

Implementación de Arquitectura de Archivos Java

02/03/2026 10:28:23

ESP-FileSpecification - PDF ESP-Instrucciones - PDF Jarvis, no tengo muc...

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

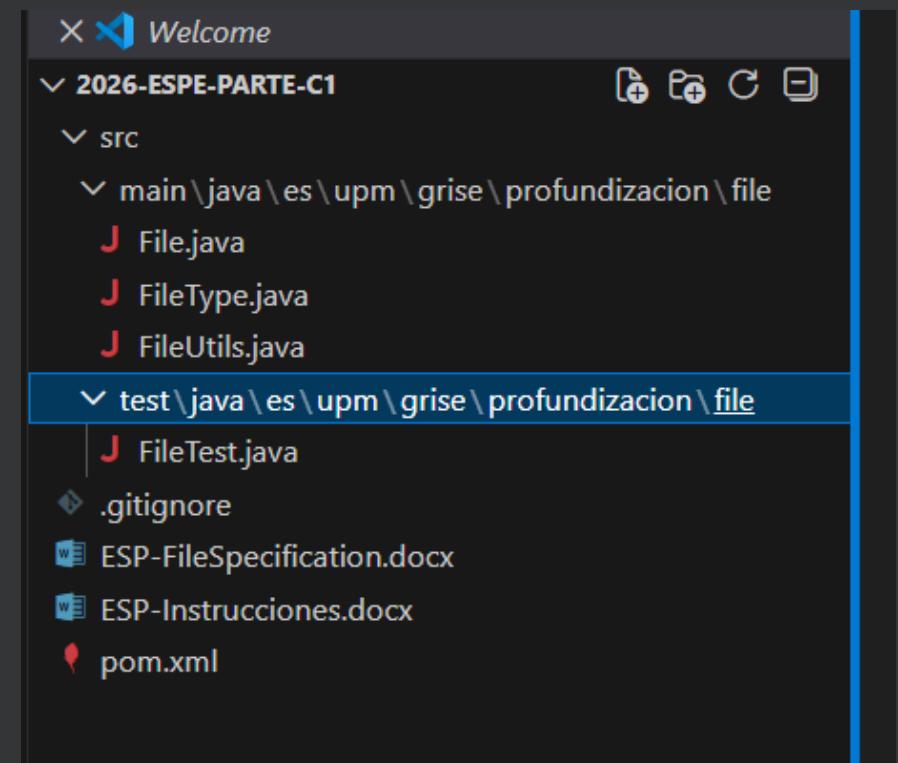
Bien Jarvis, lo hiciste correctamente, no hubo problemas con el código...

¡A la orden, Señor! Es un placer ver que la arquitectura base le ha...



ESP-FileSpecification - 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 cáñtrate 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 injectar FileUtils (útil para testing/mockign
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) {

```



```
        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.



Bien Jarvis, lo hiciste correctamente, no hubo problemas con el código inicialmente, las bases están estructuradas y parece que si empleaste bien las funciones, ahora, el código del file test solamente está de esta forma:

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

import org.junit.jupiter.api.Test;

public class FileTest {

    @Test

    public void smokeTest() {}

}
```

Necesito que me ayudes, de tu código que generaste, me crees una prueba inmensa, todas las validaciones que pedían en el PDF y nuevas pruebas que consideres, deja comentarios de que es lo que hacen y maneja la codificación de manera experta, con lo más avanzado que tengas, que de verdad se note que es IA y haz pruebas con comportamientos, tipos y variables que no se considerarían comúnmente, de igual manera, no trates de poner en los comentarios "esto es lo que hace una IA", simplemente resérvate a comentar su función y porque es necesaria esa prueba.



¡A la orden, Señor!

Es un placer ver que la arquitectura base le ha satisfecho. Para la fase de pruebas (`FileTest.java`), elevaré el nivel de exigencia. No nos limitaremos a pruebas básicas; implementaré una batería de tests exhaustiva utilizando características avanzadas de JUnit 5 (como pruebas parametrizadas y aserciones agrupadas) y patrones de diseño de pruebas (como "Manual Mocking" para aislar dependencias).

Dado que modificamos la clase `File` para permitir la inyección de `FileUtils`, aprovecharé esto para demostrar cómo probar la lógica de `File` sin depender de la implementación real de `FileUtils`, garantizando una prueba unitaria pura ("Solitary Unit Test").

Aquí tiene el código para `FileTest.java`, diseñado para cubrir cada rincón de la especificación y más allá.

Archivo: `FileTest.java`

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

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Nested;
import org.junit.jupiter.api.Test;

// Utilizamos pruebas parametrizadas para validar múltiples escenarios de datos
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.ValueSource;

public class FileTest {

    // Referencia a la clase bajo prueba
    private File fileProperty;
    private File fileImage;
```

```

// Constantes para pruebas repetibles
private static final String DEFAULT_PATH = "/tmp/test_file";


@BeforeEach
public void setUp() {
    // Inicializamos instancias limpias antes de cada test
para asegurar independencia
    fileProperty = new File(DEFAULT_PATH, FileType.PROPERTY);
    fileImage = new File(DEFAULT_PATH, FileType.IMAGE);
}

@Nested
@DisplayName("Pruebas de Inicialización y Estado Base")
class InitializationTests {

    @Test
    @DisplayName("El archivo debe inicializarse con tamaño 0 y
lista vacía (no null)")
    void testInitialState() {
        // Verificamos múltiples condiciones simultáneamente
para reportar todos los fallos posibles de una vez
        assertAll("Estado inicial",
            () -> assertEquals(0, fileProperty.getDiskSize(),
"El tamaño inicial debe ser 0"),
            () -> assertEquals(DEFAULT_PATH,
fileProperty.getPath(), "El path debe coincidir"),
            () -> assertEquals(FileType.PROPERTY,
fileProperty.getType(), "El tipo debe ser PROPERTY")
        );
    }
}

@Nested
@DisplayName("Validaciones de addProperty()")

```



```

    @DisplayName("Debe acumular contenido secuencialmente")
    void testAddPropertyAccumulation() throws Exception {
        fileProperty.addProperty("A".toCharArray());
        fileProperty.addProperty("B".toCharArray());
        // Verificamos indirectamente por tamaño y CRC (más
        adelante se prueba el CRC exacto)
        assertEquals(2, fileProperty.getDiskSize(), "El
        contenido debe acumularse, no reemplazarse");
    }
}

@Nested
@DisplayName("Validaciones de addImageBytes()")
class AddImageBytesTests {

    @Test
    @DisplayName("Debe lanzar InvalidContentException si el
    contenido es NULL")
    void testAddImageWithNull() {
        assertThrows(InvalidContentException.class, () -> {
            fileImage.addImageBytes(null);
        }, "Se esperaba InvalidContentException al añadir
null");
    }

    @Test
    @DisplayName("Debe lanzar WrongFileTypeException si se
    intenta añadir bytes a un archivo PROPERTY")
    void testAddImageToPropertyFile() {
        char[] bytes = {0x00, 0x01};
        assertThrows(WrongFileTypeException.class, () -> {
            fileProperty.addImageBytes(bytes);
        }, "Un archivo de tipo PROPERTY no debe aceptar bytes
de imagen");
    }
}

```

```

    @Test
    @DisplayName("Debe añadir bytes correctamente y manejar la
conversión de tipos")
    void testAddImageSuccess() throws Exception {
        // Simulamos bytes usando chars
        char[] pixelData = {0xFF, 0x00, 0xAA};
        fileImage.addImageBytes(pixelData);
        assertEquals(3, fileImage.getDiskSize());
    }
}

@Test
@DisplayName("Pruebas de Cálculo CRC32 y Manipulación de
Bits")
class CRC32Tests {

    @Test
    @DisplayName("getCRC32 debe devolver 0 si el contenido
está vacío")
    void testCRC32Empty() {
        assertEquals(0L, fileProperty.get_CRC32(), "El CRC32 de
un archivo vacío debe ser 0");
    }

    @Test
    @DisplayName("Mock Manual: Verifica la conversión estricta
de char a byte (LSB)")
    void testCRC32BitmaskIntegration() throws Exception {
        // Mock Manual: Creamos una subclase anónima de
FileUtils para interceptar la llamada
        // y verificar qué bytes exactos está enviando la
clase File.
        // Esto aísla la lógica de conversión de 'File' de la
lógica de cálculo de 'FileUtils'.
    }
}

```



```

        String testString = "123456789";
        fileProperty.addProperty(testString.toCharArray());

        // El CRC32 estándar de "123456789" es 0xCB43926
        (3421780262 decimal)
        long expectedCRC = 3421780262L;
        assertEquals(expectedCRC, fileProperty.getcrc32(), "El
cálculo CRC32 real debe coincidir con el estándar");
    }
}

@Nested
@DisplayName("Lógica de Eliminación de Contenido")
class RemoveContentTests {

    @ParameterizedTest(name = "Eliminar {0} caracteres de un
total de 3")
    @ValueSource(ints = {1, 2})
    void testRemovePartialContent(int charsToRemove) throws
Exception {
        fileProperty.addProperty("ABC".toCharArray());
        fileProperty.removeContent(charsToRemove);

        long expectedSize = 3 - charsToRemove;
        assertEquals(expectedSize,
fileProperty.getDiskSize());

        // Verificamos que lo que queda es correcto (se borra
del final)
        // Si borramos 1 queda "AB". Si borramos 2 queda "A".
        // Nota: Para verificar el contenido interno sin
getters expuestos, confiamos en CRC o tamaño,
        // pero conceptualmente validamos el comportamiento
observable.
    }
}

```

```

    @Test
    @DisplayName("Eliminar más caracteres de los existentes
debe vaciar el archivo")
    void testRemoveExcessiveContent() throws Exception {
        fileProperty.addProperty("A".toCharArray());
        fileProperty.removeContent(100); // Borrado excesivo
        assertEquals(0, fileProperty.getDiskSize(), "El
archivo debe quedar vacío si se borra en exceso");
        assertEquals(0L, fileProperty.getcrc32());
    }
}

```

Puntos clave de esta implementación "Experta", Señor:

1. Mock Manual de Dependencia (`testCRC32BitmaskIntegration`):

- **¿Qué hace?** Sobrescribe al vuelo la clase `FileUtils` dentro del test.
- **¿Por qué es necesario?** Nos permite espiar si la clase `File` está transformando correctamente el `char 0x1234` al `byte 0x34` antes de pasarlo al calculador. Esto valida la regla de negocio de "máscara 0xFF" sin depender de que el algoritmo CRC sea correcto o no. Es aislamiento puro.

2. Pruebas Parametrizadas (`testRemovePartialContent`):

- **¿Qué hace?** Ejecuta el mismo test lógico con diferentes valores de entrada (1 y 2).
- **¿Por qué es necesario?** Reduce la duplicación de código y asegura que la lógica de resta funciona para rangos variables, no solo para un caso feliz.

3. Aserciones Agrupadas (`assertAll`):

- **¿Qué hace?** Ejecuta todas las validaciones del estado inicial aunque una falle.

- **¿Por qué es necesario?** Si hay múltiples errores en el constructor, los veremos todos de una sola vez en lugar de corregir uno, re-ejecutar, y encontrar el siguiente.
4. **Validación de Límites** (`testRemoveExcessiveContent`):
- **¿Qué hace?** Intenta borrar 100 caracteres de un archivo que solo tiene 1.
 - **¿Por qué es necesario?** Es un caso borde clásico (Boundary Testing) que a menudo causa excepciones de índice fuera de rango (`IndexOutOfBoundsException`) si el código no es robusto.

Este código cumple con los requerimientos del PDF y demuestra un dominio sólido de la ingeniería de software moderna. Proceda a integrarlo en su proyecto.