Figura 3: Funcionalidad de la aplicación legada

4. Diseño

La implementación del wrapper y de la aplicación con interfaz gráfica de usuario se ha realizado en base a un diagrama de clases y a los diagramas de secuencia.

El diagrama de clases se ha realizado con el fin de estructurar el trabajo a realizar, mientras que los diagramas de secuencia se han realizado con el fin de entender todo el funcionamiento de la aplicación legada y de la secuencia de cada una de sus funciones.

4.1. Diagrama de clases

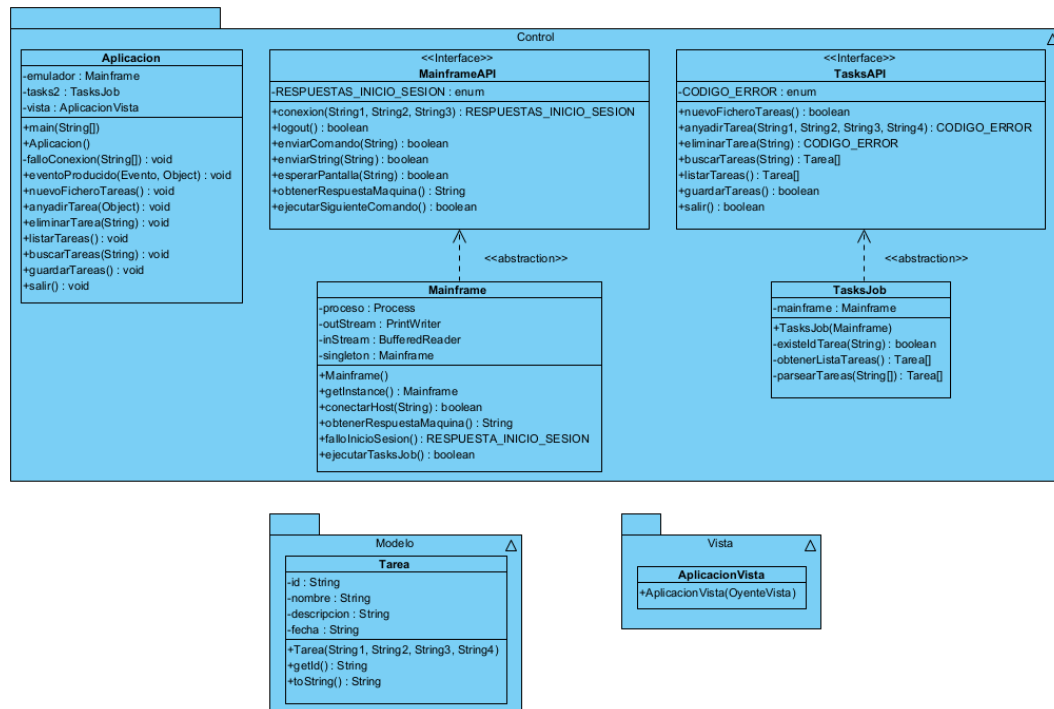


Figura 4: Diagrama de clases del wrapper y de la aplicación con interfaz gráfica.

El diseño del wrapper y de la aplicación se ha basado principalmente en el modelo MVC. Se han separado las clases e interfaces necesarias en los paquetes correspondientes para separar y tratar la implementación en conjuntos más pequeños de código. Principalmente, se han implementado dos interfaces, una para las funciones del sistema operativo MUSIC/SP llamada *MainframeAPI*, y una interfaz para los comandos de la aplicación legada, llamada *TasksAPI*. Los métodos de las interfaces son posteriormente implementados en las clases *Mainframe* y *TasksJob*, además de los otros métodos necesarios para la implementación de todo el programa.

La clase *Aplicacion* será el main del programa, que contiene sus propios métodos y atributos.

El paquete Modelo contendrá una clase *Tarea* empleada para el tratamiento de la información de las tareas procedentes de la aplicación legada, y una clase auxiliar, no reflejada en el diagrama, que representará tuplas de objetos, necesaria para la comunicación MVC.

Por último, el paquete Vista contendrá la vista de la aplicación moderna junto con una clase auxiliar no reflejada para ventanas emergentes con campos de texto.

4.2. Diagramas de secuencia

En este apartado del diseño se exponen los diagramas de secuencia construidos para entender el funcionamiento de la aplicación legada. No se ha considerado la necesidad de extender la explicación de los diagramas.

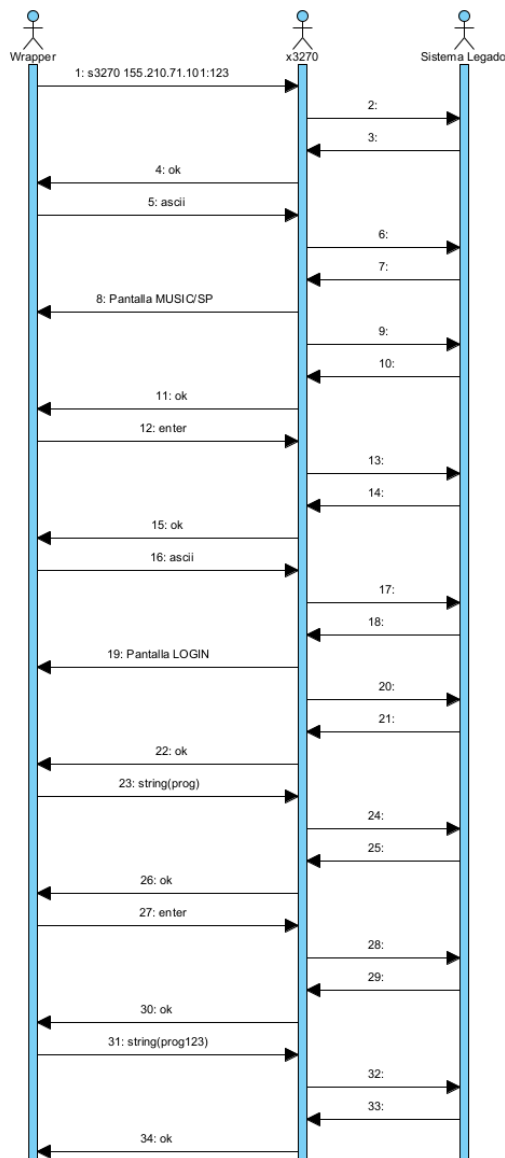


Figura 5: Primera mitad del diagrama de secuencia de la conexión con el emulador.

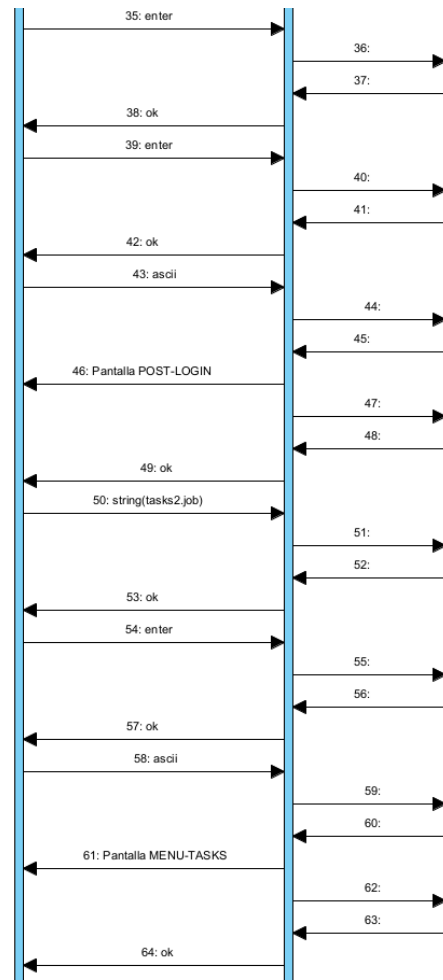


Figura 6: Segunda mitad del diagrama de secuencia de la conexión con el emulador.

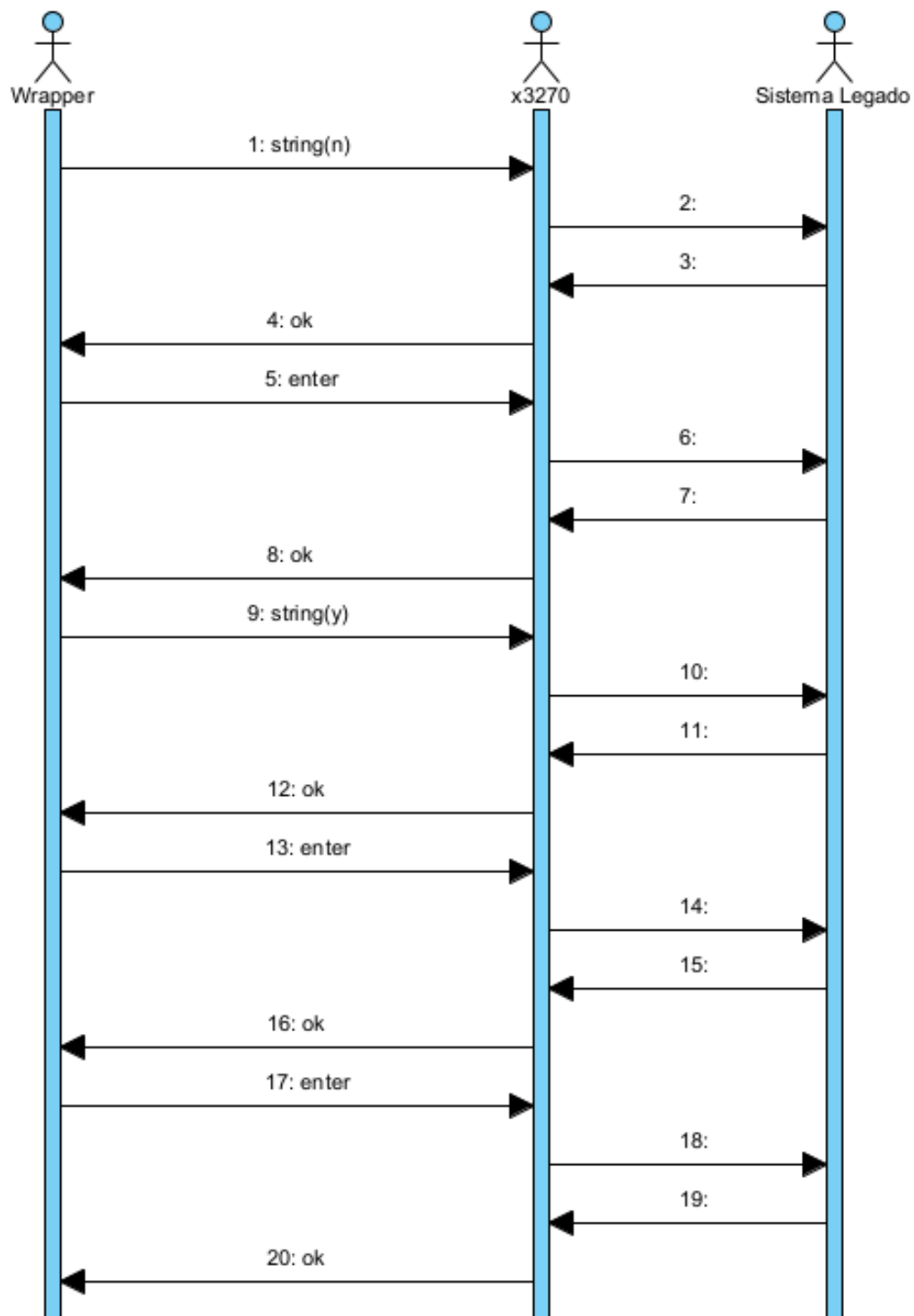


Figura 7: Diagrama de secuencia de la creación de un nuevo fichero de tareas en la aplicación legada.

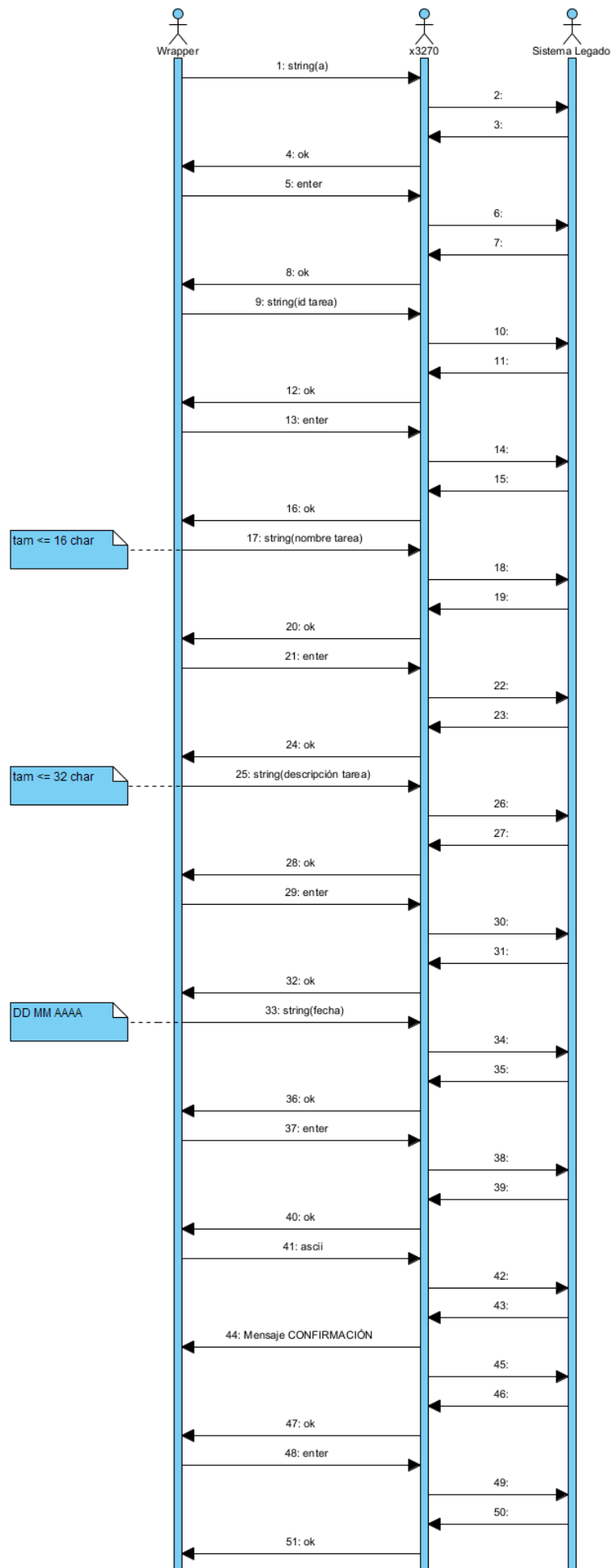


Figura 8: Diagrama de secuencia de la adición de tareas en la aplicación legada.

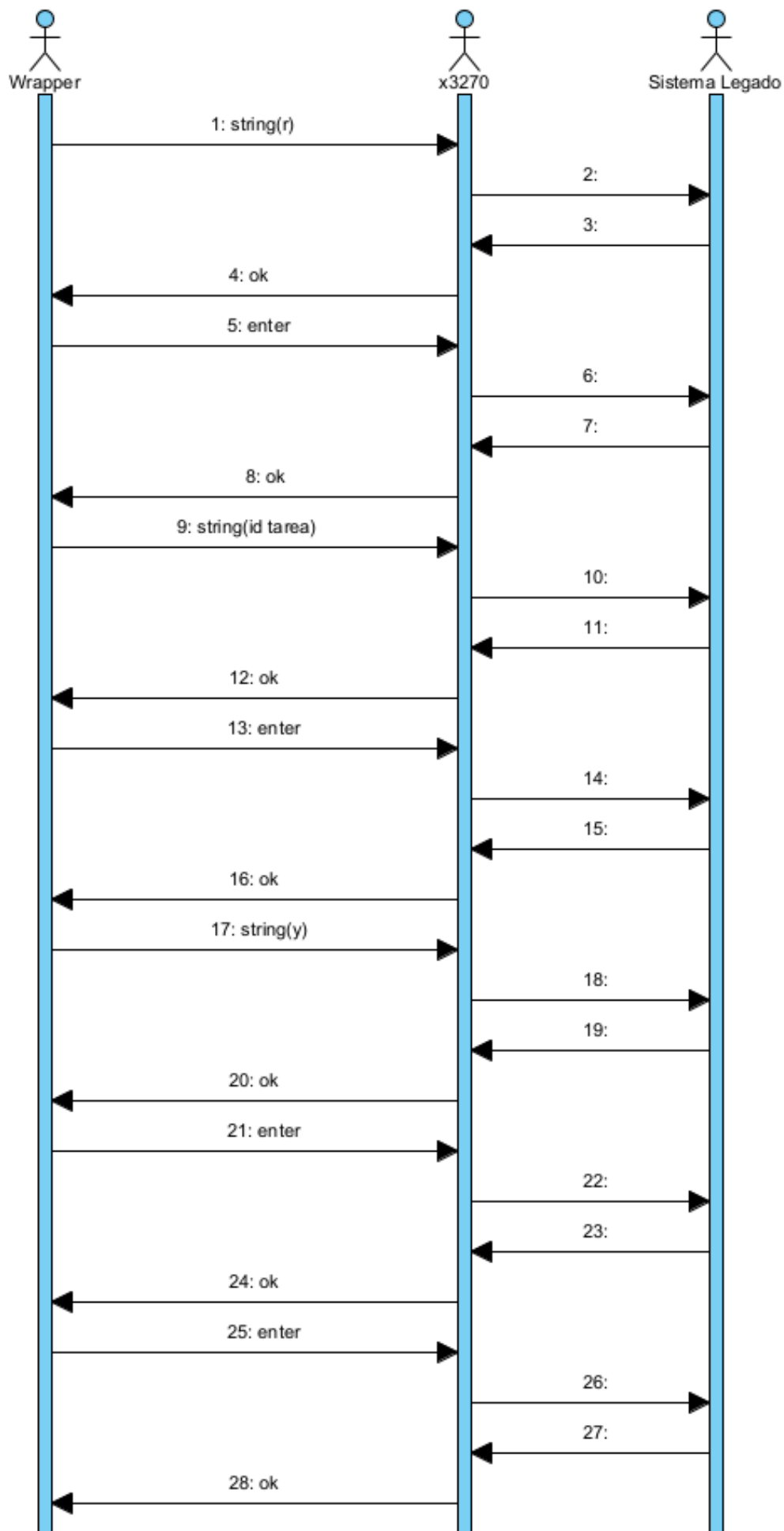


Figura 9: Diagrama de secuencia de la eliminación de tareas de la aplicación legada.

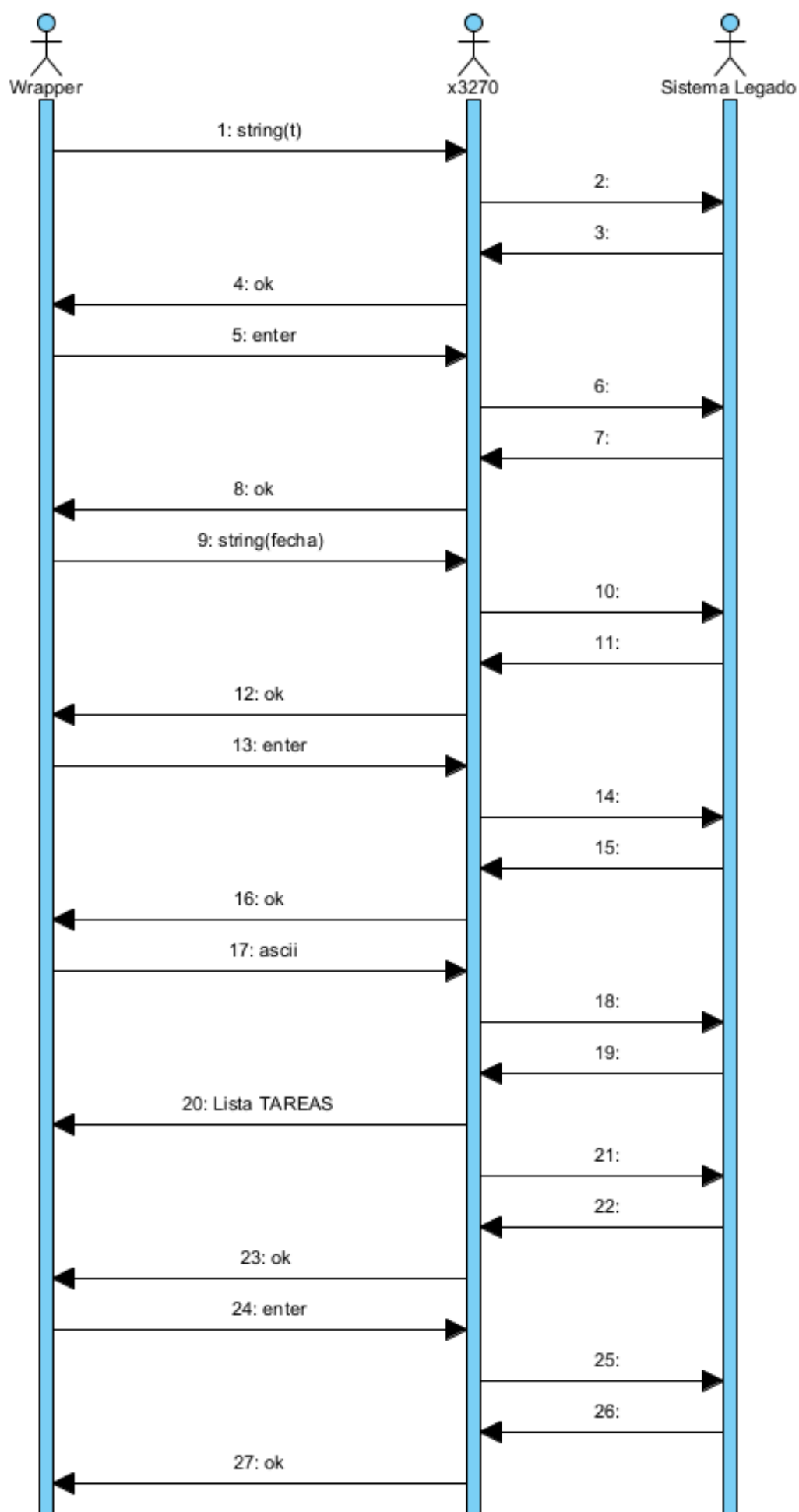


Figura 10: Diagrama de secuencia de la búsqueda de tareas en la aplicación legada.

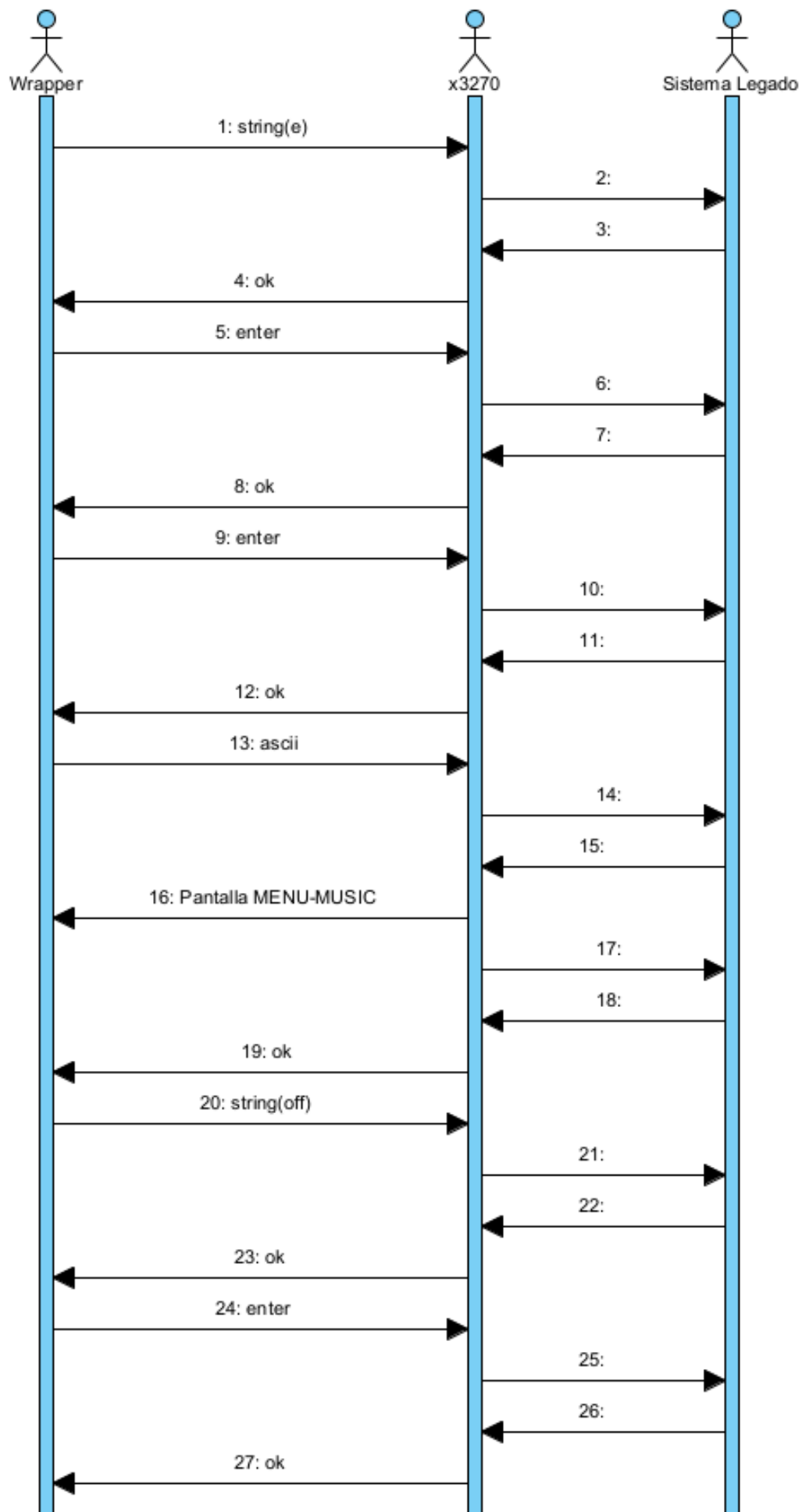


Figura 11: Diagrama de secuencia del proceso de desconexión.

5. Comunicación entre el wrapper y x3270

5.1. Redirección de los canales estándar (salida y entrada)

Para la comunicación con x3270, se utilizó la interfaz Mainframe, utilizando el patrón de diseño singleton, debido a que únicamente queremos una instancia de la clase para de esta forma evitar problemas al no permitir el sistema legado múltiples usuarios sobre un mismo host. Para su implementación, se utilizó lo siguiente:

```
private Mainframe() throws IOException {  
    proceso = Runtime.getRuntime().exec(TERMINAL_SIN_PANTALLA);  
    inStream = new BufferedReader(new InputStreamReader(  
        proceso.getInputStream()));  
    outStream = new PrintWriter(new OutputStreamWriter(  
        proceso.getOutputStream()));  
}
```

Figura 12: Constructor privado de la clase.

Como se puede observar en el código, el constructor es privado para mantener el patrón de diseño. En el código, en primer lugar se crea el proceso de conexión entre la aplicación y x3270 y posteriormente, se redirigen la entrada estándar y la salida estándar a dos buffers, los cuales sirven para poder escribir y leer en el proceso creado. Para terminar con el patrón de diseño singleton, se utiliza el método siguiente en el que se comprueba que únicamente haya una instancia de la clase:

```
public static Mainframe getInstance() throws IOException {  
    if (singleton == null) {  
        singleton = new Mainframe();  
    }  
    return singleton;  
}
```

Figura 13: Obtener instancia de la clase.

5.2. Envío de comandos

A la hora de enviar comandos, se separó entre string, connect y el resto de comandos, debido a que de esta forma, en el código se veía de una forma clara el comando que se ejecuta en x3270. En el caso del comando ASCII, se devolverá true, debido a que el método de comprobar siguiente comando lee del buffer hasta encontrar un ok, lo que haría que ya se leyese el comando ascii, siendo inservible su ejecución.

Finalmente, para cada uno de los comandos se comprobará si se ha ejecutado correctamente y se devolverá true en caso correcto, para así indicar a la aplicación que se puede seguir con el flujo de ejecución.

```
@Override
public boolean enviarComando(String comando) throws IOException {
    outputStream.println(comando);
    outputStream.flush();
    if (comando.equals(COMANDO_ASCII)) {
        return true;
    } else {
        return ejecutarSiguienteComando();
    }
}
```

Figura 14: Envío de un comando a x3270.

Se observa que en el caso del comando ASCII se retorna siempre. Esto se debe a que el tratamiento del comando string se realiza de forma distinta. Cuando se ejecuta el comando ascii, en la terminal s3270 se mostrará un vector de string que contendrá la información recibida de la aplicación legada.

En vez de esperar el string .ok como respuesta de la máquina, se espera por una cadena de texto en concreto de la aplicación con el método de espera de pantalla (figura 15, explicado en apartado 5.5).

```
@Override
public boolean esperarPantalla(String lineaABuscar)
    throws IOException, InterruptedException {
    Long maxTime = TIEMPO_EJEC_MAXIMO + System.currentTimeMillis();
    String resultado = "";
    do {
        enviarComando(COMANDO_ASCII);
        sleep(DELAY);
        resultado = obtenerRespuestaMaquina();
        if (resultado.contains(lineaABuscar)) {
            return true;
        }
    } while (maxTime > System.currentTimeMillis());

    return false;
}
```

Figura 15: Método para esperar una pantalla en concreto y tratar el funcionamiento del comando ASCII.

```
@Override
public boolean enviarString(String mensaje) throws IOException {
    outputStream.println(String.format(FORMATO_CADENA_TEXTO, mensaje));
    outputStream.flush();
    return ejecutarSiguienteComando();
}
```

Figura 16: Envío de una cadena de texto a x3270.

```
private boolean conectarHost(String host) throws IOException {  
    outputStream.println(CONNECT + host);  
    outputStream.flush();  
    return ejecutarSiguienteComando();  
}
```

Figura 17: Envío de un comando de conexión a x3270.

5.3. Comprobación de comando ejecutado correctamente

Para la comprobación de si un comando se ha ejecutado, se comprobará en el buffer de entrada si alguna línea contiene la cadena “ok” con una expresión regular. En caso de que el buffer esté vacío (null) o de error, se devolverá falso, indicando una mala ejecución del comando anterior.

```
@Override  
public boolean ejecutarSiguienteComando() throws IOException {  
    String line = "";  
    Long maxTime = TIEMPO_EJEC_MAXIMO + System.currentTimeMillis();  
    while (maxTime > System.currentTimeMillis()) {  
        line = inputStream.readLine();  
        if (line == null ||  
            line.matches(PATRON_RESPUESTA_MAINFRAME_ERROR)) {  
            System.out.println(line);  
            return false;  
        } else if (line.matches(PATRON_RESPUESTA_MAINFRAME_OK)) {  
            return true;  
        }  
    }  
    return false;  
}
```

Figura 18: Comprobación de si un comando se ha ejecutado correctamente.

5.4. Lectura de la pantalla

En esta función, se obtendrá en un string todo lo que se encuentra en el buffer de entrada, almacenándolo en la variable resultado.

```
public String obtenerRespuestaMaquina() throws IOException {
    String resultado = "";
    String line = "";
    do {
        line = inStream.readLine();
        if (line == null ||
            line.matches(PATRON_RESPUESTA_MAINFRAME_ERROR)) {
            return NOK;
        }
        resultado += line + "\n";
    } while (inStream.ready());
    return resultado;
}
```

Figura 19: *Screen scraping* de la respuesta de la máquina.

5.5. Wait(output) propio

En este método, se quiere simular la función wait de x3270, pero en este caso, se esperará por una cadena en concreto comprobando de esta forma que el método de ejecución es el correcto. Para ello, se enviará un comando ASCII, que posteriormente se leerá con el comando anterior (apartado 5.4) y finalmente, se comprobará si existe la cadena en el string obtenido de la pantalla. Este método, se realizará durante el tiempo indicado en el timeout para los casos en los que no se encuentre la cadena pasada por parámetros.

```
@Override
public boolean esperarPantalla(String lineaABuscar)
    throws IOException, InterruptedException {
    Long maxTime = TIEMPO_EJEC_MAXIMO + System.currentTimeMillis();
    String resultado = "";
    do {
        enviarComando(COMANDO_ASCII);
        sleep(DELAY);
        resultado = obtenerRespuestaMaquina();
        if (resultado.contains(lineaABuscar)) {
            return true;
        }
    } while (maxTime > System.currentTimeMillis());

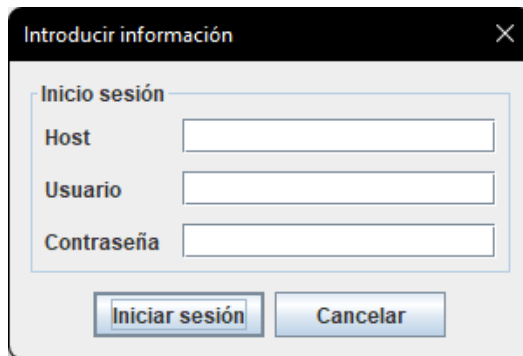
    return false;
}
```

Figura 20: Espera de una pantalla concreta.

6. Creación de la vista

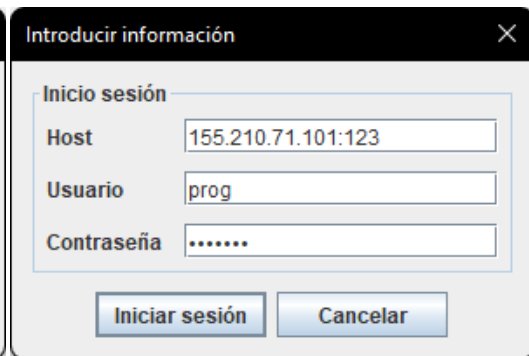
La aplicación vista implementada es una aplicación sencilla que simplemente tiene los botones necesarios para la comunicación con la aplicación legada, haciendo uso de los métodos implementados en las interfaces correspondientes.

Al ejecutar la aplicación, el usuario se encontrará con la ventana de conexión con el emulador MUSIC/SP. En esta ventana, se le pide al usuario la dirección IP del host, el usuario y la contraseña. En esta fase también se tratan los posibles errores que se pueden dar al realizar la conexión y el inicio de sesión.



The screenshot shows a window titled 'Introducir información' with a close button (X). Inside, there is a section titled 'Inicio sesión' containing three input fields: 'Host', 'Usuario', and 'Contraseña'. Below these fields are two buttons: 'Iniciar sesión' and 'Cancelar'.

Figura 21: Ventana de introducción de datos de conexión.



The screenshot shows the same 'Introducir información' window, but now the input fields are filled: 'Host' contains '155.210.71.101:123', 'Usuario' contains 'prog', and 'Contraseña' contains masked characters '.....'. The 'Iniciar sesión' and 'Cancelar' buttons remain at the bottom.

Figura 22: Ventana de introducción de datos de conexión con los datos empleados.

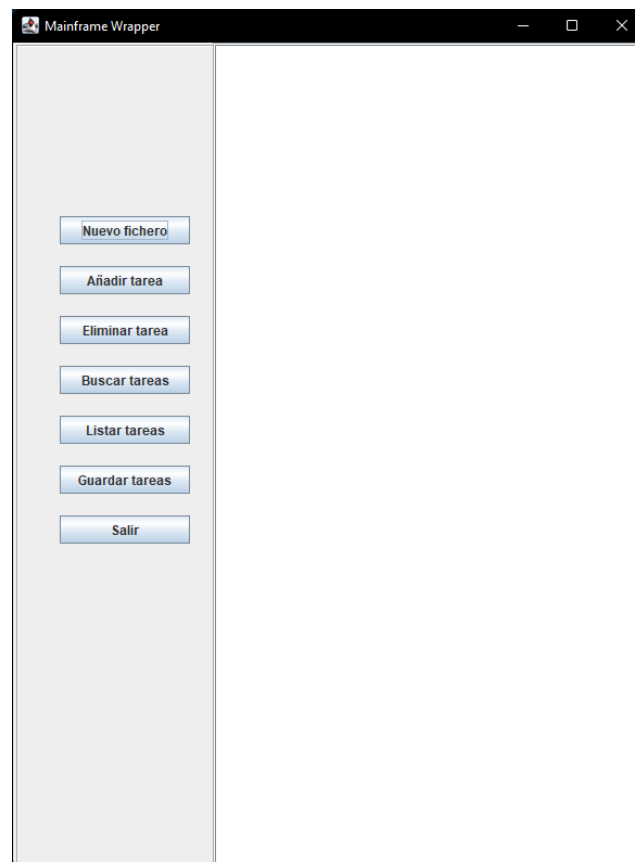


Figura 23: Menú principal de la aplicación.

La aplicación es muy sencilla. Dispone de unos botones que simulan las mismas acciones que las opciones del menú de la aplicación legada. A la derecha se encuentra

una zona blanca de texto donde se mostrarán las tareas que existen en el registro de la aplicación legada. Esta zona solamente se actualiza cuando el usuario interacciona con las correspondientes opciones.

Toda la implementación de la vista se realiza en su clase, respetando el diseño de la aplicación (figura 4).

7. Funcionamiento de la aplicación

En este apartado se detalla el funcionamiento de las distintas funciones de la aplicación moderna. En cada apartado, aparte de explicar el funcionamiento de la interfaz y sus consideraciones más importantes, se ha incluido el código del método principal para explicar algunos detalles de su implementación.

Como se ha mencionado anteriormente, es necesario que la aplicación moderna esté sincronizada con la aplicación legada. Para ello, cada función de la aplicación moderna indicará mediante ventanas emergentes al usuario cuando una acción se ha completado de forma correcta.

7.1. Nuevo fichero de tareas

La primera función de la aplicación legada es la creación de un nuevo fichero de tareas. El feedback que el usuario recibirá depende de la ejecución de la tarea en la aplicación legada.

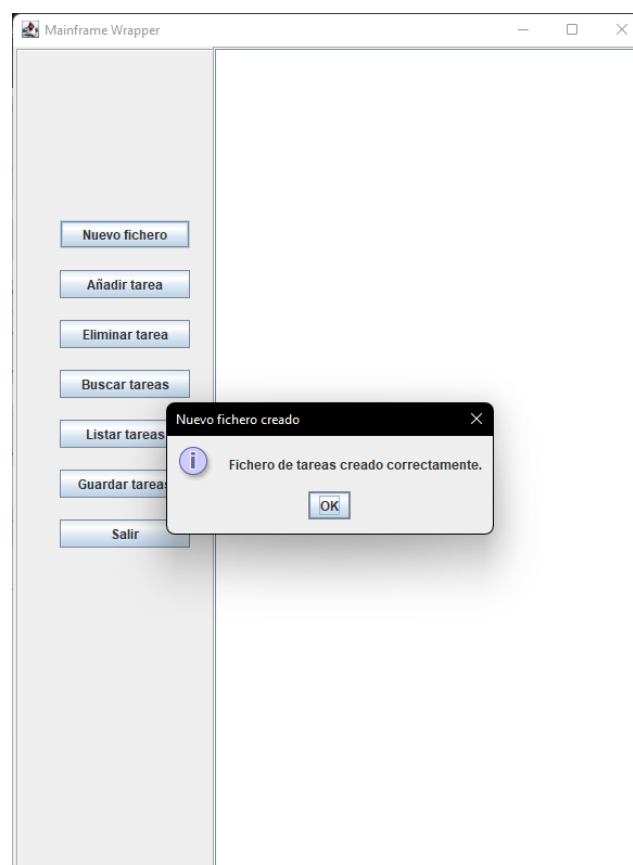


Figura 24: Ejecución corecta de la función "Nuevo fichero".

Cuando el usuario interacciona con el botón correspondiente a esta tarea, un oyente le indica a la aplicación que el usuario quiere crear un nuevo fichero de tareas, la cual ejecuta el método correspondiente.

```
public void nuevoFicheroTareas() throws IOException, InterruptedException {  
    vista.mostrarVentanaEspera();  
    if (!tasks2.nuevoFicheroTareas()) {  
        vista.notificarMensajeError(TITULO_ERROR_FICHERO_TAREAS,  
                                    MENSAJE_ERROR_FICHERO_TAREAS);  
    } else {  
        vista.notificarMensajeConfirmacion(  
            TITULO_CONFIRMACION_FICHERO_TAREAS,  
            MENSAJE_CONFIRMACION_FICHERO_TAREAS);  
    }  
}
```

Figura 25: Método de la aplicación para crear un nuevo fichero de tareas.

El método ejecuta el método de la clase TasksJob para crear un nuevo fichero y notifica al usuario del resultado de la operación.

```
@Override  
public boolean nuevoFicheroTareas()  
    throws IOException, InterruptedException {  
    if (mainframe.enviarString(NUEVO_FICHERO)) {  
        if (mainframe.enviarComando(Mainframe.COMANDO_ENTER)) {  
            if (enviarRespuestaConfirmacion() && ficheroTareasCreado()) {  
                return true;  
            }  
        }  
    }  
    return false;  
}
```

Figura 26: Método para crear un nuevo fichero de tareas en la aplicación legada.

El método que se encarga de la creación de un nuevo fichero de tareas en la aplicación legada tiene que realizar una serie de comunicaciones y sincronizaciones con la aplicación legada según el diagrama de la figura 10.

Para no utilizar demasiados anidamientos se han creado varios métodos auxiliares que se encargan de tareas específicas dentro del proceso de creación de un nuevo fichero.

7.2. Añadir tarea

La siguiente función de la aplicación legada es la creación de una nueva tarea. Es la tarea más compleja de implementar debido a que la aplicación legada requiere de 4 datos de la nueva tarea a añadir (id, nombre, descripción y fecha).

Cada uno de estos aspectos deben ser tratados para evitar errores en la aplicación legada y su cese de funcionamiento. Los identificadores deben ser secuencias de dígitos, el nombre debe tener entre 0 y 16 caracteres, la descripción debe tener entre 0 y 32 caracteres y la fecha debe tener el formato "DD MM AAAA".

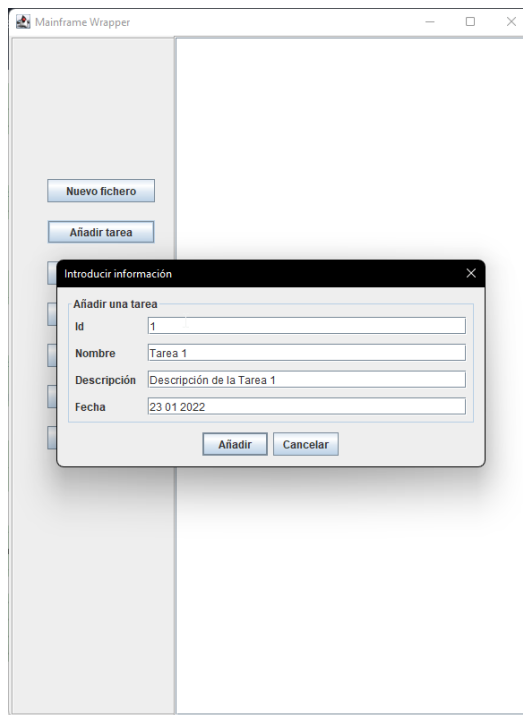


Figura 27: Campos del proceso de creación de una nueva tarea.

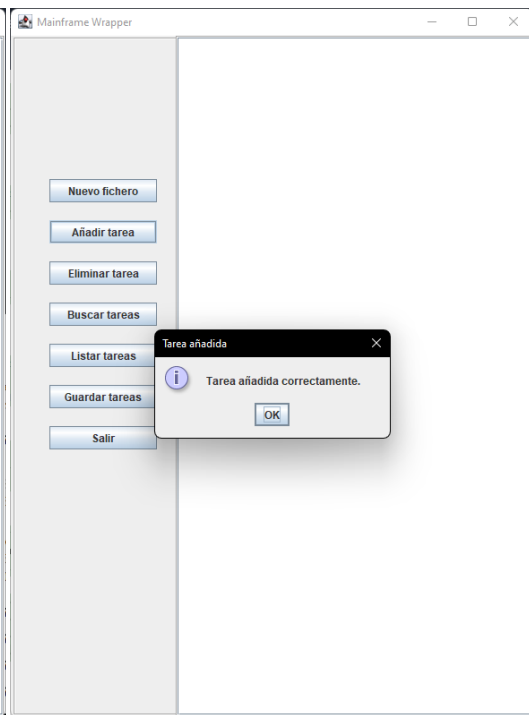


Figura 28: Mensaje de confirmación de la creación correcta de una nueva tarea.

Los errores más comunes en esta función son la introducción de un ID de tarea que no contenga únicamente dígitos y una fecha con formato incorrecto. En ambos casos, se han introducido mensajes informativos que notifiquen al usuario de los errores cometidos para que los solucione.

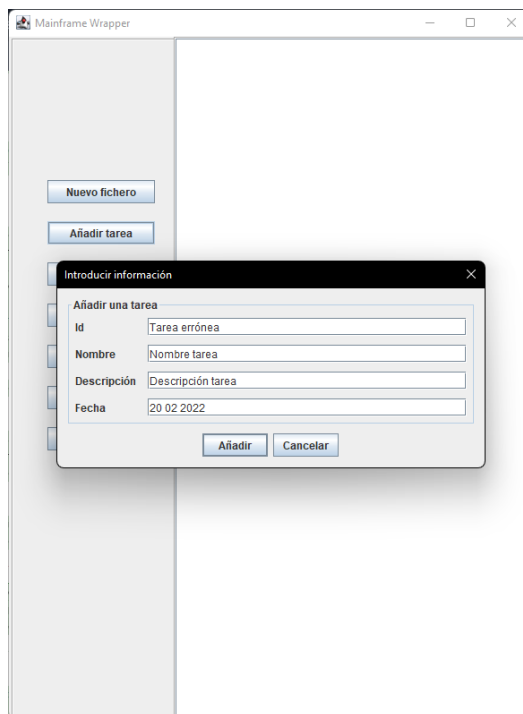


Figura 29: Campos de una tarea con ID incorrecto.

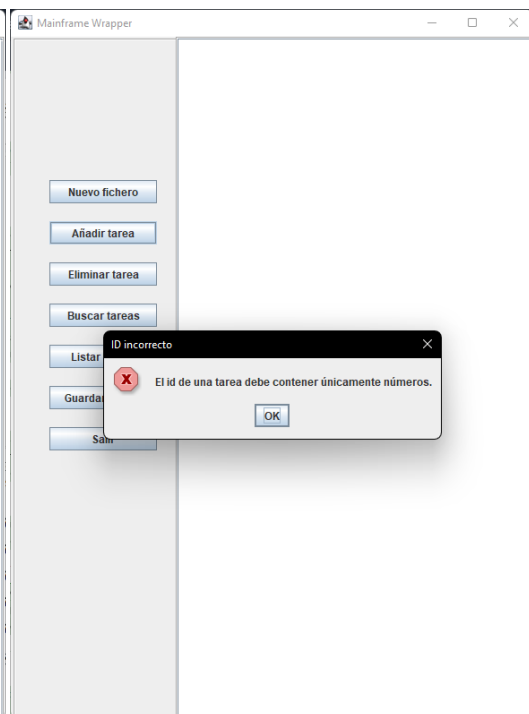


Figura 30: Mensaje de error de una tarea con ID incorrecto.

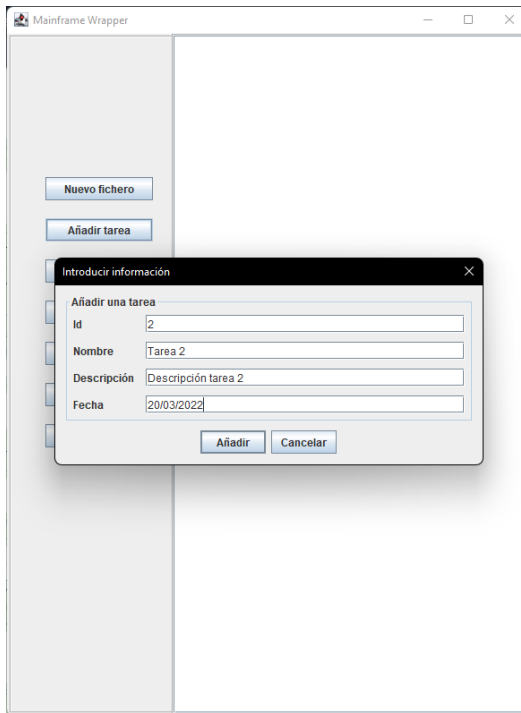


Figura 31: Campos de una tarea con fecha incorrecta.

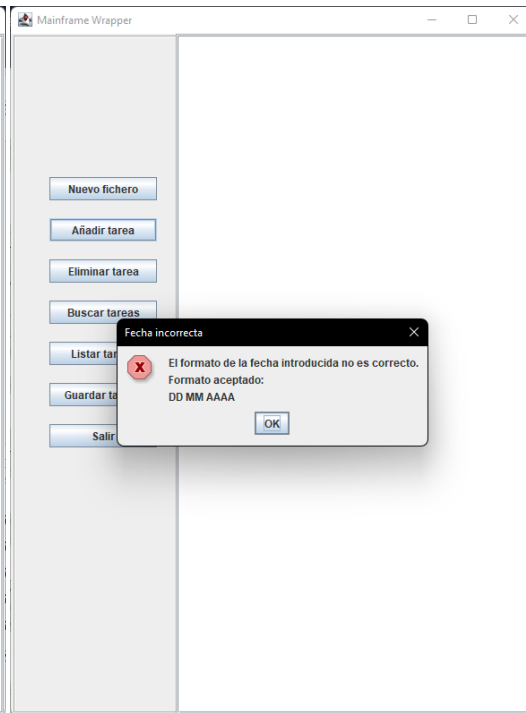


Figura 32: Mensaje de error de una tarea con fecha incorrecta.

Otros errores también tratados, pero no incluidos como capturas de pantalla son la introducción de un ID de tarea repetido, la introducción de un nombre de tarea con más de 16 caracteres o una descripción de más de 32 caracteres. Estos errores también tienen sus correspondientes mensajes de error.

Cuando el usuario interacciona con el botón de añadir tarea, el programa le pedirá introducir los datos y verificará si los datos son correctos o no. Si lo son, envía el evento que será tratado posteriormente en el método del main del programa.

```
public void anyadirTarea(Object obj)
    throws IOException, InterruptedException {
    vista.mostrarVentanaEspera();
    Tupla<Tupla, Tupla> tuplaTarea = (Tupla<Tupla, Tupla>) obj;
    Tupla<String, String> tuplaIdNombre = tuplaTarea.a;
    Tupla<String, String> tuplaDescFecha = tuplaTarea.b;

    TasksAPI.CODIGO_ERROR codigoAnyadir =
        tasks2.anyadirTarea(tuplaIdNombre.a, tuplaIdNombre.b,
            tuplaDescFecha.a, tuplaDescFecha.b);
    switch (codigoAnyadir) {
        case NOK:
            vista.notificarMensajeError(TITULO_ERROR,
                MENSAJE_ERROR_INESPERADO);
            break;
        case IDTAREA_INCORRECTO:
            vista.notificarMensajeError(TITULO_ERROR_ANYADIR_TAREA,
                MENSAJE_ERROR_ID_INCORRECTO);
            break;
        case OK:
            vista.notificarMensajeConfirmacion(
                TITULO_CONFIRMACION_ANYADIR_TAREA,
                MENSAJE_CONFIRMACION_TAREA_ANYADIDA);
            break;
    }
}
```

Figura 33: Método de la aplicación para añadir una nueva tarea.

Este método trata la información obtenida del oyente de la vista y llama al método de añadir tarea de la aplicación legada. En función del código que recibe de este método, notificará el error correspondiente al usuario.

```
@Override
public CODIGO_ERROR anyadirTarea(String idTarea, String nombreTarea,
                                String descripcionTarea, String fecha)
    throws IOException, InterruptedException {
    CODIGO_ERROR codigo = CODIGO_ERROR.NOK;

    if (existeIdTarea(idTarea)) {
        return CODIGO_ERROR.IDTAREA_REPETIDO;
    }

    if (mainframe.enviarString(ANYADIR)) {
        if (mainframe.enviarComando(Mainframe.COMANDO_ENTER)) {
            if (mainframe.esperarPantalla(PANTALLA_ANYADIR_TAREA)) {
                codigo = auxiliarEnviarDatosTarea(idTarea, nombreTarea,
                                                  descripcionTarea, fecha);
            }
        }
    }

    return codigo;
}
```

Figura 34: Método para añadir una nueva tarea en la aplicación legada.

Al igual que en el caso de la función de creación de un nuevo fichero (apartado 7.1), se ha visto necesaria la creación de métodos auxiliares para la comunicación de la aplicación moderna con la aplicación legada.

```
public CODIGO_ERROR auxiliarEnviarDatosTarea(
    String idTarea, String nombreTarea, String
    descripcionTarea,
    String fecha) throws IOException, InterruptedException {
    if (enviarIdTarea(idTarea) &&
        enviarNombreTarea(nombreTarea) &&
        enviarDescripcionTarea(descripcionTarea) &&
        enviarFechaTarea(fecha)) {
        if (esperarPantallaMenu()) {
            return CODIGO_ERROR.OK;
        }
    }

    return CODIGO_ERROR.NOK;
}
```

Figura 35: Método auxiliar para la inserción de tareas en la aplicación legada.

7.3. Eliminar tarea

La función de eliminación de tareas de la aplicación legada es relativamente sencilla. Le pide al usuario el ID de la tarea a eliminar, comprueba que el ID existe, y elimina la tarea correspondiente al ID introducido. Si el ID no existe, la aplicación devolverá el mensaje de error correspondiente.

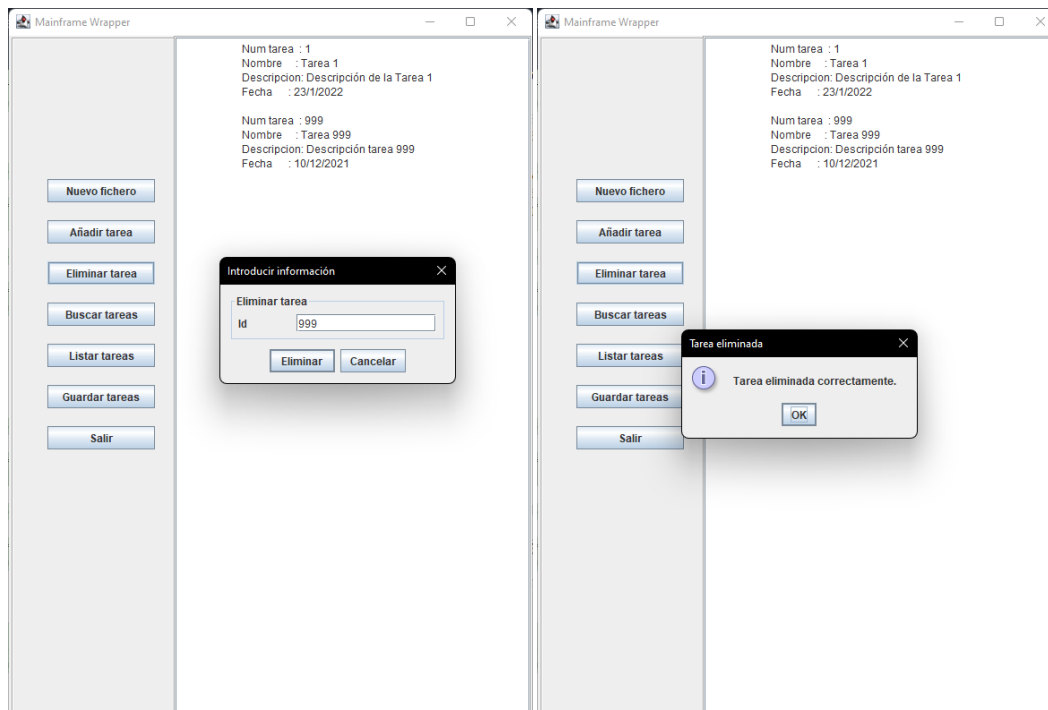


Figura 36: Campos del proceso de eliminación de una tarea

Figura 37: Mensaje de confirmación de la eliminación de la tarea

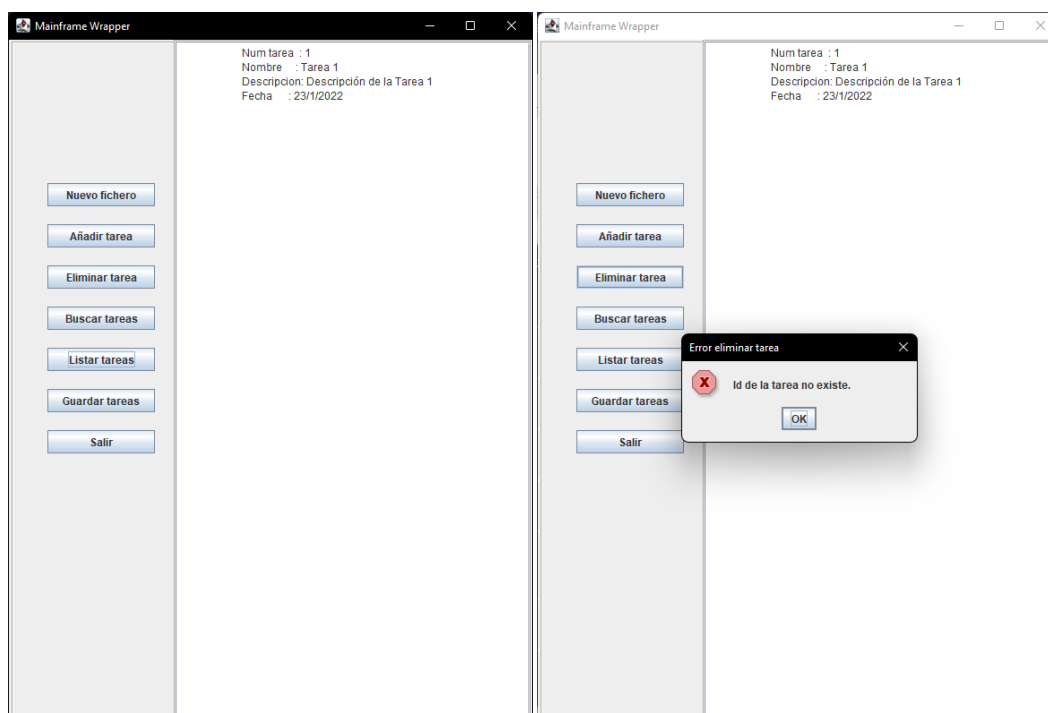


Figura 38: Confirmación de la eliminación de la tarea listando de nuevo

Figura 39: Mensaje de error de una tarea que no existe

En cuanto a la implementación, el método de la eliminación necesita dos métodos auxiliares (figura 41 y ??).

```
@Override
public CODIGO_ERROR eliminarTarea(String idTarea)
    throws IOException, InterruptedException {
    CODIGO_ERROR codigo = CODIGO_ERROR.NOK;

    if (esperarPantallaEliminar()) {
        if (enviarIdTarea(idTarea)) {
            codigo = auxiliarEliminarTarea();
        }
    }
    return codigo;
}
```

Figura 40: Método para la eliminación de tareas.

```
private boolean esperarPantallaEliminar()
    throws IOException, InterruptedException {
    if (mainframe.enviarString(ELIMINAR)) {
        if (mainframe.enviarComando(MainframeAPI.COMANDO_ENTER)) {
            if (mainframe.esperarPantalla(PANTALLA_ELIMINAR_TAREA)) {
                return true;
            }
        }
    }
    return false;
}
```

Figura 41: Primer método auxiliar de la función de eliminar tareas. Espera que la aplicación legada se encuentre en la pantalla de eliminar tareas.

```
public CODIGO_ERROR auxiliarEliminarTarea()
    throws IOException, InterruptedException {
    if (mainframe.esperarPantalla(PANTALLA_TAREA_NO_ENCONTRADA)) {
        if (esperarPantallaMenu()) {
            return CODIGO_ERROR.IDTAREA_INCORRECTO;
        }
    } else if (esperarConfirmacionEliminar()) {
        return CODIGO_ERROR.OK;
    }
    return CODIGO_ERROR.NOK;
}
```

Figura 42: Segundo método auxiliar de la función de eliminar tareas. Verifica si existe la tarea en la aplicación legada. La elimina si existe o devuelve error si no.

7.4. Buscar tareas

Para el caso de buscar tarea, se seguirá el mismo procedimiento que para eliminar, pasando como parámetro de búsqueda una fecha. En caso de éxito mostrando la lista en el panel de la derecha y en caso de no haber ninguna tarea, indicándolo mediante una ventana emergente.

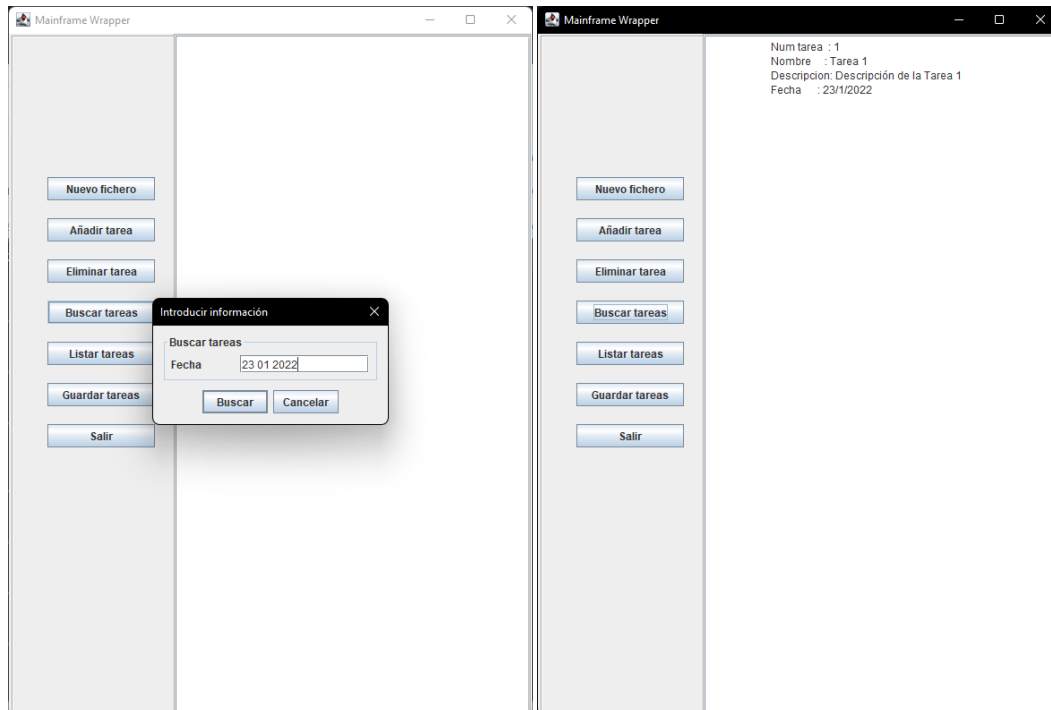


Figura 43: Campos del proceso de buscar
Figura 44: Resultado de buscar tareas co-
rrecto

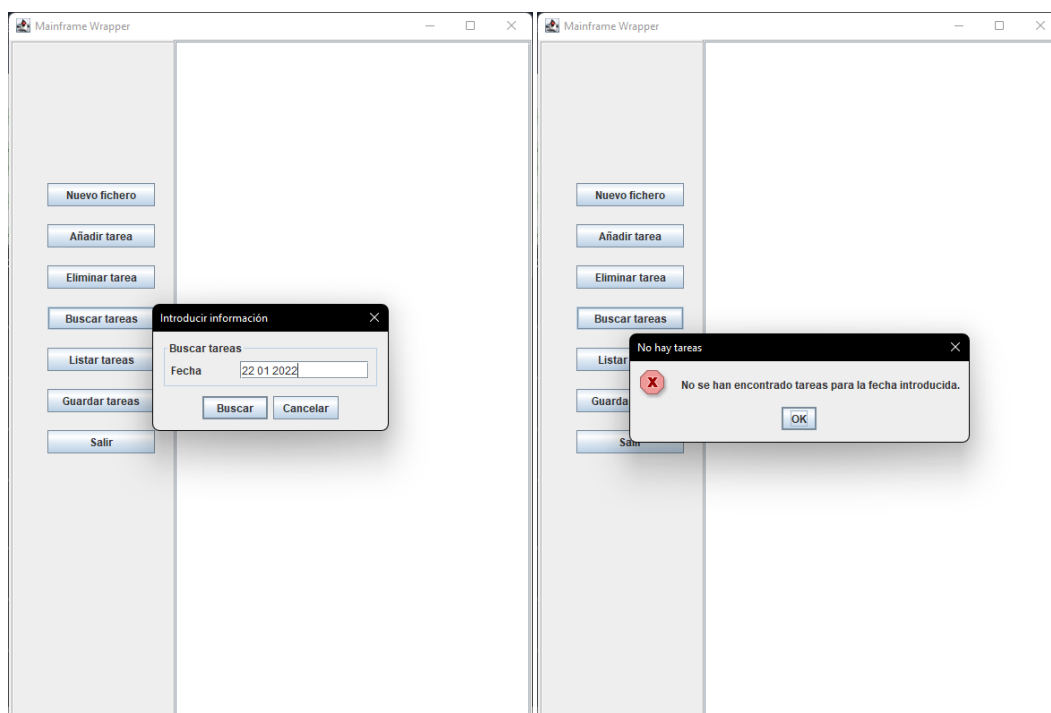


Figura 45: Proceso de búsqueda de otras
Figura 46: Mensaje de error para fechas
no válidas

```
@Override
public List<Tarea> buscarTareas(String fecha)
    throws IOException, InterruptedException {
    List<Tarea> tareas = new ArrayList();
    if (esperarPantallaBuscar()) {
        if (enviarFechaTarea(fecha)) {
            if (empezarLecturaTareas()) {
                tareas = obtenerListaTareas();
                if (esperarPantallaMenu()) {
                    return tareas;
                }
            }
        }
    }
    return tareas;
}
```

Figura 47: Método para buscar tareas en la aplicación legada.

Este método necesita 3 métodos auxiliares `esperarPantallaBuscar()`, `enviarFechaTarea()` y `empezarLecturaTareas()` para buscar y obtener las tareas de la aplicación legada.

El método principal de la función de búsqueda realiza un *screen scraping* de la información que devuelve el terminal s3270. De esta información se obtienen mediante parseo el id, nombre, descripción y fecha de las tareas encontradas por la aplicación legada. Posteriormente, se le muestra al usuario la información en la interfaz de la aplicación moderna.

```
**SEARCH TASK**
DATE(DD MM AA):
?
10 12 12
TASK NUMBER: 2
NAME       : 3
DESCRIPTION: sss
DATE       : 10/12/12

TASK NUMBER: 4
NAME       : sd
DESCRIPTION: sd
DATE       : 10/12/12

**END**

**PRESS ENTER TO CONTINUE**
?
```

Figura 48: Búsqueda de tareas en el mainframe

7.5. Listar tareas

Para el caso de listar, se realizará el mismo proceso que para buscar, pero en este caso no será necesario introducir ningún parámetro, por lo que únicamente, se mostrará un mensaje en caso de que no haya ninguna tarea en el mainframe o se mostrará la lista de tareas en caso de éxito.

7.6. Salir

La última función de la aplicación moderna es la de salida y terminación del programa. Esta función permitirá al usuario elegir si quiere guardar los cambios que ha realizado o no y terminará la conexión con la aplicación legada.

En el caso de que el usuario desea guardar los cambios, se realizará de forma implícita una llamada a la función de guardado de la aplicación legada.

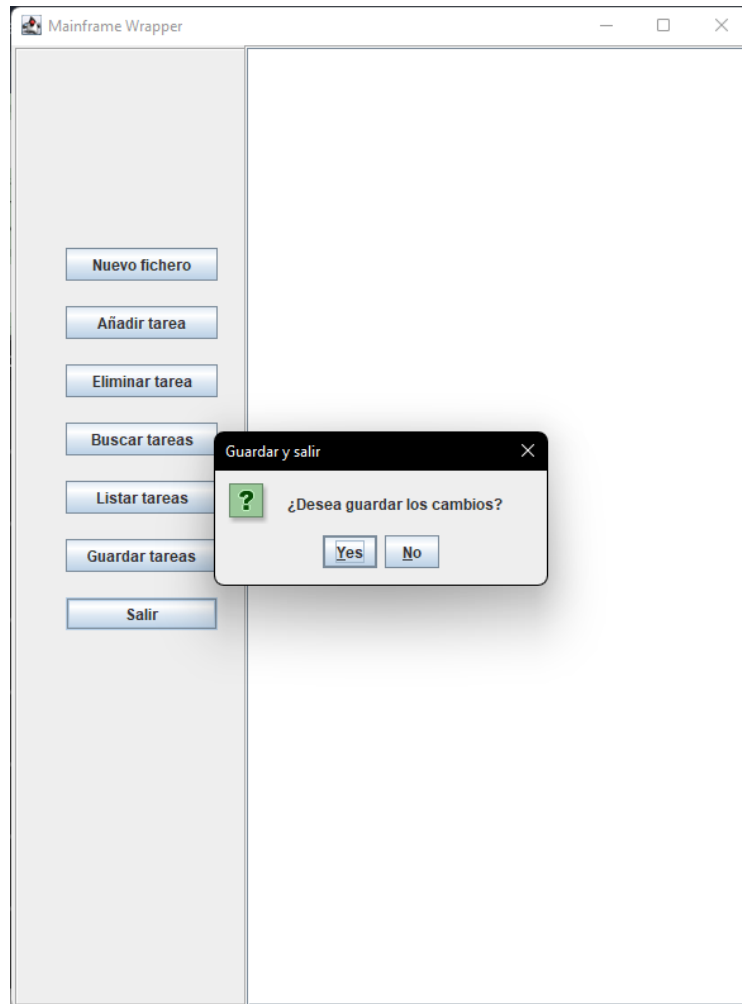


Figura 49: Función salir y guardar de la aplicación moderna.

Cuando el usuario interacciona con el botón de salir, se le muestra al usuario la ventana con la elección de guardado. Esta respuesta es enviada por el oyente al manejador de eventos. Posteriormente, la respuesta será tratada en la clase *TasksJob*.

```
public void salir(String respuesta) throws IOException, InterruptedException {  
    tasks2.salir(respuesta);  
    emulador.logout();  
    System.exit(status: 0);  
}
```

Figura 50: Método de la aplicación para salir y guardar.

El método primero llama al método de salida de la aplicación legada, pasándole la respuesta del usuario como argumento. Posteriormente, realiza la desconexión con el proceso s3270, y finalmente, realiza la terminación de la aplicación.

```
@Override
public boolean salir(String guardarTareas)
    throws IOException, InterruptedException {
    if (esperarPantallaSalir()) {
        if (realizarSalida()) {
            return true;
        } else if (esperarPantallaGuardar(guardarTareas)) {
            if (realizarSalida()) {
                return true;
            }
        }
    }
    return false;
}
```

Figura 51: Método de la aplicación legada para salir y guardar.

El método para la función de salida de la aplicación legada trata la respuesta para el guardado de las modificaciones, y luego realiza los pasos necesarios para primero cerrar la aplicación legada y luego salir del emulador MUSIC/SP.

Para ello, se han tenido que implementar varios métodos auxiliares para encargarse de los distintos procesos de comunicación con la aplicación legada y el emulador.

El primer método simplemente manda a la aplicación legada el comando para la salida y espera a que esta esté en la pantalla de salida antes de proceder.

```
private boolean esperarPantallaSalir()
    throws IOException, InterruptedException {
    if (mainframe.enviarString(MainframeAPI.COMANDO_EXIT)) {
        if (mainframe.enviarComando(MainframeAPI.COMANDO_ENTER)) {
            return true;
        }
    }
    return false;
}
```

Figura 52: Método para esperar que la aplicación legada se encuentre en la pantalla de salida.

Después, si el usuario ha decidido no guardar los cambios, el programa realiza la desconexión con el emulador.

```
private boolean realizarSalida()
    throws IOException, InterruptedException {
    if (mainframe.esperarPantalla(MENSAJE_SALIDA)) {
        if (mainframe.enviarComando(MainframeAPI.COMANDO_ENTER)) {
            return true;
        }
    }
    return false;
}
```

Figura 53: Método para realizar la salida y desconexión.

Si en cambio el usuario ha decidido guardar cambios, la aplicación moderna espera a que la aplicación legada guarde los cambios y se encuentre en la pantalla de confirmación de guardado. Una vez realizado el guardado, la aplicación realiza la salida del programa.

```
private boolean esperarPantallaGuardar(String guardarTareas)
    throws IOException, InterruptedException {
    if (mainframe.esperarPantalla(PANTALLA_GUARDAR_TAREAS)) {
        if (mainframe.enviarString(guardarTareas)) {
            if (mainframe.enviarComando(MainframeAPI.COMANDO_ENTER)) {
                return true;
            }
        }
    }
    return false;
}
```

Figura 54: Método para esperar que la aplicación legada realice el guardado de los cambios

8. Conclusiones

Para finalizar, en la práctica se ha podido comprobar el funcionamiento de un sistema legado, realizando sobre el una tarea de mantenimiento, en este caso una encapsulación, debido a que de esta forma la tarea de trabajar con la aplicación legada será más cómoda y con una interfaz de usuario actualizada.

Para la realización de la encapsulación, se tuvo que realizar un estudio previo del sistema legado, obteniendo de esta forma su funcionalidad y el proceso que había que seguir, para poder realizar cada una de las tareas por separado, quedando esto especificado en los diagramas de secuencia (apartado 4.2).

Referencias

- [1] T. x3270 Wiki, *Category:S3270 actions*, abr. de 2020. dirección: https://x3270.miraheze.org/wiki/Category:S3270_actions.