**📄 Guía de desarrollo – Aplicación de Procesamiento de Texto con Arquitectura Plugin**

**🎯 Objetivo del proyecto**

Construir una aplicación de escritorio en **Java (Maven + IntelliJ IDEA)** que permita cargar archivos de audio y texto y aplicar diferentes filtros o transformaciones mediante una **arquitectura Plugin (extensible y modular)**.

El propósito es que la aplicación sea fácilmente ampliable:

* Cada filtro será un **plugin independiente**.
* El núcleo de la aplicación no dependerá de la implementación de cada filtro, solo de una **interfaz común**.
* Nuevos filtros podrán agregarse sin modificar el núcleo.

**⚙️ Funcionamiento general**

1. El usuario carga un archivo de audio (grabado con micrófono o cargado desde disco).
2. El sistema lista los componentes (plugins) disponibles.
3. El usuario selecciona un filtro (ejemplo: reproducir audio, obtener texto, buscar cadena).
4. El núcleo carga dinámicamente el plugin y ejecuta la operación.
5. Los resultados se muestran en la interfaz gráfica.

👉 La **lógica del núcleo siempre es la misma**. Lo único que cambia son los plugins, que implementan un contrato común PluginFiltro.

**🗂️ Estructura de carpetas (Maven multi-módulo)**

text-processing-plugin-architecture/ <-- Carpeta raíz del proyecto (multi-module Maven)

│

├── app-core/ <-- Módulo principal (núcleo de la app)

│ ├── src/main/java/com/miempresa/appcore/

│ │ ├── MainApp.java <-- Punto de entrada

│ │ ├── gui/ <-- Interfaz gráfica

│ │ │ └── VentanaPrincipal.java

│ │ ├── core/ <-- Núcleo del sistema

│ │ │ ├── PluginManager.java <-- Carga dinámica de plugins

│ │ │ ├── FiltroExecutor.java <-- Ejecuta filtros

│ │ │ └── interfaces/

│ │ │ └── PluginFiltro.java <-- Contrato común

│ │ └── model/ <-- Entidades (ArchivoAudio, ArchivoTexto, etc.)

│ └── src/main/resources/

│ └── META-INF/services/ <-- Configuración para ServiceLoader

│ └── com.miempresa.appcore.interfaces.PluginFiltro

│

├── plugins/ <-- Cada submódulo = un plugin independiente

│ ├── plugin-reproducir-audio/

│ │ └── src/main/java/com/miempresa/plugins/reproducir/

│ │ └── ReproducirAudioPlugin.java

│ │

│ ├── plugin-persistir-audio-db/

│ ├── plugin-listar-audio-db/

│ ├── plugin-obtener-texto-audio/

│ ├── plugin-leer-archivo-texto/

│ └── plugin-buscar-cadena-texto/

│

└── pom.xml <-- POM padre

**📜 Interfaz común de los plugins**

package com.miempresa.appcore.interfaces;

import java.io.File;

public interface PluginFiltro {

String getNombre(); // Nombre del filtro (ej. "Reproducir Audio")

String getDescripcion(); // Breve descripción

boolean soportaTipo(String tipoArchivo); // Ej: "audio", "texto"

String ejecutar(File archivo); // Lógica principal del filtro

}

**🧩 Ejemplo de plugin con SOLID y Clean Code**

package com.miempresa.plugins.reproducir;

import com.miempresa.appcore.interfaces.PluginFiltro;

import java.io.File;

public class ReproducirAudioPlugin implements PluginFiltro {

@Override

public String getNombre() {

return "Reproducir archivo de audio";

}

@Override

public String getDescripcion() {

return "Reproduce un archivo de audio seleccionado.";

}

@Override

public boolean soportaTipo(String tipoArchivo) {

return tipoArchivo.equalsIgnoreCase("audio");

}

@Override

public String ejecutar(File archivo) {

// TODO: Implementar lógica real de reproducción de audio

System.out.println("Reproduciendo: " + archivo.getName());

return "Reproducción finalizada.";

}

}

📌 Notas de diseño:

* **S** (Single Responsibility): cada plugin hace solo una cosa.
* **O** (Open/Closed): se pueden agregar nuevos plugins sin tocar el núcleo.
* **L** (Liskov Substitution): todos los plugins pueden reemplazarse entre sí porque cumplen PluginFiltro.
* **I** (Interface Segregation): interfaz pequeña y clara.
* **D** (Dependency Inversion): el núcleo depende de la abstracción, no de implementaciones.

**🔄 Carga dinámica de plugins**

El núcleo (PluginManager) usará ServiceLoader para descubrir plugins:

ServiceLoader<PluginFiltro> loader = ServiceLoader.load(PluginFiltro.class);

for (PluginFiltro filtro : loader) {

System.out.println("Plugin cargado: " + filtro.getNombre());

}

Cada módulo plugin debe tener en:  
src/main/resources/META-INF/services/com.miempresa.appcore.interfaces.PluginFiltro

el nombre completo de la clase que implementa el plugin, por ejemplo:

com.miempresa.plugins.reproducir.ReproducirAudioPlugin

**📋 Lista de plugins a implementar (orden de prioridad)**

1. **Reproducir archivo de audio**  
   Entrada: archivo de audio.  
   Acción: reproducirlo en la aplicación.
2. **Persistir archivo en base de datos**  
   Entrada: archivo de audio.  
   Acción: guardar el archivo en la BD.
3. **Listar archivos guardados en base de datos**  
   Entrada: nada.  
   Acción: mostrar lista de audios almacenados.
4. **Obtener texto del archivo de audio (Speech-to-Text)**  
   Entrada: archivo de audio.  
   Acción: convertir voz a texto.
5. **Leer archivo de texto → Convertir a audio (Text-to-Speech)**  
   Entrada: archivo de texto.  
   Acción: generar audio a partir del texto.
6. **Buscar cadena de texto en archivo de texto**  
   Entrada: archivo de texto + cadena de búsqueda.  
   Acción: devolver coincidencias encontradas.

**📌 Lineamientos para el equipo**

* **Estandarizar interfaces** → todos implementan PluginFiltro.
* **Carpetas uniformes** → cada plugin tiene mismo esquema (src/main/java/com/miempresa/plugins/...).
* **Clases claras** → nombre + sufijo Plugin (ej. PersistirAudioPlugin).
* **Convenciones de código**:
  + Clases → PascalCase.
  + Paquetes → minúsculas.
  + Métodos/variables → camelCase.
* **Nada de lógica duplicada en el núcleo** → cada cosa va en su plugin.
* **Documentar cada plugin** con javadoc breve en getDescripcion().

**🗂️ Estructura de carpetas de un plugin**

Ejemplo: plugin-reproducir-audio

plugin-reproducir-audio/

│

├── pom.xml

│

├── src/

│ ├── main/

│ │ ├── java/com/miempresa/plugins/reproducir/

│ │ │ ├── ReproducirAudioPlugin.java <-- Clase principal (implementa PluginFiltro)

│ │ │ ├── service/ <-- Lógica interna del plugin

│ │ │ │ └── ReproductorService.java <-- Clase que hace el "trabajo real"

│ │ │ ├── utils/ <-- Métodos auxiliares (si se requieren)

│ │ │ └── model/ <-- Entidades propias del plugin (si aplica)

│ │ │

│ │ └── resources/

│ │ └── META-INF/services/

│ │ └── com.miempresa.appcore.interfaces.PluginFiltro

│ │ # Aquí va: com.miempresa.plugins.reproducir.ReproducirAudioPlugin

│ │

│ └── test/java/com/miempresa/plugins/reproducir/

│ └── ReproducirAudioPluginTest.java <-- Tests unitarios para el plugin

**📌 Explicación de carpetas**

* **ReproducirAudioPlugin.java**
  + Clase principal que implementa PluginFiltro.
  + Hace de "puerta de entrada" al plugin.
  + No debe tener lógica pesada → delega a servicios.
* **service/**
  + Contiene la lógica de negocio del plugin.
  + Ejemplo: ReproductorService que sabe reproducir archivos.
  + Permite aplicar el principio **SRP** (Single Responsibility).
* **utils/**
  + Métodos auxiliares que pueden ser reutilizables dentro del plugin.
  + Ejemplo: un validador de tipo de archivo.
* **model/**
  + Clases de datos propias del plugin (si son necesarias).
  + Ejemplo: una clase AudioMetadata con info del archivo reproducido.
* **resources/META-INF/services/**
  + Archivo que le dice a ServiceLoader qué clase implementar.
  + Ejemplo (contenido):
  + com.miempresa.plugins.reproducir.ReproducirAudioPlugin
* **test/**
  + Pruebas unitarias de cada plugin.
  + Importante para que no dependamos de probar todo en la GUI.

**📋 Aplicación a todos los plugins**

Cada plugin tendrá la **misma estructura interna**, cambiando solo el nombre del paquete y las clases:

1. plugin-reproducir-audio/
   * ReproducirAudioPlugin.java
   * service/ReproductorService.java
2. plugin-persistir-audio-db/
   * PersistirAudioPlugin.java
   * service/AudioRepository.java (para la BD)
3. plugin-listar-audio-db/
   * ListarAudioPlugin.java
   * service/AudioQueryService.java
4. plugin-obtener-texto-audio/
   * ObtenerTextoAudioPlugin.java
   * service/SpeechToTextService.java
5. plugin-leer-archivo-texto/
   * LeerArchivoTextoPlugin.java
   * service/TextToSpeechService.java
6. plugin-buscar-cadena-texto/
   * BuscarCadenaTextoPlugin.java
   * service/TextSearchService.java

📌 **Ventaja:**  
Como todos los plugins tienen la misma estructura, si alguien necesita ayudar a un compañero solo tiene que abrir service/ para ver la lógica, utils/ si hay helpers y model/ si hay clases de datos. Así no hay que leer línea por línea del plugin.

**🔗 Interacciones entre los plugins**

Aunque cada plugin es **independiente** y cumple una única responsabilidad, en la práctica algunos trabajan **en cadena**. El núcleo (app-core) es el que se encarga de **coordinar entradas y salidas** entre ellos.

**📋 Flujo de interacciones**

1. **Reproducir archivo de audio**
   * Entrada: archivo de audio desde disco o grabado.
   * Salida: solo acción (se reproduce).
   * 🔗 No genera información para otros plugins. Es autónomo.
2. **Persistir archivo en base de datos**
   * Entrada: archivo de audio.
   * Acción: lo guarda en la base de datos.
   * Salida: confirmación (ej. "Audio guardado con ID=123").
   * 🔗 Interactúa con:
     + plugin-listar-audio-db (los datos guardados luego podrán listarse).
     + plugin-obtener-texto-audio (puede tomar un archivo desde BD para convertirlo en texto).
3. **Listar archivos guardados en base de datos**
   * Entrada: nada.
   * Acción: consulta la base de datos.
   * Salida: lista de audios (metadatos + ruta o ID).
   * 🔗 Interactúa con:
     + plugin-reproducir-audio (para reproducir un audio seleccionado de la lista).
     + plugin-obtener-texto-audio (para pasar el audio listado y transformarlo a texto).
4. **Obtener texto del archivo de audio (Speech-to-Text)**
   * Entrada: archivo de audio (desde disco o desde BD).
   * Acción: transcribe el audio a texto.
   * Salida: texto plano.
   * 🔗 Interactúa con:
     + plugin-buscar-cadena-texto (sobre el texto transcrito).
     + plugin-leer-archivo-texto (se podría leer este texto transcrito y devolverlo como audio nuevamente).
5. **Leer archivo de texto → Convertir a audio (Text-to-Speech)**
   * Entrada: archivo de texto (desde disco o generado por otro plugin, como el de Speech-to-Text).
   * Acción: genera audio con voz sintética.
   * Salida: archivo de audio.
   * 🔗 Interactúa con:
     + plugin-reproducir-audio (para reproducir el audio generado).
     + plugin-persistir-audio-db (para guardar este audio en la BD).
6. **Buscar cadena de texto en archivo de texto**
   * Entrada: archivo de texto o texto generado por otro plugin.
   * Acción: busca ocurrencias de una palabra/cadena.
   * Salida: posiciones encontradas o “no encontrado”.
   * 🔗 Interactúa con:
     + plugin-obtener-texto-audio (para buscar dentro de la transcripción).
     + plugin-leer-archivo-texto (para confirmar contenido antes de convertir a audio).

**📌 Resumen gráfico de las interacciones**

Audio → Reproducir

→ Persistir en BD → Listar BD → (Reproducir | Obtener Texto)

→ Obtener Texto → (Buscar Cadena | Leer Texto→Audio → Reproducir/Persistir)

Texto → Leer Texto→Audio → (Reproducir | Persistir en BD)

**⚙️ Cómo programar estas interacciones**

1. **Entrada y salida estandarizadas**
   * Todos los plugins reciben un File o un identificador (ID en BD).
   * Si un plugin genera un archivo nuevo (ej. plugin-leer-archivo-texto), debe devolver la ruta o referencia clara para que otro plugin la use.
2. **Comunicación centralizada en el núcleo**
   * Los plugins **no se llaman entre sí directamente**.
   * El núcleo (FiltroExecutor) se encarga de:
     + Tomar la salida de un plugin.
     + Pasarla como entrada al siguiente.
3. **Resultados claros**
   * El método ejecutar(File archivo) debe devolver un String con la descripción del resultado.
   * Si el plugin genera un archivo/texto adicional, debe guardarlo en una ruta predefinida o retornarlo como referencia en el mensaje.

Ejemplo:

String resultado = filtro.ejecutar(archivo);

System.out.println("Resultado: " + resultado);

1. **Persistencia uniforme**
   * Los plugins que usan base de datos (persistir y listar) deben manejar una tabla estándar Audios con campos: id, nombre, ruta, fechaCreacion.
   * Así, cualquier otro plugin puede tomar ese ID/ruta sin importar quién lo guardó.

📌 Con esto tu equipo sabe exactamente:

* Qué plugin se conecta con cuál.
* Qué entradas y salidas manejar.
* Cómo se deben devolver resultados.
* Y que **toda la comunicación fluye a través del núcleo**, nunca plugin→plugin directo.

**APP-CORE DESARROLLADO – DESCRIPCION**

**✅ Preparación actual de app-core**

1. **Estructura Maven multi-módulo**
   * Tienes un **POM padre** (plugin-architecture/pom.xml) que define el empaquetado como pom y agrega el módulo app-core

app-core

.

* + Dentro de app-core tienes su propio pom.xml con packaging = jar, correcto para un submódulo hijo.

👉 Esto ya habilita que en el futuro agregues otros submódulos como plugin-reproducir-audio, plugin-leer-texto, etc.

1. **Interfaz común (PluginFiltro)**
2. package org.example.interfaces;
3. import java.io.File;
4. public interface PluginFiltro {
5. String getNombre();
6. String getDescripcion();
7. boolean soportaTipo(String tipoArchivo);
8. String ejecutar(File archivo);
9. }
   * Define un **contrato estándar** que todos los plugins deben implementar.
   * Es simple, clara y cumple con los principios SOLID mencionados en la guía

PLUGIN - Guía de desarrollo

.

1. 👉 Gracias a esta interfaz, cualquier plugin puede conectarse al núcleo sin necesidad de modificarlo.
2. **Descubrimiento dinámico de plugins (PluginManager)**
3. ServiceLoader<PluginFiltro> loader = ServiceLoader.load(PluginFiltro.class);
4. for (PluginFiltro filtro : loader) {
5. plugins.add(filtro);
6. System.out.println("Plugin cargado: " + filtro.getNombre());
7. }
   * Usa **ServiceLoader** de Java para buscar implementaciones de PluginFiltro.
   * Esto funciona gracias al archivo en META-INF/services/org.example.interfaces.PluginFiltro, donde se listarán las clases concretas de los plugins.
   * Con este mecanismo, el núcleo no necesita saber cuántos ni cuáles plugins existen: los carga automáticamente en tiempo de ejecución.

👉 Esto es clave: tu app-core ya está desacoplado de los plugins.

1. **Ejecución de plugins (FiltroExecutor)**
2. if (!filtro.soportaTipo(obtenerTipoArchivo(archivo))) {
3. return "El plugin no soporta este tipo de archivo.";
4. }
5. return filtro.ejecutar(archivo);
   * Valida si el plugin soporta el tipo de archivo (audio, texto, etc.).
   * Encapsula la lógica de ejecución sin que el núcleo conozca detalles de implementación.

👉 Esto asegura que el flujo siempre pasa por el núcleo, nunca de plugin a plugin directo (tal como indica la guía

PLUGIN - Guía de desarrollo

).

1. **Interfaz gráfica (VentanaPrincipal)**
   * Muestra un menú Plugins donde lista dinámicamente los que cargó PluginManager.
   * Al seleccionar un plugin, se muestra su descripción.
   * Tiene un botón "Grabar 5sg" que utiliza AudioRecorder para grabar y guardar un archivo WAV.

👉 La GUI ya está preparada para integrarse con cualquier plugin, mostrando su nombre y descripción sin modificaciones.

1. **Modelos (ArchivoAudio, ArchivoTexto)**
   * Clases simples (POJO) que encapsulan los archivos.
   * Esto prepara el terreno para que los plugins que manejen distintos tipos de archivos lo hagan de forma clara y uniforme.

**📂 META-INF/services**

Actualmente, en app-core tienes:

src/main/resources/META-INF/services/org.example.interfaces.PluginFiltro

Ese archivo **no contiene lógica**, solo nombres de clases concretas (uno por línea).

* En el núcleo (app-core) lo tendrás vacío o sin implementaciones.
* Cada plugin que desarrolles deberá incluir su clase aquí.  
  Ejemplo (en un futuro plugin-reproducir-audio):
* org.example.plugins.reproducir.ReproducirAudioPlugin

👉 Esto confirma que app-core ya está preparado para recibir cualquier plugin.

**📖 Explicación para tu equipo**

El programa funciona así:

1. **Inicio del núcleo**
   * El usuario abre la aplicación (Main), que lanza VentanaPrincipal.
   * Se muestra un menú con los plugins disponibles y un botón para grabar audio.
2. **Grabación de audio**
   * Con el botón "Grabar 5sg", el núcleo usa AudioRecorder para capturar 5 segundos desde el micrófono y guardarlos como grabacion.wav.
3. **Carga de plugins**
   * PluginManager usa ServiceLoader para buscar automáticamente todos los plugins que implementan PluginFiltro.
   * El núcleo no necesita saber cuántos son ni sus nombres: los descubre en tiempo de ejecución.
4. **Selección y ejecución de plugins**
   * El usuario selecciona un plugin desde el menú.
   * El núcleo llama a FiltroExecutor, que valida si ese plugin soporta el archivo (audio o texto) y ejecuta su lógica.
   * El resultado (mensaje, archivo generado, acción realizada) se devuelve a la interfaz.
5. **Extensibilidad**
   * Para añadir un nuevo plugin, el equipo solo debe:
     + Crear un nuevo módulo Maven (ej. plugin-reproducir-audio).
     + Implementar la interfaz PluginFiltro.
     + Registrar la clase en META-INF/services/org.example.interfaces.PluginFiltro.
   * Sin modificar app-core, automáticamente aparecerá en la GUI.

**🚀 Conclusión**

✔ Tu app-core ya está **listo para recibir cualquier plugin**.  
✔ Cumple con el diseño descrito en la guía: núcleo independiente + interfaz común + carga dinámica con ServiceLoader.  
✔ El equipo puede crear nuevos plugins sin tocar el núcleo, simplemente añadiendo un submódulo y registrándolo en META-INF/services.

**NUEVA ACTUALIZACION 2.0**

**📌 Estado actual del proyecto de arquitectura de plugins**

**1. Cambios principales en app-core**

* **Núcleo central de la aplicación que define la interfaz PluginFiltro y carga plugins con ServiceLoader.**
* **Eliminamos el archivo plano de configuración en app-core (el descriptor ahora vive en cada plugin).**
* **Se añadió soporte para ejecutar el core con Maven:**
* **mvn -pl app-core exec:java**
* **Para pruebas manuales también funciona con:**
* **java -cp "app-core/target/app-core-1.0-SNAPSHOT.jar;plugin-reproducir-audio/target/plugin-reproducir-audio-1.0-SNAPSHOT.jar" org.example.Main**

**2. POMs (padre e hijos)**

**🟢 Padre (plugin-architecture/pom.xml)**

* **packaging = pom.**
* **Declara los módulos app-core y plugin-reproducir-audio.**
* **Tiene configurado exec-maven-plugin para facilitar ejecución desde la raíz.**

**🟢 Hijo app-core/pom.xml**

* **packaging = jar.**
* **Configura exec-maven-plugin con org.example.Main.**
* **No depende de plugins → evita ciclos.**
* **Es el núcleo ejecutable.**

**🟢 Hijo plugin-reproducir-audio/pom.xml**

* **packaging = jar.**
* **Depende de app-core.**
* **Tiene el descriptor:**
* **src/main/resources/META-INF/services/org.example.interfaces.PluginFiltro**

**con el contenido:**

**org.example.plugins.reproducir.ReproducirAudioPlugin**

**3. Plugin "Reproducir Audio"**

* **Implementa PluginFiltro.**
* **Actualmente solo muestra un mensaje en la interfaz.**
* **Confirmado que se carga dinámicamente por el core.**
* **❌ Falta implementar reproducción real con Clip y controles Play/Pause/Resume/Stop.**

**4. Ejecución**

* **Se puede ejecutar con Maven:**
* **mvn -pl app-core exec:java**
* **O bien con java -cp pasando los dos JARs.**
* **En ambos casos, el core arranca → detecta el plugin → lo muestra en la GUI.**

**5. Preocupación planteada**

**📌 Cómo interactuarán los plugins entre sí y con el core.**

**Ejemplos concretos:**

* **El plugin de reproducir audio necesita saber qué archivo reproducir (puede ser el grabado por el core o uno traído de BD).**
* **El plugin de listar audios consultará la base de datos y deberá devolver esa lista al core y/o a otros plugins.**
* **El plugin de Text-to-Speech trabajará con archivos de texto en lugar de audios.**
* **El plugin de Speech-to-Text hará lo contrario (convertir audio a texto).**

**6. Propuesta: AppContext**

**Para resolverlo, el core debería proveer a los plugins un contexto común de ejecución:**

**🔹 ¿Qué es?**

**Una clase central en app-core (ejemplo: AppContext) que encapsula el estado compartido de la aplicación.**

**🔹 Contenido sugerido**

* **Último archivo de audio grabado.**
* **Lista de audios disponibles (consultados desde BD).**
* **Último archivo de texto cargado.**
* **Resultados de operaciones (ej. transcripción de audio a texto).**
* **Servicios compartidos:**
  + **Conexión a la base de datos.**
  + **Ruta de almacenamiento de archivos temporales.**

**🔹 Nueva interfaz**

**Modificar PluginFiltro para que use el contexto en lugar de parámetros sueltos:**

**public interface PluginFiltro {**

**String getNombre();**

**String getDescripcion();**

**boolean soportaTipo(String tipoArchivo);**

**String ejecutar(AppContext contexto);**

**}**

**🔹 Beneficios**

* **Plugins pueden trabajar con diferentes tipos de entrada:**
  + **Audio (File)**
  + **Texto (File o String)**
  + **Listas de resultados (List<File> o List<String>)**
* **Plugins comparten la misma información sin acoplarse entre sí.**
* **Ejemplo:**
  + **ListarAudiosPlugin rellena contexto.setListaAudios(...).**
  + **ReproducirAudioPlugin toma un archivo de contexto.getListaAudios().**
  + **SpeechToTextPlugin recibe contexto.getUltimoAudio() y devuelve contexto.setTexto(...).**

**7. Próximos pasos**

1. **Implementar clase AppContext en app-core.**

** Clase central en app-core para compartir información entre plugins y el núcleo.**

** Debe poder guardar y exponer:**

* **Último archivo cargado/guardado.**
* **Lista de audios (cuando otro plugin la obtenga de BD).**
* **Último archivo de texto.**
* **Último resultado de Speech-to-Text o Text-to-Speech.**

** Plugins podrán leer y escribir en el AppContext.**

** Servirá como “estado global” pero controlado.**

1. **Ajustar PluginFiltro para recibir AppContext.**
2. **Actualizar ReproducirAudioPlugin para usar el archivo de AppContext y no solo mostrar mensaje.  Implementar en ReproductorService la lógica real con Clip (Java Sound API). Asegurar que la reproducción se hace en un hilo separado (no bloquear la UI). Añadir ventana de control (Play / Pause / Resume / Stop). Probar con el archivo generado por Grabar 5sg.**
3. **Implementar carpeta /Plugins en la raíz para soportar plug-and-play real cargando JARs dinámicamente.**
4. **Documentar claramente el contrato AppContext para que todo el equipo desarrolle plugins con la misma base.**

**✅ Con esto, tienes un panorama claro:**

* **Lo que ya está funcionando.**
* **Cómo se ejecuta actualmente (mvn -pl app-core exec:java o java -cp ...).**
* **El estado del primer plugin.**
* **La preocupación sobre comunicación entre plugins.**
* **Una solución concreta (AppContext) que servirá de base para los 6 plugins futuros.**

**NUEVA ACTUALIZACION 2.1**

**Resumen detallado del estado actual**

**🔹 Cambios principales en app-core**

1. **Nueva clase AppContext**
   * Centraliza la comunicación entre el núcleo y los plugins.
   * Permite guardar: último archivo (audio, texto), lista de audios, último resultado y servicios compartidos.
   * Evita pasar solo un File, ahora se comparte un “contexto” más rico.
2. **Interfaz PluginFiltro actualizada**
   * Antes: ejecutar(File archivo)
   * Ahora: ejecutar(AppContext contexto)
   * Con esto todos los plugins futuros podrán recibir más información que solo el archivo.
3. **FiltroExecutor**
   * Ahora valida el tipo de archivo desde AppContext y ejecuta el plugin de forma centralizada.
4. **VentanaPrincipal**
   * Cuando el usuario selecciona un plugin:
     + Construye un AppContext con grabacion.wav (si existe).
     + Si no hay, abre un JFileChooser para que el usuario elija un archivo.
     + Llama a FiltroExecutor para pasar el contexto al plugin.
   * Resultado: la UI principal sigue siendo la “orquestadora” pero los datos se centralizan en AppContext.

**🔹 Estado del plugin Reproducir audio**

1. **ReproducirAudioPlugin**
   * Implementa PluginFiltro con la nueva firma (ejecutar(AppContext contexto)).
   * Toma el archivo desde AppContext.
   * Valida con AudioValidator (solo WAV).
   * Lanza en Swing una ventana (ReproductorFrame) con controles de reproducción.
   * Retorna al núcleo un mensaje de éxito o error (ej. *“OK: Ventana abierta”*).
2. **ReproductorService**
   * Maneja la reproducción usando **Java Sound API (Clip)**.
   * Implementado para trabajar en un **hilo separado** (Thread propio “Reproductor-Thread”), así la interfaz no se congela.
   * Soporta:
     + Play (desde cero o desde el inicio).
     + Pause (guarda posición).
     + Resume (continúa desde la posición).
     + Stop (detiene y libera recursos).
3. **ReproductorFrame**
   * Ventana Swing con botones **Play / Pause / Resume / Stop**.
   * Cada botón llama a los métodos de ReproductorService.
   * Play se ejecuta en un **hilo aparte**, evitando bloquear la UI.
4. **AudioValidator**
   * Revisa que el archivo exista, sea legible y que el formato esté soportado.
   * Actualmente solo se acepta **.wav**.

**🔹 Cómo se ejecuta todo junto**

1. El usuario en el núcleo:
   * Presiona **“Grabar 5sg”** → se genera grabacion.wav.
2. El usuario selecciona el plugin *Reproducir audio*:
   * VentanaPrincipal crea un AppContext con grabacion.wav.
   * Llama a FiltroExecutor → este pasa el contexto al plugin.
3. El plugin:
   * Valida que grabacion.wav sea un WAV válido.
   * Abre ReproductorFrame con controles.
   * El usuario controla la reproducción.
   * La reproducción real ocurre en **un hilo aparte**, así la UI nunca se congela.

**🔹 Cosas importantes a considerar**

* **Grabación en el núcleo (AudioRecorder):**
  + Ahora mismo bloquea el hilo de la interfaz 5 segundos.
  + Solución futura: usar **Thread** o mejor **SwingWorker** para grabar en segundo plano.
* **Soporte a más formatos:**
  + Actualmente solo soportamos .wav.
  + Para .mp3 u otros, habría que añadir librerías externas (ej. mp3spi, JLayer) y ampliar AudioValidator.

**📌 Consideraciones técnicas**

* **Soporte a más formatos**
  + Permitiría que los usuarios no se limiten a WAV.
  + Requiere dependencias adicionales y validación extendida.
* **Uso de SwingWorker en grabación**
  + Evitaría que la ventana se congele al grabar.
  + Además permitiría mostrar al usuario el progreso (“Grabando 1/5 seg...”).
  + Ideal para cuando quieras feedback visual en la UI.

**✅ Checklist de plugins**

Lista según el documento de planeación:

1. **Reproducir archivo de audio** – ✔️ **Implementado** (con controles Play/Pause/Resume/Stop, validación y ejecución en hilo separado).
2. Transcribir audio a texto – ⬜ Pendiente.
3. Traducir texto transcrito – ⬜ Pendiente.
4. Guardar transcripción en archivo – ⬜ Pendiente.
5. Analizar sentimiento del texto – ⬜ Pendiente.
6. Resumir texto transcrito – ⬜ Pendiente.
7. Generar subtítulos a partir del audio – ⬜ Pendiente.