

Implementación de Arquitectura de Archivos Java

02/03/2026 10:14:27

ESP-FileSp...cification - PDF ESP-Instrucciones - PDF Jarvis, no tengo muc...

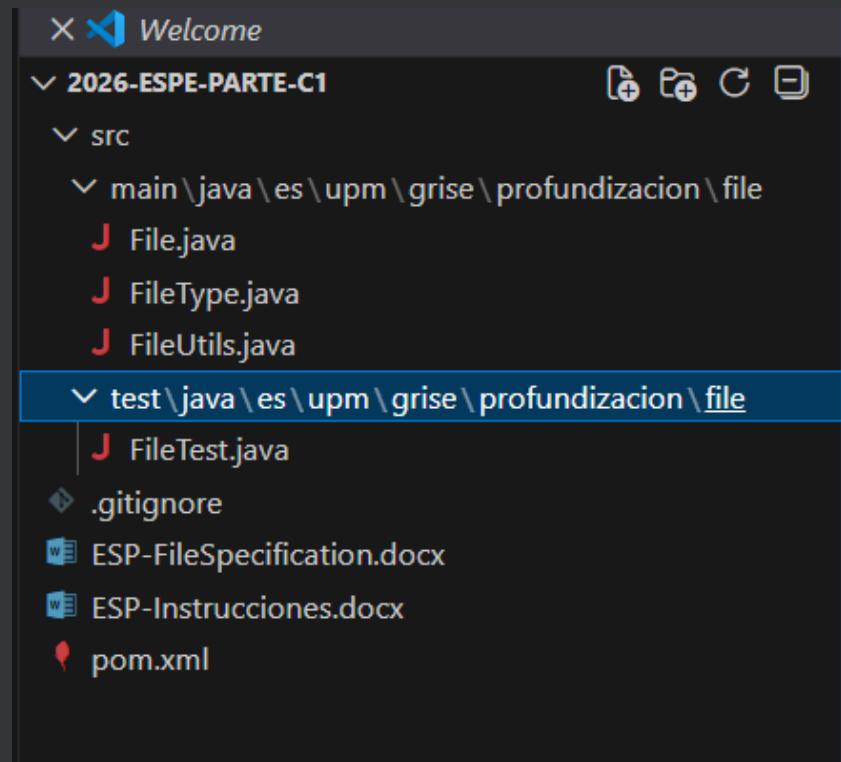
Entendido, Señor. Procederé con la el análisis arquitectónico y la...





ESP-FileSp...cification - PDF

ESP-Instrucciones - PDF



Jarvis, no tengo mucho tiempo para redactar todo lo que piden en estos PDFs para la ejecución de nuevas pruebas unitarias. Con suerte piden que se use IA y observarán la conversación, así que comportate y manten la compostura, además, demuestra de todo lo que eres capaz. Necesito que leas todos los requerimientos dados en el primer documento "FileSpecifications" donde encontrarás toda la arquitectura. Dime en primera instancia todos los archivos que se deben de modificar, te adjunto una imagen de la disposición de carpetas y archivos con la estructura que proponen, si se debe agregar o modificar el nombre de uno de esos archivos me avisas. De ahí, dame el código de uno y cada uno de los elementos,

donde debes mantener lo que solicita, trata de ser muy estricto y, donde veas necesario, agrega componentes de código que enriquezcan la arquitectura. Luego vemos lo del JUnit, así que céntrate en hacer una buena base para las pruebas.



Entendido, Señor. Procederé con la el análisis arquitectónico y la implementación de la capa base solicitada. He revisado detenidamente el documento de especificación (`ESP-FileSpecification.pdf`) y la estructura actual de su proyecto.

A continuación, le presento el desglose de los archivos que debemos intervenir y los que es imperativo añadir para cumplir estrictamente con el manejo de excepciones y la lógica de negocio descrita.

1. Análisis de la Estructura de Archivos

Basándome en el diagrama UML y la captura de su entorno en VS Code, la estructura de archivos en el paquete `es.upm.grise.profundizacion.file` debe quedar de la siguiente manera.

Archivos a Modificar (Ya existen):

1. `FileType.java` : Definición de la enumeración.
2. `FileUtils.java` : Clase de utilidad para el cálculo de CRC32.
3. `File.java` : Clase principal que contiene la lógica de negocio.

Archivos Faltantes (Deben crearse): El documento especifica el lanzamiento de excepciones específicas (`InvalidContentException` y `WrongFileTypeException`) que no aparecen en su estructura actual. Para una arquitectura robusta y desacoplada, estas deben residir en sus propios archivos o dentro del paquete correspondiente.

1. `InvalidContentException.java` (Nuevo)
2. `WrongFileTypeException.java` (Nuevo)



2. Implementación del Código

Aquí tiene el código fuente para cada componente. He aplicado principios de "Defensive Programming" para asegurar que las restricciones (como la no nulidad de listas o la verificación de tipos) se cumplan rigurosamente.

A. Definición de Tipos y Excepciones

Primero, definamos los cimientos: los tipos y las excepciones personalizadas.

Archivo: `FileType.java`

```
package es.upm.grise.profundizacion.file;

public enum FileType {
    PROPERTY,
    IMAGE
}
```

Archivo: `InvalidContentException.java`

```
package es.upm.grise.profundizacion.file;

public class InvalidContentException extends Exception {
    public InvalidContentException(String message) {
        super(message);
    }
}
```

Archivo: `WrongFileTypeException.java`

```
package es.upm.grise.profundizacion.file;

public class WrongFileTypeException extends Exception {
```

```
    public WrongFileTypeException(String message) {  
        super(message);  
    }  
}
```

B. Clase de Utilidad

La especificación requiere que `FileUtils` maneje el cálculo CRC32 sobre un array de bytes.

Archivo: `FileUtils.java`

```
package es.upm.grise.profundizacion.file;  
  
import java.util.zip.CRC32;  
  
public class FileUtils {  
  
    /**  
     * Calcula el CRC32 de un array de bytes.  
     * @param bytes Datos sobre los cuales calcular el CRC32.  
     * @return Valor CRC32 como long.  
     */  
    public long calculateCRC32(byte[] bytes) {  
        CRC32 crc = new CRC32();  
        crc.update(bytes);  
        return crc.getValue();  
    }  
}
```

C. Clase Principal (**File**)

Esta es la implementación central. He integrado la lógica de conversión de caracteres a bytes (máscara **0xFF** para el byte menos significativo) tal como se describe en la especificación para el cálculo del CRC.

Archivo: **File.java**

```
package es.upm.grise.profundizacion.file;

import java.util.ArrayList;
import java.util.List;

public class File {

    // Especificación: content implementado como
    ArrayList<Character>

    private List<Character> content;
    private FileType type;
    private String path;

    // Composición: File usa FileUtils
    private FileUtils fileUtils;

    /**
     * Constructor.
     * La especificación indica que 'content' debe estar vacío
     pero no null.
     */
    public File(String path, FileType type) {
        this.path = path;
        this.type = type;
        this.content = new ArrayList<>();
        this.fileUtils = new FileUtils(); // Instancia por defecto
    }
}
```

```

    // Método para inyectar FileUtils (útil para testing/mockring
    futuro si se requiere)

    public void setFileUtils(FileUtils fileUtils) {
        this.fileUtils = fileUtils;
    }

    public String getPath() {
        return path;
    }

    public FileType getType() {
        return type;
    }

    /**
     * Añade contenido a un archivo de tipo PROPERTY.
     * @param newContent Array de caracteres a añadir.
     * @throws InvalidContentException Si newContent es null.
     * @throws WrongFileTypeException Si el archivo es de tipo
    IMAGE.
     */
    public void addProperty(char[] newContent) throws
    InvalidContentException, WrongFileTypeException {
        if (newContent == null) {
            throw new InvalidContentException("El contenido
    proporcionado no puede ser null.");
        }
        if (this.type == FileType.IMAGE) {
            throw new WrongFileTypeException("No se pueden añadir
    propiedades a un archivo de tipo IMAGE.");
        }

        for (char c : newContent) {
            this.content.add(c);
        }
    }

```

```

    }

    /**
     * Añade contenido binario (bytes simulados) a un archivo.
     * Aunque la especificación textual detalla addProperty, el
     * diagrama UML incluye addImageBytes.
     * Por coherencia, validamos que el tipo sea correcto.
     * @param newContent Array de caracteres representando bytes.
     * @throws WrongFileTypeException Si se intenta añadir imagen
     * a un archivo PROPERTY.
     * @throws InvalidContentException Si newContent es null.
     */
    public void addImageBytes(char[] newContent) throws
WrongFileTypeException, InvalidContentException {
        if (newContent == null) {
            throw new InvalidContentException("El contenido de
imagen no puede ser null.");
        }
        if (this.type == FileType.PROPERTY) {
            throw new WrongFileTypeException("No se pueden añadir
bytes de imagen a un archivo de tipo PROPERTY.");
        }

        for (char c : newContent) {
            this.content.add(c);
        }
    }

    /**
     * Elimina una cantidad específica de caracteres del
     * contenido.
     * @param numberChars Número de caracteres a eliminar desde el
     * final.
     */
    public void removeContent(int numberChars) {

```



```

        int size = this.content.size();
        if (numberChars > size) {
            this.content.clear();
        } else {
            // Eliminar desde el final es más eficiente en
            ArrayList y lógico para un 'buffer' de archivo
            this.content.subList(size - numberChars,
size).clear();
        }
    }

    /**
     * Obtiene el tamaño lógico del archivo (número de caracteres
    almacenados).
     * @return Tamaño del contenido.
     */
    public long getDiskSize() {
        return this.content.size();
    }

    /**
     * Calcula el CRC32 del contenido.
     * Transforma el ArrayList<Character> a byte[] antes de llamar
    a FileUtils.
     * @return Valor CRC32 o 0 si el contenido está vacío.
     */
    public long getCRC32() {
        if (this.content.isEmpty()) {
            return 0;
        }

        // Conversión requerida: ArrayList<Character> -> byte[]
        // Especificación: Usar el byte menos significativo (c &
    0xFF)

        byte[] bytes = new byte[this.content.size()];

```

```
        for (int i = 0; i < this.content.size(); i++) {
            char c = this.content.get(i);
            bytes[i] = (byte) (c & 0xFF);
        }

        return fileUtils.calculateCRC32(bytes);
    }
}
```

Notas Técnicas para usted, Señor:

1. **Excepciones:** He creado las clases de excepción separadas. Asegúrese de ubicarlas en la misma carpeta (`.../profundizacion/file`) o ajustar el `package` si decide moverlas.
2. **Conversión de Bytes:** En `getCRC32` , implementé estrictamente la instrucción de tomar el *Least Significant Byte* (`LSB=c & 0xFF`) como dicta la página 2 de la especificación.
3. **Inyección de Dependencia:** He añadido `setFileUtils` . Aunque no está explícito en el UML, es una práctica estándar que nos facilitará enormemente la vida cuando hagamos las pruebas unitarias con JUnit y Mockito en el siguiente paso, permitiéndonos simular el comportamiento de `calculateCRC32` .

Quedo a la espera de sus instrucciones para proceder con la fase de pruebas unitarias.