## Selección de Tecnologías del Proyecto

### 1. Lenguaje de Programación y Framework

#### Lenguaje: Java

La elección de **Java** se fundamenta en los siguientes aspectos:

1. **Multiplataforma**: Gracias a la Máquina Virtual de Java (JVM), garantiza una ejecución coherente en Windows, Linux y macOS, lo que facilita la portabilidad sin modificaciones relevantes en el código.
2. **Orientación a objetos**: Su naturaleza pura OOP favorece una estructura modular, escalable y mantenible, esencial para proyectos a mediano y largo plazo.
3. **Concurrencia y multihilo**: Soporte nativo para tareas en segundo plano (por ejemplo, indexación continua), crucial en procesos que manejan volúmenes de datos intensivos.
4. **Ecosistema maduro**: Comunidad global activa, amplia disponibilidad de bibliotecas, actualizaciones constantes y documentación completa, lo cual reduce tiempos de resolución de errores.
5. **Experiencia del equipo**: Todo el equipo posee conocimiento sólido en Java, acortando la curva de aprendizaje y acelerando la colaboración.

**Comparativo con otras tecnologías:**

- **Python**: Gran rapidez de prototipado, pero limitado por el GIL en concurrencia real y menor rendimiento en procesos intensivos.
- **C++**: Máximo rendimiento, pero complejidad en manejo de memoria y menor portabilidad, pues requiere compilación específica por plataforma.
- **JavaScript/Electron**: Potente en web, pero presenta sobrecarga y no resulta óptimo para aplicaciones de escritorio con alta I/O de archivos.
- **C#/.NET**: Similar a Java, pero históricamente más atado al ecosistema Microsoft; aunque .NET Core mejora la portabilidad, aún depende de add-ons.

En conjunto, Java ofrece el equilibrio ideal entre portabilidad, rendimiento y experiencia del equipo, alineado con los objetivos del proyecto.

#### Framework: JavaFX

Para la capa de presentación se adoptará **JavaFX** por las siguientes razones:

- **Modernidad**: Recomendado para nuevos desarrollos frente a AWT y Swing.
- **Separación de lógica y UI**: Uso de FXML y CSS para mantener una arquitectura limpia.
- **Rendimiento**: Mejor desempeño gráfico y capacidad de animaciones más fluidas.

**Retos a considerar:**

1. **Curva de aprendizaje**: El equipo deberá familiarizarse con JavaFX y su modelo de eventos.
2. **Ecosistema joven**: Menor cantidad de soluciones «listas para usar» en comparación con Swing.
3. **Gestión de dependencias**: JavaFX no forma parte del JDK estándar y debe integrarse mediante OpenJFX.

**2. Base de Datos Relacional: PostgreSQL**

Se ha seleccionado **PostgreSQL** por:

- **Seguridad y control de acceso**: Mecanismos robustos de autenticación, superioridad sobre muchas implementaciones SQL.
- **Cumplimiento estricto de ACID**: Mantiene atomicidad, consistencia, aislamiento y durabilidad en todas las operaciones.
- **Rendimiento en grandes volúmenes**: Optimizaciones internas que mejoran el manejo de datasets extensos.
- **Open source y comunidad activa**: Desarrollo guiado por la comunidad, con actualizaciones constantes y sin costos de licencia.

Estas características satisfacen los requerimientos de integridad y escalabilidad del proyecto.

**3. Bibliotecas y Herramientas Complementarias**

- **OpenJFX**: Implementación de JavaFX para la UI.
- **PostgreSQL JDBC Driver**: Conexión de Java con la base de datos PostgreSQL.
- **Apache Lucene** (posible): Motor de búsqueda avanzada para índices de texto.
- **Lombok**: Reducción de código repetitivo mediante anotaciones en tiempo de compilación.

**4. Justificación General**

Cada componente ha sido elegido en función de:

- **Necesidades del proyecto**: Portabilidad, rendimiento y escalabilidad en la indexación y búsqueda de archivos.
- **Capacidades del equipo**: Aprovechamiento de competencias existentes en Java y rápido despliegue.
- **Objetivos del curso**: Aplicar buenas prácticas de ingeniería de software, arquitectura limpia y uso de herramientas industriales.

Esta selección garantiza una base técnica sólida, alineada con los requerimientos funcionales y no funcionales, y maximiza la eficiencia del desarrollo y mantenimiento.

**Patrón de Diseño**

**1. Definición y Propósito del Factory Method**

El **Factory Method** es un patrón de diseño creacional que define una interfaz para crear objetos en una superclase, permitiendo que las subclases (o la propia clase) determinen el tipo concreto de objetos a instanciar.

**Propósito principal:**

- Desacoplar la creación de objetos del código cliente.
- Centralizar la lógica de construcción y validación.
- Facilitar la extensión del sistema para soportar nuevos tipos sin modificar el código existente (siguiendo el Principio Abierto/Cerrado - OCP).

**Implementación en el Proyecto**

En **KoolFileIndexer**, el patrón se aplicó mediante dos métodos estáticos que actúan como fábricas:

**a) Categoria.clasificar(Archivo archivo)**

**Responsabilidad:**

- Determinar la categoría de un archivo en función de su extensión.
- Retornar una instancia predefinida de **Categoria** o **SIN_CATEGORIA** si no corresponde ninguna.

**Código relevante:**

```java
public static Categoria clasificar(Archivo archivo) {

    String ext = archivo.getExtension().toLowerCase();

        if (Set.of("jpg", "png").contains(ext)) return new Categoria("Imagen", true);

        if (Set.of("pdf", "txt").contains(ext)) return new Categoria("Documento", true);

    // ... otras categorías
```

```
    return SIN_CATEGORIA;
}
```

**b) Etiqueta.crear(String nombre)**

**Responsabilidad:**

- Validar el formato del nombre de la etiqueta (longitud, caracteres permitidos, etc.).
- Crear una instancia de **Etiqueta** normalizada (minúsculas, sin espacios redundantes).

**Código relevante:**

```java
public static Etiqueta crear(String nombre) {
    String limpio = nombre.trim().replaceAll("\\s+", " ");
    if (!ValidadorEntrada.esEtiquetaValida(limpio)) {
        throw new IllegalArgumentException("Nombre de etiqueta inválido");
    }
    return new Etiqueta(limpio.toLowerCase());
}
```

**Justificación de su Uso en el Sistema**

**a) Desacoplamiento**

El código cliente (por ejemplo, **Indexador**) no necesita conocer los detalles de construcción de las categorías o etiquetas; basta con invocar los métodos de fábrica.

**b) Extensibilidad (OCP)**

Agregar una nueva categoría (por ejemplo, para archivos .md) solo requiere actualizar **Categoria.clasificar()**, sin afectar otras clases.
De igual forma, los cambios en las reglas de validación de etiquetas se centralizan en **Etiqueta.crear()**.

**c) Cohesión y Mantenibilidad**

Toda la lógica de creación reside en un único lugar, lo cual facilita las pruebas unitarias y elimina duplicidad de código.

**Ejemplo de Uso en el Código Fuente**

Cuando el Indexador procesa un archivo:

```java
// Obtiene la categoría sin conocer la lógica interna

Categoria cat = Categoria.clasificar(archivo);



// Crea una etiqueta validada automáticamente

Etiqueta etiqueta = Etiqueta.crear("documento importante");
```

- El **Factory Method** fue fundamental para:

❖ Simplificar la creación de objetos en el dominio.
❖ Reducir el acoplamiento entre componentes.
❖ Facilitar la incorporación de futuras extensiones sin romper el código existente.

Su implementación ya forma parte integral y documentada del proyecto, demostrando su efectividad en un caso real.