MANUAL DE ESTÁNDARES - PROYECTO LIMINALIS

Tabla de Contenidos

---

1. Arquitectura del Proyecto

1.1 Estructura de Capas

El proyecto sigue una **arquitectura de tres capas**:

```
src/
├── UserInterface/           # Capa de Presentación
│   ├── Screen/             # Pantallas de la aplicación
│   ├── Utility/            # Utilidades de UI
│   └── Resources/          # Recursos visuales
├── BusinessLogic/          # Capa de Lógica de Negocio
├── DataAccessComponent/    # Capa de Acceso a Datos
│   ├── DAOs/              # Data Access Objects
│   ├── DTOs/              # Data Transfer Objects
│   ├── Helpers/           # Ayudantes de conexión
│   └── Interfaces/        # Contratos de interfaz
└── Infrastructure/         # Componentes transversales
    └── Tools/             # Herramientas generales
```

1.2 Principios de Diseño

- **Separación de responsabilidades**: Cada capa tiene una responsabilidad específica
- **Bajo acoplamiento**: Las capas se comunican a través de interfaces bien definidas
- **Alta cohesión**: Cada clase tiene una única responsabilidad
- **Uso de patrones**: DAO, DTO, Singleton para conexiones

---

2. Convenciones de Nomenclatura

2.1 Paquetes

- **Nomenclatura**: PascalCase sin espacios
- **Ejemplos**:
  - `BusinessLogic`
  - `DataAccessComponent`
  - `UserInterface`

2.2 Clases

Clases de Lógica de Negocio (BL)

- **Formato**: `<Entidad>BL`
- **Ejemplos**:
  - `UserPlayerBL.java`
  - `QuestionBL.java`
  - `AnswerBL.java`

Data Access Objects (DAO)

- **Formato**: `<Entidad>DAO`
- **Ejemplos**:
  - `UserPlayerDAO.java`
  - `CategoryDAO.java`
  - `QuestionDAO.java`

Data Transfer Objects (DTO)

- **Formato**: `<Entidad>DTO`
- **Ejemplos**:
  - `UserPlayerDTO.java`
  - `QuestionDTO.java`
  - `AnswerDTO.java`

Pantallas de UI

- **Formato**: Nombres descriptivos en PascalCase
- **Ejemplos**:
  - `CreatePlayer.java`
  - `GameScreen.java`
  - `LoginScreen.java`
  - `MainMenu.java`

Utilidades y Configuración

- **Formato**: Nombre descriptivo + "Config", "Helper", "Utility"
- **Ejemplos**:
  - `StyleConfig.java`
  - `ReusableMethods.java`
  - `AppConfig.java`

- `DataHelperSQLite.java`

2.3 Métodos

Métodos CRUD

- **Create**: `create()`
- **Read**: `readById()`, `readByName()`, `readAllStatus()`
- **Update**: `update()`, `updateAll()`
- **Delete/Status**: `changeStatus()`

Métodos de validación

- **Formato**: Verbo en presente + complemento
- **Ejemplos**:
  - `exists(String username)`
  - `getIdByUsername(String username)`
  - `getAllActivePlayers(boolean status)`

Métodos de UI

- **Formato**: Verbo + sustantivo
- **Ejemplos**:
  - `createPlayerPanel()`
  - `gameMenu()`
  - `setupKeyBindings()`

2.4 Variables

Variables de instancia

- **Formato**: camelCase
- **Ejemplos**:

```
private Integer idPlayer;
private String userName;
private String creationDate;
private String modificateDate;
```

Constantes

- **Formato**: UPPER_SNAKE_CASE
- **Ejemplos**:

```
private static final String APP_PROPERTIES = "src/app.properties";
private static final String KEY_FILE_LOG = "df.logFile";
```

```
public static final String MSG_DEFAULT_ERROR = "Caracoles, ocurrio un
error...";
```

Variables de interfaz gráfica

- **Formato**: nombreComponente + Tipo
- **Ejemplos**:

```
public static JTextField playerNameField;
JButton create;
JButton goBack;
JLabel nameLabel;
JPanel textPanel;
```

---

3. Estándares de Base de Datos

3.1 Nomenclatura de Tablas

- **Formato**: PascalCase singular
- **Ejemplos**:
  - `UserType`
  - `UserPlayer`
  - `UserAdmin`
  - `Question`
  - `Answer`
  - `Category`

3.2 Nomenclatura de Campos

Claves Primarias

- **Formato**: `id<NombreTabla>`
- **Ejemplos**:
  - `idPlayer`
  - `idQuestion`
  - `idCategory`
  - `idUserType`

Claves Foráneas

- **Formato**: `id<TablaReferenciada>`
- **Ejemplos**:
  - `idUserType` (en UserPlayer)
  - `idCategory` (en Question)
  - `idQuestion` (en Answer)

Campos regulares

- **Formato**: PascalCase
- **Ejemplos**:
  - `Name`
  - `Description`
  - `Status`
  - `Score`
  - `UserName`
  - `Password`

Campos de auditoría

- **Formato**: PascalCase con sufijo descriptivo
- **Campos estándar**:

```
CreationDate DATETIME NOT NULL DEFAULT(datetime('now','localtime'))
ModificateDate DATETIME DEFAULT NULL
Status VARCHAR(10) NOT NULL DEFAULT 'Activo'
```

3.3 Tipos de Datos

- **IDs**: `INTEGER PRIMARY KEY AUTOINCREMENT`
- **Nombres/Textos cortos**: `VARCHAR(n)`
- **Textos largos**: `VARCHAR(255)`
- **Números enteros**: `INTEGER`
- **Fechas**: `DATETIME`
- **Estado**: `VARCHAR(10)` con valor por defecto 'Activo'

3.4 Constraints y Valores por Defecto

```
-- Clave primaria con autoincremento
idPlayer INTEGER PRIMARY KEY AUTOINCREMENT

-- Campos únicos
Name VARCHAR(50) NOT NULL UNIQUE

-- Claves foráneas
idUserType INTEGER REFERENCES UserType(idUserType)

-- Estado con valor por defecto
Status VARCHAR(10) NOT NULL DEFAULT 'Activo'

-- Fecha de creación automática
CreationDate DATETIME NOT NULL DEFAULT(datetime('now','localtime'))

-- Fecha de modificación nullable
ModificateDate DATETIME DEFAULT NULL
```

3.5 Scripts de Base de Datos

- **DDL (Data Definition Language)**: `01DDL.txt` / `01DDL.sql`
  - Contiene CREATE TABLE, DROP TABLE
  - Debe incluir DROP IF EXISTS antes de CREATE
  - Orden: Eliminar tablas dependientes primero
- **DML (Data Manipulation Language)**: `DML.txt` / `DML.sql`
  - Contiene INSERT, UPDATE
  - Datos de prueba y seed data
  - Formato multiline para INSERT múltiples

---

4. Estándares de Código Java

4.1 Estructura de Clases

```
package <paquete>;

// Imports organizados por grupos
import java.util.*;          // Java standard
import javax.swing.*;        // Java extended

import BusinessLogic.*;      // Capa BL
import DataAccessComponent.*; // Capa DAC
import Infrastructure.*;      // Infraestructura

public class NombreClase {
    // 1. Constantes
    private static final String CONSTANTE = "valor";

    // 2. Variables de clase (static)
    private static Connection connection;

    // 3. Variables de instancia
    private Integer id;
    private String name;

    // 4. Constructores
    public NombreClase() {}

    // 5. Métodos públicos
    public void metodoPublico() {}

    // 6. Métodos privados/protected
    private void metodoPrivado() {}
}
```

4.2 Comentarios

- **Comentarios de línea**: Para explicaciones breves

```
// Cerrar recursos
rs.close();
```

- **Comentarios de bloque**: Evitar en exceso, el código debe ser autodocumentado
- **Comentarios TODO**: No utilizados en el proyecto
- **Comentarios de base de datos en SQL**:

```
-- database: ../Database/triv.sqlite
```

4.3 Formato de Código

- **Indentación**: 4 espacios
- **Llaves**: Estilo K&R (llave de apertura en la misma línea)

```
public void metodo() {
    if (condicion) {
        // código
    }
}
```

- **Longitud de línea**: Máximo ~120 caracteres (flexibilidad para consultas SQL)
- **Espacios en blanco**:
  - Después de comas: `metodo(param1, param2)`
  - Alrededor de operadores: `a = b + c`
  - No antes de paréntesis en llamadas a métodos

---

5. Capa de Lógica de Negocio (BL)

5.1 Responsabilidades

- Implementar reglas de negocio
- Validar datos antes de persistir
- Coordinar operaciones entre DAOs
- Manejar transacciones lógicas

5.2 Estructura Estándar

```java
package BusinessLogic;

import DataAccessComponent.DAOs.*;
import DataAccessComponent.DTOs.*;
import Infrastructure.AppException;
import java.util.List;

public class EntidadBL {

    // Métodos CRUD principales
    public static Boolean create(params) throws AppException {
        try {
            // Validaciones de negocio
            if (exists(username)) {
                // Lógica alternativa
            }

            // Preparar DTO
            EntidadDTO dto = new EntidadDTO();
            dto.setAtributo(valor);

            // Llamar al DAO
            EntidadDAO dao = new EntidadDAO();
            return dao.create(dto);

        } catch (AppException e) {
            throw e;
        } catch (Exception e) {
            throw new AppException("Mensaje descriptivo", e,
                                   EntidadBL.class, "create");
        }
    }

    // Métodos de validación
    public static Boolean exists(String identifier) throws AppException {
        // Implementación
    }

    // Métodos de consulta
    public static List<EntidadDTO> getAll(boolean status)
            throws AppException {
        // Implementación
    }
}
```

5.3 Estándares de Métodos BL

- **Métodos estáticos**: Preferir métodos estáticos cuando no se requiere estado
- **Manejo de excepciones**:

    - Siempre usar try-catch
    - Relanzar AppException con información contextual
    - Diferenciar entre AppException (esperada) y Exception (inesperada)

- **Validaciones**: Realizar validaciones antes de llamar al DAO
- **Mensajes de error**: Descriptivos en español para el usuario

---

6. Capa de Acceso a Datos (DAOs)

6.1 Responsabilidades

- Ejecutar operaciones CRUD contra la base de datos
- Mapear ResultSet a DTOs
- Gestionar PreparedStatements
- No contener lógica de negocio

6.2 Estructura Estándar

```
package DataAccessComponent.DAOs;

import DataAccessComponent.DTOs.*;
import DataAccessComponent.Helpers.DataHelperSQLite;
import DataAccessComponent.Interfaces.IDAO;
import Infrastructure.AppException;

import java.sql.*;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
import java.util.ArrayList;
import java.util.List;

public class EntidadDAO extends DataHelperSQLite
                        implements IDAO<EntidadDTO> {

    @Override
    public boolean create(EntidadDTO entity) throws AppException {
        DateTimeFormatter dtf = DateTimeFormatter
                                .ofPattern("yyyy-MM-dd HH:mm:ss");
        LocalDateTime now = LocalDateTime.now();

        String query = "INSERT INTO Tabla (Campo1, Campo2, CreationDate) "
                     + "VALUES (?, ?, ?);";
        try {
            Connection conn = openConnection();
            PreparedStatement pstmt = conn.prepareStatement(query);
            pstmt.setString(1, entity.getCampo1());
            pstmt.setString(2, entity.getCampo2());
            pstmt.setString(3, dtf.format(now).toString());
            pstmt.executeUpdate();
            return true;
```

```
        } catch (Exception e) {
            throw new AppException("Error al crear: " + entity.getNombre(),
                                   e, getClass(), "create");
        }
    }

    @Override
    public List<EntidadDTO> readAllStatus(boolean status)
            throws AppException {
        String query = "SELECT * FROM Tabla";
        if (status) {
            query += " WHERE Status = 'Activo'";
        }

        List<EntidadDTO> lista = new ArrayList<>();
        try {
            Connection conn = openConnection();
            PreparedStatement pstmt = conn.prepareStatement(query);
            ResultSet rs = pstmt.executeQuery();

            while (rs.next()) {
                EntidadDTO dto = new EntidadDTO();
                dto.setId(rs.getInt("id"));
                dto.setNombre(rs.getString("Name"));
                // Mapear todos los campos
                lista.add(dto);
            }
            rs.close();
            pstmt.close();

        } catch (Exception e) {
            throw new AppException("Error al leer entidades",
                                   e, getClass(), "readAllStatus");
        }
        return lista;
    }
}
```

6.3 Estándares de DAOs

Formato de Fechas

```
DateTimeFormatter dtf = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
LocalDateTime now = LocalDateTime.now();
pstmt.setString(n, dtf.format(now).toString());
```

Consultas SQL

- **Queries legibles**: Una cláusula por línea cuando es necesario

- **PreparedStatements**: SIEMPRE usar para prevenir SQL injection
- **Concatenación de queries**: Usar + para continuación

```
String query = "SELECT campo1, campo2, campo3 "
             + "FROM Tabla "
             + "WHERE Status = ?";
```

Manejo de Conexiones

- **No cerrar conexión**: Dejar comentado `// conn.close();`
- **Cerrar recursos**: Siempre cerrar ResultSet y PreparedStatement

```
rs.close();
pstmt.close();
// conn.close(); // Mantener comentado
```

Mapeo de ResultSet a DTO

```
while (rs.next()) {
    EntidadDTO dto = new EntidadDTO();
    dto.setIdEntidad(rs.getInt("idEntidad"));
    dto.setNombre(rs.getString("Name"));
    dto.setEstado(rs.getString("Status"));
    dto.setFechaCreacion(rs.getString("CreationDate"));
    dto.setFechaModificacion(rs.getString("ModificateDate"));
    lista.add(dto);
}
```

---

7. Objetos de Transferencia de Datos (DTOs)

7.1 Responsabilidades

- Transportar datos entre capas
- No contener lógica de negocio
- Proporcionar getters y setters para todos los campos

7.2 Estructura Estándar

```
package DataAccessComponent.DTOs;

public class EntidadDTO {
    // Atributos que mapean a la tabla
    private Integer idEntidad;
```

```
    private String name;
    private String status;
    private String creationDate;
    private String modificateDate;

    // Constructor vacío
    public EntidadDTO() {
    }

    // Constructor con campos mínimos
    public EntidadDTO(String name) {
        this.name = name;
    }

    // Constructor completo (sin fechas de auditoría)
    public EntidadDTO(Integer idEntidad, String name) {
        this.idEntidad = idEntidad;
        this.name = name;
    }

    // Constructor con fechas
    public EntidadDTO(Integer idEntidad, String name,
                      String creationDate, String modificateDate) {
        this.idEntidad = idEntidad;
        this.name = name;
        this.creationDate = creationDate;
        this.modificateDate = modificateDate;
    }

    // Getters y Setters
    public Integer getIdEntidad() {
        return idEntidad;
    }

    public void setIdEntidad(Integer idEntidad) {
        this.idEntidad = idEntidad;
    }

    // ... más getters y setters
}
```

7.3 Estándares de DTOs

Constructores

- **Constructor vacío**: Siempre presente
- **Constructores sobrecargados**: Según casos de uso
  - Constructor con campos mínimos requeridos
  - Constructor completo sin auditoría
  - Constructor completo con auditoría

Tipos de Datos

- **IDs y cantidades**: `Integer` (no int, permite null)
- **Textos**: `String`
- **Fechas**: `String` (formato "yyyy-MM-dd HH:mm:ss")
- **Booleanos**: `Boolean` (no boolean, permite null)

Nombres de Atributos

- Deben coincidir con los nombres de campos de la base de datos
- Usar camelCase en Java, PascalCase en BD
- Ejemplos:
  - BD: `CreationDate` → Java: `creationDate`
  - BD: `ModificateDate` → Java: `modificateDate`
  - BD: `idPlayer` → Java: `idPlayer`

---

8. Interfaz de Usuario (UI)

8.1 Estructura de Pantallas

Organización

```
UserInterface/
├── Screen/          # Pantallas principales
│   ├── MainFrame.java           # Frame principal
│   ├── LiminalisSystem.java     # Sistema de arranque
│   ├── LoadingScreen.java       # Pantalla de carga
│   ├── MainMenu.java            # Menú principal
│   ├── CreatePlayer.java        # Crear jugador
│   ├── GameScreen.java          # Pantalla de juego
│   └── ...
├── Utility/         # Utilidades reutilizables
│   ├── StyleConfig.java         # Configuración de estilos
│   ├── ReusableMethods.java     # Métodos reutilizables
│   └── ImageBackgroundPanel.java # Paneles personalizados
└── Resources/       # Recursos visuales
```

8.2 Estándares de Pantallas

Método Principal

Cada pantalla debe tener un método estático que retorne JPanel:

```
public static JPanel nombrePantalla() {
    // Configuración de componentes
    JPanel panel = new JPanel();
    // ...
```

```
    return panel;
}
```

Configuración de Fuentes

```
Font titleFont = new Font("Cooper Black", Font.BOLD, 40);
Font textFont = new Font("Comic Sans MS", Font.BOLD, 18);
Font buttonFont = new Font("Comic Sans MS", Font.PLAIN, 16);
```

Colores Estándar

```
// Definidos en StyleConfig
Color buttonPrimary = new Color(217, 163, 187);    // Rosa
Color buttonSecondary = new Color(171, 157, 204);  // Morado
Color keyboardButtons = new Color(218, 209, 237);  // Lila claro
Color titleColorPanel = new Color(173, 160, 219);  // Morado suave
```

Creación de Botones

```
JButton button = StyleConfig.createButton(
    "Texto del Botón",
    StyleConfig.buttonPrimary(),
    200,  // width
    50    // height
);
```

Navegación entre Pantallas

```
button.addActionListener(e -> {
    MainFrame.setContentPane(OtraPantalla.pantalla());
});
```

8.3 Manejo de Eventos

ActionListeners

```
button.addActionListener(e -> {
    try {
        // Validaciones
        if (campo.getText().trim().isEmpty()) {
```

```
            JOptionPane.showMessageDialog(panel,
                "Mensaje de error",
                "Error",
                JOptionPane.ERROR_MESSAGE);
            return;
        }

        // Lógica de negocio
        EntidadBL bl = new EntidadBL();
        bl.metodo(parametros);

        // Navegación exitosa
        MainFrame.setContentPane(OtraPantalla.pantalla());

    } catch (AppException appEx) {
        JOptionPane.showMessageDialog(panel,
            appEx.getMessage(),
            "Error",
            JOptionPane.ERROR_MESSAGE);
        appEx.printStackTrace();
    }
});
```

8.4 Layouts Comunes

```
// FlowLayout para alineación simple
JPanel panel = new JPanel(new FlowLayout(FlowLayout.CENTER));

// BorderLayout para divisiones
JPanel panel = new JPanel(new BorderLayout());

// BoxLayout para apilar verticalmente
JPanel panel = new JPanel();
panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));

// GridLayout para grillas
JPanel panel = new JPanel(new GridLayout(rows, cols, hgap, vgap));
```

8.5 Validaciones en UI

```
// Validar campo vacío
if (campo.getText().trim().isEmpty()) {
    JOptionPane.showMessageDialog(panel,
        "Nombre no Válido",
        "Error",
        JOptionPane.ERROR_MESSAGE);
    return;
}
```

```
// Validar formato
if (!campo.getText().matches(regex)) {
    // Mostrar error
}
```

---

9. Manejo de Excepciones

9.1 Clase AppException

Estructura

```
package Infrastructure;

public class AppException extends Exception {
    public AppException(String showMsg) {
        super(showMsg);
        saveLogFile(null, null, null);
    }

    public AppException(String showMsg, Exception e,
                        Class<?> clase, String metodo) {
        super(showMsg);
        saveLogFile(e.getMessage(), clase, metodo);
    }

    private void saveLogFile(String logMsg, Class<?> clase,
                             String metodo) {
        // Guarda en archivo de log
    }
}
```

9.2 Uso de AppException

En Capa BL

```
public static Boolean metodo(params) throws AppException {
    try {
        // Lógica de negocio
        EntidadDAO dao = new EntidadDAO();
        return dao.metodo(params);

    } catch (AppException e) {
        throw e;  // Re-lanzar AppException sin modificar
    } catch (Exception e) {
        throw new AppException(
            "Mensaje descriptivo para el usuario",
            e,
```

```
            ClaseActual.class,
            "nombreMetodo"
        );
    }
}
```

En Capa DAO

```
@Override
public boolean metodo(params) throws AppException {
    try {
        // Operación de base de datos

    } catch (Exception e) {
        throw new AppException(
            "Error al realizar operación: " + detalle,
            e,
            getClass(),
            "metodo"
        );
    }
}
```

En Capa UI

```
try {
    // Llamada a BL
    EntidadBL.metodo(params);

    // Acción exitosa
    MainFrame.setContentPane(OtraPantalla.pantalla());

} catch (AppException appEx) {
    JOptionPane.showMessageDialog(
        panel,
        appEx.getMessage(),  // Mensaje amigable
        "Error",
        JOptionPane.ERROR_MESSAGE
    );
    appEx.printStackTrace();  // Log en consola
}
```

9.3 Mensajes de Error

- **Para el usuario**: Claros, en español, sin tecnicismos
- **Para el log**: Detallados, incluyen clase y método
- **Formato**: "No se pudo [acción]: [detalle]"

Ejemplos:

```
"No se pudo crear el jugador: PlayerName"
"Error al obtener el ID del jugador: PlayerName"
"No se pudo establecer conexión con la base de datos"
"Caracoles, ocurrió un error inesperado! Contacta al administrador"
```

---

10. Configuración y Propiedades

10.1 Archivo app.properties

```
# Ubicación en: src/app.properties

# Base de datos
db.File=Storage/Database/triv.sqlite

# Logs
df.logFile=Logs/app.log

# Otras configuraciones
app.name=Liminalis
```

10.2 Clase AppConfig

Estructura

```
package Infrastructure;

import java.io.FileInputStream;
import java.io.IOException;
import java.util.Properties;

public class AppConfig {
    private static final Properties props = new Properties();

    // Nombres de claves
    private static final String APP_PROPERTIES = "src/app.properties";
    private static final String KEY_FILE_LOG = "df.logFile";
    private static final String KEY_DB_NAME = "db.File";

    // Métodos de acceso
    public static String getDatabase() {
        return getProperty(KEY_DB_NAME);
    }

    public static String getLogFile() throws IOException {
```

```
        String relativePath = getProperty(KEY_FILE_LOG);
        Path path = Paths.get(System.getenv("APPDATA")
                              + "\\Liminalis\\"
                              + relativePath);
        // Crear directorios si no existen
        Path dirPath = path.getParent();
        if (!Files.exists(dirPath)) {
            Files.createDirectories(dirPath);
        }
        return path.toString();
    }

    // Mensajes por defecto
    public static final String MSG_DEFAULT_ERROR =
        "Caracoles, ocurrió un error inesperado! Contacta al administrador";
    public static final String MSG_DEFAULT_CLASS = "undefined";
    public static final String MSG_DEFAULT_METHOD = "undefined";

    // Bloque de inicialización
    static {
        try (InputStream appProperties =
                new FileInputStream(APP_PROPERTIES)) {
            props.load(appProperties);
        } catch (IOException e) {
            System.err.println("ERROR al cargar ⚠ " + e.getMessage());
        }
    }

    private static String getProperty(String key) {
        return props.getProperty(key);
    }
}
```

10.3 Rutas de Almacenamiento

Estructura en APPDATA

```
%APPDATA%/Liminalis/
├── Storage/
│   └── Database/
│       └── triv.sqlite
└── Logs/
    └── app.log
```

Obtención de Rutas

```
// Base de datos
protected static Path getDatabasePath() {
```

```
    String appData = System.getenv("APPDATA");
    return Paths.get(
        appData,
        "Liminalis",
        "Storage",
        "Database",
        "triv.sqlite"
    );
}

// Archivo de log
Path logPath = Paths.get(
    System.getenv("APPDATA") + "\\Liminalis\\Logs\\app.log"
);
```

Checklist de Revisión de Código

Nomenclatura

- [ ] Clases en PascalCase con sufijos correctos (BL, DAO, DTO)
- [ ] Métodos en camelCase con verbos descriptivos
- [ ] Variables en camelCase
- [ ] Constantes en UPPER_SNAKE_CASE
- [ ] Paquetes en PascalCase sin espacios

Base de Datos

- [ ] Tablas en singular PascalCase
- [ ] IDs con formato `id<NombreTabla>`
- [ ] Campos de auditoría presentes (CreationDate, ModificateDate, Status)
- [ ] Uso de AUTOINCREMENT en PKs
- [ ] Constraints de UNIQUE donde corresponda

Código

- [ ] Excepciones manejadas con AppException
- [ ] Try-catch en cada método que puede fallar
- [ ] Mensajes de error descriptivos en español
- [ ] PreparedStatements en lugar de concatenación de SQL
- [ ] Recursos cerrados (ResultSet, PreparedStatement)
- [ ] Fecha y hora con DateTimeFormatter estándar

Capas

- [ ] BL contiene lógica de negocio, no acceso directo a BD
- [ ] DAO solo hace operaciones CRUD, sin lógica de negocio
- [ ] DTO solo transporta datos, sin lógica
- [ ] UI no contiene lógica de negocio

UI

- [ ] Validaciones antes de llamar a BL
- [ ] Manejo de excepciones con JOptionPane
- [ ] Uso de StyleConfig para estilos consistentes
- [ ] Navegación mediante MainFrame.setContentPane()

🎯 Buenas Prácticas Adicionales

Seguridad

- Siempre usar PreparedStatements
- No exponer mensajes de error técnicos al usuario
- Validar entrada del usuario
- Usar STATUS lógico en lugar de DELETE físico

Performance

- Mantener conexión singleton
- Cerrar solo ResultSet y PreparedStatement
- No cargar datos innecesarios en DTOs

Mantenibilidad

- Un método, una responsabilidad
- Nombres descriptivos sobre comentarios
- Consistencia en el uso de patrones
- Separación clara de responsabilidades por capa

Testing

- Probar validaciones en BL
- Probar operaciones CRUD en DAO
- Probar flujos completos en UI

Versión: 1.0
Fecha: Enero 2026
Proyecto: Sistema de Trivia Liminalis
Equipo: InFismal, Bachily, Pandy06, Grandote4x4, KarliBoo