

MANUAL TÉCNICO

Sistema de Monitoreo y Prevención de Incendios

PYRALINE – Versión 1.0

Resumen Ejecutivo

El presente manual técnico documenta la estructura interna, criterios de diseño y lineamientos de desarrollo del sistema **Pyraline**, una aplicación desarrollada en Java cuyo objetivo es la detección temprana de anomalías térmicas y la prevención de incendios mediante el monitoreo continuo de sensores ambientales.

Este documento está dirigido a **desarrolladores, mantenedores o colaboradores externos**, y tiene como finalidad facilitar la comprensión del proyecto, permitir su extensión segura y garantizar la coherencia técnica del sistema a lo largo del tiempo.

1. Introducción y Visión General

1.1 Propósito del Sistema

Pyraline es un sistema de monitoreo ambiental que:

- Lee datos de temperatura desde un sensor Arduino
- Evalúa dichos datos contra un umbral configurable
- Activa alertas visuales y sonoras ante estados críticos
- Registra eventos térmicos en una base de datos SQLite
- Permite la visualización histórica de alertas
- Mantiene configuraciones persistentes entre reinicios

El sistema está diseñado para operar como una **herramienta preventiva**, priorizando la robustez, la claridad visual y la tolerancia a fallos de hardware.

1.2 Alcance del Proyecto

El proyecto cubre las siguientes funcionalidades técnicas:

- Comunicación serial automática con Arduino
- Monitoreo de temperatura en tiempo real
- Gestión de umbrales térmicos persistentes
- Registro histórico de eventos (alerta / normalización)
- Interfaz gráfica basada en Java Swing
- Manejo centralizado de errores y mensajes
- Soporte para recursos multimedia (audio e imágenes)

Limitaciones actuales

- No incluye comunicación remota (red / internet)
 - No contempla múltiples sensores simultáneos
 - No implementa autenticación cifrada
 - No incluye reportes estadísticos avanzados
-

2. Tecnologías y Dependencias Principales

Componente	Tecnología
Lenguaje principal	Java JDK 17+
Interfaz gráfica	Java Swing
Base de datos	SQLite
Comunicación serial	JSerialComm
Audio	Archivos .wav
Control de versiones	Git / GitHub
IDE	Visual Studio Code

Nota: El sistema no depende de frameworks externos pesados, priorizando una arquitectura ligera y portable.

3. Estructura General del Proyecto

La organización del proyecto sigue un enfoque modular y desacoplado, separando claramente las responsabilidades del sistema.

```
src/
| App.java
| TestDatabase.java
|
|--- BusinessLogic/
|--- DataAccess/
|--- Infrastructure/
└--- UserInterface/
```

Descripción general de carpetas

- **App.java**
Punto de entrada principal de la aplicación.
- **BusinessLogic/**
Contiene la lógica central del sistema, procesamiento de datos y reglas de negocio.
- **DataAccess/**
Maneja toda la interacción con la base de datos mediante DAOs y DTOs.
- **Infrastructure/**
Provee servicios de soporte: configuración, errores, estilo, hardware y recursos.
- **UserInterface/**
Implementa las pantallas gráficas y componentes visuales.

Las dependencias entre carpetas deben ser siempre unidireccionales, evitando acoplamientos circulares.

4. Convenciones de Desarrollo

4.1 Convenciones de Nomenclatura

Clases

- PascalCase
- Nombre representativo de la responsabilidad
- Ejemplo:
 - ArduinoSensor

- PyralineDashboard
- UsuarioBL

Métodos

- camelCase
- **Siempre en infinitivo**
- Ejemplos:
 - leerTemperatura()
 - guardarConfiguracion()
 - validarUsuario()

Variables

- camelCase
- Nombres descriptivos
- Sin abreviaturas ambiguas

4.2 Convenciones de Paquetes

- No se usan prefijos innecesarios
- Los paquetes representan **capas funcionales**
- No se mezclan responsabilidades dentro del mismo paquete

Ejemplo correcto:

BusinessLogic/UsuarioBL.java

DataAccess/DAOs/UsuarioDAO.java

Ejemplo incorrecto:

BusinessLogic/UsuarioDAO.java

5. Manejo de Errores y Mensajes

5.1 Excepciones Personalizadas

Todas las excepciones del sistema deben encapsularse en:

- AppException

Esto permite:

- Control centralizado de errores
- Mensajes claros al usuario
- Prevención de cierres inesperados

Regla:

Ninguna excepción técnica debe propagarse directamente a la interfaz.

5.2 Mensajes y Estilo Visual

- AppMSG: Centraliza los mensajes mostrados al usuario
- AppStyle: Define colores, fuentes y estilos gráficos

Esto garantiza:

- Coherencia visual
- Facilidad de mantenimiento
- Separación entre lógica y presentación

6. Clases Principales del Sistema

Esta sección describe las **clases más relevantes del proyecto**, sus responsabilidades y los métodos clave que concentran la lógica fundamental del sistema.

Nota importante

No se documentan todas las clases del proyecto, únicamente aquellas consideradas **estructurales o críticas**.

Las clases no incluidas siguen las mismas convenciones y lineamientos definidos en este manual.

6.1 App.java

Responsabilidad

Clase **punto de entrada** del sistema.

Inicializa la aplicación y garantiza que la interfaz gráfica se ejecute en el hilo correcto.

Responsabilidades técnicas

- Arrancar la aplicación

- Inicializar la interfaz principal
- Encapsular el arranque en SwingUtilities.invokeLater

Métodos relevantes

- main(String[] args)

Regla de desarrollo

Esta clase **no debe contener lógica de negocio**, acceso a datos ni manejo de hardware.

6.2 PyralineLogin.java

Responsabilidad

Gestiona la **pantalla de inicio y autenticación** del sistema.

Responsabilidades técnicas

- Capturar usuario y contraseña
- Validar credenciales mediante lógica de negocio
- Mostrar mensajes de error o permitir el acceso
- Redirigir al panel principal (PyralineDashboard)

Métodos relevantes (conceptuales)

- validarCredenciales()
- iniciarSesion()
- mostrarError(String mensaje)

Regla de desarrollo

- No debe acceder directamente a la base de datos
 - Toda validación debe delegarse a UsuarioBL
-

6.3 PyralineDashboard.java

Responsabilidad

Pantalla principal del sistema.

Muestra el **estado del sensor**, la **temperatura**, las **alertas** y los distintos módulos de la aplicación.

Responsabilidades técnicas

- Mostrar temperatura en tiempo real
- Reflejar el estado del sistema (Normal / Crítico / Desconectado)
- Cambiar entre vistas (Home, Configuración, Alertas)
- Mostrar imágenes y alarmas visuales
- Reproducir sonidos de alerta

Métodos relevantes (conceptuales)

- actualizarTemperatura(double temperatura)
- actualizarEstado(String estado)
- mostrarAlarma()
- detenerAlarma()
- cambiarVista(String vista)

Regla de desarrollo

- La UI no debe tomar decisiones de negocio
 - Solo muestra información ya procesada
-

6.4 ArduinoSensor.java

Responsabilidad

Encapsula la **comunicación directa con el hardware Arduino**.

Responsabilidades técnicas

- Detectar puertos seriales disponibles
- Establecer conexión automática
- Leer datos del sensor
- Detectar desconexión del dispositivo

Métodos relevantes (conceptuales)

- conectar()
- desconectar()
- leerTemperatura()

- `estaConectado()`

Regla de desarrollo

- Esta clase **no interactúa con la interfaz**
 - No guarda datos en base de datos
 - Se limita estrictamente al hardware
-

6.5 ArduinoPollingService.java

Responsabilidad

Servicio encargado del **monitoreo continuo** del sensor.

Responsabilidades técnicas

- Leer temperatura periódicamente
- Comparar valores contra el umbral
- Detectar estados críticos
- Notificar cambios de estado
- Mantener el monitoreo en segundo plano

Métodos relevantes (conceptuales)

- `iniciarMonitoreo()`
- `detenerMonitoreo()`
- `procesarLectura(double temperatura)`
- `evaluarUmbral(double temperatura)`

Regla de desarrollo

- No debe manipular directamente la UI
 - Debe notificar eventos mediante callbacks o lógica intermedia
-

6.6 UsuarioBL.java

Responsabilidad

Gestiona la **lógica de negocio relacionada con usuarios**.

Responsabilidades técnicas

- Validar credenciales
- Verificar estado del usuario (activo/inactivo)
- Aplicar reglas de seguridad

Métodos relevantes (conceptuales)

- validarUsuario(String usuario, String clave)
- esUsuarioActivo(String usuario)

Regla de desarrollo

- Toda validación debe pasar por esta clase
 - Nunca se consulta directamente el DAO desde la UI
-

6.7 FactoryBL.java

Responsabilidad

Centraliza la **creación de instancias de lógica de negocio**.

Responsabilidades técnicas

- Evitar creación directa de objetos desde la UI
- Controlar dependencias
- Facilitar mantenimiento y pruebas

Métodos relevantes (conceptuales)

- getUsuarioBL()
- getArduinoPollingService()

Regla de desarrollo

- Todas las instancias BL deben obtenerse desde esta fábrica
-

6.8 DataHelperSQLiteDAO.java

Responsabilidad

Gestiona la **conexión con la base de datos SQLite**.

Responsabilidades técnicas

- Abrir y cerrar conexiones

- Ejecutar consultas SQL
- Manejar errores de base de datos

Métodos relevantes (conceptuales)

- getConnection()
- cerrarConexion()
- ejecutarConsulta(String sql)
- ejecutarActualizacion(String sql)

Regla de desarrollo

- Ningún DAO debe manejar conexiones directamente
 - Todo acceso pasa por este helper
-

6.9 TestDatabase.java

Responsabilidad

Clase auxiliar para **verificar la conectividad y estructura de la base de datos.**

Responsabilidades técnicas

- Probar conexión SQLite
- Validar tablas existentes
- Verificar integridad básica

Uso

- Solo en desarrollo y pruebas
 - No forma parte del flujo normal de la aplicación
-

6.10 AppConfig.java

Responsabilidad

Gestiona la **configuración persistente del sistema.**

Responsabilidades técnicas

- Leer archivo .properties
- Guardar configuración de umbral

- Mantener valores entre reinicios

Métodos relevantes (conceptuales)

- cargarConfiguracion()
- guardarConfiguracion()
- getUmbral()
- setUmbral(double valor)

Regla de desarrollo

- La configuración nunca debe codificarse “hardcodeada”
-

6.11 AppStyle.java

Responsabilidad

Centraliza la **identidad visual** de la aplicación.

Responsabilidades técnicas

- Definir colores
- Definir fuentes
- Mantener coherencia visual

Regla de desarrollo

- Ningún color o fuente debe definirse directamente en la UI
-

6.12 AppException.java

Responsabilidad

Clase base para **manejo de errores controlados** del sistema.

Responsabilidades técnicas

- Encapsular errores técnicos
- Proveer mensajes amigables
- Evitar cierres inesperados

Regla de desarrollo

- Toda excepción del sistema debe derivar de AppException

7. Convenciones y Reglas de Desarrollo

Esta sección define las normas que deben seguirse al crear **nuevas carpetas, paquetes, clases y métodos**, independientemente de cambios futuros en la arquitectura o nomenclatura.

7.1 Convenciones generales de nomenclatura

Paquetes / carpetas

- Se utilizan **nombres en inglés**
- En **CamelCase** o **lowerCamelCase** según el estándar del proyecto
- El nombre debe reflejar **responsabilidad**, no tecnología

Correcto:

BusinessLogic

DataAccess

UserInterface

Infrastructure

Incorrecto:

Clases

MisArchivos

NuevoPaquete1

Regla clave

Un paquete = **una responsabilidad clara**

Clases

- Usar **PascalCase**
- El nombre debe representar **qué es, no qué hace**

Ejemplos:

- UsuarioBL
- ArduinoSensor
- AppConfig

- EstadoAlertaDAO

Incorrecto:

- Clase1
 - Manager
 - Helper2
-

Métodos

Regla obligatoria del proyecto

Todos los métodos se nombran en INFINITIVO

Correcto:

- validarUsuario()
- leerTemperatura()
- guardarConfiguracion()
- ejecutarConsulta()

Incorrecto:

- validaUsuario()
 - leyendoSensor()
 - configuracionGuardada()
-

7.2 Reglas para creación de nuevas clases

Antes de crear una clase nueva, el desarrollador debe preguntarse:

1. ¿Pertenece a UI, lógica, datos o infraestructura?
2. ¿Existe ya una clase con una responsabilidad similar?
3. ¿Puede reutilizarse una interfaz o helper existente?

Prohibido

- Clases “comodín”
 - Clases con más de una responsabilidad fuerte
-

7.3 Reglas para la capa de Lógica de Negocio (BL)

Responsabilidad

- Validaciones
- Reglas del sistema
- Decisiones importantes

Permitido

- Llamar a DAOs
- Lanzar AppException
- Procesar datos antes de enviarlos a la UI

Prohibido

- Código de interfaz gráfica
- Acceso directo a hardware
- SQL embebido

Toda clase BL:

- Debe tener un nombre terminado en **BL**
 - Debe obtenerse mediante **FactoryBL**
-

7.4 Reglas para Acceso a Datos (DAO / DTO)

DAO

- Un DAO por entidad
- Implementan una interfaz común (IDAO)
- No contienen lógica de negocio

Ejemplo:

UsuarioDAO

LugarDAO

TipoAlertaDAO

DTO

- Clases simples

- Solo atributos + getters/setters
- Sin lógica

Ejemplo:

UsuarioDTO

LugarDTO

Regla estricta

La UI nunca accede directamente a un DAO

7.5 Reglas para SQL y Base de Datos

- SQL **no debe** escribirse en:
 - UI
 - BL
- El acceso se centraliza en:
 - DAOs
 - DataHelperSQLiteDAO

Buenas prácticas

- Consultas claras
 - Nombres de tablas en singular
 - Claves primarias explícitas
 - Manejo de errores con AppException
-

7.6 Manejo de excepciones

Regla global

Toda excepción del sistema debe encapsularse en AppException

Correcto:

```
throw new AppException("Error al conectar con la base de datos");
```

Incorrecto:

```
throw new Exception(e);
```

7.7 Estilo visual y recursos

- Colores, fuentes y estilos → AppStyle
- Imágenes → carpeta resources/img
- Sonidos → carpeta resources/sounds

Nunca definir estilos directamente en la UI

8. Persistencia de Datos y Base de Datos (SQLite)

8.1 Objetivo

Esta sección describe los criterios técnicos para el uso de la base de datos SQLite en Pyraline, de forma que cualquier desarrollador pueda comprender, mantener o extender el sistema de persistencia sin afectar la estabilidad del proyecto.

8.2 Motor de Base de Datos

- **Motor:** SQLite
- **Tipo:** Persistencia local ligera
- **Archivo físico:** .db
- **Acceso:** Centralizado mediante clases DAO

Justificación: SQLite es ideal para aplicaciones de escritorio por su bajo acoplamiento, portabilidad y cero configuración de servidor.

8.3 Reglas Generales de Persistencia

1. La **UI no accede directamente** a la base de datos.
 2. Toda operación SQL debe pasar por un **DAO**.
 3. La lógica de negocio **no contiene SQL**.
 4. Las excepciones SQL deben transformarse en AppException.
 5. Las entidades de la base se representan mediante **DTOs**.
-

8.4 Clase Central de Acceso a Datos

DataHelperSQLiteDAO

Responsabilidad principal: Centralizar la conexión, ejecución de sentencias y manejo de errores de SQLite.

Funciones clave:

- Abrir y cerrar conexiones
- Ejecutar consultas (SELECT)
- Ejecutar actualizaciones (INSERT, UPDATE, DELETE)
- Manejo seguro de excepciones

Regla:

Ningún DAO debe crear conexiones directamente; todas se obtienen a través de esta clase.

8.5 Patrón DAO

Cada tabla de la base de datos tiene su DAO correspondiente.

Convenciones:

- Nombre de clase termina en DAO
- Implementa la interfaz IDAO<T>
- Contiene únicamente operaciones CRUD

Ejemplos:

- UsuarioDAO
- EstadoAlertaDAO
- TipoAlertaDAO

⚠ Prohibido:

- Validaciones de negocio
 - Lógica de interfaz
-

8.6 DTOs (Data Transfer Objects)

Convenciones:

- Nombre termina en DTO
- Atributos privados
- Getters y setters públicos
- Sin lógica de negocio

Ejemplo:

- UsuarioDTO
- EstadoAlertaDTO

Validaciones: Los rangos y valores críticos deben validarse antes de persistirse (ej. temperatura válida).

8.7 Estructura lógica de la Base de Datos

Criterios generales:

- Tablas en singular
- Clave primaria INTEGER PRIMARY KEY AUTOINCREMENT
- Fechas almacenadas en formato ISO (YYYY-MM-DD HH:MM:SS)

Ejemplo conceptual de tablas:

- Usuario
- EstadoAlerta
- TipoAlerta
- Lugar

(La definición exacta puede variar según la evolución del proyecto)

8.8 Registro de Eventos (Bitácora)

Cada evento térmico relevante se registra automáticamente:

- Estado sobre umbral
- Estado normalizado
- Fecha y hora
- Temperatura detectada

Regla:

El registro es automático y no depende de interacción del usuario.

8.9 Mantenimiento de Datos

- El sistema permite eliminar todos los registros históricos
- La acción requiere confirmación explícita
- La eliminación es **física** (DELETE)

Responsabilidad: Esta operación debe ejecutarse únicamente desde la capa de lógica de negocio.

8.10 Pruebas de Base de Datos

TestDatabase

Clase destinada a:

- Verificar conexión
- Probar consultas
- Validar estructura inicial

Uso: Solo en entornos de desarrollo y pruebas.

9. Clases Principales del Sistema

Esta sección documenta las **clases principales reales** del proyecto Pyraline, basándose directamente en el código fuente actual (versión 2.2). Su objetivo es permitir que cualquier desarrollador externo comprenda **qué hace cada clase, por qué existe y cómo debe extenderse**, sin necesidad de leer todo el código desde cero.

9.1 ArduinoPollingService

Paquete: BusinessLogic.services

Rol en N-Tier: Business Logic

Responsabilidad: Servicio central del sistema. Procesa las lecturas térmicas recibidas desde ArduinoSensor, compara la temperatura contra el umbral configurado, controla el estado de alerta (normal / crítico) y coordina la persistencia de eventos térmicos.

Dependencias directas:

- PYRALINEDAO (persistencia de eventos)
- PyralineDashboard (notificación visual y sonora)
- AppConfig (lectura y escritura de umbral)
- AppException (manejo de errores)

Atributos clave:

- float umbralAlarma
- boolean alertaActiva
- PYRALINEDAO pyralineDAO
- PyralineDashboard dashboard

Métodos principales:

- ArduinoPollingService(PyralineDashboard dash) → inicializa el servicio y carga el umbral persistido.
- notificarEstadoConexion(boolean conectado) → informa a la UI el estado físico del hardware.
- setUmbraAlarma(float nuevoUmbra) → actualiza y persiste el umbral en disco.
- procesarLectura(String rawData) → núcleo de la lógica térmica del sistema.

Métodos internos:

- registrarEvento(float valor, int tipo) → guarda eventos de alerta o normalización en SQLite.

Reglas técnicas:

- No accede directamente al hardware.
- No realiza operaciones gráficas directas.
- Toda excepción crítica se encapsula en AppException.

9.2 ArduinoSensor

Paquete: Infrastructure.hardware

Rol en N-Tier: Infrastructure

Responsabilidad: Gestionar la comunicación serial con el Arduino, incluyendo detección automática de puertos, lectura continua de datos, detección de desconexión física y reconexión automática.

Métodos principales:

- detectarPuertoAutomatico() → identifica puertos compatibles (USB / Arduino).
- conectar(String puertoNombre, ArduinoPollingService service) → establece comunicación serial.

Métodos internos:

- manejarDesconexion(ArduinoPollingService service) → activa el modo de reconexión infinita.
- registrarErrorEnLog(String mensaje) → registra errores físicos en el log del sistema.

Reglas técnicas:

- Nunca interactúa con la UI directamente.
- Opera en hilos independientes (Thread daemon).

9.3 PyralineDashboard

Paquete: UI.form

Rol en N-Tier: User Interface

Responsabilidad: Pantalla principal del sistema. Centraliza la visualización de temperatura, estado del sistema, configuración del umbral y bitácora histórica de eventos térmicos.

Componentes funcionales:

- Vista HOME (monitoreo en tiempo real)
- Vista CONFIGURACIÓN (ajuste de umbral)
- Vista ALERTAS (bitácora HTML)

Métodos clave:

- `setModoAlerta(boolean activa)` → controla sirena sonora y parpadeo visual.
- `actualizarEstadoHardware(boolean conectado)` → indica desconexión o conexión del sensor.
- `actualizarMonitoreo(float temp, boolean esAlerta)` → refresca temperatura y estado.
- `refrescarHistorial()` → consulta SQLite y genera HTML dinámico.
- `setPollingService(ArduinoPollingService service)` → enlaza la lógica de negocio con la UI.

Reglas técnicas:

- Toda actualización gráfica se ejecuta en `SwingUtilities.invokeLater`.
 - No contiene lógica de negocio ni acceso directo a hardware.
-

9.4 PyralineLogin

Paquete: UI.form

Rol en N-Tier: User Interface

Responsabilidad: Gestionar el acceso inicial al sistema, validando credenciales y orquestando el arranque de los servicios principales.

Flujo principal:

1. Captura usuario y contraseña.
2. Valida acceso mediante `UsuarioBL`.
3. Inicializa `PyralineDashboard`.
4. Arranca `ArduinoPollingService` y `ArduinoSensor`.

Regla técnica: La autenticación nunca accede directamente a la base de datos.

9.5 UsuarioBL

Paquete: BusinessLogic.logic

Rol en N-Tier: Business Logic

Responsabilidad: Aplicar reglas de negocio para la autenticación de usuarios antes de acceder a la capa de datos.

Método principal:

- validarAcceso(String email, String password)

Reglas técnicas:

- Evita consultas innecesarias si los datos son inválidos.
 - Toda falla técnica se encapsula en AppException.
-

9.6 DataHelperSQLiteDAO

Paquete: DataAccess.helpers

Rol en N-Tier: Data Access Component

Responsabilidad: Proveer una implementación genérica para operaciones CRUD usando SQLite y reflexión.

Métodos implementados:

- readAll()
- mapResultSetToEntity(ResultSet rs)

Reglas técnicas:

- Los DAOs concretos heredan de esta clase.
 - El mapeo campo-columna se realiza por nombre exacto.
-

9.7 BusinessFactory

Paquete: BusinessLogic.factory

Responsabilidad: Centralizar la creación y uso de DAOs mediante una fábrica genérica.

Métodos principales:

- getAll()
 - getBy(Integer id)
 - add(T oT)
 - upd(T oT)
 - del(Integer id)
-

9.8 AppConfig

Paquete: Infrastructure.config

Responsabilidad: Gestionar la configuración global del sistema mediante app.properties.

Métodos clave:

- getDATABASE()
 - getLOGFILE()
 - getUmbralPersistido()
 - guardarUmbral(float valor)
-

9.9 AppException

Paquete: Infrastructure.exception

Responsabilidad: Centralizar el manejo de errores técnicos y su persistencia en archivos de log.

Regla global:

Toda excepción del sistema Pyraline debe encapsularse mediante AppException.

9.10 AppStyle

Paquete: Infrastructure.style

Responsabilidad: Definir la identidad visual del sistema de forma centralizada.

Regla: Nunca definir colores, fuentes o bordes directamente en formularios.

10. Convenciones de Desarrollo

Esta sección define las **reglas obligatorias de desarrollo** del proyecto Pyraline. Su cumplimiento garantiza coherencia, mantenibilidad y alineación con la arquitectura N-Tier. Todo desarrollador que modifique o extienda el sistema **debe seguir estas convenciones**.

10.1 Convenciones de Paquetes

La estructura de paquetes refleja directamente la arquitectura N-Tier:

Capa	Paquete base
------	--------------

User Interface	UserInterface
----------------	---------------

Business Logic	BusinessLogic
----------------	---------------

Data Access	DataAccess
-------------	------------

Infrastructure	Infrastructure
----------------	----------------

Reglas:

- Los paquetes se nombran en **PascalCase**.
 - No se permite acceso cruzado entre capas no adyacentes.
 - La UI nunca accede directamente a DataAccess.
-

10.2 Convenciones de Clases

Elemento	Convención
----------	------------

Nombre de clase	PascalCase
-----------------	------------

Clases DAO	Sufijo DAO
------------	------------

Clases DTO	Sufijo DTO
------------	------------

Servicios	Sufijo Service
-----------	----------------

Excepciones	Sufijo Exception
-------------	------------------

Reglas:

- Una clase debe tener **una sola responsabilidad**.
 - No se permite lógica de negocio en formularios (UI).
 - No se permiten clases utilitarias genéricas sin propósito claro.
-

10.3 Convenciones de Métodos

Tipo	Convención
------	------------

Estilo	camelCase
--------	-----------

Forma verbal	Infinitivo
--------------	------------

Ejemplos correctos:

- validarAcceso()
- procesarLectura()
- guardarUmbral()
- actualizarEstadoHardware()

Reglas:

- Los métodos deben expresar **acciones claras**.
 - Evitar abreviaturas ambiguas.
 - Un método no debe exceder una responsabilidad lógica.
-

10.4 Convenciones para DAOs y DTOs

DAO (Data Access Object):

- Encapsulan únicamente acceso a datos.
- Heredan de DataHelperSQLiteDAO.
- No contienen lógica de negocio.

DTO (Data Transfer Object):

- Contienen solo atributos y validaciones.
 - Validan rangos permitidos (ej. temperatura).
 - No contienen lógica de persistencia.
-

10.5 Convenciones de Base de Datos (SQLite)

Reglas SQL:

- Tablas en snake_case.
- Claves primarias autoincrementales.
- Fechas almacenadas en formato ISO (YYYY-MM-DD HH:MM:SS).

Ejemplo:

```
CREATE TABLE estado_alerta (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
```

descripcion TEXT NOT NULL

);

10.6 Convenciones de Manejo de Errores

- Toda excepción debe encapsularse en AppException.
 - No se permiten printStackTrace() en producción.
 - Los errores críticos se registran en log físico.
-

10.7 Convenciones de Interfaz Gráfica

- Colores y fuentes se definen exclusivamente en AppStyle.
 - Toda actualización gráfica se ejecuta en el hilo EDT.
 - No se permite lógica de negocio en componentes Swing.
-

10.8 Convenciones de Recursos

Tipo Ubicación

Imágenes Infrastructure/resources/img

Sonidos Infrastructure/resources/sounds

Formatos permitidos:

- Imágenes: PNG
- Audio: WAV

11. Flujo General del Sistema

Esta sección describe el **flujo completo de funcionamiento de Pyraline**, desde el inicio de sesión hasta la detección de eventos térmicos y su persistencia. El objetivo es que cualquier desarrollador comprenda **cómo interactúan las capas del sistema** y en qué orden se ejecutan los procesos.

11.1 Flujo de Inicio y Autenticación

Capas involucradas: User Interface → Business Logic → Data Access

Secuencia:

1. El usuario abre la aplicación.
2. Se muestra la pantalla PyralineLogin.
3. El usuario ingresa credenciales.
4. PyralineLogin invoca a UsuarioBL.validarAcceso().
5. UsuarioBL consulta UsuarioDAO.
6. Si las credenciales son válidas y el usuario está activo:
 - Se inicializa PyralineDashboard.
 - Se cargan configuraciones persistidas.
7. Si falla la autenticación:
 - Se muestra mensaje de error mediante AppMSG.

Regla: La UI no accede directamente a la base de datos.

11.2 Flujo de Inicialización del Sistema

Capas involucradas: Infrastructure → Business Logic → User Interface

Secuencia:

1. PyralineDashboard solicita instancia de ArduinoPollingService.
2. ArduinoPollingService:
 - Carga el umbral desde AppConfig.
 - Inicializa variables internas.
3. Se instancia ArduinoSensor.
4. ArduinoSensor detecta automáticamente el puerto serial.
5. Se establece la comunicación con el Arduino.

Caso especial: Si el sensor no está conectado, el sistema continúa en modo monitoreo pasivo.

11.3 Flujo de Monitoreo Continuo

Capas involucradas: Infrastructure → Business Logic → User Interface

Secuencia:

1. ArduinoSensor recibe datos del Arduino.
2. La lectura se envía a ArduinoPollingService.
3. ArduinoPollingService:
 - Convierte la lectura a valor numérico.
 - Compara la temperatura con el umbral.
 - Determina el estado térmico.
4. Se notifica a PyralineDashboard.
5. La UI actualiza:
 - Temperatura mostrada.
 - Estado visual del sistema.

Regla: Las actualizaciones gráficas se ejecutan en el hilo EDT.

11.4 Flujo de Detección de Alerta

Capas involucradas: Business Logic → Data Access → User Interface

Secuencia:

1. La temperatura supera el umbral.
2. ArduinoPollingService detecta el cambio de estado.
3. Se registra el evento en SQLite.
4. Se activa el modo de alerta.
5. PyralineDashboard:
 - Activa sirena sonora.
 - Muestra parpadeo visual.
 - Indica estado CRÍTICO.

11.5 Flujo de Normalización

Capas involucradas: Business Logic → Data Access → User Interface

Secuencia:

1. La temperatura retorna a valores seguros.
 2. Se detecta la normalización.
 3. Se registra el evento correspondiente.
 4. Se desactiva la alerta.
 5. La UI actualiza el estado a NORMAL.
-

11.6 Flujo de Gestión de Umbral

Capas involucradas: User Interface → Business Logic → Infrastructure

Secuencia:

1. El usuario ajusta el umbral desde CONFIGURACIÓN.
 2. PyralineDashboard notifica a ArduinoPollingService.
 3. ArduinoPollingService:
 - Valida el nuevo valor.
 - Persiste el cambio mediante AppConfig.
 4. Se muestra confirmación visual.
-

11.7 Flujo de Bitácora de Alertas

Capas involucradas: User Interface → Data Access

Secuencia:

1. El usuario accede a ALERTAS.
 2. PyralineDashboard consulta eventos históricos.
 3. Los registros se formatean en HTML.
 4. Se muestra el historial con scroll.
-

11.8 Flujo de Vaciado de Historial

Capas involucradas: User Interface → Data Access

Secuencia:

1. El usuario presiona “Vaciar Historial Físico”.
 2. Se solicita confirmación.
 3. Si confirma:
 - o Se eliminan los registros de SQLite.
 - o Se muestra mensaje de éxito.
 4. La UI refresca la vista.
-

11.9 Flujo de Cierre de Sesión

Capas involucradas: User Interface

Secuencia:

1. El usuario presiona “Cerrar Sesión”.
2. Se detienen servicios activos.
3. Se retorna a PyralineLogin.

12. Arquitectura N-Tier Aplicada a Pyraline

El sistema Pyraline está diseñado siguiendo el patrón de **Arquitectura en Capas (N-Tier)**, con el objetivo de lograr **desacoplamiento, escalabilidad, mantenibilidad y claridad estructural**. Esta sección describe cómo dicha arquitectura se implementa **de forma concreta** en el proyecto.

12.1 Descripción General de la Arquitectura

Pyraline se estructura en **cuatro capas claramente definidas**, cada una con responsabilidades específicas:

1. **User Interface (UI)**
2. **Business Logic (BL)**
3. **Data Access (DA)**
4. **Infrastructure (INF)**

Cada capa solo puede comunicarse con la capa inmediatamente inferior, garantizando bajo acoplamiento.

12.2 Mapeo de Capas vs Estructura del Proyecto

Capa N-Tier Paquetes reales del proyecto

User Interface UserInterface.Form, UserInterface.Style

Business Logic BusinessLogic

Data Access DataAccess.DAO, DataAccess.DTO, DataAccess.Helpers,
DataAccess.Interfaces

Infrastructure Infrastructure, Infrastructure.resources, Infrastructure.Tools

Este mapeo permite identificar rápidamente dónde debe implementarse cada nueva funcionalidad.

12.3 Capa User Interface (UI)

Responsabilidad:

- Interacción directa con el usuario
- Renderizado gráfico
- Captura de eventos

Clases representativas:

- PyralineLogin
- PyralineDashboard
- BackgroundPanel

Reglas:

- No contiene lógica de negocio.
 - No accede a base de datos ni hardware.
 - Toda actualización gráfica se ejecuta en el hilo EDT.
-

12.4 Capa Business Logic (BL)

Responsabilidad:

- Aplicar reglas de negocio
- Coordinar flujos del sistema
- Validar información

Clases representativas:

- ArduinoPollingService
- UsuarioBL
- FactoryBL

Reglas:

- No conoce detalles de la UI.
 - Accede a datos exclusivamente mediante DAOs.
 - Orquesta la comunicación entre capas.
-

12.5 Capa Data Access (DA)

Responsabilidad:

- Persistencia de datos
- Ejecución de sentencias SQL
- Mapeo objeto–relacional

Componentes:

- DAOs (UsuarioDAO, EstadoAlertaDAO, etc.)
- DTOs (UsuarioDTO, PYRALINEDTO, etc.)
- DataHelperSQLiteDAO

Reglas:

- No contiene lógica de negocio.
 - No interactúa con la UI.
 - Todas las consultas deben ser encapsuladas.
-

12.6 Capa Infrastructure (INF)

Responsabilidad:

- Servicios transversales
- Comunicación con hardware
- Configuración del sistema
- Estilos visuales

Componentes clave:

- ArduinoSensor
- AppConfig
- AppException
- AppStyle
- Recursos multimedia

Reglas:

- No contiene reglas de negocio.
 - Puede ser reutilizada por múltiples capas.
-

12.7 Justificación del Uso de N-Tier

El uso de N-Tier en Pyraline permite:

- ✓ Aislar fallos por capa
- ✓ Facilitar pruebas y mantenimiento
- ✓ Permitir cambios de tecnología (ej. otra BD)
- ✓ Escalar el sistema sin reescritura completa

Esta arquitectura es especialmente adecuada para sistemas de monitoreo continuo y aplicaciones críticas como la prevención de incendios.

12.8 Relación con Diagramas de Diseño

La arquitectura implementada corresponde directamente con:

- Diagrama de Arquitectura N-Tier
- Diagrama de Clases
- Modelo Entidad–Relación (MER)

Estos diagramas deben mantenerse sincronizados con la estructura del código.